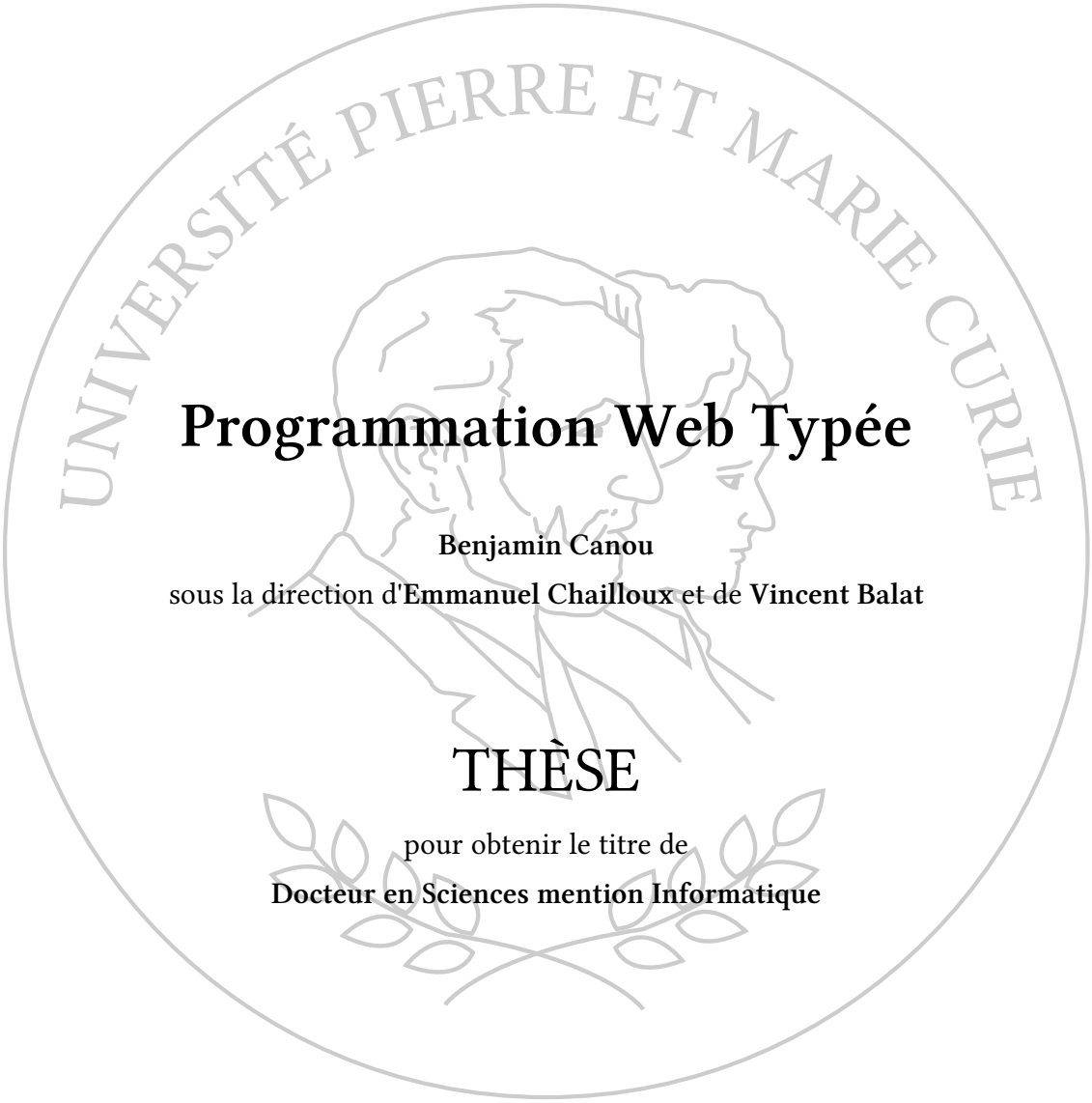
Université Pierre et Marie Curie
École Doctorale Informatique, Télécommunications et Électronique



Programmation Web Typée

Benjamin Canou
sous la direction d'**Emmanuel Chailloux** et de **Vincent Balat**

THÈSE

pour obtenir le titre de
Docteur en Sciences mention Informatique

soutenue le **4 octobre 2011**, devant le jury composé de

Président
Christian Queinnec — Université Pierre et Marie Curie

Directeurs
Emmanuel Chailloux — Université Pierre et Marie Curie
Vincent Balat — Université Paris Diderot

Rapporteurs
Peter Van Roy — Université Catholique de Louvain
Jacques Garrigue — Nagoya University

Examinateurs
Giuseppe Castagna — CNRS
Alain Frisch — LeXiFi
Manuel Serrano — INRIA Sophia Antipolis

Document mis en forme avec XeLaTeX et le jeu de fontes Libertine.

« If I had some duct tape, I could fix that. »
— Mac Gyver

Remerciements

Les premières personnes que je veux remercier sont mes directeurs, Emmanuel et Vincent, pour avoir su m'aiguiller grâce à leur recul scientifique et leur expertise technique, tout en me laissant la latitude nécessaire à m'exprimer librement au niveau des directions de recherche comme des solutions apportées.

Je souhaite ensuite dire un grand merci à tout l'équipage de l'APR, qui a su, lors de son appareillage sous la tempête, maintenir son bon esprit, sa cohésion sans faille, et a réussi à affirmer son identité. En particulier, merci à Michèle, notre capitaine, pour avoir su mener le navire jusqu'à son rythme de croisière.

Et puis bien sûr, si j'ai su conserver bonne humeur et motivation au cours de cette thèse, c'est en large partie grâce à la fine équipe du bureau 325, son savoir-vivre, ses discussions politiques et sociétales enjouées, ses réparties de bon goût et ses douces musiques. Et j'y inclus bien entendu la machine à café et le Père La Grolle.

Je remercie aussi bien entendu les membres des projets Ocsigen et PWD pour toutes les séances de discussion menées, qui furent bien souvent très intéressantes et constructives.

C'est grâce à cet environnement de travail en coopération, de liberté de recherche, de bonne entente et d'amitié que je n'hésiterais pas une seconde à me lancer à nouveau dans cette thèse, si le choix m'en était redonné.

Un grand merci aussi à mes rapporteurs, Jacques et Peter, pour avoir eu le courage d'affronter les deux-cent cinquante pages de ce document et pour leurs remarques constructives. Merci aussi à Alain, Christian, Giuseppe et Manuel d'avoir accepté de faire partie du jury de ma soutenance, et pour leur réactivité et flexibilité lors du choix de la date.

La thèse, c'est le moment où on apprend le métier de chercheur, mais pour beaucoup dont moi, celui d'enseignant. Je veux donc saluer les enseignants et équipes pédagogiques qui m'ont aidé à mes débuts et fait confiance par la suite, Pascal, Tong et les autres qui ont déjà été mentionnés ou pourront faire comme si je ne les avais pas (involontairement bien entendu) oubliés en écrivant leur nom ici : [. . . . .]. Merci aussi aux secrétaires et administratifs qui nous aident au quotidien dans notre travail, Odile, Irphane, Marilyn, Sabrina et les autres.

Mais la thèse c'est aussi plusieurs années d'une vie, qui ne se passe pas entièrement au laboratoire (encore que parfois…).

Mes premiers remerciements vont donc bien sûr à Carine (et Spirou et Flamme) de m'avoir accompagné durant ces années, de m'avoir épaulé, motivé, écouté, relu, et surtout supporté dans les moments difficiles.

Merci aussi à mes parents, mes frères et au reste de ma famille et de mes amis pour m'avoir soutenu et ne pas avoir trop râlé de ne pas me voir durant les périodes intensives de ces quatre années de travail.

Et pour finir, je ne peux oublier de remercier le Docteur, qui a permis indirectement l'aboutissement de ce travail en empêchant depuis son T.A.R.D.I.S. l'extermination du peuple de la Terre par les Daleks (un certain nombre de fois). Il en va bien sûr de même pour K-9 et Sarah-Jane, Jack Harkness et Gwen Cooper, Samantha Carter et toute l'équipe SG1 de l'armée américaine, ainsi que le grand et vaillant Son Goku.

Table des matières

II Modifications du document

III Vers un langage multi-tiers

🕮 Références

🕮 Annexes

1 Introduction

Le Web d'aujourd'hui est le résultat de vingt ans de progression permanente, en termes d'accès, d'usages et de techniques. Concrètement, de plus en plus de personnes ont accès au Web, depuis de plus en plus d'endroits de plus en plus variés et mobiles, en particulier depuis l'apparition des réseaux WiFi et 3G, peuplés de téléphones, de lecteurs multimédia ou de livres électroniques, proposant tous un accès au Web avec des avantages et des restrictions spécifiques. De même, grâce à l'évolution des moyens d'accès à internet, le nombre de connexions augmente, de même que la quantité de données transmises, permettant des sites visuellement riches ou au contenu volumineux comme la vidéo à la demande. Les connexions se font par ailleurs de plus en plus rapidement, permettant la mise à jour fréquente d'informations, mais aussi des applications complexes de travail en collaboration et de partage en temps réel.

Cette croissance de la dimension du Web, de sa complexité et de son audience ainsi que l'augmentation de la puissance et de la diversité de son parc de machines est allée de paire avec le développement des logiciels qui le font marcher : les navigateurs et les serveurs Web. À chaque étape, de nouvelles technologies ont été introduites, tout en continuant à faire fonctionner l'ensemble des pages Web existantes.

Malheureusement, la programmation du Web n'a pas reçu la même attention, et programmer un site Web aujourd'hui demande de maîtriser un nombre bien trop important de technologies, ainsi que leurs interactions. Cette complexité est aggravée par le fait que ces plates-formes et langages ont bien souvent une conception archaïque et un très faible niveau de fiabilité et de sécurité par rapport aux langages utilisés pour les applications classiques. Paradoxalement, les applications Web modernes, avec leur masse d'utilisateurs connectés en permanence et échangeant des données personnelles, demandent un niveau de fiabilité et de sécurité bien plus élevé que la plupart des applications classiques.

Le but de cette thèse est donc de proposer une nouvelle façon de programmer le Web, qui soit homogène, permettant de programmer de la même façon les parties de l'application fonctionnant dans le navigateur et celles fonctionnant sur le serveur, et intégrant dans le langage les mécanismes de communication. Un autre point important de notre approche est d'assurer la stabilité et la sécurité des applications Web en s'appuyant sur des vérifications préalables, effectuées par un système de types. La section 1.1 éclaircira ces notions de langage et de systèmes de types, et précisera le contexte dans lequel nous avons choisi de nous placer. Le lecteur déjà familier avec les langages et systèmes de types, en particulier avec la famille ML, peut s'affranchir de la lecture de cette introduction volontairement très générale.

Nous continuerons cette introduction en donnant une vision de l'état actuel du Web et de sa programmation. Pour ceci, dans la section 1.2 nous décrirons chronologiquement les grandes étapes de la construction du Web, en donnant un aperçu des technologies introduites à chaque avancée, en observant les problèmes apparus et souvent non résolus pour les programmer. Ensuite, la section 1.3 reprendra et étendra ces éléments historiques pour donner un aperçu de l'état des techniques, langages et environnement de développement Web répandus actuellement, illustré d'exemples concrets d'applications Web les utilisant. Nous évoquerons aussi rapidement d'autres travaux de recherche dans le domaine ; une revue de l'état de l'art spécifique plus approfondie sera donnée au cours de la thèse.

Nous serons alors à même d'exposer dans la fin de cette introduction notre vision du Web et de sa programmation, les problématiques spécifiques qui en découlent sur lesquelles nous nous sommes concentrées, et les solutions associées, que nous développons dans cette thèse.

1.1 Langages et systèmes de types

Avant de parler du Web et de sa programmation, nous devons expliciter deux mots clefs importants de cette thèse, la *programmation* et les *types*. Nous présentons les grandes familles de langages et de systèmes de types, ainsi que leurs critères de choix. Finalement nous expliquons dans quel cadre nous nous plaçons.

1.1.1 Langages et paradigmes

Les langages de programmation reposent en général sur un *paradigme* principal, c'est-à-dire une vision plus ou moins abstraite de la machine et des calculs qu'elle effectue. Les langages généralistes modernes donnent souvent accès à quelques primitives d'un ou plusieurs autres paradigmes.

Un langage impératif considère la machine comme un automate auquel on donne des instructions, les unes après les autres. Les instructions modifient l'état mémoire de la machine, effectuent des entrées/sorties avec le matériel, ou influent sur le flot d'instruction lui-même, par exemple pour répéter une série d'instructions un certain nombre de fois, ou choisir un chemin différent dans le programme en fonction d'une entrée de l'utilisateur. En général, un programme dans ce type de langages est structuré en blocs de code indépendants appelées *procédures*, ayant un nom pour pouvoir être appelées depuis les autres procédures par celui-ci, et prenant éventuellement des paramètres. Ce type de langages est le plus répandu, probablement car il ressemble au fonctionnement interne des ordinateurs.

Un langage fonctionnel abstrait au contraire la machine pour proposer une vision proche de la vision mathématique constructive du calcul. Le code est structuré en *fonctions*, et n'effectue pas directement de modifications sur les valeurs déjà présentes dans la mémoire, créant plutôt de nouvelles valeurs à partir de celles existantes, comme en mathématiques. Dans les langages fonctionnels, contrairement aux langages impératifs, la mémoire de la machine est le plus souvent abstraite au programmeur, et est automatiquement, et de façon sûre, gérée par le langage.

Dans les langages orientés objets, les valeurs manipulées sont les *objets*, des entités *encapsulant* à la fois des données, et les traitements associés à ces données. Le corps des traitements est parfois écrit en fonctionnel, le plus souvent en impératif. La large majorité des langages grand-public modernes est impératif et orientée objets.

Les langages logiques, moins répandus, partagent le qualificatif de langages *déclaratifs* avec les langages fonctionnels, mais au lieu de manipuler des fonctions, le programmeur manipule des relations. La programmation logique permet souvent de résoudre un problème à un niveau de détail plus haut que les langages impératifs, voire même fonctionnels. En contrepartie, le langage implique l'utilisation d'une machinerie plus complexe, et moins prévisible, que celles des types de langages précédents.

1.1.2 Exécution d'un programme

Un processeur ne comprend qu'un langage spécifique et difficile à programmer pour un humain qu'on appelle le langage machine. Pour exécuter un programme écrit dans un langage de plus haut niveau comme ceux décrits à la section précédente, destinés à être écrits et lus par des humains, il faut utiliser une technique pour le transformer en instructions compréhensibles par le processeur. Il existe plusieurs grandes techniques pour arriver à cette fin, que nous présentons dans cette section. Le choix de la technique à utiliser se fait en fonction du langage et du système de types utilisés, mais aussi des besoins de l'utilisateur en termes de fiabilité, simplicité, performances ou sécurité.

Compilation Compiler un programme signifie transformer le programme source en langage de haut niveau vers un programme équivalent en langage machine. Cette transformation est faite entièrement avant d'exécuter le programme, par un programme tiers appelé compilateur.

Les langages de haut niveau, comme les langues naturelles, permettent des formes d'expressions complexes, composant des sous-expressions. Le premier travail d'un compilateur est donc d'analyser la syntaxe du programme pour en trouver la structure.

Au contraire, les langages machine ont une structure très simple de séquence d'instructions numérotées. Le compilateur utilise alors un ensemble de schémas de compilation, décrivant pour chaque forme d'expression du langage de haut niveau la suite d'instructions du langage machine à produire. Il faut bien entendu composer ces schémas entre eux dans le cas d'expressions composées, pour refléter dans le langage machine produit la structure du langage initial.

Bien sûr, il s'agit d'une vision simplificatrice ; un compilateur réel effectue de nombreuses passes supplémentaires (optimisations, vérifications, etc.) entre l'analyse initiale du programme source et la production du code machine.

Interprétation Un interprète, au contraire d'un compilateur, ne produit pas de code machine. À la place, il lit le programme et applique un ensemble de règles d'évaluation pour calculer son résultat, comme un humain le fait pour calculer le résultat d'une expression arithmétique.

Pour cela, il contient un ensemble de sous-programmes, chacun permettant d'interpréter une forme spécifique d'expression du langage de haut niveau. Le travail de l'interprète pour un programme donné est alors de parcourir sa structure, en appelant à chaque étape le sous-programme correspondant à la forme de l'expression rencontrée. Dans le cas d'une expression composée, c'est l'interprète qui se charge de la composition, en appelant les sous-programmes correspondant à chaque sous-expression, et en transmettant leurs résultats au sous-programme dédié à l'expression principale.

Un interprète est souvent plus facile à écrire qu'un compilateur, mais en général moins performant, puisqu'il faut payer le coût de l'analyse du programme à chaque exécution, mais surtout du parcours de la structure, et du choix du sous-programme à chaque étape de ce parcours. De plus, un interprète utilisant la structure originale du programme, il ne peut effectuer d'optimisation nécessitant de la modifier, or les optimisations de ce type (propagation des constantes, dépliage des boucles, etc.) ont une rôle important dans les bonnes performances d'un compilateur.

L'avantage principal de l'interprétation est que le même programme peut être utilisé sur différents processeurs et systèmes, du moment que l'interprète y est disponible. Avec un compilateur, il faut que le compilateur dispose de schémas de compilation pour chacun des processeurs, et il faut utiliser un programme compilé différent pour chaque processeur, limitant les possibilité de distribution et d'échange de programmes.

Machine virtuelle Une approche intermédiaire entre la compilation et l'interprétation est l'utilisation d'une machine virtuelle. L'idée est de compiler le programme vers un langage machine, puis d'interpréter ce langage machine au lieu de le donner à exécuter à un processeur réel. En général le langage machine cible ne correspond pas à celui d'un modèle de processeur réel, mais à celui d'une machine abstraite (on utilise le terme de machine abstraite pour désigner le modèle théorique, et celui de machine virtuelle pour désigner l'interprète de langage machine).

Les performances de cette technique se situent entre l'interprétation et la compilation. En effet, interpréter un langage très simple comme celui d'une machine virtuelle implique un sur-coût plus faible que l'interprétation d'un langage de haut niveau. De plus, il est possible d'effectuer des optimisations lors de la compilation vers le langage de la machine virtuelle. On obtient donc une technique avec des performances raisonnables, tout en conservant la facilité d'échange de programmes de l'interprétation.

Compilation à la volée Les machines virtuelles récentes utilisent un mécanisme dit de compilation à la volée, ou JIT (*Just In Time*). Au lieu d'interpréter les instructions, la machine virtuelle compile à la demande des parties du programme vers le langage de la machine réelle, pour le faire exécuter par le processeur.

Grâce à cette technique, on conserve la facilité d'une machine virtuelle classique pour l'utilisateur, et on obtient d'excellentes performances, rivalisant avec celles de la compilation classique. Par contre, la conception et l'implantation d'une telle machine est aussi complexe, sinon plus que celle d'un compilateur.

1.1.3 Systèmes de types

Qu'il soit fonctionnel, impératif et/ou par objets, un langage manipule des valeurs. Si le langage n'est pas complètement trivial, ces valeurs peuvent être de plusieurs types. Un langage de programmation classique manipule par exemple des nombres entiers, des nombres à virgule, des caractères et chaînes de caractères, ainsi que des valeurs composées, regroupant plusieurs autres valeurs.

Les sous-parties d'un programme (fonctions, procédures, expressions, etc.) s'appliquent souvent à des valeurs d'un type précis. L'addition, par exemple, ne peut s'appliquer que sur des nombres. C'est à ce moment qu'intervient le système de types, pour empêcher l'exécution d'un traitement sur une valeur qui n'est pas du bon type.

Typage dynamique ou typage statique Avec un système de types dynamique, les types sont vérifiés au cours de l'exécution. Lors de l'exécution du programme, et lorsqu'il faut appliquer un traitement à une valeur, le système vérifie que la valeur est du bon type. Par exemple, il vérifie juste avant d'effectuer une addition que les entrées sont bien des nombres. Si une telle vérification échoue, le programme échoue brutalement, sans pouvoir se terminer.

Avec un système de types statique, les types sont vérifiés avant d'exécuter le programme. Si le programme est déclaré bien typé par un système de types statique, alors on a l'assurance que le programme ne pourra pas échouer à cause d'une erreur de type.

Dans bien des cas, cette sûreté d'exécution du typage statique est un précieux atout. C'est par exemple le cas pour les programmes destinés à effectuer de longs et coûteux calculs ; où un arrêt brutal serait une perte de temps et d'argent considérable. C'est encore plus le cas pour les programmes destinés à s'exécuter dans des environnements critiques, comme le système de freinage assisté d'une voiture. Un autre atout non négligeable du typage statique est qu'on peut aussi s'affranchir des vérifications lors de l'exécution du programme, et gagner en performances.

En contrepartie, la tâche de vérifier statiquement les types est réputée difficile. En effet, il s'agit pour simplifier de prédire l'exécution du programme, uniquement en parcourant son code et sans l'exécuter. Par conséquent, un système de types statique peut rejeter certains programmes pourtant valides en pratique, et oblige le programmeur à comprendre et se plier à son fonctionnement, afin de ne pas écrire de tels programmes trop souvent.

Typage fort ou typage faible Le typage statique à la sûreté infaillible tel que présenté ci-dessus, est qualifié de typage fort. Mais il existe aussi des langages dans lequel le typage est dit plus faible, considérant comme bien typés des programmes produisant pourtant une erreur de types à l'exécution.

Concrètement, cela peut être parce que le système n'est pas assez subtil pour détecter toutes les erreurs de types, ou une volonté de conception donnant la possibilité au programmeur de contourner la décision du système de types par une instruction spéciale.

Les langages très répandus C et Java sont tous les deux des langages à typage faible, comme décrit au paragraphe précédent. Les deux ont un système de types qui peut laisser passer des programmes erronés, mais surtout, les deux disposent d'une instruction dite de transtypage (ou *typecast*), permettant

au programmeur de forcer à changer le type d'un élément du programme, sans demander son avis au système de types [1].

Langage machine typé Il y a une différence majeure entre C et Java, qui fait que l'exécution d'un programme Java reste sûre malgré un système de types faible, alors que l'exécution d'un programme C peut s'arrêter brutalement ou donner un résultat inattendu, à cause d'une erreur de types.

La raison est qu'un programme C est compilé en langage machine pour processeur classique, ce dernier ne faisant que des vérifications minimales à l'exécution (validité du format des instructions, division par zéro, etc.), alors qu'un programme Java est compilé en langage machine pour la JVM (*Java Virtual Machine*), dont les instructions portent des informations de types qui sont vérifiées à l'exécution.

Les machines virtuelles récentes utilisent de plus en plus un langage machine typé, l'exemple le plus connu étant l'environnement .Net de Microsoft. Ce système permet la compilation de langages aux systèmes de types statiques ou dynamiques et faibles ou forts, en s'assurant qu'ils auront tous la même sûreté d'exécution. En contrepartie, étant donné qu'il faut fixer le format des types dans la machine, ce mécanisme peut rendre difficile la compilation d'un système de types exotique. D'autre part, les vérifications dynamiques peuvent nuire aux performances, surtout dans le cas d'un langage typé statiquement, et qui n'en aurait donc pas eu besoin.

Vérification ou inférence de types Un système de types statique peut prendre deux formes. Dans la plus simple, et la plus répandue, c'est au programmeur d'indiquer le type de chaque élément (variable, paramètre, etc.) de son programme. Le système de type effectue alors la vérification que les types donnés par le programmeur sont corrects. Une autre possibilité est de laisser le soin au système de types de deviner (*inférer* ou *synthétiser*) les types, rendant le code moins verbeux. Il existe aussi des langages mélangeant les deux approches, permettant de ne pas indiquer les types sur certaines constructions locales du langage où le compilateur peut les déduire facilement.

1.1.4 Concurrence et parallélisme

Un grand nombre de programmes ont besoin de réaliser plusieurs tâches dans le même intervalle de temps. Par exemple, un lecteur de musique doit être capable d'envoyer les données sonores à la carte son, en même temps afficher les informations du morceau joué, et répondre aux ordres de l'utilisateur. On dit alors que ces tâches sont exécutées en concurrence, car elles accèdent en concurrence aux ressources de la machine (mémoire et temps processeur). Plusieurs modèles existent pour introduire la concurrence dans un langage de programmation. Nous présentons dans cette section les principales caractéristiques des modèles de concurrence.

Concurrence et parallélisme Le parallélisme désigne la capacité d'une architecture matérielle à exécuter simultanément plusieurs tâches. C'est par exemple le cas d'un réseau d'ordinateurs, chacun disposant de son propre processeur, ou des processeurs modernes disposant de plusieurs cœurs. Il faut bien noter qu'un système ne disposant pas de parallélisme peut quand même être concurrent, simplement en partageant le temps d'exécution entre les différentes tâches.

Mémoire partagée ou passage de messages Dans un système concurrent, les tâches doivent communiquer des informations entre elles. Par exemple, au sein du lecteur de musique, lorsque l'utilisateur clique sur le bouton pause, la tâche chargée de prendre en charge ce clic doit parler à la tâche gérant la carte son pour que celle-ci arrête l'envoi des données sonores.

1. Java ajoute cependant une vérification dynamique permettant de vérifier lors de l'exécution, mais cette vérification ne prend pas en compte les types composés. Il est par exemple impossible d'utiliser un fichier à la place d'un nombre, mais il est possible d'utiliser une liste de fichiers à la place d'une liste de nombres.

Il y a deux approches courantes, (1) les tâches peuvent s'envoyer des messages par des canaux de communication, ou bien (2) avoir accès à un même espace mémoire, dans lequel les informations peuvent être lues et mises à jour par toutes.

L'utilisation du passage de messages s'avère en général plus prévisible et donc moins source de bogues, mais elle est plus restrictive et moins efficace que la mémoire partagée (on peut par exemple simuler efficacement le passage de message avec la mémoire partagée, mais pas le contraire avec les modèles d'ordinateurs courants).

Coopération ou préemption Dans un système concurrent (et non parallèle), il faut répartir le temps d'exécution entre les différentes tâches. Il est alors possible de laisser le système choisir le moment où changer la tâche à exécuter (modèle préemptif), ou laisser au programmeur le travail d'insérer explicitement dans son programme des points où la tâche à exécuter doit changer (modèle coopératif).

Dans le modèle préemptif, pour maîtriser les accès en concurrence à la mémoire, le programmeur doit utiliser un mécanisme de verrous, pour s'assurer qu'une seul tâche accède à un moment donné à une zone mémoire verrouillée. Ce mécanisme est réputé difficile à mettre en place, et source de bogues, et de blocages du programme dans le cas où deux tâches essaient de verrouiller la même zone en même temps. Dans le modèle coopératif, puisque les tâches ne peuvent être interrompues qu'à des endroits spécifiés, ce mécanisme n'est pas nécessaire. Le modèle coopératif n'est cependant exempt ni de bogues, ni de blocages, si le programmeur place au mauvais endroit ou oublie de placer un point de coopération.

D'autre part, si le modèle préemptif peut être utilisé naturellement sur une architecture parallèle (il en est en fait une simulation), le modèle coopératif ne peut au contraire pas profiter du parallélisme, puisque les tâches ne peuvent s'exécuter en même temps.

En pratique Les systèmes d'exploitations actuels implantent tous le modèle préemptif et mémoire partagée pour permettre la concurrence dans les programmes comportant plusieurs tâches.

Ce modèle est le plus difficile à programmer, car l'exécution d'un programme est sujet à un très grand nombre de variations, suivant les endroits du programme où ont lieu les préemptions, l'ordre dans lequel sont faits les verrouillages, etc. Mais c'est le mécanisme le plus général, par dessus lequel il est possible de simuler tous les autres modèles.

Des modèles hybrides existent, permettant de mélanger les deux modèles (on peut citer les fair-threads [📖40], où un programme peut contenir des groupes de tâches la concurrence étant coopérative au sein de chaque groupe, mais préemptive entre les groupes).

1.1.5 Notions d'ingénierie du logiciel

Outre ses qualités théoriques intrinsèques en termes de sûreté et d'expressivité, un langage doit fournir des avantages pratiques pour avoir du succès. Le domaine de l'ingénierie du logiciel définit des critères pour analyser l'utilisation pratique des logiciels, cette section donne un aperçu de ces critères (en se concentrant sur ceux importants pour le domaine du Web) et un petit lexique des termes associés que nous utiliserons par la suite.

Portabilité Ce terme désigne la capacité d'un programme ou d'un langage à être utilisé dans des environnements variés. Elle est fonction du modèle d'exécution du langage (nous avons déjà vu qu'un langage interprété est plus portable qu'un langage compilé), mais aussi du niveau d'abstraction des primitives d'interaction entre le programme et le système sur lequel il s'exécute.

Déploiement Il s'agit de la phase de passage en production d'un logiciel. Pour qu'un logiciel ait du succès, son déploiement doit pouvoir être fait sans connaissance poussée de son fonctionnement interne. En général, cette phase est plus liée au logiciel lui-même qu'au langage, mais certains environnements

de développement imposent un modèle de distribution et déploiement (on peut citer en exemple les boutiques d'applications des téléphones portables).

Maintenance et ré-utilisation Ces termes désignent la capacité d'un logiciel à rester utilisable sur la durée. Concrètement, les deux nécessitent que le langage propose un mécanisme de structuration de haut niveau (*modules* ou *composants*), permettant d'isoler proprement une partie du code pour la mettre à jour ou la ré-utiliser ailleurs.

Environnement de développement De plus en plus, le succès des langages est corrélé non pas à ses capacités intrinsèques mais à la quantité et qualité des outils associés : aide au déverminage, édition automatisée du code source, outils de tests, etc. On parle d'IDE (*Integrated Development Environment (Environnement de développement)*) lorsque tous ces outils sont intégrés en un unique logiciel.

1.1.6 Le langage ML

Dans cette thèse, nous utilisons comme base de travail le langage ML. C'est une des familles de langages prisées dans la recherche en langages et en programmation. Dans sa déclinaison la plus utilisée OCaml, il intègre les paradigmes fonctionnel, impératif et objet. La grande force du langage est de laisser le choix au programmeur, et permettant de traiter des problèmes différents avec les solutions les plus appropriées, plutôt que de rechercher des solutions de contournement pour s'intégrer dans un paradigme unique. La sûreté d'exécution est assurée par un système de types statique fort, et une gestion automatique et sûre de la mémoire. C'est un langage concis, avec inférence de types, et une syntaxe déclarative proche de la rédaction mathématique.

Le langage dispose de plus de bonnes propriétés du point de vue ingénierie du logiciel. Le système de types utilise un mécanisme appelé *polymorphisme paramétrique*, permettant de donner un type générique au code pouvant fonctionner avec des valeurs de différents types. Ce mécanisme rend possible la ré-utilisation de code en conservant la sûreté d'exécution. Le langage propose aussi un système de *modules* évolué, permettant la structuration du code, et la création de composants réutilisables, et bénéficiant de la même sûreté que le reste du langage puisque lui-aussi typé. Il existe d'autre part des implantations portables et très efficaces. OCaml [📖57] fournit un compilateur vers une machine virtuelle pour la portabilité, et un compilateur vers du code machine optimisé.

1.2 Petit historique du Web et de sa programmation

Nous continuons cette introduction générale en présentant le troisième mot clef de cette thèse, le Web. Nous allons pour cela nous appuyer sur une vision historique, en partant des premiers sites Web statiques pour aboutir aux complexes applications Web actuelles. Concrètement, cette section est découpée en sous-parties, chacune introduisant un nouvel aspect du Web. À chaque fois, nous exposons les difficultés pour programmer ce nouveau trait, en particulier dans un environnement typé, ainsi que les solutions appliquées en pratique.

1.2.1 Le Web *statique* et ses documents hypertexte

La notion primordiale du Web est celle de *document hypertexte*, désignant un document textuel enrichi de formatage et d'*hyperliens* vers d'autres documents hypertexte. Par la suite on parlera simplement de documents et de liens. Un site Web est un ensemble de documents, chacun possédant un nom unique, servant à constituer les liens. Un site Web contient un document principal, l'*index* ou la *racine*, présenté à l'utilisateur accédant au site sans spécifier de nom de document particulier.

Ce concept de document hypertexte est aussi utilisé dans nombre de logiciels : encyclopédies, fichiers d'aide de logiciels, logiciels de classement de livres ou de documents, etc. Le Web se différencie

de ces utilisations par son caractère distribué. Les sites Web sont situés sur des machines distinctes connectées par le réseau internet, qui transmettent aux utilisateurs les documents via des logiciels appelés serveurs Web. Pour que les serveurs et les clients se comprennent, tous parlent entre eux en utilisant le protocole HTTP (*Hyper Text Transfer Protocol*).

Côté serveur, pour faire simple, on utilise souvent un fichier par document, et les noms donnés aux documents par le serveur sont leurs chemins dans le système de fichiers. Un lien vers un document au sein du site sera alors simplement son nom de fichier.

Mais il est aussi possible sur le Web d'utiliser des liens complexes, vers d'autres sites et via d'autres protocoles. Pour ceci on utilise un format spécifique de lien appelé URL (*Uniform Resource Locator*), qui contient l'adresse sur le réseau de la machine sur laquelle le document est situé, le protocole pour y accéder, et le nom du document sur la machine.

L'utilisateur, de son côté, utilise un logiciel de navigation Web, qui se charge de réaliser l'affichage du formatage et des liens. Lorsque l'utilisateur clique sur un lien, le navigateur calcule alors son URL, et émet une requête au serveur correspondant, dans le format spécifié par le protocole HTTP. Le serveur lui répondra alors, toujours via le protocole HTTP, soit par le contenu du document, soit par une page d'erreur si le document n'est pas trouvé. On parle alors de *lien mort*. Pour que tous les navigateurs affichent les pages de la même façon, les documents sont écrits dans un format spécifique, le XHTML (*eXtensible Hyper Text Markup Language*).

Un petit exemple de site Web Afin de concrétiser ces notions de document et de lien, ainsi que les formats utilisés, penchons nous sur un exemple de petit site Web statique, que nous appellerons site alpha. La figure 1.1 donne une représentation graphique des liens au sein du site et vers l'extérieur. Un lien est représenté par une flèche, dirigée de la page contenant le lien vers la page pointée. Le site alpha dispose d'une page principale, usuellement nommée `index.html`, d'un plan et de deux pages de contenu. Il permet d'accéder via des liens externes à un autre site (le site bêta) et au serveur de fichiers du site alpha en utilisant le protocole classique de transfert de fichiers FTP (*File Transfer Protocol (Protocole de transfert de fichiers)*).

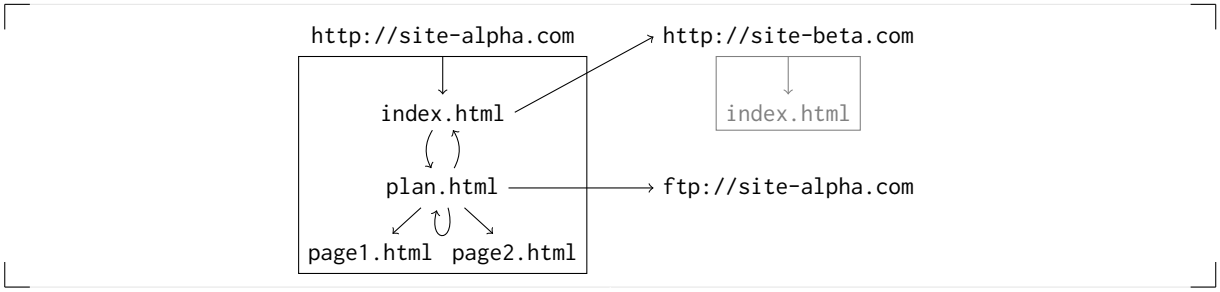


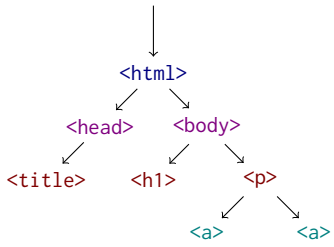
Figure 1.1: Plan du site alpha

La figure 1.2 donne le code source XHTML de la page d'accueil du site. Le format XHTML est une version spécifique du format XML (*eXtensible Markup Language*) [2]. En XML, le formatage du document se fait en encadrant les parties à formater par des *balises*, de façon similaire à des parenthèses. À la différence de parenthèses classiques, il y a plusieurs types de balises, différenciés par des étiquettes. Une balise ouvrante étiquetée *n* est écrite `<n>` et la balise fermante correspondante `</n>`. Les balises peuvent de plus être paramétrées, à l'aide d'*attributs* nommés. Par exemple, la balise *n* avec l'attribut *id* valant *3* s'écrira `<n id="3"></n>`.

2. Pour être chronologiquement correct, les pages Web n'étaient pas en XHTML mais en HTML (*Hyper Text Markup Language*), un format plus laxiste pour être écrit plus facilement par un humain, mais moins facile à traiter automatiquement. Le format XML a été inventé après, comme généralisation des formats à balises apte au traitement automatique, et XHTML est l'adaptation XML du vénérable HTML.

Le format XHTML utilise ce formatage à balises de XML, et précise en plus quelles balises peuvent apparaître, et leur signification en termes de formatage de la page. Par exemple, si on encadre par les balises `<i>` et `</i>` une partie de texte, celle-ci sera affichée en italique. De plus, XHTML définit comment les balises s'imbriquent les unes aux autres. Concrètement, une page XHTML est structurée en deux balises principales : `<head>` contenant des informations sur la page, et `<body>` encadrant son contenu, toutes deux contenues dans une balise principale `<html>`. Dans cette page, on peut voir un paragraphe de texte (`<p>`) et des liens (`<a>`, dont le texte est contenu dans la balise, et la cible est précisée dans l'attribut `href`) internes au site alpha, et externe vers le site béta.

```
1 <html>
2   <head>
3     <title>site alpha - page d'accueil</title>
4   </head>
5   <body>
6     <h1>Bienvenue sur le site  </h1>
7     <p> Vous pouvez consulter
8         <a href="plan.html">le plan du site alpha</a>.
9         Accédez au site béta par
10        <a href="http ://www.site-beta.com/">ce lien</a>. </p>
11  </body>
12 </html>
```



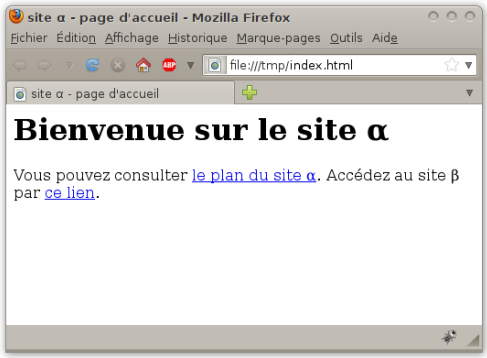
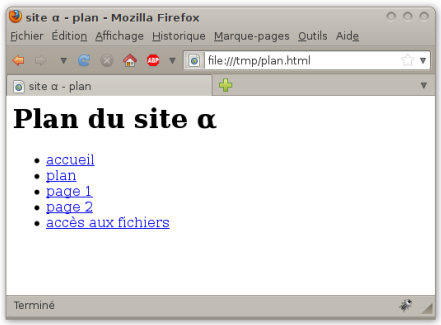


FIGURE 1.2: Fichier `index.html`

La figure 1.3 donne le code de la page `plan.html`. Elle contient une liste à puces (`<ul>`) contenant des éléments de liste (`<li>`). Elle contient aussi un lien vers la page elle-même, et un lien dont l'URL utilise un autre protocole qu'HTTP. La figure donne aussi une représentation arborescente décrivant la hiérarchie d'imbrication des balises, ainsi que le rendu fait par un navigateur.




```
1 <html>
2   <head>
3     <title>site alpha - plan</title>
4   </head>
5   <body>
6     <h1>Plan du site alpha</h1>
7     <ul>
8       <li> <a href="index.html">accueil</a> </li>
9       <li> <a href="#">plan</a> </li>
10      <li> <a href="page1.html">page 1</a> </li>
11      <li> <a href="page2.html">page 2</a> </li>
12      <li> <a href="ftp ://site-alpha.com/">accès aux fichiers</a> </li>
13    </ul>
14  </body>
15 </html>
```

FIGURE 1.3: Fichier `plan.html`

La figure 1.4 montre une consultation du site par un utilisateur. On y voit un exemple concret de transmission via le protocole HTTP : chaque flèche représente un échange par le réseau entre le navigateur et le serveur, le texte en dessous de la flèche étant la commande conforme au protocole HTTP transmise sur le réseau.

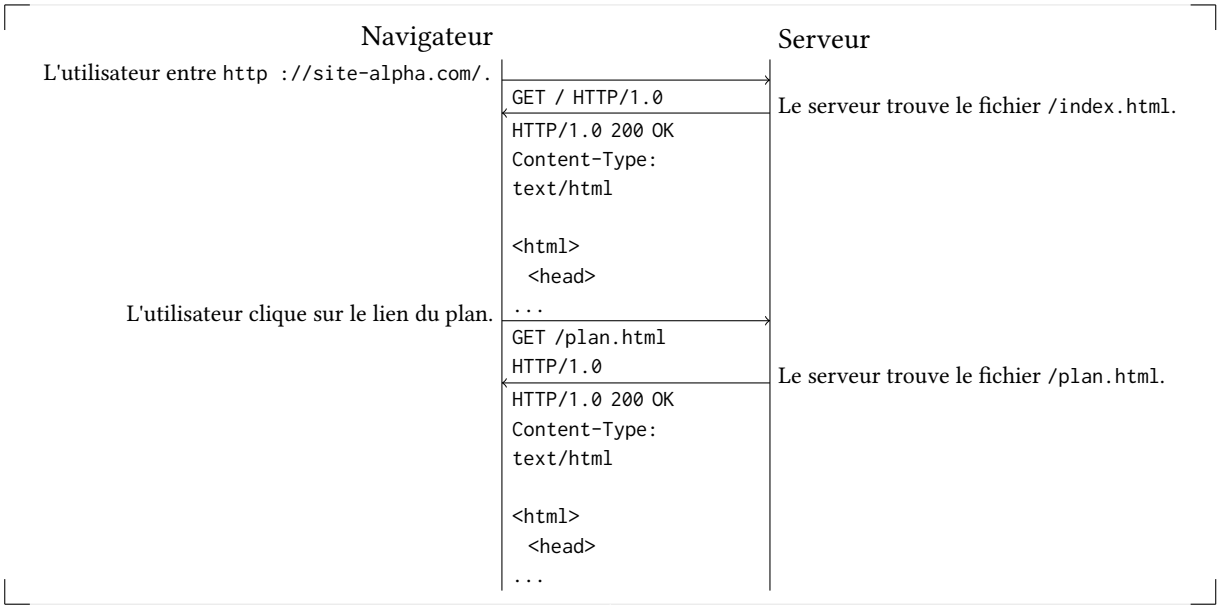


Figure 1.4: Exemple de consultation du site par HTTP

Du point de vue typage, comme nous l'avons vu dans l'exemple, les documents XML doivent respecter des règles, principalement un ensemble de balises et une structure spécifiques. Nous avons présenté les documents XHTML, mais d'autres formats XML existent sur le Web, tel SVG (*Scalable Vector Graphics*) pour les dessins ou MathML pour les formules mathématiques, utilisant le format de base XML, mais avec des balises et des structures différentes.

Ces formats de documents XML du Web sont spécifiés par des recommandations, établies par le W3C (*World Wide Web Consortium*). Elles sont en général décrites en deux parties. La première partie est destinée à être lue par une machine, et décrit l'ensemble des balises et leurs attributs associés, ainsi que les imbrications possibles. On appelle en général cette spécification la *grammaire XML* ou le *type de document*, et elle est donnée dans le format standard DTD (*Document Type Definition*). Lorsqu'un document XML respecte les règles fixées par une DTD, on dit qu'il est *valide* par rapport à cette DTD. La seconde partie de la spécification est en langue anglaise, et décrit non pas le document lui-même, mais comment le navigateur doit l'interpréter.

Le respect de ces formats est important, car les navigateurs Web doivent s'assurer que les documents sont valides pour pouvoir les afficher correctement [3]. De ce point de vue, on peut assimiler la vérification de validité d'un document à une vérification de typage, où la DTD est le type, et le document la donnée dont le type doit être vérifié pour que le programme fonctionne correctement.

Si on veut typer statiquement (pour rappel, cela signifie vérifier à l'avance, afin qu'aucune erreur ne se produise par la suite) un site Web dans son ensemble, il faut alors typer chacun de ses documents, mais aussi vérifier qu'il n'existe pas de lien mort au sein du site.

En pratique, il est encore fréquent de trouver des pages Web statiques ne répondant pas aux recommandations, car la plupart des éditeurs grand public WYSIWYG (*What You See Is What You Get*) génèrent

3. En réalité, la quantité de documents non valides présents sur le Web fait que les navigateurs sont obligés de faire preuve de souplesse, et interprètent comme ils peuvent les documents non valides.

du XHTML approximatif, et les éditeurs de texte se contentent de vérifier la forme XML des documents (principalement que toutes les balises ouvertes soient bien fermées, et dans le bon ordre) mais pas leur validité. Il est par ailleurs fréquent de rencontrer des pages affirmant être valides avec un joli logo W3C mais ne l'étant en fait pas. De même, il ne faut pas chercher bien loin pour trouver des liens morts sur le Web, y compris sur des sites récents.

Mais ces problèmes sont principalement le fait de l'inertie des technologies et des pratiques. Il existe en effet de nombreux outils permettant de vérifier qu'un document est *valide* par rapport à sa DTD. Le W3C propose par exemple un validateur en ligne [⚓69], permettant de vérifier une page à une URL donnée.

1.2.2 Le Web *dynamique* et son contenu généré à la volée

Pour étendre les possibilités fournies par les sites Web statiques ont été introduits les formulaires. Ainsi, la page Web reçue à l'issue d'un envoi de formulaire sera confectionnée à la volée en fonction des données envoyées, et non simplement lue depuis un fichier statique.

Techniquement, pour permettre de construire des pages à la volée, il est possible de programmer un serveur Web spécifique, prenant en charge l'ensemble de la chaîne, de la requête du client à la génération de la page Web qui en découle. Mais la façon la plus répandue est la possibilité donnée par les serveurs Web généralistes de prendre en charge toute la partie liée aux communications, et de déléguer à un programme externe uniquement la génération du contenu de la page (en général via le protocole CGI (*Common Gateway Interface*)).

Il est alors possible de programmer des pages générées dynamiquement, aussi facilement qu'on imprimerait du texte à l'écran dans un environnement de programmation classique. Le langage de programmation le plus connu utilisant cette technologie est PHP (*Personal Home Page*), un langage simple et facile d'accès pour les débutants, qui au sein de l'alliance de logiciels libres LAMP (*Linux Apache MySQL PHP*) (qui regroupe le système d'exploitation, le serveur Web généraliste, le langage de script, et le moteur de bases de données) a démocratisé la programmation de sites dynamiques.

La figure 1.5 montre un exemple de formulaire, et la figure 1.6 un exemple de page dont certains éléments sont générés dynamiquement par un script PHP sur le serveur, en fonction des éléments obtenus par le formulaire.

```
1  <html>
2    <head>
3      <title> Page de bienvenue </title>
4    </head>
5    <body>
6      <form method="POST" action="hello.php">
7        Entrez votre nom : <input type="text" name="champNom" />
8        <input type="submit" />
9      </form>
10   </body>
11 </html>
```

Figure 1.5: Exemple de formulaire

Pour réaliser cette dynamicité de contenu, les liens hypertexte ne sont plus simplement les noms des documents, mais sont en plus paramétrés par le contenu des formulaires. Un lien peut donc être mort même si le script qu'il cible existe, si ses paramètres ne correspondent pas à ceux attendus par le script. De plus, le protocole HTTP propose deux types de paramètres :

- Les paramètres *get* sont inclus dans l'URL qui permet d'accéder au document. Ils sont par conséquent visibles à l'utilisateur, et facilement modifiables à la main. Ils sont de plus enregistrés dans les marques pages du navigateur.
- Les paramètres *post* sont issus des champs d'un formulaires et transmis dans le corps du message HTTP. Ils n'apparaissent pas dans l'URL, ne sont pas stockés dans les marques pages et sont

```php
<? $nom = $_POST["champNom"] ?>
<html>
  <head>
    <title> Page de bienvenue à <?php echo $nom ?> </title>
  </head>
  <body>
    <p> <?php echo "Bienvenue " + $nom ?> </p>
  </body>
</html>
```

FIGURE 1.6: Exemple de page Web générée dynamiquement en PHP

difficilement modifiables manuellement.
Si un utilisateur enregistre l'adresse d'une page à l'issue d'un envoi de formulaire, il manquera alors au script les variables *post* attendues, ce qui crée potentiellement un lien mort. C'est le cas si, dans l'exemple précédent, l'utilisateur appelle directement la page `hello.php` sans passer par le formulaire. C'est ce qui explique en partie que le mécanisme de marque pages des navigateurs ne semble pas fonctionner correctement pour un grand nombre de sites dont la navigation interne n'est pas triviale.

Du point de vue typage, il s'agit de produire un document dynamiquement, en fonction de paramètres passés à la page. Il faut donc typer le document produit par rapport à sa grammaire. Dans ce contexte, le typage statique demande de typer le langage produisant le document statiquement, de façon à être certain que les documents produits seront tous valides. Le typage dynamique revient à générer le document, et vérifier sa validité avant de l'envoyer au client ; reste alors à choisir que faire en cas d'erreur.

Pour les liens morts, il faut s'assurer que les URL générées pointent vers des scripts existants, que le nombre, les noms et les types des arguments *get* générés sont corrects, et que les scripts nécessitant des arguments *post* ne sont appelés que par des formulaires avec les champs appropriés.

En pratique, les environnements de programmation Web grand public tels que LAMP montrent des lacunes dès ce niveau de dynamicité, ne prenant en compte ni la vérification de la validité, ni la génération des liens. C'est au programmeur de s'assurer par la relecture de son code que les pages seront correctes et les liens générés vivants.

À côté des langages grand public, plusieurs environnements de programmation Web issus de la recherche en langages, que nous présenterons dans la partie III, permettent d'assurer la plupart des propriétés de typage introduites par les sites dynamiques que nous venons de voir.

1.2.3 Le Web *social*, ses utilisateurs et ses données en masse

Depuis de nombreuses années (et bien avant l'arrivée des *réseaux sociaux* à la mode), le Web a un impact social. Il n'est plus seulement vecteur d'information, mais un support de communication, permettant de regrouper des communautés d'utilisateurs, de faire ses courses ou d'effectuer des démarches administratives en ligne. Le phénomène est en cours d'explosion grâce à la démocratisation de l'accès au Web, la puissance des machines permettant de réaliser des sites très sophistiqués, et le Web mobile.

Techniquement, ceci introduit deux nouvelles difficultés. D'une part, il faut reconnaître l'utilisateur et lui délivrer des informations personnalisées. D'autre part, il faut pouvoir stocker, rapatrier et mettre à jour de grandes quantités de données.

Le protocole HTTP est intrinsèquement non connecté, puisque la communication entre un client et le serveur se fait par requêtes indépendantes, et non via un canal de communication établi pour une session de dialogue. La réalisation d'applications connectées nécessite donc l'implantation au dessus d'HTTP d'une *couche session* (dans la terminologie réseau). Il faut donc concrètement que le serveur puisse identifier un même client entre deux requêtes, avec une méthode ne permettant pas qu'un client

ne puisse pas se faire passer pour un autre. Pour ceci, en général, des informations spécifiques au client (adresse réseau, nom du navigateur, nom et mot de passe haché d'utilisateur dans le cas d'un site à inscription, etc.) sont compilées en un identifiant de session, qui est transmis du serveur au client à chaque envoi de page, puis à nouveau du client au serveur à chaque requête. Concrètement, les *cookies* ont été introduits dans les navigateurs pour enregistrer ces informations de session. Un cookie est associé à un site Web, il peut être mis à jour par le serveur lors de l'envoi d'une page, et est systématiquement envoyé dans les requêtes au site auquel il est attaché.

Afin de stocker les données des profils utilisateur, les systèmes de fichiers arrivant rapidement à leur limite de commodité pour le programmeur, les langages de script Web tels PHP se sont très rapidement vus greffer des interfaces vers les SGBDR (*Système de Gestion de Base de Donnée Relationnelle*) classiques. Étant donné la part de développement amateur dans le Web, des solutions gratuites ou libres de bases de données relationnelles ont été majoritairement adoptées, en particulier MySQL au sein de LAMP.

Du point de vue typage, à ce niveau de dynamicité, le typage rejoint la sécurité, et un programme correctement typé ne doit pas permettre la fuite, la suppression ou l'injection malveillantes d'informations de la base de données, ainsi que l'usurpation d'identité.

En pratique, les environnement de programmation Web proposent un mécanisme de session fiable et prêt à l'emploi, mais laissent au programmeur la charge d'implanter son mécanisme d'authentification utilisateur, laissant libre champ aux failles de sécurité.

L'interface avec les SGBDR étant faite en général de façon médiocre, en dialoguant textuellement via le langage SQL (*Search and Query Language (Langage de requêtes de bases de données)*), elle est la principale source de failles de sécurité. Nous reviendrons plus en détails sur ce sujet à la section suivante.

Dans la partie III, nous donnerons un aperçu des solutions de recherche permettant d'interagir de manière typée et sûre avec des SGBDR.

1.2.4 Le Web *2.0* et l'utilisation massive d'AJAX

AJAX (*Asynchronous JavaScript And XML*), est le mot clef désignant les applications qui utilisent le langage JavaScript [📖60] muni de sa fonctionnalité permettant d'effectuer une requête HTTP sans changer de page, et de recevoir de cette manière des données sous forme XML afin de les utiliser dans la page.

JavaScript est le langage utilisé pour programmer du code intégré dans un document Web, qui s'exécute au sein du navigateur et a accès en lecture et écriture au document. Ces accès au document se font via une interface appelée le DOM (*Document Object Model*).

Ainsi, il est possible de mettre à jour une page en téléchargeant uniquement une partie des informations nécessaires à sa constitution et en la mettant à jour côté client via le DOM. Un comportement typique est le parcours de catalogue où on ne met à jour que la sous-partie visible et pas toute la page.

En pratique, JavaScript est un langage typé dynamiquement et particulièrement riche, qui est resté jusqu'à peu particulièrement lent, et ne disposant pas d'environnement de développement correct, en particulier pour le déverminage. À ceci, il faut ajouter que le DOM est une API (*Application Programming Interface*) de bas niveau, et varie suivant les navigateurs et leurs versions. Il est donc très facile d'écrire du code bogué, très difficile de corriger ses erreurs, et même parfois difficile de se rendre compte qu'elles existent étant donné que le langage est particulièrement permissif. Pour pallier ceci, la solution la plus courante est d'utiliser de grosses bibliothèques JavaScript multi-usages, largement testées, et d'écrire le minimum de code dédié. Les bibliothèques de scripts serveur complexes proposent même souvent des composants prêt à l'emploi générant du JavaScript à la volée pour rendre automatiquement les pages générées un peu plus dynamiques.

Du point de vue typage, même si on avait réussi à valider les pages générées, le problème de typage du document se repose si celui-ci est modifié au cours de sa vie. Si des éléments de document sont générés dynamiquement côté client, on aimerait pouvoir les typer. D'autre part, on voudrait aussi être sûrs que les modifications faites au DOM préservent son typage. C'est ce problème que nous aborderons dans la partie II.

De plus, le code client pouvant générer des liens, ou directement effectuer des requêtes HTTP, il faudrait pouvoir typer ces appels, afin d'être sur de ne pas générer de lien mort.

1.3 Programmer le Web aujourd'hui

Nous venons d'introduire, en nous appuyant sur une vision historique, les principales problématiques de programmation du navigateur, du serveur, et de leurs communications, ainsi que les techniques majoritairement utilisées en pratique pour les traiter et leurs lacunes, en particulier en matière de typage.

Dans cette section nous décrivons de façon plus rigoureuse les technologies existantes, en passant en revue les problématiques que nous avons aperçus à la section précédente, et en décrivant pour chacune les solutions disponibles. Nous commençons par les solution de programmation du client, celles du serveur, ainsi que les solutions de programmation client/serveur intégrées, puis nous abordons la programmation du document, et enfin la communication avec les bases de données.

1.3.1 Technologies client

JavaScript Le moyen le plus répandu pour programmer le client est l'utilisation du langage JavaScript, dont une introduction est donnée dans l'annexe A. Le principal avantage de JavaScript est qu'il est présent sur presque tout le parc des navigateurs installés. Son inconvénient majeur est la trop importante diversité entre les navigateurs et versions de navigateurs, en termes de fonctionnalités implantées, d'extensions propriétaires, de bogues et de performances. La tendance est cependant à l'amélioration et à l'homogénéisation des fonctionnalités.

Bibliothèques JavaScript Pour pallier les incompatibilités et la difficulté de développement et de déverminage de JavaScript, les développeurs Web utilisent de façon quasi systématique des bibliothèques. Il en existe un nombre important, citons parmi les plus connues jQuery, DOJO ou ExtJS. Ces bibliothèques contiennent, d'une part une couche d'abstraction du langage et du DOM de plus ou moins haut niveau, si bien que s'il se limite aux primitives de sa bibliothèque, le programmeur n'a plus à se soucier du navigateur. D'autre part, ces bibliothèques fournissent une collection exhaustive de composants, allant de simples petites animations jusqu'à des composants d'interfaces graphiques complexes. L'idée est que la plupart des besoins du programmeur débutant sera résolue par l'assemblage de composants, et que le programmeur avancé pourra à la fois bénéficier des composants, et d'un langage assaini.

Greffons En présence de besoins spécifiques, comme une bonne performance graphique ou vidéo, ou une plus grande portabilité et indépendance du navigateur, les développeurs Web font souvent appel à des greffons. En général, l'application se présente dans une sous-fenêtre du navigateur, avec une interaction faible avec le reste de la page. Certains sites, plus rares, sont entièrement écrits avec ces technologies. On peut citer dans l'ordre de popularité Flash (de Adobe), Java (de Sun Microsystems), Silverlight (de Microsoft). Concrêtement, les greffons reposent sur l'interface NPAPI (*Netscape Plug-in API*) de Netscape, qui fournit un moyen limité d'interaction avec la page Web et JavaScript, justifiant leur aspect détaché du document. Ils sont en langage machine, ce qui, d'une part, leur impose d'être fournis pour chaque architecture, et d'autre part, leur donne les droits d'exécution d'un programme classique, introduisant de potentielles failles de sécurité. C'est pourquoi les greffons majeurs (Flash,

Java et Silverlight) sont des machines virtuelles, fournissant une interaction cadrée entre le programme hôte et le système, limitant ainsi les failles de sécurité et les bogues. La technologie NaCL (pour Native Client, de Google), propose d'exécuter des programmes en code machine plutôt que d'utiliser une machine virtuelle, en utilisant une vérification avant l'exécution (une analyse statique de flot de contrôle) pour s'assurer que les fonctions système appelées appartiennent à une liste de fonctions autorisées. Côté recherche, François Rouaix et al ont proposé le navigateur MMM [🕮48], écrit en OCaml et offrant des possibilités de programmation d'applets de façon bien typée [🕮38], limitant ainsi les problèmes précédents tout en assurant une bonne efficacité, mais cette voie n'a pas eu le succès escompté.

1.3.2 Technologies serveur

PHP, Perl, ASP, JSP, etc. cette liste rassemble des langages aux propriétés différentes. Les langages comme PHP ou ASP (*Active Server Pages*) de Microsoft sont des langages dédiés au Web, tandis que Perl ou JSP (*Java Server Pages*) de Sun sont des ré-utilisations de langages existants. PHP est un langage de script typé dynamiquement de façon basique, tandis que Java est un langage applicatif expressif statiquement typé.

Nous les avons regroupés car ce sont les plus populaires, et ce pour des raisons pratiques. L'essor de ces langages n'est pas dû à leurs qualités propres mais à leur facilité d'accès au programmeur débutant, et à leur large support par les hébergeurs. Dans un environnement hébergé classique, le concepteur de site Web dispose d'un accès en écriture à un répertoire sur une machine partagée entre plusieurs sites Web, où il dépose ses pages Web sous forme de fichiers HTML. Ces langages ont été conçus pour s'intégrer facilement à cet environnement, et ainsi se sont très vite répandus chez les hébergeurs, payants comme gratuits. Le concepteur dépose ses programmes sous forme de fichiers de script, une page dynamique étant associée à un fichier. Le langage se charge de toute la machinerie relative au réseau et à la gestion des URL, en général en se reposant sur un serveur généraliste via CGI, si bien que le programmeur se concentre sur le cœur des fonctionnalités. En contrepartie, le modèle d'exécution et de liaison des URL n'est pas programmable, ce qui est limitant pour le programmeur, mais séduisant du point de vue de la facilité d'administration pour l'hébergeur.

À de rares exceptions, les applications importantes écrites dans ces langages sont couplées à un SGBDR, auquel est délégué la gestion des accès et mises-à-jour en concurrence des données.

Un autre point commun à ces langages est le traitement du document Web généré comme du texte simple. Ainsi, le programmeur débutant peut conserver sa connaissance du HTML et rajouter quelques bribes de dynamisme à sa page simplement en générant des morceaux de texte. Bien sûr, cette approche complètement non typée est à l'opposée de ce que nous voulons dans cette thèse.

JavaScript côté serveur Afin d'uniformiser la programmation entre le client et le serveur, et puisqu'il est difficile de se passer de JavaScript sur le client, de nombreux projets (64 recensés sur WikiPédia) cherchent à programmer le serveur en JavaScript. Parmi les plus anciennes technologies, on peut citer ASP de Microsoft, qui permettait de programmer dans son dialecte JScript. Pour les plus en vogue, on peut citer Node.JS [⚓101], qui utilise l'interprète V8 du navigateur Chrome de Google. Il propose le même modèle d'exécution que le navigateur, avec une programmation entièrement événementielle et une boucle d'évènement implicite. Un des arguments est de se passer des threads systèmes et d'utiliser un modèle coopératif avec communication par passage de messages, afin de réduire l'empreinte en mémoire et en nombre de processus et ainsi mieux pouvoir passer à l'échelle. Les standards E4X (*ECMAScript for XML*) [🕮61] et CommonJS participent à cet effort de programmation du serveur en JavaScript en fournissant respectivement une API de manipulation de XML depuis JavaScript et une bibliothèque standard étendue pour JavaScript.

Autres langages dynamiques Les projets Django [⚓83] et Ruby on Rails [⚓102] sont deux environnements de programmation Web pour les langages dynamiques Python et Ruby. Ils sont tous les deux

fondés sur l'architecture MVC (*Modèle Vue Contrôleur*), un motif de programmation très apprécié dans le monde de l'ingénierie du logiciel, découplant le cœur des fonctionnalités (le modèle) de l'affichage (la vue) en deux modules séparés, dialoguant entre eux via un troisième module (le contrôleur). Ici, le *modèle* est lié à la base de données, et la *vue* est un modèle HTML. Ces technologies restent proches des langages à scripts plus classiques, dans le sens où le SGBDR reste l'élément central, définissant les données manipulées par l'application et assurant la cohérence des accès et mises-à-jour.

Continuations Christian Queinnec explique dans [📖46] l'importance de l'état dans la navigation Web, en particulier à cause du retour en arrière et de la fonction de duplication de fenêtre, et explique que les continuations sont le bon modèle pour le gérer. Concrètement, une continuation permet de figer un calcul et son environnement d'exécution, pour éventuellement le stocker ou le dupliquer comme une autre valeur du langage, et le lancer plus tard. Il s'agit donc de stocker des continuations au cours de la navigation, qui pourront être re-déclenchées (resp. dupliquées) lorsque l'utilisateur utilise le bouton de retour arrière (resp. duplique la fenêtre en cours) de son navigateur. L'avantage est de ne pas maintenir un fil d'exécution vivant par client sur le serveur, puisqu'ils peuvent être figés et relancés. C'est une version améliorée et intégrée au langage de la programmation événementielle que nous avons vu pour JavaScript, dans laquelle c'est au programmeur de figer lui-même l'environnement du calcul. Seaside [📖10] est un environnement de développement Web reposant sur le langage Smalltalk [📖63], et permettant une telle navigation non triviale via les continuations. Il bénéficie d'un certain succès commercial. D'autres solutions du monde de la recherche, que nous présenterons plus loin, se basent aussi sur ce principe.

1.3.3 Solutions multi-tiers

Dans l'industrie comme dans la recherche, il existe des solutions cherchant à programmer l'ensemble de l'application dans un même environnement. De façon peu surprenante, les solutions industrielles sont principalement basées sur l'assemblage de composants pré-fabriqués dans le code serveur. Mais, dans la recherche comme dans l'industrie, il existe aussi quelques solutions plus originales, cherchant à fournir un environnement homogène, tout en laissant au programmeur la possibilité de programmer des comportements spécifiques, et non simplement d'assembler des composants.

Bibliothèques de composants MVC Étant donné que les langages que nous avons présentés section 1.3.2 sont les plus répandus, leur public s'étend du développement amateur jusqu'aux très populaires applications sociales actuelles (Facebook, par exemple, est écrit en PHP) dont la taille du code n'a plus rien à voir avec de simples scripts. Leur approche très peu typée du langage, du document et des interactions rend alors difficiles le développement, la maintenance et la sécurisation de ces applications. C'est pourquoi, de la même façon qu'avec JavaScript côté client, les développements industriels reposent sur des bibliothèques cherchant à s'abstraire de ces bases fragiles. Quasiment toutes ces bibliothèques (on utilise plus souvent le terme *Framework*, importé de l'anglais) reposent sur un mécanisme MVC. Ce modèle permet, en plus d'une meilleure conception architecturale des applications, d'accéder à une première forme de solution complète de programmation Web. La définition des structures de données utilisées par l'application se fait en assemblant des composants, et les accès à la base de données sont alors automatiquement générés de façon bien formée. De même, les composants d'interface, en plus de générer du HTML, comportent souvent une partie script client, et le code JavaScript final est généré à la volée à partir de l'assemblage. Cette approche est intéressante car elle permet facilement de gérer les différentes parties à partir du langage serveur, mais elle est limitée sur deux points importants. D'abord, l'expressivité est limitée, et se limite à la composition de composants préfabriqués, liant ainsi le style de programmation à celui dicté par la bibliothèque. D'autre part, les composants sont implantés dans un langage non sûr, et le programmeur doit faire confiance aveuglément au fournisseur de sa

bibliothèque qui, de son côté, doit donc fournir un travail exhaustif de test de ses composants et de leurs assemblages possibles. Cela implique aussi que l'écriture de nouveaux composants nécessite une connaissance du langage d'implantation et une maîtrise du fonctionnement interne de la bibliothèque et de son mécanisme d'assemblage de composants.

Assemblages cohérents Ces dernières années, un certain nombre de solutions ont émergé pour développer des applications Web en assemblant et étendant des solutions logicielles existantes. Toutes se basent sur le même principe : un langage compilé vers JavaScript, une bibliothèque standard minimale implantée côté serveur et côté client en JavaScript afin d'améliorer la cohérence, et un mécanisme de communication. On peut citer notablement :

- GWT [⚓80] de Google, est un portage du langage Java dans les navigateurs, par compilation vers JavaScript, ainsi qu'un portage d'une partie de la bibliothèque standard Java, et une bibliothèque spécifique au navigateur.
- HaXe [⚓98] de la petite entreprise française Motion Twin spécialisée dans les jeux Web, est une solution constituée d'un langage riche et statiquement typé, d'une bibliothèque minimale comportant le support du langage et des communications, et un compilateur ciblant divers plates-formes classiques du Web.
- Dans le monde OCaml : OcamlJS + Lwt + Orpc est un assemblage permettant d'utiliser le langage OCaml, via le compilateur OcamlJS côté client, en harmonisant le modèle de concurrence entre les deux parties via la bibliothèque de threads coopératifs Lwt, et en permettant les communications HTTP via la bibliothèque Orpc.

Si elles ne définissent pas une vision globale de la programmation d'application Web et que la cohérence reste limitée, ces solutions sont déjà un grand pas en avant par rapport aux technologies précédentes. Elles sont aussi séduisantes de part leur modularité intrinsèque. Certains aspects de ces solutions seront discutés plus en détails dans la partie III.

Ocsigen, Links, HOP, OPA : les langages multi-tiers Ces solutions, issues des avancées de la recherche, et au public pour l'instant plus restreint, cherchent à utiliser les avancées de la recherche en langages de programmation et à fournir un environnement complet pour la programmation d'applications Web. Elles incorporent les différents aspects (serveur, client, données et communications) au niveau du langage et/ou du système de types de base. Ces solutions, proches du travail présenté dans cette thèse, ne peuvent être décrites en quelque phrases, elles seront détaillées au chapitre 14.

1.3.4 Création et manipulation de documents

Le premier concept que nous avons présenté, et qui est pour nous l'essence même du Web, est celui de document, en particulier de document interactif. Malheureusement, aucune des solutions, industrielles ou de recherche, ne reflète correctement cette notion. Ce constat est la motivation principale d'une large partie du travail qui sera développé dans cette thèse.

Côté serveur Dans les langages serveur les plus répandus, comme PHP, Perl ou Java CGI, la notion de document n'existe tout simplement pas. Ces langages manipulent simplement des flux de texte, et le programmeur doit produire à la main les balises XML correspondant au document qu'il veut envoyer. Pour Perl ou Java, cela est dû au fait qu'il s'agit de langages généralistes, utilisés de façon détournée pour fonctionner avec un serveur Web. Pour PHP, cela pourrait venir du fait que le projet est issu d'un langage de script ad-hoc écrit par Rasmus Lerdorf pour ses propres pages personnelles, et que les versions suivantes n'ont jamais cherché à améliorer le modèle, laissant le travail aux *frameworks*.

Côté client Le navigateur expose une API JavaScript dédiée à la manipulation de la représentation du document. Mais cette API est très impérative et bas niveau, et est beaucoup plus adaptée aux modification locales de document qu'à la création de morceaux de document. Et même si la situation s'améliore, l'API n'est pas homogène entre les différents navigateurs. De plus, et alors que la notion de grammaire de document existait avant l'apparition de JavaScript, l'API ignore complètement la grammaire du document. Outre les problèmes posés sur le client, cet aspect bas niveau ne pousse donc pas à reprendre une telle interface pour la manipulation coté serveur, encourageant l'hétérogénéité entre les parties.

Passage du serveur au client Indépendamment des problèmes intrinsèques aux représentations du document sur le serveur et le client, la différence entre les deux est elle-même source de problèmes. En particulier, une difficulté arrive lorsque le programmeur veut faire le lien entre des parties du document fabriquées sur le serveur et les parties correspondantes dans le document sur le client. Ce problème est récurrent : il apparaît dès que le programmeur veut, une fois sur le client, effectuer une action sur un élément de la page, alors que cet élément peut être placé dans la page de différentes (ou même multiples) façons par le serveur.

Pour ceci, il existe une solution classique, mais malheureusement, celle-ci n'est pas satisfaisante car entièrement manuelle et donc source de nombreux bogues. Concrètement, sur le serveur, le programmeur doit insérer des *identifiants* (attributs *id* des balises) dans le document généré, en s'assurant bien que le même identifiant n'est pas donné à plusieurs balises. Il doit alors faire très attention au choix de ces identifiants. Par exemple, s'il veut ajouter plusieurs fois un morceau de page contenant des identifiants, il devra en générer automatiquement des différents pour chaque occurrence. De même, il faut s'assurer qu'aucun conflit n'existe entre les identifiants insérés par les éventuelles bibliothèques utilisées. Une fois sur le client, le programmeur peut retrouver la partie du document marquée par une étiquette spécifique. Il faut alors assurer la cohérence des identifiants entre le programme serveur et le programme client, ce qui est d'autant plus difficile si les identifiants sont générés automatiquement.

Solutions de plus haut niveau Les solutions industrielles du type GWT, cherchent alors à résoudre ces problèmes en s'abstrayant des modèles bas-niveau. Mais la solution donnée consiste à proposer une collection extensive de *widgets* pré-fabriqués, et à ne laisser au programmeur que la liberté de les assembler, comme dans les interfaces graphiques classiques. Si cette solution peut être vue comme une avancée en termes d'ingénierie, elle ne nous convainc pas, car elle masque complètement la notion de document. D'autre part, le gain de temps apporté peut être très positif comme très négatif. En effet, si jamais le programmeur a besoin de sortir du modèle imposé, il doit alors (1) fournir les implantations bas-niveau de ses composants, et donc en apprendre les techniques, ce qui implique (2) de comprendre le fonctionnement interne spécifique à la bibliothèque, ce qui est souvent non trivial car les programmeurs de ces bibliothèques ont souvent un niveau bien meilleur que celui du programmeur moyen.

Solutions de recherche Plusieurs solutions de recherche, que nous détaillerons plus tard dans ce document cherchent à améliorer la génération de document côté serveur. Le programmeur manipule vraiment un document arborescent dont les éléments sont ceux de la grammaire visée, et non un flux de texte ou des composants pré-fabriqués d'un modèle plus abstrait. De plus, plusieurs solutions permettent de faire le lien entre les parties du document serveur et les parties correspondantes du document client automatiquement, ainsi que de créer des parties de document côté client avec la même API que côté serveur.

Par contre, toutes les solutions à notre connaissance pêchent sur la modification du document côté client, qui reste très impérative, bas niveau et très différente de l'API de création. C'est d'autant plus problématique que ce caractère impératif des modifications se propage insidieusement, affaiblissant la correction de la création de document côté client et même côté serveur, ainsi que la liaison entre les documents client et serveur.

1.3.5 Accès aux données

Outre la création de document, le travail essentiel d'une application Web est le traitement de données. On trouve aujourd'hui sur le Web une grande variété d'applications, et autant de types, quantités et utilisations des données qu'elles manipulent. La première caractéristique est bien sûr la quantités de données de l'application. Cette quantité se recoupe souvent avec le nombre d'utilisateurs. De même, suivant les types d'applications, la fréquence de modification de ces données peut aller de très peu fréquente (par exemple dans un Wiki) jusqu'à plusieurs centaines de fois par seconde (par exemple sur un site d'enchères). Les trois solutions répandues pour stocker ces données sont (1) le vénérable système de fichiers, (2) les SGBDR SQL et (3) les nouvelles bases de données fortement distribuées de la mouvance NoSQL.

Bases de données SQL Le moyen le plus répandu pour stocker les données dans les applications Web est l'utilisation d'un SGBDR de type SQL. Le succès de cette approche est très facile à expliquer. Dans la plupart des applications Web courantes (Wiki, forum, CMS (*Content Management System (gestionnaire de contenu)*), etc.), l'intégrité et la cohérence des données est très importante. Étant donné la trop grande simplicité des langages de scripts courants, et leur difficulté de déverminage, l'utilisation du modèle transactionnel est une aubaine pour le programmeur Web, qui peut déléguer la gestion des problèmes de concurrence au SGBDR. Par ailleurs, étant donné le nombre de développeurs Web amateurs, l'existence des moteurs gratuits MySQL et PostgreSQL a permis l'ancrage définitif du modèle SQL à la programmation Web.

Mais ces SGBDR atteignent leur limite d'adaptabilité avec les applications à très grand nombre d'utilisateurs, du type Facebook ou Twitter. Comme les solutions de stockage de masse présentées juste après n'étaient pas à même d'être utilisées en production à temps, la plupart de ces sites à fort trafic utilisent toujours l'architecture LAMP. Mais pour que le modèle fonctionne, il faut l'utiliser de façon détournée pour supporter la charge. Par exemple, l'architecture de WikiPédia [📖54] utilise une dizaine de logiciels auxiliaires pour aider à distribuer l'architecture LAMP, et implante des mécanismes de mise en cache spécifiques. D'autres sites à très forte fréquentation du type Facebook utilisent encore LAMP, mais en plus de devoir utiliser une architecture distribuée complexe, se limitent aux fonctionnalités basiques de SQL pour des raisons de performances, sans laisser le SGBDR assurer la cohérence des mises-à-jour.

Du point de vue sécurité, l'accès à la base de données SQL est bien souvent le principal point noir, car les langages de script se contentent de fournir un accès aux bases de données via des requêtes textuelles, forgées dans des chaînes de caractères du langage. Il est alors très difficile de s'assurer qu'une donnée issue d'un formulaire ne se retrouvera pas mal insérée dans une requête, offrant un emplacement facile pour injecter des ordres arbitraires au moteur de base de données à une personne malveillante. La figure 1.7 présente quelques exemples d'injection SQL.

```
$result = "SELECT * FROM users"
        + "WHERE user = '" + $user + "' ;"
```

Si un utilisateur arrive à modifier son nom, en lui substituant la chaîne "' OR '42'='42", la commande PHP ci-dessus rapatriera les données de tous les utilisateurs au lieu de celles de l'utilisateur uniquement. **Solution usuelle** : protéger manuellement tous les caractères spéciaux dans les chaînes.

(a) Échappement de chaîne et conditionnelles

```
$result = "SELECT title FROM books"
        + "WHERE title = '" + $title + "' ;"
```

Une recherche en utilisant le nom de livre "'; DROP TABLE 'books"* suffit à détruire la table des livres. **Solution usuelle** : protéger tous les caractères spéciaux dans les chaînes et/ou n'autoriser qu'une instruction par requête.

(b) Échappement de chaîne et séquences

FIGURE 1.7: Exemples d'injection SQL en PHP

Pour faciliter la programmation, et limiter les failles, certains moteurs plus évolués proposent un mécanisme de liaison automatique entre les objets du langage et des données de la base. Ce type de

mécanisme, s'il apporte une certaine sécurité, limite l'expressivité des requêtes possibles, et donc l'intérêt du modèle transactionnel, et fait à nouveau reposer l'intégrité des données sur le langage.

Système de fichiers Certaines applications ne nécessitent pas systématiquement le recours à un modèle transactionnel. Dans ce cas, il est toujours possible d'utiliser le système de fichiers. Cette approche a de nombreux avantages : (1) elle peut être déployée quasiment partout, en particulier chez les hébergeurs gratuits, (2) elle permet facilement d'inter-opérer avec d'autres applications, ou de manipuler les fichiers à la main, (3) elle réduit les risques de corruption des données par injection de code SQL dont nous venons de parler (mais en introduit d'autres).

Mais outre les quelques cas spécifiques où le système de fichiers est suffisant pour stocker les données de l'application, il est utilisé comme source de stockage auxiliaire par quasiment toutes les applications Web, pour mettre en cache des résultats de requêtes complexes, voire même parfois des pages Web complètes.

Stockage dans le Nuage Avec les nouvelles applications sociales à très grande échelle, le moyen traditionnel d'accès aux données n'est plus adapté, ou avec des efforts très lourds, et ce pour plusieurs raisons, techniques comme commerciales.

Les premières raisons pour rejeter l'utilisation des SGBDR dans le domaine des applications à très grande échelle sont techniques. Tout d'abord, La capacité nécessaire pour stocker les informations de centaines de millions d'utilisateurs ne peut être regroupée dans une seule machine, ni même une petite grappe assimilable à une machine unique. À cette contrainte technique, il faut ajouter le fait que de nombreuses applications Web modernes ont une envergure mondiale, et qu'il est important de disposer de serveurs près des utilisateurs pour améliorer les temps d'accès, en particulier car ces applications sont de plus en plus basées sur AJAX, et font en permanence des requêtes depuis le client. Ainsi, dans une application à très grande échelle, il faut obligatoirement une architecture distribuée, avec des temps de latence entre les nœuds non négligeables. Les SGBDR classiques n'ont simplement pas été pensés pour ce modèle de distribution, et l'explosion rapide de ces applications Web n'a pas permis leur adaptation.

Outre ce caractère fortement distribué difficile à mettre en œuvre dans les SGBDR, un autre point plus grave est que le modèle à transaction, et la cohérence forte des données qu'il implique, ne peut passer suffisamment à l'échelle. Dans ce monde du Web à très grande échelle, s'il est très important que l'utilisateur obtienne les résultats qu'il demande le plus rapidement possible, l'exactitude des réponses est le plus souvent moins importante. Il n'est pas très grave, par exemple, qu'un utilisateur de Tokyo qui lit les nouvelles du jour de ses amis au petit déjeuner rate une mise à jour d'un de ses amis de Paris expliquant qu'il a acheté un nouveau rasoir. Bien sûr, il faut que la nouvelle lui parvienne, et ce dans un ordre cohérent avec le reste des informations données par l'utilisateur de Paris. Par contre, ces informations peuvent être désordonnées par rapport à celles des autres utilisateurs, et apparaître un peu en décalage temporel selon l'endroit où elles sont consultées, sans effet notable sur la fiabilité ou la sécurité de l'application. Dans ce but précis, plusieurs projets (citons Cassandra ou CouchDB) de bases de données, connus sous le nom de mouvance NoSQL, cherchent à permettre une distribution à grande échelle avec des lectures et mises-à-jour rapides, en proposant un modèle plus simple et en relaxant les contraintes de cohérence des SGBDR classiques.

Une autre notion, nécessaire à de nombreux concepts d'applications Web à la mode, et encore une fois incompatible avec les SGBDR classiques, est celle d'*élasticité* [📖25]. Si quelques applications à très grande envergure (citons Facebook ou YouTube) ont les moyens financiers pour faire construire d'immenses centres de données répartis dans le monde, c'est loin d'être le cas de toutes. D'autre part, beaucoup d'applications n'ont pas des besoins constants, en stockage comme en calcul. Plusieurs entreprises (citons Amazon ou OVH) ont alors cherché à exploiter cette filière commerciale, et louent des machines et de l'espace de stockage à la demande. Si l'architecture des machines est élastique, il faut alors bien entendu que les logiciels de base de données le soient aussi, et puissent s'adapter aux changements de

l'architecture sur laquelle ils s'exécutent.

1.4 Point de vue développé dans cette thèse

Dans cette introduction, nous avons vu qu'il existe de nombreuses façons d'aborder le problème de la programmation Web, et autant de solutions industrielles : langage de scripts classiques, assemblage de composants sur le serveur, bibliothèques à-tout-faire en JavaScript, etc.

Toutes ces solutions ont leurs avantages, allant de la grande facilité d'abord par les débutants à une fiabilité maximale sur architectures distribuées à grande échelle et élastiques. Mais mis à part leurs avantages et inconvénients spécifiques, toutes les solutions grand public ou industrielles souffrent à nos yeux de deux lacunes principales.

1. De façon générale, ces solutions sont soit des assemblages de composants existants (langages de scripts, SGBDR, etc.), soit des implantations sur le Web de concepts existant (interfaces graphiques). Aucune n'a réellement bénéficié d'une réflexion globale, prenant en compte toute l'architecture de la solution.
2. Et plus précisément, comme nous l'avons présenté en détail, ces solutions ne considèrent pas comme centrale la notion de document interactif, que nous considérons comme essentielle. Et cela est vrai qu'il s'agisse de solutions bas-niveau du type langages de scripts, ou la notion de document n'existe même pas, ou de celles de haut-niveau, du type assemblage de composants d'interface graphiques, s'abstrayant complètement du document, et même en partie des solutions de recherche.

Dans cette thèse, nous cherchons à traiter ces deux thèmes, en proposant une solution de programmation Web homogène et architecturée autour d'une unique notion de document interactif.

Vers un langage Web centré sur le document Nous avons fait le choix d'une approche de conception et implantation de langage, qui est pour nous celui qui permet le plus de liberté pour concevoir une solution vraiment adaptée au domaine. C'est, d'autre part et plus objectivement, le choix le plus approprié pour faire en sorte que la programmation soit vraiment homogène entre des différentes parties, tout en maximisant les possibilités offertes au programmeur.

Un seul langage Clairement, le principal obstacle pour le programmeur débutant, et la principale source de bogues pour tous, est la multiplicité des langages et bibliothèques à maîtriser pour écrire une application Web complète. Ce problème est aggravé par le fait que les interfaces entre ces technologies ne sont satisfaisantes ni sur le plan du développement et du déverminage (exemple des identifiants dans le document), ni sur celui de la sécurité (exemple de l'interface textuelle avec les BDR (*Base de Données Relationnelle*)).

Si les solutions industrielles adoptent le choix pragmatique de la composition et l'adaptation de solutions existantes, nous avons préféré faire le choix d'un langage unique permettant de développer l'ensemble de l'application dans un environnement cohérent. C'est par ailleurs le même choix que les solutions de recherche similaires.

Il y a deux possibilités pour arriver à ce but (1) partir d'un langage existant, suffisamment expressif et adaptable pour implanter les solutions théoriques sans avoir à trop les simplifier, ou (2) partir de zéro, en prenant toutes les libertés pour exprimer exactement les solutions théoriques en concevant le langage en ce sens.

Pour cette thèse, nous avons cherché à concilier les deux approches. D'une part, nous donnons une spécification de langage donnant une solution théorique au problème, et nous avons cherché à garder la spécification la plus simple possible pour qu'elle soit adaptable au maximum. D'autre part, nous expliquons comment implanter les solutions proposées dans un cadre existant. Concrètement,

cette thèse s'inscrit en partie dans le projet Ocsigen. C'est dans ce cadre et, de façon plus générale, sur le langage OCaml sur lequel il se base, que nous donnerons des solutions pratiques détaillées. Nous donnerons aussi, lorsque cela a du sens, des pistes d'implantation dans d'autres langages.

Un seul modèle de document En plus de proposer un seul langage, nous souhaitons que le document retrouve sa place au centre de la programmation Web. Pour cela, nous proposons un unique modèle de document, de haut niveau, pouvant servir à la création comme à la manipulation, sur le serveur comme sur le client. En plus d'apporter de la cohérence au langage, le mécanisme permet d'utiliser les références aux parties du document prises sur le serveur dans le code client, supprimant complètement la nécessité de la manipulation d'identifiants et les problèmes liés.

Typage statique à tous les étages De notre point de vue le typage statique est un bon moyen pour l'aide au développement et au déverminage, mais aussi pour garantir un certain niveau de sécurité, ce qui est un point primordial sur le Web. Nous avons donc fait le choix de concevoir une solution reposant sur le typage statique.

Si des solutions existent pour assurer par typage les communications client/serveur, ou les requêtes de bases de données, ce n'est pas le cas du document. Des solutions existent pour générer de façon bien typée du XML, mais, comme nous l'expliquerons le moment voulu, elles ne sont pas directement applicables au modèle du document sur le client.

Les solutions industrielles dans des langages typés statiquement à la Java, comme GWT, permettent déjà de construire et de modifier des documents de façon bien typée, mais elles reposent sur des fondations fragiles. En effet, si les objets de haut niveau sont bien typés, ils sont implantés dans une couche basse non typée utilisant les mécanismes de base sur le client comme sur le serveur.

Nous avançons que bénéficier d'une couche basse typée permet de faciliter la programmation de composants de haut niveau implantés par dessus, mais surtout d'être plus sûr de leur correction [4]. D'autre part, nous pensons qu'une couche basse typée unique autorise le programmeur à plus facilement étendre les bibliothèques de composants existantes, et limite les mauvaises interactions entre celles-ci. Pour ces raisons, le modèle de document que nous proposons est donc typé statiquement, au niveau de la création comme de la modification.

1.5 Plan de la thèse

Cette thèse est organisée en trois parties. La première partie est plutôt pratique, et présente plusieurs expériences autour de la programmation du client. La seconde est plus théorique, et présente nos travaux sur le modèle de document. La troisième partie présente les travaux connexes, et conclut sur une proposition de langage, utilisant les résultats des deux parties précédentes. Concrètement, les deux premières parties peuvent être lues indépendamment, mais il est préférable de les avoir abordées avant de lire la dernière partie.

Dans la partie I, nous présentons OBrowser, une expérience réalisée durant cette thèse, qui permet d'exécuter du code OCaml dans le navigateur. Ce résultat a été très important car, d'une part il nous a permis d'avoir une plate-forme expérimentale de programmation client/serveur dans le même langage, et d'autre part il a montré la faisabilité de s'abstraire du modèle de langage et d'exécution de JavaScript.

Nous présentons aussi plusieurs expériences permettant d'inter-opérer avec l'environnement du navigateur, en particulier le document, ainsi qu'avec les bibliothèques tierces existantes, là encore en utilisant les structures et le système de types du langage OCaml.

4. Cette vision peut être reliée à celle des machines virtuelles modernes du type .Net, dont le langage machine est typé, permettant d'utiliser différents modèles de langage source, sans sacrifier la sûreté.

Au moment où cette thèse a débuté, très peu de solutions similaires existaient, mais la situation à changé depuis. Cette partie sera donc conclue par la comparaison avec les travaux récents, et les perspectives possibles pour ce travail.

Dans la partie II, nous commençons par définir une spécification formelle d'une notion de document impératif, proche de celui présent dans les navigateurs. Puis nous définissons pour ce document impératif une sémantique alternative, permettant son utilisation de façon statiquement bien typée, pour la création comme la modification dynamique du document, et qu'il est possible d'implanter sur le navigateur comme le serveur, avec la même sémantique.

Ensuite, nous proposons un langage généraliste à la ML, pas aussi expressif qu'OCaml mais pas jouet pour autant, et muni de primitives de manipulations de ce document impératif. Nous décrivons formellement son évaluation, et son système de types, permettant de garantir le typage de la création du document, mais aussi la conservation de ce typage lors des modifications, grâce à la sémantique alternative du document.

Dans la partie III, nous commençons par donner un tour d'horizon des langages multi-tiers issus de la recherche. Nous précisons pour chacun le rapport au document côté client comme côté serveur, mais aussi le reste des problématiques inhérentes à la conception d'un langage pour le Web, et que nous avons pu apercevoir dans cette introduction.

À partir de cette comparaison, et en utilisant les résultats des parties I et II, nous présentons les grandes lignes d'un langage multi-tiers statiquement typé, à la programmation et au modèle d'exécution homogène et flexible, utilisant un modèle unique de document, côté client comme côté serveur, et en précisons les apports par rapport aux solutions existantes.

1 Introduction

I Première partie
Programmation des navigateurs en OCaml

2 Présentation générale d'OBrowser

Dans la première partie de cette thèse, nous avons cherché à programmer le client dans un langage de haut niveau et statiquement typé. Étant donné le peu de recul sur la programmation des navigateurs, nous avons préféré, plutôt que de nous lancer dans la conception d'un nouveau langage, fournir une plate-forme expérimentale en utilisant un langage existant.

Notre choix de langage s'est porté sur OCaml. La raison principale derrière ce choix est qu'OCaml autorise l'utilisation de la plupart des paradigmes de programmation et de concurrence, permettant de tester en pratique un large panel de styles de programmation. De même son système de types est suffisamment souple et expressif pour encoder un grand nombre de propriétés, tout en ne s'éloignant pas trop de ceux langages plus grand public, permettant que les solutions soient exploitables en pratique. Bien sûr, d'autres langages fournissent ces propriétés (citons F#, SML (*Standard ML*) ou Scala), nous avons alors choisi OCaml car nous le connaissons bien, et pour permettre d'avoir une plate-forme de test pour la programmation client au sein du projet Ocsigen, dont la partie serveur est écrite en OCaml.

Concrètement, pour utiliser dans le navigateur un langage différent de JavaScript, il y a deux possibilités principales : l'utilisation d'un greffon ou la compilation vers JavaScript. Dans cette partie, nous décrivons une troisième approche : OBrowser [🕮6], une machine virtuelle OCaml, écrite en JavaScript.

Motivations Il aurait été possible d'écrire un nouveau compilateur, ou de modifier le compilateur OCaml existant pour ajouter une cible JavaScript. Ce travail a d'ailleurs été réalisé indépendamment et sera présenté au chapitre 6. Mais l'implantation sous forme d'une machine virtuelle offre un certain nombre d'avantages, dont voici les plus importants.

- Le compilateur OCaml est largement éprouvé, il est donc préférable de le réutiliser plutôt que d'écrire un compilateur qui n'atteindra jamais le même niveau de maturité.
- Le format du code-octet d'OCaml change rarement d'une version à l'autre d'OCaml, ce qui n'est pas le cas de la chaîne de compilation, la maintenance est donc plus facile.
- Nous verrons que l'abstraction du modèle d'exécution de JavaScript est plus facile avec une machine virtuelle qu'avec des techniques de compilation. Par exemple, OBrowser fournit la concurrence préemptive, qui n'existe pas en JavaScript.
- Dans le cadre d'expérimentations client/serveur avec le serveur Ocsigen écrit en OCaml, il est important d'avoir exactement la même sémantique des deux côtés, ce qui est plus facile à assurer avec une machine virtuelle qu'avec un compilateur.
- Le déverminage de code JavaScript est difficile, est encore plus si celui-ci est généré. Celui d'une machine virtuelle est plus facile, il est par exemple possible de l'instrumenter pour comparer son exécution pas à pas avec celle de la machine originale.
- Et bien sûr, l'idée semblant un peu folle et amusante, alors nous avons voulu tenter le coup.

Contexte historique Une large partie de ces raisons et des solutions techniques données dans cette partie reste valide aujourd'hui. Néanmoins, certains de ces points sont à replacer dans le contexte de l'état du parc des navigateurs et des outils de développement Web en 2007, époque où l'expérience a commencé. Depuis, les outils de développement et de déverminage JavaScript ont évolué, et les interprètes JavaScript ont été optimisés de façon impressionnante. Le chapitre 6 présentera des alternatives plus récentes, expliquera en quoi elles sont en partie mieux adaptées aux navigateurs d'aujourd'hui, et présentera des travaux possibles pour mieux prendre en compte ces évolutions dans OBrowser.

Dans ce chapitre d'introduction, maintenant que nous avons présenté le contexte et les motivations de sa conception, nous donnons une présentation d'OBrowser du point de vue utilisateur.

Nous commençons par décrire l'architecture générale du système, et son utilisation par un exemple minimal de programme OCaml s'exécutant dans le navigateur. Puis, nous décrivons les traits du langage et de la bibliothèque OCaml implantés dans OBrowser, en précisant les adaptations faites pour s'intégrer à l'environnement du navigateur. Nous présentons aussi les traits avancés d'OCaml non triviaux à implanter, ainsi que les traits ajoutés à la bibliothèque, spécifiques à la programmation du navigateur. Ensuite, nous présentons les possibilités d'interface avancée avec le monde JavaScript, importantes pour utiliser les bibliothèques existantes. Enfin, nous donnons un petit exemple pour résumer plus concrètement cette introduction, et le plan détaillé de la suite de cette partie.

Dans les annexes A et B, le lecteur pourra trouver, si besoin, une introduction aux deux technologies avec lesquelles il est nécessaire d'être familiarisé pour comprendre cette partie sur OBrowser : le langage JavaScript et son environnement d'exécution au sein des navigateurs, et la machine virtuelle et la bibliothèque d'exécution d'OCaml.

2.1 Architecture générale

La figure 2.1 présente comment, à partir du compilateur OCaml standard, des sources téléchargées d'OBrowser et d'un programme OCaml utilisateur, exécuter ce dernier dans un navigateur Web. Une application OBrowser est composée de trois fichiers : (1) Un fichier HTML page.html ouvrable dans un navigateur, servant à lancer une instance de la machine virtuelle OBrowser avec le programme OCaml souhaité, (2) la machine virtuelle OBrowser dans un fichier vm.js, et enfin (3) le programme utilisateur compilé prog.uue.

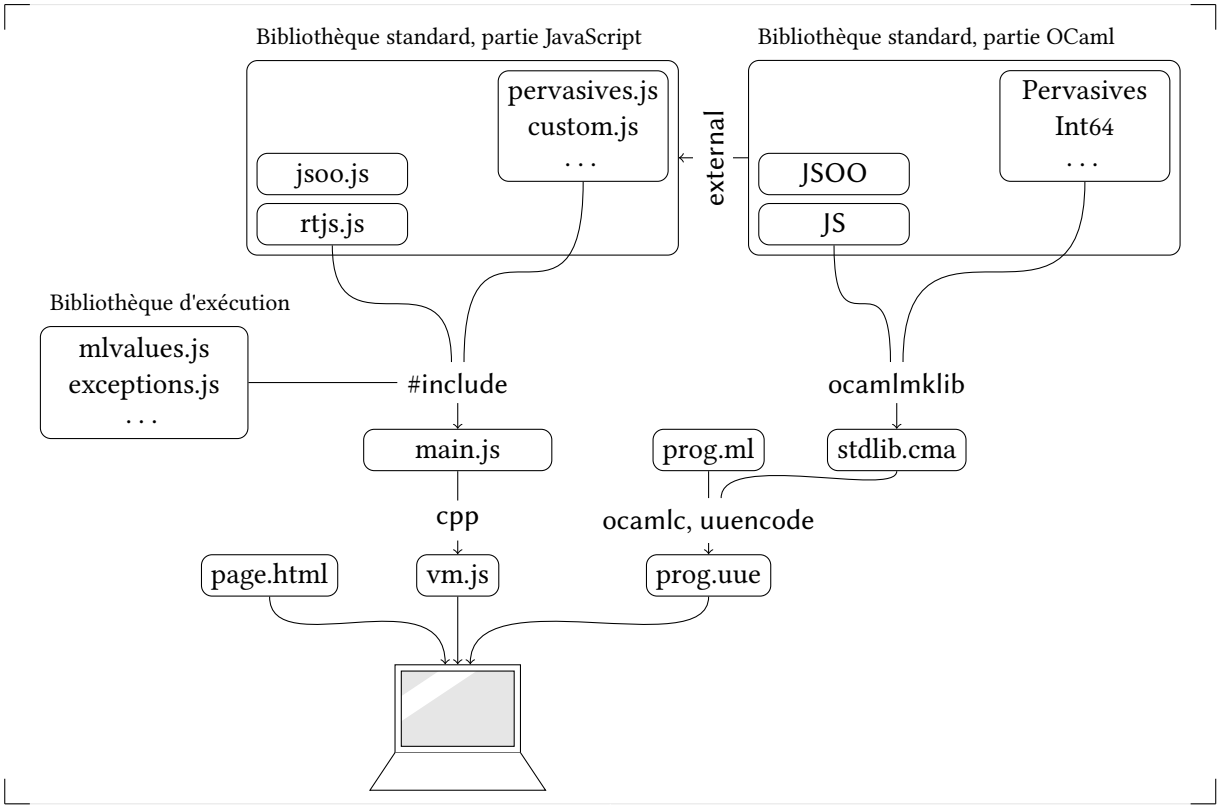


FIGURE 2.1: Architecture d'OBrowser.

```
1 Js.alert "Hello World" ;;
```

(a) Programme principal

```
1 export CAMLLIB=/usr/lib/ocaml/obrowser
2 ocamlc prog.ml -o prog
3 uuencode prog stdout > prog.uue
```

(b) Compilation et encodage

```
1 <html>
2   <head>
3     <title>Hello World</title>
4     <script type="text/javascript" src="vm.js"></script>
5     <script type="text/javascript">
6       window.onload = function (e) {
7         exec_caml ("prog.uue", "main");
8       }
9     </script>
10  </head>
11  <body> Hello World </body>
12 </html>
```

(c) Page Web lançant le programme

FIGURE 2.2: Un *«Hello World»* en OBrowser

1. Le fichier HTML doit charger la machine virtuelle via une balise liant le fichier de script vm.js, et appeler la machine virtuelle sur le programme utilisateur avec la fonction JavaScript exec_caml. Nous verrons que si ce fichier contient un corps, celui-ci pourra être parcouru et modifié par le programme OBrowser.
2. La machine virtuelle OBrowser est distribuée dans les sources sous forme d'un ensemble de fichiers JavaScript. Pour faciliter l'utilisation tout en conservant du code lisible et modulaire, le fichier unique vm.js est produit par le pré-processeur cpp (d'autres fonctionnalités du pré-processeur sont utilisées, notamment les macros pour optimiser le code en expansant certaines structures et constantes utilisées fréquemment). Ce fichier contient l'interprète de code-octet, mais aussi les fonctions externes de la bibliothèque standard OCaml, et les implantations de fonctions spécifiques à la bibliothèque d'OBrowser pour accéder à l'environnement du navigateur.
3. Pour obtenir le programme prog, on utilise le compilateur ocamlc standard, mais il faut utiliser une version alternative de la bibliothèque OCaml, distribuée avec OBrowser. La bibliothèque d'OBrowser est celle d'OCaml un peu modifiée et augmentée, comme nous l'expliquons dans la section qui suit. D'autre part, afin d'assurer la compatibilité avec certains navigateurs ne prenant en charge le téléchargement de fichiers qu'en texte brut 7 bits, on utilise le classique uuencode pour permettre la transmission des fichiers.

La figure 2.2 donne un exemple concret de programme, ainsi que la méthode de compilation. On utilise le compilateur existant, mais il faut le paramétrer via une variable d'environnement, pour qu'il utilise la bibliothèque standard d'OBrowser à la place de celle du système.

2.2 Bibliothèque standard OCaml

Deux méthodes existent pour adapter la bibliothèque standard OCaml à un environnement particulier. La possibilité la plus évidente est (1) d'éditer les sources de la bibliothèque standard, en supprimant les fonctions qui n'ont pas de sens dans l'environnement, et éventuellement en éditant certaines fonctions afin de leur donner du sens. Alternativement, on peut (2) se contenter de conserver telle quelle la signature de la bibliothèque standard, et ne pas implanter les primitives externes. Il est alors possible de lancer une exception OCaml à l'exécution en cas d'appel d'une primitive non implantée, ou de vérifier après compilation qu'un appel à une telle primitive n'apparaît pas dans le fichier de code-octet.

Avec la méthode 1, les fonctions non prises en charge n'apparaissent plus dans la signature de la bibliothèque, et donc dans la documentation. Avec la méthode 2, on suppose que le programmeur saura

repérer les fonctions qui n'ont pas de sens dans le navigateur. Cependant, cette décision n'est pas toujours évidente, et elle l'est encore moins pour les fonctions de bibliothèques de haut niveau appelant indirectement des primitives non prises en charge. La méthode 1 implique par définition la suppression ou l'adaptation de toutes les parties de la bibliothèque amenées à effectuer un appel à une primitive non prise en charge. La méthode 1 est donc clairement la plus élégante, c'est par exemple celle choisie dans le portage d'OCaml vers l'environnement très spécifique des micro-contrôleurs PIC [📖33], et celle que nous avions choisie initialement pour OBrowser.

Cependant, si la méthode 2 est clairement moins propre, dans le cadre de programmes client/serveur en OCaml, c'est la seule possibilité si on veut permettre le partage de code entre les parties, sans avoir à recompiler séparément le code partagé pour chaque partie. En particulier, c'est la seule solution autorisant l'utilisation de bibliothèques externes pré-compilées pour la bibliothèque standard OCaml. La raison technique vient du mécanisme de compilation séparée d'OCaml, qui n'est pas assez fin pour accepter d'utiliser un module compilé avec une version différente de la bibliothèque standard, même dans le cas où celui-ci n'utilise pas les fonctions enlevées. La méthode 2, de conservation de la signature de bibliothèque standard OCaml, a donc été choisie lors de l'utilisation d'OBrowser comme plate-forme expérimentale du projet Ocsigen.

Adaptations Une large majorité de la bibliothèque standard étendue d'OCaml est implantée dans OBrowser. Néanmoins, quelques modules de la bibliothèque standard n'ont pas pu être implantés ou seulement en partie, et d'autres adaptés. Les principales adaptations sont les suivantes.

- Naturellement, les primitives de gestion de fichiers (entrées/sorties et gestion de l'arborescence) ne sont pas implantées. De même pour le module Unix.
- Une exception au point précédent est faite pour le flux de sortie standard, qui est affiché dans une zone de la page servant de console texte. Ce mécanisme est implanté principalement afin de pouvoir tester les programmes existants effectuant des affichages, avant de les adapter pour un affichage spécifique au navigateur.
- La fonction Sys.argv, au lieu de renvoyer les arguments de la ligne de commande, renvoie les arguments passés à la fonction exec_caml, encore une fois principalement pour tester des programmes existants sans avoir à les modifier.
- Les modules liés au ramasse-miettes Gc et Weak ne sont pas implantés, car OBrowser utilise le ramasse-miette de JavaScript. Or celui-ci ne peut être configuré et ne propose pas de mécanisme de références faibles.
- Le module Str n'est pas implanté, mais un module RegExp est fourni, permettant d'utiliser les expressions rationnelles de JavaScript.

Simulation d'un environnement classique Lors de l'écriture initiale de la machine virtuelle, nous avons cherché à reproduire l'environnement classique d'exécution OCaml. Dans cette approche, plutôt que de ne pas implanter les fonctions d'entrée/sortie, nous avons cherché à les simuler dans le navigateur. Par exemple, chaque canal d'entrée/sortie disposait d'une fenêtre à l'intérieur du navigateur, de même pour la fenêtre graphique. La figure 2.3 montre une capture d'écran de cette version.

Cette première version a principalement servi à développer la machine, elle incorporait pour cela un mode de déverminage pas à pas et d'exploration des valeurs OCaml. Cette expérience, si elle n'a pas subsisté pour notre utilisation, pourrait être réutilisée dans un but pédagogique.

2.3 Traits avancés d'OCaml

En plus du langage de base, et de la bibliothèque standard, OBrowser implante plusieurs traits avancés du langage, permettant de se rapprocher au maximum de la sémantique standard d'OCaml, afin d'avoir un environnement d'expérimentation client/serveur homogène.

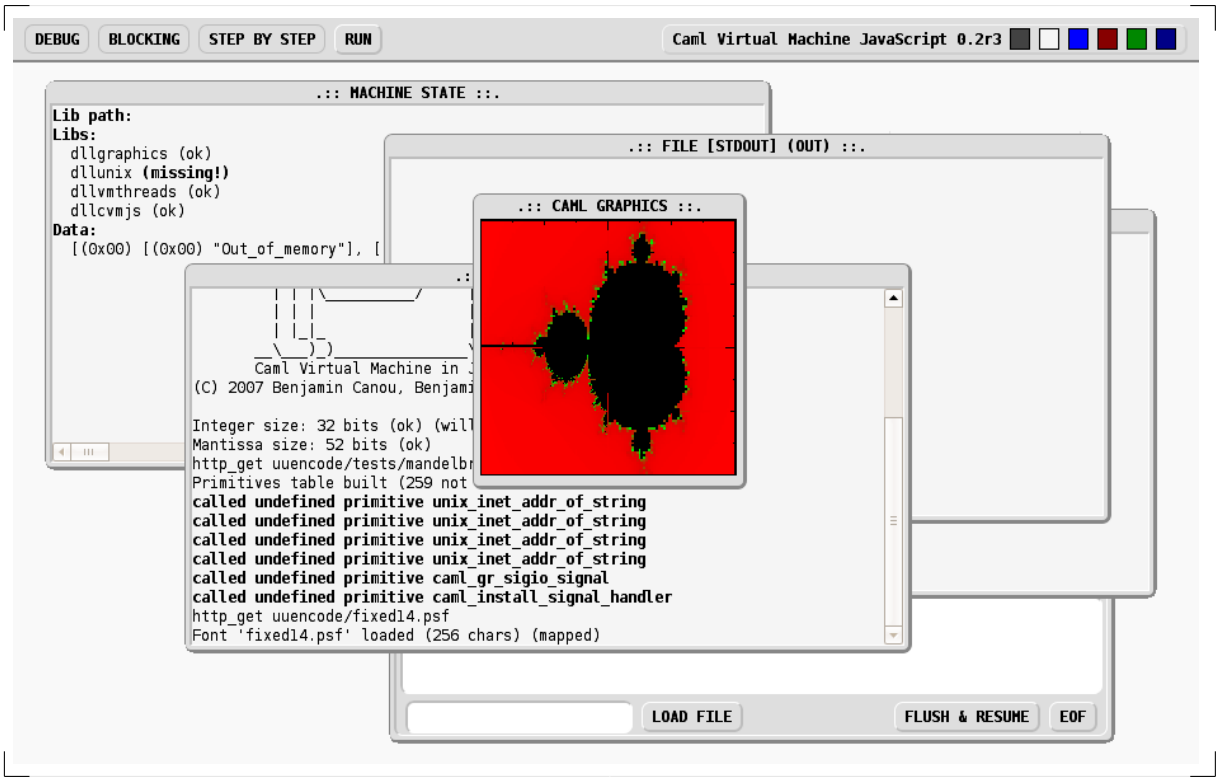


FIGURE 2.3: La première machine virtuelle Caml en JavaScript

Threads préemptifs contrairement à la plupart des solution de programmation du navigateur, OBrowser n'impose pas d'utiliser le modèle événementiel de JavaScript. Il implante pour cela les threads préemptifs, avec l'API classique d'OCaml. Outre le fait que cela permet d'avoir le même modèle de concurrence qu'OCaml, le modèle des threads préemptifs est suffisamment souple pour implanter la plupart des autres modèles de concurrence, et est donc une bonne solution pour une plate-forme expérimentale. Par exemple, pour l'utilisation dans le projet Ocsigen, la bibliothèque Lwt a été implantée pour OBrowser, au dessus des threads préemptifs.

Interface avec JavaScript OBrowser fournit plusieurs méthodes pour l'inter-opérabilité entre OCaml et JavaScript.

1. Tout d'abord, un module Js est présent dans la bibliothèque standard, qui intègre des fonctions prédéfinies spécifiques à l'environnement du navigateur, et qui sera revu plus en détails à la section suivante.

2. Pour le programmeur souhaitant inter-opérer avec des parties du navigateur non présentes dans Js, ou avec des bibliothèques externes, OBrowser définit une FFI (*Foreign Function Interface (interface de fonctions externes)*) en JavaScript semblable à celle avec d'OCaml en C, utilisant le mot-clef external d'OCaml.

3. OBrowser offre une seconde FFI, dans le module OCaml JSOO. À l'inverse de la FFI OCaml classique, et de façon proche de la FFI de Haskell, il s'agit là de manipuler les valeurs et fonctions JavaScript depuis OCaml. Ces deux FFI seront présentées plus tard dans ce chapitre.

4. Enfin, nous présenterons, plus tard dans cette partie, une couche d'inter-opérabilité de haut niveau utilisant les mécanismes objet des deux langages.

Attentes bloquantes En JavaScript, les attentes de ressources, que ce soient celles prédéfinies du navigateur, ou celles introduites par des bibliothèques tierces, sont écrites de façon asynchrone. Concrètement, le programmeur JavaScript, lors de la demande de ressource, enregistre une fonction qui sera rappelée lorsque la ressource sera prête. Ce mécanisme est bien adapté au modèle événementiel de JavaScript, mais il n'est pas naturel pour OCaml et le modèle préemptif. OBrowser définit pour ceci un mécanisme, permettant de faire passer une attente asynchrone JavaScript pour un simple appel de fonction depuis OCaml. Ce mécanisme sera présenté en détail dans la présentation de le FFI, et son implantation au chapitre 4.

Sérialisation Afin de pouvoir communiquer des valeurs OCaml directement entre le client et le serveur, OBrowser prend en charge la (dé)sérialisation de valeurs, dans le même format qu'OCaml. En particulier, ce format étant très lié à la représentation mémoire des valeurs, OBrowser utilise volontairement une représentation très proche de celle d'OCaml.

Module Graphics Afin de permettre d'exécuter les programmes graphiques OCaml, sans avoir à utiliser une bibliothèque alternative de dessin, OBrowser implante le module Graphics d'OCaml, en utilisant les primitives du navigateur.

2.4 Programmation du navigateur

En plus des modules provenant de la distribution d'OCaml, OBrowser fournit dans sa bibliothèque standard les modules Js et RegExp, permettant d'utiliser en OCaml les principales primitives de la bibliothèque standard de JavaScript.

Communications HTTP Via les fonctions http_get et http_post, le programmeur peut effectuer une requête HTTP, et récupérer le contenu sous forme de chaîne OCaml. Côté JavaScript, une attente asynchrone est effectuée via un objet XMLHttpRequest. Du point de vue OCaml, la fonction est bloquante, et retourne simplement le résultat une fois l'attente terminée. Bien sûr, si le programmeur nécessite une attente asynchrone, il peut simplement l'encapsuler dans un thread.

```
(* renvoie le résultat d'une requête GET à l'url donnée, retourne le code et le texte résultat *)
val http_get : string -> int * string

(* renvoie le résultat d'une requête POST à l'url donnée, retourne le code et le texte résultat
   les paramètres supplémentaires sont le ContentType attendu, et les paramètres POST *)
val http_post : string -> string -> string -> int * string
```

Parcours et modification du document OBrowser fournit un sous-module Js.Node, regroupant les primitive de parcours et de manipulation des nœuds de l'arbre du document. L'interface est bas-niveau, proche du DOM, et ne prend pas du tout en compte le typage de la grammaire du document. Elle apporte cependant un minimum de sûreté d'exécution, en séparant les accesseurs des attributs texte, des évènements et des nœuds enfants. La signature simplifiée et commentée du module Js.Node est donnée dans la figure 2.4.

Le type abstrait t du module Node représente un nœud du document. Pour commencer à parcourir l'arbre de la page Web, et exploiter le corps HTML existant, document donne le nœud racine, et get_element_by_id $n\ i$ permet de retrouver sous une racine r donnée le nœud possédant l'attribut *id* i spécifié,

Pour parcourir l'arborescence, on peut utiliser les primitives à la JavaScript n_children, child, ainsi qu'une petite interface fonctionnelle spécifique à OBrowser avec children, iter et fold_left.

```
module Node : sig
  type t (* abstract type of nodes *)

  (* document root *)
  val document : t
  (* retrieve only element with given id under root *)
  val get_element_by_id : t -> string -> t

  (* create text node *)
  val text : string -> t
  (* create element node *)
  val element : string -> t

  (* get/set/remove node attribute *)
  val get_attribute : t -> string -> string
  val set_attribute : t -> string -> string -> unit
  val remove_attribute : t -> string -> unit

  (* set/remove event handler *)
  val register_event : t -> string -> ('a -> unit) -> 'a -> unit
  val clear_event : t -> string -> 'a -> unit

  (* children accessors *)
  val children : t -> t list
  val n_children : t -> int
  val child : t -> int -> t
  val iter : (t -> unit) -> t -> unit
  val fold_left : ('a -> t -> 'a) -> 'a -> t -> 'a

  (* children modifiers *)
  val append : t -> t -> unit
  val remove : t -> t -> unit
end
```

FIGURE 2.4: Signature du module Js.Node.

Il est possible de construire de nouveaux nœuds, text permet de créer un nœud texte à partir de son contenu, et element permet de créer un nœud conteneur (élément de page) en spécifiant son étiquette (pour HTML : "div", "p", etc.). Les nœuds éléments sont vides lors de leur construction, il faut ensuite les initialiser par modification.

Pour modifier les enfants, on utilise les fonctions append, remove. Pour lire, affecter et supprimer les attributs texte des nœuds on utilise respectivement les fonctions get_attribute, set_attribute et remove_attribute. Ces primitives apportent un peu de sûreté d'exécution par rapport aux équivalents JavaScript. Par exemple, get_attribute ne peut renvoyer effectivement qu'une chaîne, et lève une exception si l'attribut n'est pas défini ou n'est pas convertible en une chaîne.

Événements Pour définir les attributs correspondants à des rattrapeurs d'événements, il faut utiliser les fonctions spécifiques register_event et clear_event. Celles-ci permettent d'affecter une fonction OCaml, à exécuter quand l'évènement précisé se produit sur le nœud.

Le modèle d'exécution des évènements est différent de celui de JavaScript. Chaque rattrapeur d'évènement est lancé dans un nouveau thread de la machine virtuelle. Ce modèle a été choisi car il s'intègre le mieux avec la concurrence préemptive. En particulier, il permet l'expérimentation de différents modèles de concurrence et de gestion d'évènements. Par exemple, ce mécanisme permet de retrouver le modèle natif de JavaScript simplement en utilisant un verrou global pour les évènements.

Création de document XHTML Il existe différentes façons de construire des documents XHTML bien formés en OCaml. Par exemple, au sein du projet Ocsigen, le module XHTML.M est préconisé pour générer du XHTML bien formé côté serveur, qui utilise un encodage via les variants polymorphes d'OCaml. Alternativement on peut utiliser le compilateur OCamlDuce, qui introduit en partie le typage XML du

```
module Html : sig
  val create : string -> ?attrs:(string * string) list -> unit -> Node.t
  val string :
    string -> Node.t
  val a :
    ?style:string -> ?onclick:(unit -> unit) ->
    ?href:string -> ?name:string ->
    ?attrs:(string * string) list -> Node.t list -> Node.t
  val div :
    ?style:string ->
    ?attrs:(string * string) list -> Node.t list -> Node.t
  val span :
    ?style:string ->
    ?attrs:(string * string) list -> Node.t list -> Node.t
  val img :
    src:string -> alt:string ->
    ?style:string -> ?attrs:(string * string) list -> unit -> Node.t
  val ul :
    ?style:string ->
    ?attrs:(string * string) list -> Node.t list -> Node.t
  val li :
    ?style:string ->
    ?attrs:(string * string) list -> Node.t list -> Node.t
  (* ... *)
end
```

FIGURE 2.5: Signature simplifiée du module Js.Html.

langage CDuce au langage OCaml. De même, d'autres bibliothèques permettent de produire, soit directement soit à partir d'une représentation intermédiaire, des documents XHTML bien formés. Et bien entendu, on voudrait aussi pouvoir utiliser la solution qui sera présentée dans la partie II de cette thèse.

Nous ne nous attardons pas sur la description de ces solutions, qui sera donnée en temps voulu dans la partie II, cette liste est donnée simplement pour montrer qu'en tant que plate-forme d'expérimentation, fixer une interface pré-définie de création de documents HTML dans OBrowser aurait peu de sens. Cependant, afin qu'OBrowser soit utilisable dès le téléchargement pour écrire de petites applications, sans avoir à utiliser les primitives très bas-niveau de Node, ni à l'inverse de devoir utiliser et apprendre le fonctionnement d'une bibliothèque de haut niveau, OBrowser fournit un module Html, permettant de créer de façon facile et concise des nœuds XHTML. Un extrait de la signature du module Js.Html est donnée dans la figure 2.5.

Ce module, implanté à l'aide des primitives de Node, définit une fonction par élément de la grammaire XHTML. Chaque fonction prend en paramètre une liste de nœuds qui seront insérés en tant qu'enfants. Puisque le type Node.t est complètement générique, c'est au programmeur de vérifier que la composition est bien correcte vis-à-vis de la grammaire XHTML. Par contre, les attributs spécifiques à chaque type de nœud sont reflétés par des arguments optionnels/étiquetés d'OCaml. Le typage assure donc la correction de la présence et des types des attributs. Par exemple, l'attribut onclick des nœuds le prenant en compte, doit être une fonction, et les attributs src et alt des images sont obligatoires.

2.5 Interface de fonctions externes (FFI)

Dans un environnement de programmation Web client, il est intéressant de pouvoir exploiter les bibliothèques tierces existantes. Pour ceci, de façon similaire à l'interface avec C d'OCaml, OBrowser permet l'utilisation de fonctions externes en JavaScript, et définit des fonctions JavaScript d'accès, de modification et de conversion des données OCaml. La FFI définit de plus des primitives de gestion de l'exécution au sein du programme OCaml, en particulier du mécanisme de threads, afin que l'utilisation de bibliothèques externes en OCaml puisse se faire en respectant le modèle préemptif.

Fonctions externes En OCaml, une déclaration de fonction externe se fait avec le mot clef external, en spécifiant les types des arguments et de retour, et le nom de la fonction C implantant le code de la primitive.

OBrowser interprète ces déclarations de façon très similaire à l'interface avec C. L'implantation d'une fonction OCaml externe est simplement la fonction JavaScript dont le nom est le nom externe spécifié par le programmeur. Les paramètres passés depuis OCaml à la fonction externes sont transmis comme paramètres de la fonction JavaScript, et la valeur retournée à OCaml est la valeur de retour de la fonction JavaScript.

```
external nom_caml
  : type_arg1
    -> type_arg2
    -> ...
    -> type_retour
  = "nom_externe"
```

```
function nom_externe(arg1, arg2, ...) {
  /* ... */
  return /* ... */ ;
}
```

Les valeurs transitant entre les mondes sont des valeurs OCaml. Comme pour l'interface avec C, L'interface avec JavaScript n'effectue pas la conversion automatique en valeurs JavaScript natives, c'est au programmeur de le faire via la FFI décrite dans le suite.

Compilation Le compilateur ocamlc étant prévu pour être utilisé avec C, il demande, lors de la compilation d'une déclaration externe, à ce que le fichier objet C contenant le symbole demandé soit présent, pour l'intégrer à l'exécutable. Il faut alors simuler la présence des symboles C pour réussir à compiler un programme utilisant des fonctions externes. La distribution d'OBrowser donne pour cela un Makefile générique, construisant à la volée une bibliothèque temporaire factice en fonction des déclarations trouvées dans le fichier source.

Types simples La FFI définit des fonctions de conversion entre les principaux types simples et leurs équivalent en JavaScript, similaires aux macros C de la FFI d'OCaml.

- val_string(s) (resp. string_val(v)) pour convertir (par copie) une chaine JavaScript en une chaine OCaml (resp. une chaine OCaml en une chaine JavaScript),
- val_float(f) (resp. float_val(v)) pour convertir un nombre JavaScript en un flottant OCaml (resp. un flottant OCaml en un nombre JavaScript),
- val_int(i) (resp. int_val(v)) pour convertir un nombre JavaScript représentable en un entier (sinon il sera approximé) en un entier OCaml (resp. un entier OCaml en un nombre JavaScript),
- comme dans la FFI pour C, des raccourcis UNIT (= val_int(0)), TRUE, FALSE, etc. sont définis pour les types simples pré-définis principaux.

Blocs De même, des fonctions JavaScript permettent de dé-construire et modifier les valeurs structurées, et d'accéder à leurs méta-données.

- is_block détermine si une valeur est un bloc OCaml,
- block_tag(v) donne l'étiquette d'un bloc, les étiquettes spéciales étant disponibles sous forme de variables globales, avec le même nom que dans l'interface avec C,
- block_size(v) donne la taille d'un bloc (contrairement à l'interface avec C, il s'agit effectivement du nombre de valeurs, et non du nombre de mots machine, il n'y donc pas de traitement particulier à faire pour les tableaux de flottants ou les chaînes),
- mk_block(s,t) crée un nouveau bloc de taille s et d'étiquette t, valable pour les blocs normaux, il faut utiliser les fonctions spécifiques pour les étiquettes spécifiques,
- field(b,i) et store_field(b,i) permettent de lire et modifier les sous-valeurs d'un bloc.
- des raccourcis ATOM, mk_pair, etc. sont définis pour les types construits pré-définis principaux.

Données externes Il est possible d'encapsuler les données du monde JavaScript avec les deux méthodes habituelles d'OCaml.

1. Les blocs de données abstraites permettent d'encapsuler une valeur externe, de façon complètement opaque pour OCaml. On peut construire et ouvrir un te bloc via les fonctions box_abstract(v) et unbox_abstract(b),
2. Les blocs de données personnalisées permettent d'encapsuler une valeur externe, généralement opaque pour OCaml sauf pour certaines opérations spécifiques. Pour chaque type de valeur externe, il faut définir un objet JavaScript ops contenant un champ id devant être unique et les fonctions hash(v), compare(v1,v2), serialize(v,writer) et deserialise(reader). Puis il faut enregistrer cet objet dans une table globale avec register_custom. On peut ensuite encapsuler des données d'un type externe ops donné avec mk_custom(ops,val), et les ouvrir avec custom_val(b).

La bibliothèque standard fournit de base les types externes int32 et int64, et les fonctions val_int32, int32_val, val_int64 et int64_val.

Gestion du contrôle La partie de la FFI décrite dans les paragraphes précédents permet d'utiliser du code JavaScript existant, en convertissant les données de, et vers, OCaml. Pour que l'inter-opérabilité soit complète, la FFI d'OCaml définit des primitives avancées permettant au programmeur d'influer sur l'exécution au sein de la partie programme OCaml.

Certaines de ces primitives sont plus limitées que leurs équivalents en C, en grande partie à cause de la difficulté à simuler un modèle préemptif classique au dessus de celui de JavaScript. Nous présenterons plus en détails le fonctionnement de ces primitives au chapitre 4, en expliquant les raisons précises des limitations.

Une différence majeure avec l'implantation standard de la machine virtuelle OCaml est qu'OBrowser est ré-entrant. Il permet de lancer plusieurs programmes OCaml au sein du même environnement JavaScript. De ce fait, certaines primitives de la FFI nécessitent de connaître la machine virtuelle en cours d'exécution. Pour ceci, le mot-clef this désigne l'objet représentant cette dernière dans le corps des fonctions externes. Ainsi, les primitives de gestion du contrôle ne sont pas définies comme de simples fonctions, mais comme des méthodes de la machine virtuelle, et doivent être appelées avec this.primitive par le programmeur.

Exceptions En C, la fonction caml_raise (ainsi qu'une série de fonction plus spécifiques caml_failwith, caml_invalig_arg, etc.) permettent de lever des exceptions OCaml. Concrètement, un appel à une de ces primitives, interrompt brutalement l'appel externe, et au lieu de renvoyer une valeur à l'appel de fonction OCaml, lève l'exception spécifiée dans la machine virtuelle.

OBrowser définit de façon similaire des méthodes raise et (failwith, invalid_arg, etc.), effectuant le même travail.

Fonctions de rappel Comme la fonction C caml_callback, la méthode callback(o,a) permet d'appeler une fonction OCaml depuis JavaScript. Elle prend en paramètre une fermeture OCaml, et un tableau JavaScript de valeurs OCaml contenant les arguments à passer.

De façon similaire, la méthode callback_method(o,m,a) permet d'appeler la méthode de nom m d'un objet OCaml depuis JavaScript.

Threads et attente de ressource Il est possible de piloter les threads préemptifs d'OBrowser depuis JavaScript. Le lancement d'un thread se fait avec la primitive thread_new(c), en fournissant la fermeture OCaml c à exécuter. Cette méthode renvoie l'identifiant id du thread, permettant par exemple de le stopper avec la méthode thread_kill(id).

Les méthodes thread_wait(r,c) et thread_notify_all(r) sont définies dans la machine, pour implanter les attentes de ressources dans les fonctions externes.

Pour rappel, en JavaScript, une attente asynchrone d'événement (ou de ressource) se fait en donnant une fonction de rappel, exécutée par la boucle d'événement lorsque l'événement s'est produit (ou que la ressource a changé d'état). Pour faire passer une telle attente pour un simple appel de fonction OCaml, le programmeur doit procéder comme suit.

1. Il sépare en deux fonctions la définition de sa fonction externe, la partie avant l'attente de ressource, et la partie une fois la ressource obtenue. On appellera la seconde partie la continuation, qui classiquement peut être une fonction définie localement dans la première, afin de pouvoir accéder à ses variables.
2. Il choisit la ressource r sur laquelle attendre. Par exemple, pour une requête HTTP, ce pourra être l'objet XMLHttpRequest associé. S'il s'agit d'une attente d'événement non associée à un objet, il faut allouer une ressource fraîche qui peut être un objet JavaScript quelconque alloué pour l'occasion.
3. À la fin de la première partie, il initialise l'attente asynchrone JavaScript, et dans la fonction de rappel JavaScript, place simplement un appel à thread_notify_all(r).
4. Puis, la toute dernière ligne de code exécutée de la fonction est l'appel de thread_wait(r, c), où c est la continuation.
5. Le thread est alors endormi (mais les autres threads peuvent continuer de s'exécuter, il sera réveillé lorsque la fonction de rappel JavaScript sera appelée, grâce à thread_notify_all(r).
6. Au réveil du thread, la continuation est exécutée, et son résultat constitue le résultat de l'appel externe. Concrètement, il n'y a alors aucune différence avec un appel de fonction normal, du point de vue OCaml.

2.6 Interface de fonctions externes inversée

Si la FFI classique est adaptée à la création d'interfaces avec des bibliothèques existantes, elle s'avère un peu lourde lorsque le programmeur souhaite interagir ponctuellement avec l'environnement du navigateur depuis son programme OCaml, de façon non prévue pas la bibliothèque standard. On peut citer comme exemple l'accès à des informations spécifiques comme le nom ou la version du navigateur, l'utilisation d'une fonction mathématique de JavaScript non présente dans la bibliothèque OCaml, etc.

OBrowser fournit pour ce type d'utilisation le module JSOO (pour *JavaScript Object Operations*), permettant de lire et manipuler les valeurs JavaScript, et d'appeler des méthodes et fonctions JavaScript, entièrement depuis OCaml.

Valeurs JavaScript JSOO définit un type abstrait obj, représentant l'ensemble des valeurs JavaScript. Il définit des fonctions de conversion depuis les types pré-définis d'OCaml, et des fonction d'extraction d'une valeur JavaScript vers un type OCaml demandé, avec un test dynamique levant une exception si la valeur n'est pas du bon type.

```
1 val string s : string -> obj
2 val float f : float -> obj
3 val int i : int -> obj
4 val block b : 'a -> obj
5 val as_string : obj -> string
6 val as_int : obj -> int
7 val as_float : obj -> float
8 val as_block : obj -> 'a (* dangerous *)
```

Pour la création d'objet, JSOO définit une fonction new_obj, ainsi que la fonction eval, évaluant une expression JavaScript et renvoyant le résultat avec le type abstrait obj.

```
1 val new_obj : obj -> obj
2 val eval : string -> obj
```

Structure des objets Les propriétés des objets sont accessibles, modifiables et supprimables par les fonctions get, set et unset. Il n'y a pas de distinction entre les attributs, et les gestionnaires d'évènements comme dans le module Node, c'est au programmeur de confectionner une valeur du type JavaScript souhaité avec les fonctions vues précédemment.

```
1 val get : string -> obj -> obj
2 val set : string -> obj -> obj -> unit
3 val unset : string -> obj -> unit
```

Appels de fonctions et méthodes JavaScript De la même façon que les accès aux propriétés, on définit l'appel de fonction JavaScript prenant en paramètre l'objet fonction, et l'appel de méthode prenant un nom de méthode et l'objet cible. Les deux prennent aussi un tableau OCaml de valeurs JavaScript, contenant les arguments.

```
1 val call_method : string -> obj array -> obj -> obj
2 val call_function : obj array -> obj -> obj
```

Réciproquement, on donne la possibilité de transformer une fonction OCaml en une fonction JavaScript, ou en une fonction lançant un thread, appropriée pour affecter comme rattrapeur évènement. La différence entre les deux a déjà été présentée dans la section sur la FFI.

```
1 (** wrap a closure into a JS function *)
2 val wrap_closure : (obj -> obj) -> obj
3 (** wrap a closure into a JS function launching it in a new thread *)
4 val wrap_event : (obj -> unit) -> obj
```

Chaînage L'objet cible de ces fonctions est toujours placé en dernier argument afin de pouvoir chaîner les appels par un combinateur >>> (`let (>>>) x f = f x`), l'idée étant d'obtenir un code proche de JavaScript, comme dans l'exemple qui suit.

```
JavaScript  :  document.getElementById("ex").tagName
OCaml       :  eval "document" >>> call_method "getElementById" [| string "ex" |]
                               >>> get "tagName" >>> as_string
```

2.7 Exemple

La figure 2.6 montre un exemple de programme OBrowser un peu moins trivial que le *Hello World* de la figure 2.2, qui illustre l'utilisation des fonctions spécifiques à l'environnement du navigateur que nous venons de présenter. Il s'agit d'un document contenant un emplacement vide, qui doit être rempli au démarrage de la page par le programme OCaml, en fonction d'un fichier texte externe.

1. Le programme charge du contenu depuis un fichier texte via la fonction http_get.
2. Une fois le contenu obtenu, le module Regexp est utilisé pour découper son contenu en lignes, et les lignes vides supprimées avec un filtrage de liste OCaml classique.
3. Le programme récupère alors le nœud destiné à recevoir le contenu dans l'arbre, qui avait été marqué d'un "basket" dans le fichier HTML, avec la fonction get_element_by_id.
4. Puis, à partir de la liste calculé plus haut, le programme construit une liste XHTML ordonnée, en utilisant les fonctions du module Html. On peut voir que le format de ce module se combine de façon concise avec les opérations sur les listes d'OCaml.
5. Enfin, le contenu mis en forme en HTML est ajouté au document avec Node.append.

```
open Js


let items =
  List.filter
    ((<>) "")
    (Array.to_list
       (Regexp.split
          (Regexp.make "\n")
          (http_get "list.txt")))

let container =
  Node.get_element_by_id Node.document "basket"

let _ =
  Node.append
    container
    (Html.ol
       (List.map
          (fun n -> Html.li [Html.string n])
          items))
```

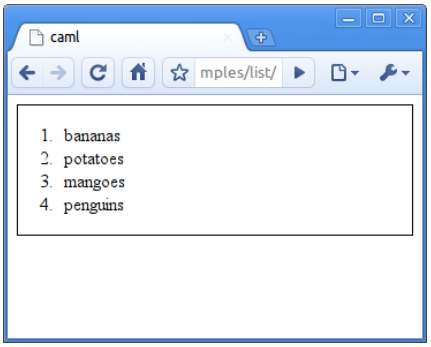
(a) Code OBrowser

```
1: ananas
2: potatoes
3: mangoes
4: penguins
```

(b) Fichier de données

```
<html>
  <head>
    <title>caml</title>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <script type="text/javascript" src="vm.js"></script>
    <script type="text/javascript">
      /* <![CDATA[ */
      window.onload = function (e) {
        exec_caml ("main.uue", "main");
      }
    /* ]]> */
    </script>
  </head>
  <body>
    <div id="basket"
         style="border :1px black solid">
    </div>
  </body>
</html>
```

(c) Fichier HTML



(d) Rendu par le navigateur

FIGURE 2.6: Affichage de listes depuis un fichier en OBrowser.

2.8 Plan de cette partie

Nous avons présenté le contexte du projet, et donné une vue d'ensemble de son architecture et de son fonctionnement. Dans la suite de cette partie, nous allons présenter plus en détail l'utilisation et l'implantation d'OBrowser, ainsi que des expériences autour de l'inter-opérabilité avec JavaScript, selon le déroulement suivant.

Au chapitre 3, nous donnerons deux exemples complets d'applications pour navigateur en OBrowser : un petit jeu spécifiquement conçu pour le navigateur, et un petit logiciel de gribouillage porté d'un exemple existant, et adapté pour le navigateur.

Au chapitre 4, nous détaillerons les points importants de l'implantation de la machine virtuelle, la représentation des données OCaml en JavaScript, et l'implantation des différents traits du langage, en particulier l'implantation du modèle préemptif.

Au chapitre 5, nous présenterons une expérience permettant l'inter-opérabilité de haut niveau entre les modèles objets d'OCaml et de JavaScript, pour utiliser des bibliothèques existantes et l'environnement du navigateur à travers les objets OCaml.

Au chapitre 6, nous donnerons les perspectives du projet OBrowser, une présentation des autres solutions et travaux connexes en les comparant avec OBrowser, puis conclurons sur cette partie.

3 Exemples d'applications en OBrowser

Dans ce chapitre, nous présentons deux exemples de programmes OBrowser. Le premier montre comment porter un code existant, un petit logiciel de gribouillage utilisant Graphics, et l'interfacer de façon minimale avec la page Web. Le second, au contraire, est un exemple spécifiquement conçu pour l'exécution dans le navigateur, un petit clone du jeu Boulder Dash.

Pour tester directement sur le web :
Gribouillage ..... http://www.pps.jussieu.fr/~canou/scribble
Boulder Dash ..... http://www.pps.jussieu.fr/~canou/bdash

3.1 Portage d'un exemple OCaml : gribouillage

Ce premier exemple est un classique des tutoriels de programmation d'interfaces : le dessin à souris levée. Concrètement, le programme original utilise une boucle infinie, en appelant à chaque tour la fonction wait_next_event du module Graphics. Lorsque l'utilisateur appuie sur un bouton, une boucle interne dessine des segments entre les positions successives, jusqu'à ce que le bouton soit relâché. L'utilisateur peut aussi influer sur le pinceau avec le clavier.

```
1  let size = ref 5 and color = ref black in
2  open_graph "500x500" ;
3  while true do
4    let e = wait_next_event [Button_down ; Key_pressed] in
5    if e.keypressed then (
6      match e.key with
7       | 'p' -> size := 1     | 'm' -> size := 5      | 'g' -> size := 10
8       | 'b' -> color := blue | 'n' -> color := black | 'j' -> color := yellow
9       | _ -> ()
10   ) else (
11     let x = e.mouse_x and y = e.mouse_y in
12     let rec draw x y =
13       let e = wait_next_event [Button_up ; Mouse_motion ] in
14       let x' = e.mouse_x and y' = e.mouse_y in
15         if e.button then (
16           set_line_width !size ; set_color !color ;
17           moveto x y ; lineto x' y' ;
18           draw x' y'
19         )
20     in draw x y
21   )
22 done
```

.

Portage simple Un tel exemple est très simple à porter de façon basique, puisqu'OBrowser implante le module d'affichage graphique d'OCaml. Le code suivant fonctionne donc dans le navigateur sans avoir à modifier le cœur du programme. En particulier, la fonction wait_next_event peut être utilisée, le mécanisme d'attente de ressources d'OBrowser ayant permis son implantation.

Il y a tout de même un petit effort d'adaptation, car la fonction open_graph ouvre normalement une fenêtre, ce qui n'a pas de sens dans un navigateur. Dans OBrowser, la fonction renvoie un élément de page de type Node.t représentant le canevas graphique, que le programmeur ajoutera où il veut. On modifie alors l'appel de cette fonction comme suit, en ayant pris soin dans le fichier XHTML de marquer l'élément de la page de devant contenir la zone de dessin de l'identifiant graphics.

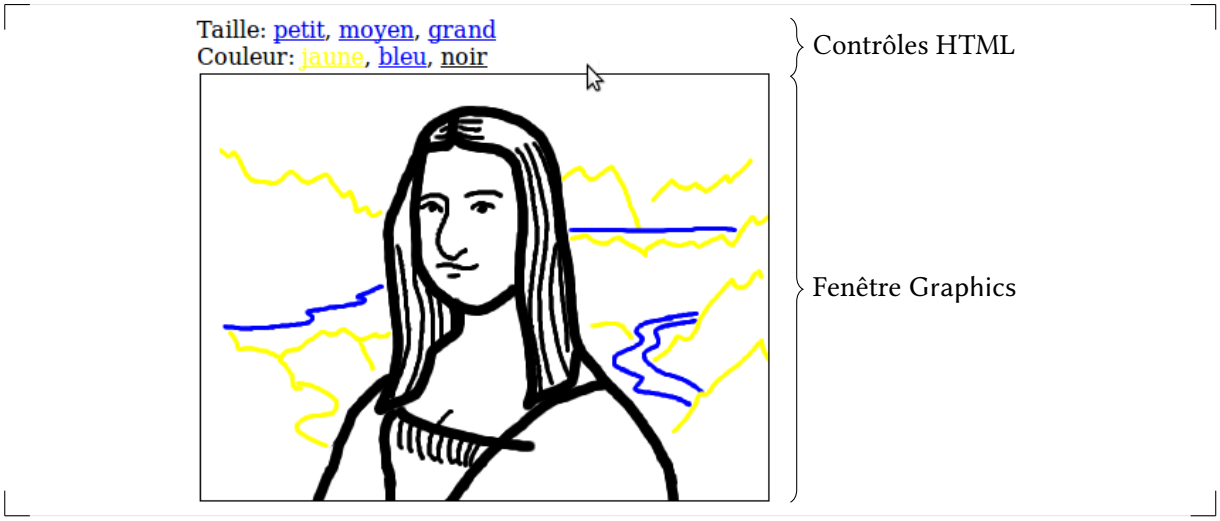


FIGURE 3.1: Gribouillage dans le navigateur.

```
Node.append (get_element_by_id "graphics") (open_graph 500 500) ;
```

Mélange des genres Lors d'un portage vers le navigateur, au lieu d'utiliser une interaction au clavier et un affichage textuel, on veut en général permettre à l'utilisateur de visualiser les résultats et d'intervenir sur le comportement du programme via des éléments de page interactifs XHTML.

Dans le code suivant, on a conservé le corps du programme de gribouillage tel-quel, mais en remplaçant les appuis de touches par des liens XHTML pour piloter le choix de la taille et de la couleur du pinceau. La figure 3.1 montre une capture d'écran de cette version intégrée au document.

```
let size = ref 5 and color = ref black in
Node.append (get_element_by_id "graphics")
  (Html.div
     [ Html.string "Taille : " ;
       Html.a  ~onclick:(fun () -> size := 1) [Html.string "petit"] ;
       Html.string ", " ;
       Html.a  ~onclick:(fun () -> size := 5) [Html.string "moyen"] ;
       Html.string ", " ;
       Html.a  ~onclick:(fun () -> size := 10) [Html.string "grand"] ;
       Html.br () ;
       Html.string "Couleur : " ;
       Html.a  ~onclick:(fun () -> color := yellow) ~style:"color :yellow" [Html.string "jaune"] ;
       Html.string ", " ;
       Html.a  ~onclick:(fun () -> color := blue) ~style:"color :blue" [Html.string "bleu"] ;
       Html.string ", " ;
       Html.a  ~onclick:(fun () -> color := black) ~style:"color :black" [Html.string "noir"] ;
       Html.br () ;
       open_graph 500 500 ]) ;
while true do
  let e = wait_next_event [Button_down] in
  let x = e.mouse_x and y = e.mouse_y in
  let rec draw x y =
    let e = wait_next_event [Button_up ; Mouse_motion ] in
    let x' = e.mouse_x and y' = e.mouse_y in
    if e.button then (
      set_line_width !size ; set_color !color ;
      moveto x y ; lineto x' y' ;
      draw x' y'
    )
  in draw x y
```

```
31 done
```

3.2 Un exemple conçu pour navigateur : Boulder Dash

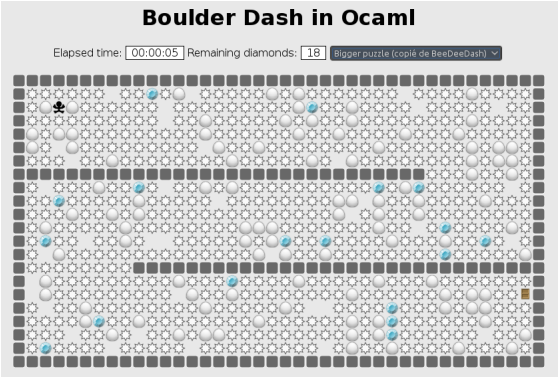
Dans cette section, nous proposons un exemple dans lequel l'affichage et l'interaction utilisent explicitement l'environnement du navigateur, au contraire du précédent dans lequel ce dernier était bien précédent, mais masqué par l'émulation du module Graphics d'OCaml.

3.2.1 Présentation du jeu

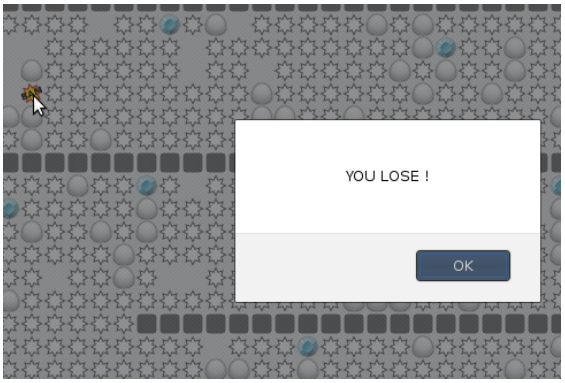
Boulder Dash est un type de jeu très simple, dont il existe de nombreuses déclinaisons, dans lequel il s'agit de collecter toutes les pierres d'un tableau pour gagner.

C'est un jeu de réflexion de type labyrinthe classique, il est divisé en problèmes indépendants à résoudre, chacun exprimé par une carte du niveau sous forme de tableau à deux dimensions, dans laquelle on déplace un petit personnage jusqu'à la sortie, en résolvant ou contournant les situations ou éléments de la carte qui posent problème.

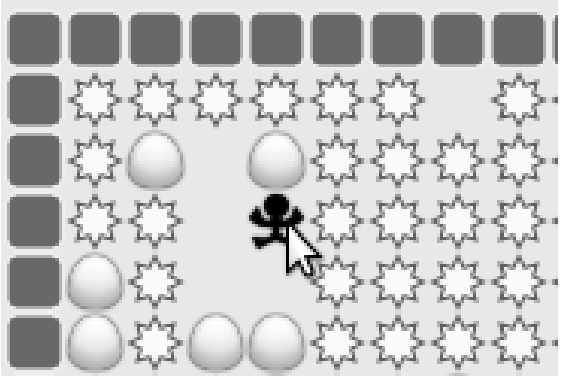
Le principe du jeu, et plus particulièrement la déclinaison que nous implantons ici, est décrit plus en détails par la série de captures d'écran commentées suivante :



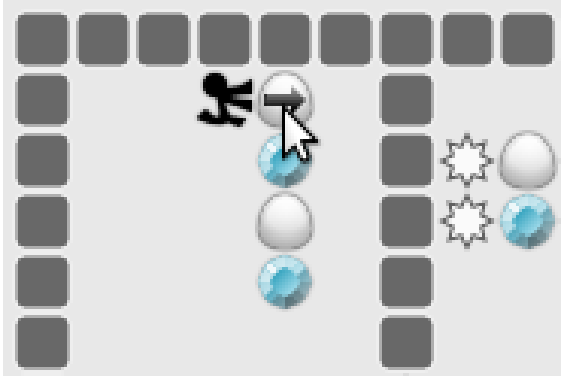
1. Le personnage se trouve à une position de départ prédéfinie, le tableau est entièrement rempli de cases sur lesquelles il peut marcher, de pierres précieuses et de rochers.



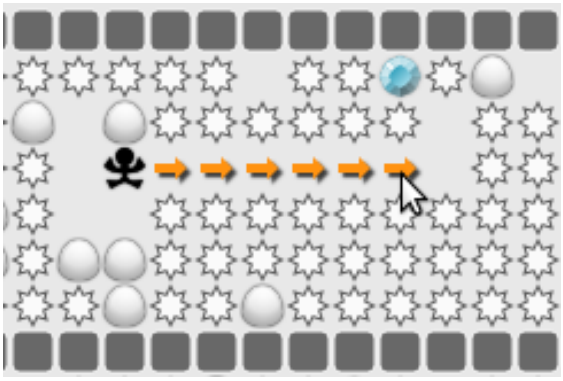
2. Lorsqu'il marche sur une case puis qu'il en sort, elle devient vide. Les pierres peuvent tomber si la case en dessous est vide. Si une pierre tombe sur le personnage, il meurt et la partie est perdue.



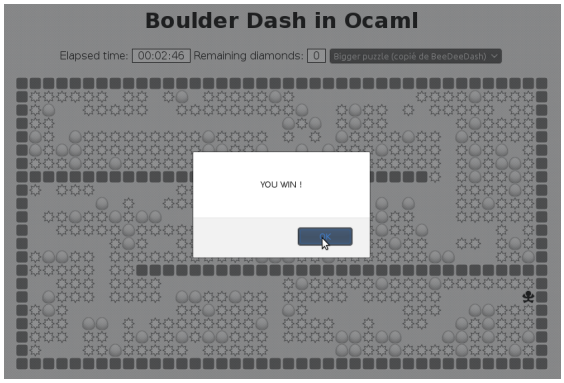
3. Le personnage peut néanmoins se glisser dessous.



4. Où les pousser.



5. Lorsqu'on passe la souris sur une case, le jeu montre alors le déplacement qui sera fait, qu'on valide en cliquant.



6. Le but est de collecter toutes les pierres précieuses, la porte de sortie est alors ouverte et le niveau est gagné.

3.2.2 Initialisation de l'interface

Une possibilité avec OBrowser est d'utiliser une page HTML statique, que l'on vient ajuster, et rendre interactive en parcourant et modifiant l'arbre du document à l'initialisation.

Dans cet exemple, on part au contraire d'une page vide, et on construit entièrement l'interface, à l'aide des fonctions du module `Html`, comme montré dans le code ci-après. Ainsi, on peut directement ajouter des rattrapeurs d'évènements sur les éléments de la page à leur construction, plutôt que d'avoir à les retrouver dans l'arbre avec des *id*.

```
(* initialize page *)
let init () =
  let board_div = Html.div [] in
  let clock_div,clock_start,clock_stop = make_clock () in
  let rem_div, rem_set = make_rem () in
  Node.set_attribute body "style"
    "font-family : sans-serif;
   text-align : center;
   background-color : #e8e8e8;" ;
  Node.append body (Html.h1 [ Html.string "Boulder Dash in Ocaml " ]) ;
  let levels = load_levels () in
  Node.append body
    (Html.div
       [Html.string "Elapsed time : " ; clock_div ;
        Html.string " Remaining diamonds : " ; rem_div ;
        Html.string " " ;
        Html.select
          (Html.option [Html.string "Choose a level"]
           :: (List.map (fun (f, n) ->
                 Html.option
                   ~onclick:(fun () ->
                     load_level f
                       board_div clock_start clock_stop rem_set)
                   [ Html.string n ])
                levels)) ;
        Html.br () ; Html.br () ;
        board_div ])
```

Affichage de l'horloge et du score La fonction suivante est un exemple typique de la construction d'éléments de page personnalisés : comme HTML n'est pas extensible, on utilise un élément racine que l'on paramètre et remplit afin de simuler un composant plus complexe, éventuellement en y insérant des sous-composants fournis par le programmeur. Les traitements disponibles sont alors retournés, en même temps que l'élément de page racine du composant, éventuellement dans un objet.

Ici, la fonction `make_box` crée un affichage de texte, avec un style particulier et initialisé à un texte par défaut. Elle renvoie l'élément lui-même et une fonction pour l'éditer. La fonction `make_rem` crée l'affichage des pierres précieuses restantes à collecter.

```
(* generic display panel *)
let make_display default =
  let div = Html.div
    ~style:"border : 1px black solid ; background-color : white ;
       display : inline ; padding-right : .5em ; padding-left : .5em ;"
    [ Html.string default ] in
  let set v = Node.replace_all div (Html.int v) in
  (div, set)

(* score display panel *)
let make_rem () =
  let div, set = make_box "--" in
  (div, (fun i -> set (sprintf "%02d" i)))
```

La fonction suivante fabrique une horloge, là encore sous forme d'un élément de page que le programmeur peut ajouter où il veut et de traitements. La fonction lance un thread qui met à jour l'affichage chaque seconde. Ce comportement est simplement obtenu par une boucle effectuant un appel à `Thread.delay` à chaque tour.

Elle retourne l'élément et deux fonctions permettant de (re)démarrer et d'arrêter l'horloge. Pour ceci, elle utilise une référence sur un booléen qui est vérifié à chaque réveil du thread d'affichage, et met à jour le temps initial lors du (re)lancement.

```
(* clock display panel and start/stop*)
let make_clock () =
  let div, set = make_box "-- :-- :--" in
  let t0 = ref 0. and running = ref false in
  let run () =
    while true do
      if !running then (
        let dt =  in
        Node.replace_all div [
          Node.text
            (let secs = int_of_float (Sys.time () -. !t0) in
             sprintf "%02d :%02d :%02d"
               (secs / 3600) ((secs / 60) mod 60) (secs mod 60))
        ]
      ) ;
      Thread.delay 1.
    done
  and start () = t0 := Sys.time () ; stopped := false
  and stop () = stopped := true in
  ignore (Thread.create update ()) ;
  (div, start, stop)
```

3.2.3 Chargement de niveau

Les niveaux de jeu ne sont pas encodés dans le programme, ils sont situés chacun dans un fichier séparé. La liste des noms de fichiers correspondant aux niveaux disponibles se trouve elle aussi dans un fichier externe, maps.txt.

Chargement de fichier Afin de matérialiser le chargement, le programme affiche un petit rectangle rouge contenant le texte "LOADING..." dans le coin en haut à droite de la fenêtre. La fonction `process_file` définie ci dessous prend en paramètre un nom de fichier et applique un traitement sur son contenu. Bien entendu, la fonction de lecture bloquante utilise le mécanisme décrit au chapitre 4, et donne seulement l'illusion d'être bloquante au niveau du programme OCaml, laissant la possibilité à d'autre threads ou à l'interface du navigateur de s'exécuter. La fonction `with_loading` gère le témoin,

elle prend une fonction en paramètre, crée un nouvel élément *div*, l'ajoute par effet de bord à la page, applique la fonction et l'enlève une fois le résultat obtenu.

```
(* display "LOADING" while performing a task *)
let with_loading task arg =
  let div = Html.div
    ~style:"background-color : red ;
      color : white ; display :inline ;
      position : absolute ;
      top :0 ; right :0;"
    [ Html.string "LOADING" ] in
  Node.append body div ;
   (* give browser control to redraw *)
  Thread.delay 0. ;
  let res = task arg in
  Node.remove body div ;
  res

(* process the content of a file *)
let process_file name process =
  with_loading process (http_get name)
```

Liste des niveaux Le fichier maps.txt dans lequel se trouve la liste des niveaux est constitué de lignes de la forme `"fichier" "titre"`. On le lit donc assez naturellement avec le module Regexp utilisant les expressions rationnelles de JavaScript.

```
(* load list of levels *)
let load_levels () =
  process_file
    "maps.txt"
    (fun txt ->
      let lines = Regexp.split (Regexp.make "\\n") txt in
      let pair_exp = "\"([^\"]+)\".*\"([^\"]+)\"" in
      let scan_pair line =
        let res = Regexp.exec (Regexp.make pair_exp) line in
        (res.(1), res.(2))
      in
      List.map scan_pair
        (List.filter
           ((<>) "")
           (Array.to_list lines)))
```

Lecture et affichage de niveau Chaque niveau est enregistré dans un fichier écrit de façon lisible par un humain, chaque ligne (resp. case) représentée par une ligne (resp. un caractère) du fichier. La lecture se fait alors simplement en chargeant le contenu textuel du fichier et en l'analysant caractère par caractère. Dans le cas de formats plus complexes, OBrowser est compatible avec ocamllex/menhir.

```
(* load level from file *)
let load_level_map file =
  process_file file (fun data ->
    let res = ref [] and row = ref [] in
    for i = 0 to String.length data - 1 do
      match data.[i] with
        | '\n' -> res := List.rev (!row) :: !res ; row := []
        | '#' -> row := Wall :: !row     | '.' -> row := Grass :: !row
        | ' ' -> row := Empty :: !row    | '+' -> row := Diamond :: !row
        | 'X' -> row := Boulder :: !row  | 'W' -> row := Guy :: !row
        | 'E' -> row := Door :: !row     | 'S' -> row := Guy :: !row
        | _ -> failwith "malformed level"
    done ;
    Array.of_list (List.map Array.of_list (List.rev !res)))
```

L'affichage se fait avec un tableau HTML dont les cases contiennent des images. Au chargement du fichier de niveau, le programme construit un état avec le type ci-dessous, qui sera mis à jour au fil des actions du joueur.

```
type cell = (* type of cells *)
  | Empty | Grass | Diamond | Boulder | Bam | Door | End | Guy | Wall
(* game state *)
and state = {
  (* data *)
  map : cell array array ;      (* level map *)
  imgs : Node.t array array ;   (* references to img elements *)
  mutable pos : int * int ;     (* current pos *)
  mutable endpos : int * int ;  (* door *)
  mutable rem : int ;           (* remaining diamonds *)
  mutable dead : bool ;         (* are you dead ? *)
  (* call backs mutex *)
  mutable cb_mutex : Mutex.t ;
  (* workaround onMouseOut event dropping *)
  mutable pending_out_cb : (unit -> unit) option ref ;
}
```

La fonction de chargement de niveau principale suivante récupère donc la carte du niveau, en déduit les positions du personnage et de la sortie, compte les diamants et construit les images afin de construire l'état initial du jeu.

```
(* update game state with level *)
let load_level file board_div clock_start clock_stop rem_set =
  process_file file
    (fun data ->
      let map = load_level_map file in
      let gx = ref 0 and gy = ref 0 and ex = ref 0 and ey = ref 0 and rem = ref 0 in
      let imgs =
        Array.mapi (fun y ->
          Array.mapi (fun x cell ->
            (match cell with
              | Guy -> gx := x ; gy := y
              | Diamond -> incr rem
              | Door -> ex := x ; ey := y
              | _ -> ()) ;
            Html.img ~src:(List.assoc cell img_assoc) ())) map
      in
      let table =
        Html.map_table
          ~style:"border-collapse :collapse; line-height : 0;"
          ~attrs:["align", "center"]
          ~td_style:"padding : 0; width : 20px; height : 20px;"
          imgs
      in
      loop
        { map = map;                    imgs = imgs ;
          pos = (!gx, !gy) ;            endpos = (!ex, !ey) ;
          dead = false ;                rem = !rem ;
          cb_mutex = Mutex.create () ;
          pending_out_cb = ref None }
        rem_set clock_stop ;
      Node.replace_all board_div table ;
      fade table 2. ;
      clock_start ())
```

modification de la carte L'état d'une case de la carte du niveau pourra être mis à jour via la fonction suivante `set_cell`, qui fait en même temps la mise à jour de l'affichage. Pour ceci, elle redéfinit l'attribut *src* de l'objet image stocké dans l'état du niveau. Cette modification est interceptée par le navigateur, et provoque le changement de l'image automatiquement.

```
let img_assoc =
  [ (Empty, "sprites/empty.png");      (Bam, "sprites/bam.png");
    (Grass, "sprites/grass.png");     (Diamond, "sprites/diamond.png");
    (Boulder, "sprites/boulder.png"); (End, "sprites/end.png");
    (Door, "sprites/door.png");       (Guy, "sprites/guy.png");
    (Wall, "sprites/wall.png")]

(* updates a cell and its associated display *)
let set_cell state x y v =
  state.map.(y).(x) <- v ;
  Node.set_attribute state.imgs.(y).(x) "src" (List.assoc v img_assoc)
```

Affichage progressif Il est assez facile, avec le mécanisme de threads, de coder des transitions visuelles. Par exemple, le niveau, une fois chargé apparaît sur 2 secondes en modifiant progressivement son opacité grâce à la fonction `fade`. Le code s'adapte à la réactivité de la machine et limite à un maximum de vingt étapes, en utilisant une boucle classique de la programmation de jeux.

```
(* fade in a page element *)
let fade elt t =
  let sty = Node.get_attribute elt "style" in
  let t0 = Sys.time () in
  while Sys.time () -. t0 <= t do
    Node.set_attribute elt "style"
      (sprintf "%s opacity :%g;" sty ((Sys.time () -. t0) /. t)) ;
    Thread.delay 0.05 (* 20FPS max *)
  done ;
  Node.set_attribute elt "style" (sty ^ " opacity :1;")
```

3.2.4 Moteur de jeu et interactions

Pour programmer un jeu de ce type dans un navigateur, il faut épouser le modèle évènementiel. En OBrowser, cela peut être fait de deux façons :

1. En masquant ce modèle depuis OCaml grâce au mécanisme de fonctions bloquantes, c'est ce que nous avons fait pour le premier exemple.
2. En utilisant explicitement le modèle évènementiel, avec une attente asynchrone, c'est ce que nous allons faire ici.

Boucle principale Le calcul des fonctions de rappel, appelé la première fois après le chargement du niveau, est le suivant :

1. On associe à chaque case du niveau une fonction de rappel sur l'évènement click.

   Cette fonction va :

   (a) Effectuer une modification adaptée pour sa position et de l'état actuel de la carte.

   (b) Recalculer toutes les fonctions de rappel de la même façon.
2. Et le programme s'arrête, en attente d'être relancé par un évènement.

Ainsi, c'est chaque action du joueur qui met en place sous forme de fonctions de rappel l'ensemble des prochaines actions possibles, et c'est de cette façon très asynchrone et évènementielle qu'est introduite la boucle de jeu.

La fonction principale du jeu est donc la fonction loop, cette fonction est récursive, mais n'est pas appelée directement récursivement, comme expliqué, elle insère ses appels récursifs dans des fonctions de rappel.

```
let rec loop state rem_set clock_stop =
  if state.pos = state.endpos then (
    clock_stop () ; clear_cbs state ; alert "YOU WIN !"
  ) else
    if state.dead then (
      clock_stop () ; clear_cbs state ; alert "YOU LOSE !"
    ) else (
      if state.rem = 0 then (
        (* open door *)
        let x,y = state.endpos in
        Node.set_attribute state.imgs.(y).(x) "src" "sprites/end.png" ;
        state.map.(y).(x) <- End
      ) ;
      clear_cbs state ;
      install_cbs state rem_set clock_stop
    )
and clear_cbs state = (* ... *)
and install_cbs state rem_set clock_stop = (* ... *)
```

Cycle d'interaction Comme nous l'avons montré dans la présentation du jeu, dans cette version de Boulder Dash, l'utilisateur peut cliquer sur n'importe quelle case à la verticale ou à l'horizontale de la position du personnage. On autorise un déplacement collectant plusieurs pierres précieuses, mais on n'autorise le déplacement que jusqu'au prochain rocher. Lorsque le joueur passe la souris sur une case vers laquelle il existe un chemin, le jeu le matérialise en remplaçant les images de ce chemin par des petites flèches appropriées. Si le joueur clique, le déplacement s'effectue, avec un déplacement animé case par case, et si une étape du déplacement provoque une chute de pierre, alors elle doit se produire, avant que le personnage ne continue son chemin.

Pour ceci, on installera trois fonctions de rappel sur chaque case ciblable :

1. Lors d'un évènement over, on attache une fonction remplaçant les cases entre le personnage et la case ciblée par des flèches.
2. Lors d'un évènement out, on restaure les images originales. Il faudra faire attention d'une part à ce que les threads liés aux évènements over et out ne s'entrelacent pas, et d'autre part au fait que le navigateur oublie parfois d'envoyer les évènements out.
3. Lors d'un évènement click, on restaure les images originales, et on effectue, case par case le déplacement, avec un petit temps entre chaque, et à chaque fois la chute de pierres.

Chutes de pierres À chaque déplacement du personnage, le calcul des chutes de pierres est effectué. Afin de rendre l'effet visuel de la chute, ce calcul est fait étape par étape, en effectuant une petite pause entre chaque. On parcourt la carte de bas en haut, et on fait tomber les pierres ne reposant sur rien d'une hauteur d'une case maximum. L'exception `Death` est lancée lorsqu'une pierre tombe sur le personnage.

```
(* 5th Symphony *)
exception Death

(* make boulders fall, and kill the guy if it must be so *)
let rec fall state =
  (* assumes wall borders *)
  let changed = ref false in
  for y = Array.length state.map - 2 downto 1 do
    for x = 1 to Array.length state.map.(y) - 2 do
      let sustaining = state.map.(y + 1).(x) = Guy && state.map.(y).(x) = Boulder in
      if (state.map.(y).(x) = Empty
         && state.map.(y - 1).(x) = Boulder) then (
        set_cell state x (y - 1) Empty ;
        set_cell state x y Boulder ;
        changed := true ) ;
      if (state.map.(y).(x) = Empty && state.map.(y - 1).(x) = Empty
          && state.map.(y).(x - 1) = Boulder && state.map.(y - 1).(x - 1) = Boulder) then (
```

```
18        set_cell state (x - 1) (y - 1) Empty ;
19        set_cell state x y Boulder ;
20        changed := true ) ;
21      if (state.map.(y).(x) = Empty && state.map.(y - 1).(x) = Empty
22          && state.map.(y).(x + 1) = Boulder && state.map.(y - 1).(x + 1) = Boulder) then (
23        set_cell state (x + 1) (y - 1) Empty ;
24        set_cell state x y Boulder ;
25        changed := true ) ;
26      if (not sustaining) && state.map.(y + 1).(x) = Guy && state.map.(y).(x) = Boulder then (
27        set_cell state x (y + 1) Bam ;
28        raise Death )
29    done
30  done ;
31  if !changed then (
32    Thread.delay 0.05 ;
33    fall state)
```

Mise en place des fonctions de rappel Cette mise en place est effectuée par le code suivant, code qui montre les possibilités offertes par OBrowser pour programmer le navigateur de plusieurs façons différentes, y compris en mélangeant les genres.

- Le mécanisme de threads et verrous est utilisé pour empêcher l'entrelacement des fonctions de rappel et complètement ignorer certains évènements, par exemple l'apparition d'un évènement de souris alors qu'on est déjà en train d'effectuer la prise en compte d'un clic.
- On utilise les primitives impératives du document pour nettoyer les fonctions de rappels précédentes, et installer les nouvelles à chaque cycle.
- Un style très fonctionnel, à la CPS (*Continuation Passing Style*), est utilisé pour la confection de ces fonctions de rappels : la fonction pour une case à distance n du personnage est fabriquée à partir de la fonction fabriquée pour la case à distance $(n-1)$, et est elle-meme utilisée pour la création de la fonction de rappel pour la distance $(n+1)$.
- Pour prendre en compte le fait que le navigateur rate certains évènements out, on utilise des combinateurs pour encapsuler les fonctions de rappel de façon à ce que si un évènement over a été traité, la fonction de rétablissement soit systématiquement appelée avant tout autre traitement si l'évènement out n'est pas encore arrivé.

```
1 (* remove all event cb *)
2 and clear_cbs state =
3   for y = 0 to Array.length state.map - 1 do
4     for x = 0 to Array.length state.map.(y) - 1 do
5       Node.clear_event state.imgs.(y).(x) "onmouseover" () ;
6       Node.clear_event state.imgs.(y).(x) "onmouseout" () ;
7       Node.clear_event state.imgs.(y).(x) "onclick" ()
8     done
9   done
10 (* install cbs for a loop step *)
11 and install_cbs state rem_set clock_stop =
12   (* is a cell walkable ? *)
13   let walkable = function
14     | Empty | Grass | Diamond | End -> true
15     | _-> false
16   in
17   (* event cb combinators *)
18   let inhibit f () =
19   (* do not execute if another cb is executing *)
20     if Mutex.try_lock state.cb_mutex then
21       (f () ; Mutex.unlock state.cb_mutex)
22   and set_pending_out f out () =
23   (* set a corresponding out cb when called *)
24     f () ; state.pending_out_cb := Some out
25   and with_pending_out f () =
26   (* call previous out cb if present *)
27     match !(state.pending_out_cb) with
```

.

```
    | None -> f ()
    | Some out -> out () ; state.pending_out_cb := None ; f ()
in
(* install move event cbs *)
let rec install_move (x, y) (dx, dy) img over_cont out_cont click_cont =
  if walkable state.map.(y).(x) then (
    let cur_img = Node.get_attribute state.imgs.(y).(x) "src" in
    let over () = Node.set_attribute state.imgs.(y).(x) "src" img ; over_cont ()
    and out () = Node.set_attribute state.imgs.(y).(x) "src" cur_img ; out_cont ()
    and click () =
      try
        click_cont () ;
        if state.map.(y).(x) = Diamond then state.rem <- state.rem - 1 ;
        Thread.delay 0.05 ;
        set_cell state (x - dx) (y - dy) Empty ;
        set_cell state x y Guy ;
        state.pos <- (x,y) ;
        fall state ;
      with Death -> state.dead <- true ;
    in
    Node.register_event state.imgs.(y).(x) "onmouseover"
      (inhibit (set_pending_out (with_pending_out over) out)) () ;
    Node.register_event state.imgs.(y).(x) "onmouseout"
      (inhibit (with_pending_out (fun () -> ()))) () ;
    Node.register_event state.imgs.(y).(x) "onclick"
      (inhibit (with_pending_out (fun () ->
        click () ; loop state rem_set clock_stop))) () ;
    if state.map.(y).(x) <> End then
      install_move (x + dx, y + dy) (dx, dy) img over out click
  )
(* install push event cbs *)
and update_push (x, y) (dx, dy) img img_guy=
  let x' = x + dx and y' = y + dy in
  let x'' = x' + dx and y'' = y' + dy in
  if (try
        state.map.(y').(x') = Boulder && state.map.(y'').(x'') = Empty
    with Invalid_argument "index out of bounds" -> false) then (
    let over () =
      Node.set_attribute state.imgs.(y).(x) "src" img_guy ;
      Node.set_attribute state.imgs.(y').(x') "src" img
    in
    let out () =
      Node.set_attribute state.imgs.(y).(x) "src" "sprites/guy.png" ;
      Node.set_attribute state.imgs.(y').(x') "src" "sprites/boulder.png"
    in
    let click () =
      set_cell state x y Empty ;
      set_cell state x' y' Guy ;
      state.pos <- (x', y') ;
      set_cell state x'' y'' Boulder ;
      (try fall state with Failure "DEAD" -> state.dead <- true) ;
      loop state rem_set clock_stop
    in
    Node.register_event state.imgs.(y').(x') "onmouseover"
      (inhibit (set_pending_out (with_pending_out over) out)) () ;
    Node.register_event state.imgs.(y').(x') "onmouseout"
      (inhibit (with_pending_out (fun () -> ()))) () ;
    Node.register_event state.imgs.(y').(x') "onclick"
      (inhibit (with_pending_out click)) () ;
  )
in
let nil_cont () = () in
let cx, cy = state.pos in
install_move (cx + 1, cy) (1,  0) "sprites/R.png" nil_cont nil_cont nil_cont ;
install_move (cx - 1, cy) (-1, 0) "sprites/L.png" nil_cont nil_cont nil_cont ;
install_move (cx, cy - 1) (0, -1) "sprites/U.png" nil_cont nil_cont nil_cont ;
install_move (cx, cy + 1) (0,  1) "sprites/D.png" nil_cont nil_cont nil_cont ;
update_push (cx, cy) (1,  0) "sprites/bR.png" "sprites/push_r.png" ;
update_push (cx, cy) (-1, 0) "sprites/bL.png" "sprites/push_l.png"
```

3.3 Conclusion sur les exemples

Au travers de ces deux exemples, nous avons deux possibilités données par OBrowser :

1. L'intégration dans le navigateur de programme existants, pratique, par exemple, pour une démonstration en ligne d'un programme écrit en OCaml. Nous avons vu qu'il était raisonnablement facile d'ajouter un peu d'interaction avec l'utilisateur via des éléments HTML, sans avoir à modifier la structure du programme.
2. La conception d'applications spécifiquement conçues pour le modèle d'exécution du navigateur. Dans ce cas, nous avons pu voir que le mécanisme de threads d'OBrowser permet d'écrire des comportements qui seraient complexes à écrire en JavaScript comme les animations.

Et bien sûr, dans les deux cas, on obtient en seulement quelques centaines de lignes des programmes intéressants, avec des interactions complexes et des animations, sans pour autant devenir difficiles à déverminer ou incompréhensibles, comme ce serait quasi obligatoirement le cas en JavaScript, à cause du modèle événementiel impératif bas niveau du langage.

4 Conception et implantation d'OBrowser

En introduction, nous avons donné une présentation générale d'OBrowser du point de vue de l'utilisateur. Nous avons présenté l'architecture générale, les traits avancés du langage implantés, et les possibilités d'interaction avec l'environnement du navigateur, via le module Js, ou via les deux types de FFI.

Dans ce chapitre, nous décrivons OBrowser du point de vue de l'implantation. Cette présentation se concentre sur les points de l'implantation spécifiquement adaptés à JavaScript et l'environnement du navigateur et significativement différents de l'implantation existante en C.

Nous commençons par décrire la représentation des valeurs OCaml, l'implantation des différents types numériques par le type Number de JavaScript, celle des types construits d'OCaml par les objets de JavaScript, ainsi que la sérialisation des données. Puis, nous décrivons comment OBrowser effectue le chargement et l'analyse d'un exécutable en code-octet OCaml, et l'initialisation de la machine. Nous serons alors à même de décrire l'interprétation du programme, le principe de base, la prise en charge des threads préemptifs, ainsi que les traits avancés de la FFI : exceptions, attentes bloquantes et gestion d'événements.

4.1 Représentation des valeurs

Cette section présente comment les différentes formes de valeurs OCaml sont encodées dans la bibliothèque standard d'OBrowser, en utilisant les objets et types primitifs de JavaScript. Nous commençons par présenter comment implanter les différents types numériques d'OCaml en utilisant ceux de JavaScript, puis nous présentons l'implantation des types construits avec les objets de JavaScript.

4.1.1 Valeurs numériques

Le langage OCaml permet de manipuler différentes sortes de valeurs numériques. Les entiers de base sur 31 ou 63 bits (suivant si l'architecture est 32 ou 64 bits, comme expliqué dans l'annexe B), mais aussi les flottants en double précision, et les entiers sur 32 et 64 bits.

JavaScript ne dispose que d'un type Number regroupant les flottants et les entiers. Comme en Scheme, l'interprète décide de la représentation la plus appropriée pour les constantes et les résultats d'opérations, entre les flottants double précision et les entiers 32 bits. On utilise donc ce même type Number comme base pour implanter les différents types numériques d'OCaml.

Entiers Comme en OCaml, à l'exécution, les entiers ne désignent pas seulement les entiers du langage, mais toutes les valeurs immédiates pouvant être représentées dans un entier : caractères, constructeurs constants, booléens, etc.

Le fait de ne pas maîtriser la représentation des nombres en JavaScript pose problème pour implanter les entiers. En effet, en OCaml, les opérations de base sur les entiers ont obligatoirement un résultat entier. Mais en JavaScript, lorsque le résultat d'une addition ou multiplication dépasse la capacité des entiers, et de même lors d'une division au reste non nul, le résultat est un flottant. On ne peut donc pas directement utiliser les opérations de JavaScript pour implanter celles de la machine virtuelle.

Heureusement, JavaScript fournit des opérations logiques bit-à-bit, n'utilisant que la partie entière sur 32 bits de leurs arguments, et renvoyant un résultat entier sur 32 bits. À chaque instruction sur les entiers pouvant effectuer un changement de représentation, on utilise donc une ruse consistant à

ajouter une opération binaire idempotente, forçant l'interprète à conserver une représentation entière sur 32 bits (Par exemple (e|0) ou (e&-1)).

Malheureusement, pour certaines opérations comme la multiplication, l'utilisation de l'opération flottante de JavaScript peut entraîner une perte d'information. Il convient donc de réécrire ces opérations, la solution classique étant de découper en mots de 16 bits les opérandes, d'utiliser les opérations flottantes sur ces sous-valeurs suffisamment petites pour que les sous-résultats soient exacts, et de recombiner ces sous-résultats.

Flottants Les flottants d'OCaml sont implantés par le type Number de JavaScript. Ils sont systématiquement encapsulés dans un bloc, afin de conserver la même représentation, et donc la même sémantique et le même format de sérialisation qu'OCaml. La racine du problème qui oblige cette encapsulation est que plusieurs primitives de la bibliothèque d'exécution exigent la capacité à faire la distinction dynamiquement entre les flottants et les entiers.

En OCaml, les entiers sont en général utilisés de façon très fréquente, le choix d'encapsuler les flottants est donc raisonnable. Pour une utilisation alternative dans laquelle l'utilisation des entiers serait anecdotique par rapport à celle des flottants, il serait possible d'encapsuler les entiers. Pour une utilisation où la conservation de la sémantique d'OCaml est moins importante que les performances, il serait aussi possible de n'encapsuler aucun des deux.

Entiers 64 bits Le langage JavaScript n'implante pas les entiers 64 bits. OBrowser fournit donc une bibliothèque les simulant, en encapsulant deux entiers 32 bits, et en implantant les opérations par composition des opérations flottantes de JavaScript sur les sous-mots de 16 bits, comme expliqué précédemment pour les opérations 32 bits problématiques.

4.1.2 Valeurs construites

On définit pour les représenter le type objet JavaScript Block donné ci-dessous. Comme en OCaml, chaque bloc a une taille et une étiquette, encodés par les champs size et tag de l'objet JavaScript. Les valeurs du bloc sont stockées dans le tableau accessible par le champ content de l'objet. La distinction entre les valeurs immédiates et les blocs est alors simplement faite avec un test de type JavaScript (b instanceof Block).

Fermetures Le choix de ne pas stocker directement le contenu dans l'objet mais d'utiliser une indirection supplémentaire a été fait principalement pour permettre de conserver la représentation exacte des fermetures mutuellement récursives d'OCaml. Dans la machine virtuelle en C, comme présenté dans l'annexe B, les différentes fonctions et l'environnement sont regroupés dans une même zone mémoire, et le corps de chaque fonction accède aux valeurs de l'environnement via des adresses relatives à sa propre adresse de base au sein de la zone.

Pour ceci, l'objet Block définit un champ offset, initialement à zéro à l'allocation d'un bloc, et une fonction shift permettant de définir un nouveau bloc, partageant le contenu du bloc originale, mais avec des indices décalés. L'accès aux valeurs contenues dans le bloc doit alors être fait via les méthodes get et set, qui calculent l'indice en prenant en compte le décalage par rapport au début du bloc original.

```
function Block(size, tag) {
    this.size = size;
    this.tag = tag;
    this.content = [];
    this.offset = 0;
}
Block.prototype.get = function (i) {
    return this.content[this.offset + i];
}
Block.prototype.set = function (i, v) {
    this.content[this.offset + i] = v;
}
Block.prototype.shift = function (o) {
    var nsize = this.size - o >= 0
                ? this.size - o : 0;
    var b = new Block (nsize, this.tag);
    b.content = this.content;
    b.offset = this.offset + o;
    return b;
}
```

Chaînes Il n'est pas possible d'utiliser directement les chaînes de JavaScript, puisqu'elles ne sont pas mutables, contrairement à celles d'OCaml. On utilise une représentation simple à base de tableau de caractères Unicode. Si le programme utilise beaucoup l'interaction avec JavaScript, il faut alors réaliser souvent des conversions entre les deux modèles de chaînes. Si besoin, il serait possible assez facilement d'améliorer les performances, nous donnerons des pistes au chapitre 6.

4.1.3 Sérialisation

Comme nous l'avons déjà dit, nous cherchons à avoir exactement le même format de sérialisation qu'OCaml, que nous décrivons dans l'annexe B. Mis à part la documentation inexistante, impliquant de lire le code original d'OCaml pour connaître l'algorithme et le format de sérialisation, l'implantation de la (dé)sérialisation en elle-même ne pose pas de difficulté particulière, en grande partie grâce au fait que la machine virtuelle OBrowser conserve une représentation largement similaire à celle de la machine d'origine. Il y a cependant plusieurs difficultés cachées.

1. Comme nous l'avons vu précédemment, il n'est pas facile de gérer précisément la représentation interne des nombres. Or le format de sérialisation utilise des données binaires de différents tailles et signes.
   Pour résoudre ces difficultés de façon à conserver un algorithme principal lisible, nous avons abstrait la notion de flux binaire via deux types objets JavaScript Reader (resp. Writer) définissant une primitive de lecture (resp. écriture) pour chaque couple taille/signe, read8s, read8u, read16s, read16u (resp. write*), etc.

2. En OCaml, les constantes de chaînes sont stockées octet par octet comme elles apparaissent dans le source. OBrowser demande alors que les chaînes données par le programmeur soient en UTF-8, et les (dé)sérialise comme telles. Ce comportement est différent d'OCaml de base, qui ne fait aucun a priori sur l'encodage, mais similaire à celui des bibliothèques d'interface graphique pour OCaml (par exemple LablGtk+).

3. Pour obtenir la représentation binaire IEEE754 d'un flottant, afin de le (dé)sérialiser, il suffit en langage C de le transtyper en entier non signé 64 bits, et d'utiliser les opérations classiques de manipulations de bits sur les entiers. Malheureusement JavaScript n'offre aucun moyen d'accéder à la représentation binaire des nombres. On utilise alors les propriétés numériques des flottants pour retrouver, à l'aide d'une série de tests et d'opérations arithmétiques flottantes sur la valeur, les différentes parties de la représentations binaire des flottants. Les fonctions float_of_bytes et bytes_of_float, qui en C prennent une ligne chacune, deviennent en JavaScript les "monstres" de la figure 4.1.

```
function float_of_bytes (bytes) {
  /* sign & exponent */
  var sign = ((bytes[0] >>> 7) == 1);
  var exponent = (((bytes[0] & 0x7F) << 4)
                  | (bytes[1] >> 4 )) - 1023;
  /* mantissa in a bool array */
  var ba = [];
  for (var b = 1;b < 8;b++)
    for (var d = 0;d < 8;d++)
      ba[(b - 1) * 8 + d - 4] =
        (((bytes[b] >> (7 - d)) & 1) == 1);
  /* proceed */
  var m = Number (1);
  for (var i = 0;i < 52;i++)
    if (ba[i])
      m += Math.pow (2, -(i + 1));
  return box_float ((sign ? (-1) : 1)
                    * m
                    * Math.pow (2, exponent));
}
```

```
function bytes_of_float (x) {
  var x = unbox_float (x);
  var e = Math.ceil (Math.log (Math.abs (x))
                     / Math.log (2));
  var m = Math.abs (x * Math.pow(2,-e)) * 2 - 1
  e += 1022;
  var bits = [];
  bits[0] = (x < 0);
  for (var i = 0;i <= 52 ; i++) {
    bits [11 + i] = (m >= 1);
    m = (m - Math.floor (m)) * 2;
  }
  for (var i = 0;i <= 10 ; i++) {
    bits [11 - i] = (((e >>> i) & 1) == 1);
  }
  var bytes = [0,0,0,0,0,0,0,0];
  for (var i = 0;i < 8 ; i++) {
    for (var j = 0;j < 8 ; j++) {
      bytes[i] = (bytes[i] * 2)
                 | (bits[8 * i + j] ? 1 : 0);
    }
  }
  return bytes;
}
```

FIGURE 4.1: Représentation IEEE754 des nombres JavaScript.

4.2 Chargement et analyse d'un fichier de code-octet

Le chargement et l'analyse du programme est souvent une partie triviale de l'implantation d'une machine virtuelle. Mais en JavaScript, et dans un navigateur, le problème est un peu plus difficile que dans un environnement classique.

Structure de la machine L'implantation de la machine virtuelle en OBrowser est bien encapsulée dans un type objet VM. Elle est réentrante, on peut charger plusieurs programmes dans plusieurs objets VM, et même les lancer en concurrence.

Le constructeur de VM prend en premier paramètre l'URL du programme à charger, télécharge et analyse le code-octet du programme, et initialise le contexte d'exécution initial de la machine en conséquence. Une fois la machine prête, on peut appeler sa méthode run pour démarrer l'interprétation du code.

Obtention du code-octet Pour accéder à des données externes depuis l'interprète JavaScript du navigateur, il faut utiliser le mécanisme XMLHttpRequest, qui permet de récupérer le contenu d'un fichier dont on connaît l'URL relative sur le serveur. Le résultat peut être un document arborescent si le fichier récupéré était au format XML, ou du texte brut. C'est ce second cas que nous utilisons.

Nous avons vu en introduction que le fichier devait être encodé après compilation. Ceci est du au fait que le code-octet est une succession de mots binaires de 32 bits, alors que certains navigateurs ne savent récupérer de façon fiable que des données texte 7 bits. Au final, la procédure pour permettre la lecture du code-octet est la suivante :

1. Côté serveur (ou de façon statique), on encode le flux d'octets en 7 bits, avec uuencode.
2. Depuis le navigateur, on récupère ce flux d'octets au format texte, avec une requête HTTP.
3. On décode ce flux avec une implantation de uudecode en JavaScript, et on groupe les octets quatre par quatre.
4. On obtient alors un tableau d'entiers 32 bits représentant le fichier programme prêt à être analysé.

Analyse de l'exécutable Une fois la difficulté d'obtenir le contenu du fichier de code-octet passée, il faut l'analyser afin d'initialiser la machine virtuelle en conséquence avant de pouvoir lancer l'interprétation. Comme expliqué dans l'annexe B, le code-octet est divisé en sections nommées.

- La section CODE n'est pas touchée, elle reste un tableau d'entiers de 32 bits et sera utilisée sans modification, via le champ code de l'objet VM. Le contexte initial de la machine est initialisé au début de ce segment de code.
- La section DATA doit être désérialisée, la valeur obtenue est stockée dans le champ data de la l'objet représentant la machine.
- La section DLLS est juste lue afin de ne pas tenter d'interpréter un programme demandant des bibliothèques externes non implantées (ex. Unix).
- La section PRIM, contenant les noms externes des primitives séparés par des octets nuls, est décodée en un tableau de chaînes syms, associant le nom de chaque primitive au numéro utilisé dans les appels de fonctions externes au sein du code-octet.

Liaison des fonctions externes Lors de l'initialisation de la machine, une fois le tableau des symboles obtenu, on constitue le tableau prims associant à chaque numéro de primitive son implantation en JavaScript.

On utilise alors le code ci-dessous pour effectuer la liaison des fonction externes. Pour chaque symbole, on cherche si une fonction JavaScript de ce nom existe à l'aide de la primitive JavaScript eval. Si ce n'est pas le cas, on affecte à cette primitive une fonction levant une exception, afin que le programme soit stoppé avec un message intelligible si la primitive est appelée au cours du programme.

```
1  function undefined_primitive (name){
2    return function () {
3      throw (new Error("undefined primitive " + name))
4    }
5  }
6  for (n in this.syms) {
7    try {
8      this.prims[n] = eval (this.syms[n]);
9    } catch (e) {
10     this.prims[n] = undefined_primitive (this.syms[n]);
11   }
12 }
```

4.3 Mécanisme d'interprétation

Dans cette section, nous présentons la façon dont OBrowser représente l'état de la machine et implante une boucle d'interprétation du code-octet en utilisant les primitives disponibles en JavaScript. En particulier, nous expliquons la prise en charge des threads préemptifs et attentes bloquantes. La figure 4.2 page 70 illustre cette section par une version (simplifiée pour la lisibilité) de l'initialisation de la machine et de la boucle d'interprétation.

Contexte d'exécution et initialisation Le contexte d'exécution d'OBrowser contient les mêmes éléments que la machine virtuelle en C, qui sont implantés comme suit.

- OBrowser utilise sa propre pile, et non celle de JavaScript. La raison principale à cela est de s'abstraire de l'appel de fonction de JavaScript, principalement car il ne définit pas d'appels terminaux, et que la profondeur d'appels récursifs est très limitée sur certains navigateurs.
  La pile est donc présente dans le contexte sous forme d'un tableau JavaScript (champ stack). Comme la taille des tableaux JavaScript est dynamiquement ajustée en fonction de leur utilisation, la taille de la pile peut grandir automatiquement.

En C, la pile est une zone pré-allouée, et le sommet de pile est simplement un pointeur au sein de cette zone. Les opérations de pile sont implantées par l'incrémentation de ce pointeur. Comme ce mécanisme n'existe pas en JavaScript, on utilise en plus de la pile un indice dans le tableau indiquant le sommet de la pile (champ sp).

- Le pointeur vers le dernier rattrapeur d'exceptions (champ trap_sp), au lieu de contenir un pointeur dans la pile contient aussi un indice relatif, comme le pointeur de pile.
- De la même façon, le contexte contient le tableau représentant le segment de code (champd code), et l'indice de l'instruction courante (champ pc), là où la machine en C ne stocke qu'un pointeur au sein du segment de code.
  Le même système est utilisé dans les fermetures, ou le pointeur de code est remplacé par un objet contenant le segment de code et l'indice dans ce segment. Le segment est sauvegardé afin de permettre d'utiliser plusieurs segments de code au sein d'une même machine, par exemple pour pouvoir prendre en charge le chargement dynamique de modules.
- Contrairement à la pile et au code, l'accumulateur, l'environnement et le compteurs d'arguments de la ZAM (*ZINC Abstract Machine (machine virtuelle d'OCaml/Caml Light)*) ne posent pas de problème particulier et sont dans les champs accu, env et extra_args du contexte.

Sélection d'interprètes d'instructions À chaque étape de l'évaluation, il faut sélectionner le code JavaScript à exécuter en fonction de l'instruction. Comme cela implique un sur-coût systématique à chaque instruction, c'est une partie qui peut gravement entacher les performances de la machine si elle est négligée. Pour l'implanter, il y a plusieurs possibilités en JavaScript.

- La structure de contrôle switch, est celle classiquement utilisée pour écrire une machine virtuelle en C. Le problème en JavaScript est que les cas ne sont pas des constantes mais des expressions, le branchement pourrait donc se faire en temps linéaire par rapport au nombre d'instructions de la machine, suivant les optimisations faites ou non par l'interprète.
- Dans le cas ou les expressions non constantes ne sont pas optimisées, un arbre d'alternatives if/else peut permettre de diminuer ce temps de façon logarithmique (pour la machine virtuelle caml disposant d'environ 150 instructions, cela revient à 7 branchements).
- La troisième possibilité est l'utilisation de la résolution de JavaScript. Concrètement, on utilise un objet, dont les noms des champs sont les codes des instructions de la machine virtuelle. Les valeurs associées sont alors des fonctions JavaScript, chacune contenant le code d'interprétation de l'instruction en question.

Pour discriminer entre ces trois solutions, nous en avons comparé les performances en pratique, sur les différents navigateurs, et la solution finalement implantée est la troisième. L'ensemble des interprètes d'instructions est donc stocké dans un objet global i_tbl, dont un extrait est donné ci-contre. Chacun prend en paramètre l'objet la machine virtuelle courante et le contexte d'interprétation. La valeur de retour des interprètes est utilisée pour indiquer si la machine doit continuer ou non.

```
1 var i_tbl = {
2   IACC0: function (vm, c) {
3     c.accu = c.stack[c.sp];
4     return true ;
5   },
6   /* ... */
7   ISTOP: function (vm, c) {
8     c.status = STOP;
9     return false ;
10  }
11 }
```

Exceptions De même que les appels de fonctions OCaml n'utilisent pas la pile de JavaScript, les exceptions d'OBrowser ne sont pas implantées en utilisant celles de JavaScript. Ce choix a été fait en cohérence avec le reste du mécanisme d'interprétation, et parce que les exceptions de JavaScript sont beaucoup trop coûteuses par rapport à celles d'OCaml (où en tout cas l'étaient avant les toutes dernières générations d'interprètes, et le restent sur certains). Dans la ZAM, et donc dans OBrowser, l'ouverture

d'un bloc try se fait en posant un rattrapeur d'exceptions sur la pile OCaml, et la levée d'exception en modifiant le contexte de la machine virtuelle comme indiqué dans le dernier rattrapeur posé, pour que la prochaine instruction soit celle du bloc with, et le niveau de pile restauré.

Mais en plus des exceptions levées en OCaml, nous avons vu en introduction que la FFI d'OBrowser permet le lancement d'exception OCaml depuis une fonction externe en JavaScript. En C, ce mécanisme est implanté avec les étiquettes dynamiques setjmp/longjmp. Pour implanter cette fonctionnalité en JavaScript, on utilise donc une exception JavaScript, seule possibilité pour briser le flot de contrôle. La méthode raise restaure le contexte d'exécution de la machine virtuelle de la même façon que l'instruction RAISE, puis lève une exception spéciale MAGIC_CAML_EX, afin d'arrêter l'appel externe. Cette exception est alors rattrapée par l'interprète de l'instruction ayant appelé la fonction externe, comme montré dans le code ci-dessous.

```
VM.prototype.raise = function (e) {
  with (this.ctx) {
    accu = e;
    if (trap_sp == -1) {
      throw new Error ("Fatal");
      this.thread_kill (pid);
    } else {
      sp = trap_sp;
      cur_code = unbox_code (stack[sp]);
      pc = 0;
      trap_sp = stack[sp + 1];
      env = stack[sp + 2];
      extra_args = stack[sp + 3];
      sp += 4;
      throw MAGIC_CAML_EX;
    }
  }
}
```

```
IJS_CALL1: function (vm, c) {
  try {
    var prim = vm.prims [c.cur_code[c.pc]];
    var r = prim.call (vm, c.accu);
    c.accu = r;
    c.pc++;
  } catch (e) {
    if (e != MAGIC_CAML_EX) throw e;
  }
  return true;
},
```

Threads Le mécanisme de base de la concurrence dans OBrowser est implanté de façon classique comme suit.

- Au lieu d'un contexte d'exécution unique, la machine virtuelle contient en réalité une liste circulaire des contextes de tous les threads en cours d'exécution, et un pointeur vers le thread courant.
- Au lieu d'une simple boucle exécutant les instructions du programme et s'arrêtant sur une instruction d'arrêt, on utilise une double boucle. La boucle externe effectue la rotation du thread courant, et s'arrête quand la liste de threads vivants est vide. La boucle interne effectue un quota d'instructions pour chaque thread, en s'arrêtant prématurément sur une éventuelle instruction d'arrêt.
- La machine fournit une primitive thread_new qui lance un nouveau thread.
- L'instruction STOP, en plus d'arrêter la boucle interne, supprime le thread courant de la liste. La machine fournit aussi une fonction thread_kill, permettant de tuer dynamiquement un thread.

Exécution incrémentale Dans le modèle d'exécution de JavaScript, aucun événement ni mise à jour de l'affichage ne peut se produire tant que du code s'exécute. Avec certains navigateurs, l'interface est même complètement bloquée tant qu'un script s'exécute sur la page. Il n'est d'ailleurs pas rare sur le Web de trouver des pages Web bloquant le navigateur quelques secondes, au chargement ou lors d'une action demandant un traitement complexe.

Ce modèle d'exécution est d'ailleurs une des principales raisons pour lesquelles la partie client des applications Web se contente d'effectuer des actions courtes, limitées à l'interaction, et délègue les calculs complexes au serveur. En effet, si le programmeur veut que la page Web reste réactive lors d'un calcul, il doit découper son programme en étapes, et rendre la main explicitement après chaque étape à la boucle d'événements de JavaScript. Or la seule façon pour rendre la main au navigateur est d'utiliser

la fonction setTimeout, qui prend en paramètre une fonction à appeler après un certain délai. Il faut alors écrire les programmes complexes en forme CPS, ou se limiter à des petits programmes (courts et ne nécessitant pas d'état).

Si on se limite au schéma décrit au paragraphe précédent, l'exécution souffre du même problème, et le programme devra être entièrement exécuté avant que l'utilisateur ne puisse agir sur la page. Mais grâce à la structure de machine virtuelle, l'état du programme est complètement encapsulé dans le contexte d'exécution de la machine, et il suffit alors d'arrêter la boucle externe régulièrement, pour la relancer à l'aide d'un setTimeout. Au final, on obtient alors non seulement un modèle de concurrence préemptive entre les threads du programme, mais aussi avec le moteur du navigateur, de façon complètement transparente pour le programmeur.

Mécanisme d'attente de ressource Le mécanisme de base présenté jusqu'ici implante la concurrence préemptive, mais n'est pas suffisant pour implanter le mécanisme d'attente de ressource présentée en introduction, qui pour rappel permet de faire passer une attente asynchrone de ressource JavaScript sous forme d'un simple appel de fonction bloquant en OCaml.

Pour implanter cette fonctionnalité, on ajoute un état au contexte de chaque thread, pouvant valoir RUN, WAIT_RES ou GOT_RES, et les trois primitives d'attente de ressource sont alors implantées comme suit.

- thread_wait(r,c), qui doit être appelée depuis un appel de fonction externe, permet de mettre en pause le thread courant, en attendant qu'une ressource r change d'état. Son implantation est en trois étapes : (1) elle passe l'état du thread à WAIT_RES, (2) enregistre la continuation du calcul à exécuter une fois le thread réveillé sous forme d'une fonction JavaScript, dans le champ continuation du contexte du thread, et (3) lève une exception JavaScript spéciale MAGIC_CAML_CONT. Cette exception permet d'une part, comme pour les exceptions, de couper brutalement l'exécution de la fonction externe JavaScript, et d'autre part de forcer un changement de thread, puisque celle-ci est rattrapée uniquement dans la boucle externe de l'interprète. Il faut noter que si le thread était le dernier actif dans la machine, cette dernière s'arrête, rendant la main au navigateur jusqu'à ce que la machine soit réveillée.
- thread_notify_one(r) permet de signaler un changement d'état de la ressource r à un des threads en attente, en réveillant ce dernier. Elle passe son état à GOT_RES. Puisqu'elle peut être appelée depuis n'importe quel contexte, et non uniquement depuis un appel de fonction externe, cette primitive relance la machine si jamais elle était arrêtée.
- thread_notify_all(r) effectue le même travail, mais pour tous les threads en attente sur r.

Lorsque la boucle externe de l'interprète passe la main à un thread à l'état GOT_RES, elle appelle la continuation précisée dans son contexte. Le résultat de la continuation est placé dans l'accumulateur de la machine, avant de passer à l'interprétation du code-octet, si bien qu'il n'y a aucune différence du point de vue du code-octet OCaml entre un appel de fonction externe normal et un tel appel externe s'effectuant en deux temps.

Éventuellement, la continuation peut elle-même devoir effectuer une attente et appeler thread_wait. dans ce cas l'état du thread revient à WAIT_RES, et l'interprétation du code-octet n'est pas reprise, il faudra pour cela attendre le thread_notify correspondant.

Primitives de threads avancées Le mécanisme d'attente de ressource est suffisant pour implanter tous les traits de haut niveau liés au modèle préemptif. La liste suivante en donne quelques exemples.

- La fonction Thread.join est simplement implantée comme une attente sur l'identifiant du thread à attendre. À la mort d'un thread, celui-ci appelle thread_notify_all avec son propre identifiant.
- Un mutex est un objet JavaScript contenant éventuellement le numéro du thread qui le détient. Lors de la réservation de mutex, il y a alors deux possibilités. Si aucun thread ne détient le mutex, dans ce cas la fonction Mutex.lock y inscrit l'identifiant du thread courant, et retourne. Dans le

cas contraire, elle appelle thread_wait en passant le mutex comme ressource. La continuation effectue le même test, ainsi le thread sera remis en pause tant que le mutex n'est pas libre, et son exécution ne reprendra que lorsqu'il aura effectivement réussi à le réserver. Quant à la fonction Mutex.unlock, elle se contente de mettre à null le détenteur du mutex si le thread courant en est bien propriétaire, et à appeler thread_notify_one sur ce dernier.

Fonctions de rappel Dans l'interface avec C d'OCaml, il est possible d'appeler des fonctions OCaml depuis C avec la fonction caml_callback. Celle-ci prend en paramètre une fermeture provenant du monde OCaml et un paramètre à lui passer, effectue l'application, et renvoie le résultat de la fonction OCaml.

Cette fonction est implantée dans OBrowser, mais avec des restrictions. En effet, puisque caml_callback doit retourner le résultat de la fonction de rappel OCaml, l'interprétation ne peut pas être mise en pause au milieu de l'exécution. L'implantation utilise donc une boucle d'interprétation plus simple, interprétant d'une seule traite tout le code de la fonction. Elle lève donc une exception si le code de la fonction appelle une primitive d'attente de ressource.

En pratique, cette implantation n'est pas aussi limitative qu'il n'y parait. En effet, l'utilisation principale des valeurs fonctionnelles en JavaScript, à part la confection de rattrapeurs d'événements, est le passage de fonctions utilisateur à des fonctionnelles provenant de bibliothèques, pour paramétrer une partie du calcul par du code personnalisé. C'est par exemple le cas dans la bibliothèque d'expressions rationnelles RegExp, où on peut paramétrer la substitution de motifs reconnus en donnant directement la fonction de substitution. Dans ce cas, il convient que ces fonctions soient courtes, pour ne pas bloquer le navigateur, ce qui est déjà le cas lors d'une utilisation de ce type en JavaScript.

Rattrapeurs d'événements Pour confectionner les rattrapeurs d'événements OCaml depuis JavaScript, il ne faut donc pas utiliser une fonction de rappel classique, puisque la gestion d'événements doit se faire en OBrowser en utilisant les threads. Une autre fonction caml_wrap_event est définie pour cela dans la FFI, qui permet d'encapsuler une fermeture OCaml dans une fonction JavaScript. Lors de l'appel du rattrapeur d'événement JavaScript, celui-ci lancera un thread dans la machine virtuelle, qui exécutera la fermeture OCaml, à l'aide de la fonction thread_new. Il est alors possible que la machine soit complètement arrêtée lorsque l'événement se produit. Dans ce cas, La fonction thread_new, si elle détecte que la machine était à l'arrêt, relance son exécution en appelant sa méthode run, de façon similaire à la notification de ressource.

C'est aussi cette fonction qui est utilisée en interne par la fonction OCaml Node.register_event, si bien qu' on a donc un fonctionnement similaire pour les rattrapeurs d'événements, que ceux-ci soient mis en place depuis OCaml ou depuis JavaScript avec la FFI.

4.4 Conclusion

Nous avons présenté dans ce chapitre l'implantation d'OBrowser, en insistant sur les traits avancés liés au modèle de concurrence préemptive. Clairement, cette implantation n'est pas la plus optimisée (nous verrons des perspectives d'amélioration possibles au chapitre 6), mais le but de construire une plate-forme d'expérimentation de programmation Web client polyvalente est bien atteint. En effet, le point le plus important à retenir de ce chapitre, ainsi que des exemples du chapitre précédent, est qu'il est possible de s'abstraire du modèle classique de programmation du navigateur, au niveau du langage de base comme de la concurrence, et ce en permettant d'utiliser l'environnement du navigateur et les bibliothèques tierces existantes. Au chapitre suivant, nous continuerons sur cette voie, en montrant qu'il est possible d'interagir avec l'environnement du navigateur en utilisant un système de types statiques et un modèle objet différent de celui de JavaScript.

```
function VM(url, argv) {
  var program = load_program (url);
  this.syms = /* ... */
  this.prims = /* ... */
  this.data = /* ... */
  this.status = VM_WAITING;
  this.max_pid = 1;
  this.ctx = {
    cur_code : program.code,
    pc : 0,
    sp : 0,
    trap_sp : -1,
    accu : UNIT,
    stack : [],
    env : ATOM,
    extra_args : 0,
    status : RUN,
    pid : this.max_pid++
  };
  this.ctx.n_ctx = this.ctx ;
  this.ctx.p_ctx = this.ctx ;
}
```

```
VM.prototype.run = function () {
  running_vm = this;
  if (this.status != VM_RUNNING) {
    this.status = VM_RUNNING;
    var vm = this;
    function sched_run () {
      var t1 = (new Date ()).getTime ();
      for (var i = 0;i < TIMEOUT_INTERVAL;i++) {
        for (var j = 0;j < RESCHED_INTERVAL;j++) {
          if (vm.ctx == null) break;
          var c = vm.ctx;
          if (c.status == GOT_RES) {
            try {
              c.accu = c.iocontinuation ();
              c.status = RUN;
              continue;
            } catch (e) {
              if (e == MAGIC_CAML_CONT) break;
              if (e != MAGIC_CAML_EX) throw e;
              break;
            }
          } else {
            if (c.status != RUN)
              /* SLEEP, WAIT & WAIT_RES */
              break;
          }
          if (! i_tbl [c.cur_code.get (c.pc++)] (vm, c)) {
            break;
          }
        }
        var t2 = (new Date ()).getTime ();
        if (!vm.thread_yield ()) {
          vm.status = VM_WAITING;
          break;
        }
        if (t2 - t1 > TIMEOUT_STOP) {
          t1 = t2;
          break;
        }
      }
      if (vm.status != VM_WAITING)
        window.setTimeout (sched_run, 0);
    }
    sched_run ();
  }
  running_vm = null;
}
```

FIGURE 4.2: Contexte d'exécution et boucles d'interprétation.

5 Inter-opérabilité des modèles objet

Au chapitre 2, nous avons présenté deux façons de faire interagir OCaml et JavaScript dans OBrowser au travers des deux FFI. Les deux approches ont en commun de se situer à un niveau très bas, celui de la représentation des données, et d'être effectuées par le programmeur. Dans cette section, nous présentons une interface de haut niveau entre les objet de JavaScript et OCaml, automatisée et fondée sur les types.

Plusieurs travaux [🕮8] [🕮7] ont déjà été menés afin de faire cohabiter les modèles objets d'OCaml et de langages à classes plus classiques. Or, de nombreuses bibliothèques JavaScript utilisent un modèle objet à classes à la Java implanté au dessus du modèle de JavaScript. Nous avons donc naturellement voulu essayer d'appliquer les résultats de ces travaux pour utiliser ces bibliothèques OCaml. Plus particulièrement, cette expérience a commencé lorsque nous avons essayé d'écrire un binding de la bibliothèque JavaScript Google Closure [⚓72].

La technique qui en résulte, que nous présentons ici, permet effectivement de binder une bibliothèque utilisant un modèle à classes. Mais elle va plus loin, en permettant d'exploiter le typage canard de JavaScript d'une part, et le typage des objets OCaml d'autre part.

5.1 Rappels sur les modèles objets de JavaScript et OCaml

Si une grande partie des travaux présentés ici peuvent s'appliquer au modèle à classes classique, certains points requièrent du lecteur une connaissance des spécificités des modèles objet d'OCaml et JavaScript. Voici donc quelques rappels essentiels sur ces modèles pour le lecteur non spécialiste de l'un ou l'autre des langages.

Objets en JavaScript Le modèle objet de JavaScript est un modèle à *prototypes* et est (d'après son créateur [⚓91]) repris du langage Self [🕮32]. Concrètement, les objets sont des tables d'association extensibles entre des noms et des valeurs (les valeurs fonctionnelles jouant le rôle de méthodes). Chaque objet est de plus lié à un autre objet que l'on appelle son *prototype*. Lors de la résolution d'un nom (appel d'une méthode ou accès à un champ), si l'objet lui-même ne définit pas le nom recherché, alors la recherche est effectuée dans son prototype (et récursivement, les prototypes pouvant être chaînés).

En JavaScript, le programmeur peut définir de nouveaux types objets et modifier les prototypes associés à ces types. L'association entre un objet et son prototype est faite à sa création, et dépend de son type objet. Les objets d'un même type partagent le même prototype, un peu comme les variables de classes des modèles objets à classes. Une présentation pratique avec un exemple des objets JavaScript est donnée dans l'annexe A.

Objets en OCaml Le langage OCaml n'est pas un langage *à objets*, mais un langage fonctionnel et impératif à la ML *avec une extension objet*. Ainsi, le modèle à objets d'OCaml [🕮49] cherche à donner les mêmes possibilités d'expression que les langages à objets (classes, héritage multiple, méthodes binaires, etc.) tout en restant dans l'esprit du polymorphisme paramétrique.

Contrairement aux langages à classes, le sous-typage n'est pas lié à l'héritage mais utilise la structure des types. Concrètement, les types objets ont la forme de types enregistrements extensibles : le type d'un objet est l'ensemble de ses méthodes (et les types associés). Un objet est sous-type d'un autre si l'ensemble de méthodes du premier est inclus dans celui du second, et que les types associés aux méthodes du premier sont bien des sous-types de ceux des méthodes du second.

Contrairement aux langages à classes, il est alors possible en OCaml de définir deux classes dont les objets peuvent être interchangés sans problème, sans pour autant avoir à utiliser une relation d'héritage entre les deux. Le lecteur pourra se référer à [📖59] pour une introduction aux objets en OCaml.

Modèle à classes simulé Comme le présentent les créateurs de Self, langage dont le modèle objet de JavaScript est issu, dans [📖51], le modèle à prototypes de JavaScript est suffisamment expressif pour simuler les mécanismes du modèle à classes. En pratique, beaucoup de bibliothèques JavaScript (ex. GWT [⚓80], Prototype [⚓76]) simulent spécifiquement le modèle de Java/.Net, soit parce qu'il est mieux connu des programmeurs, soit pour homogénéiser avec du code serveur écrit dans ces langages. Cette simulation se fait en définissant sous forme de fonctions JavaScript des équivalents aux mots-clefs objet de Java (extends, instanceof, implements, super, etc.).

Typage canard (*duck typing*) Comme l'exhibent les auteurs de Typed-Scheme [📖20], les programmeurs adeptes des langages dynamiques se restreignent en général à un sous-langage qui pourrait être accepté par un système de types statique. Dans ce cadre, le modèle objet à prototypes est très souple sur le système de types implicite utilisé par le programmeur. Si, comme nous venons de le voir, les programmeurs JavaScript issus du monde à classes ont souvent tendance à penser dans ce dernier, le modèle le plus souvent utilisé par les programmeurs du monde dynamique est le typage canard.

Selon le typage canard, *un animal qui a des ailes, un bec et qui fait "coin coin" est un canard.* En d'autres termes : si, dans un contexte donné, un objet a toutes les caractéristiques qu'on lui demande, alors il est du bon type. En JavaScript cela se traduit par : si, pour un code donné, un objet définit tous les champs accédés et méthodes appelées par ce code, alors cet objet est du bon type. On peut étendre la définition en disant que le contenu des champs doit aussi correspondre à la structure demandée par le code (et récursivement). En fait, on peut dire que les types objets d'OCaml sont une forme statique de typage canard.

5.2 Présentation du système

Pour le programmeur, l'utilisation de ce système de bindings automatisés s'effectue en deux étapes : (1) Il définit une interface pour chaque classe JavaScript qu'il veut utiliser en OCaml. (2) Il lance alors la génération automatique, et obtient une classe OCaml pour chaque interface.

Description d'interfaces Nous avons choisi d'intégrer l'IDL (*Interface Description Language*) dans le code source OCaml, en reprenant un sous-ensemble de la syntaxe des types objet. L'idée est que le programmeur n'ait pas à apprendre un langage supplémentaire, l'interface étant de toutes façons très proche du type objet OCaml résultant. On utilise pour cela le pré-processeur CamlP4 pour étendre d'une syntaxe external class la règle *definition* de la grammaire d'OCaml donnée dans le manuel [📖57] (dont nous reprenons les notations).

```
        definition   +=   ext-class-definition
ext-class-definition   ::=  external class ext-class-binding { and ext-class-binding }
  ext-class-binding   ::=  class-name : [ ext-class-type -> ]* object [ext-method]* end = string-litteral
        ext-method   ::=  method method-name : ext-class-arrow-type { = string-litteral }
    ext-class-type   ::=  int | float | bool | unit | string
                      |   ( ext-class-arrow-type )
                      |   ext-class-type array
                      |   class-name
ext-class-arrow-type   ::=  ext-class-type { -> ext-class-arrow-type }
```

Pour chaque classe interfacée, le programmeur donne :
- Le nom qui désignera la classe depuis OCaml, et le nom de la classe en JavaScript, de façon similaire aux définitions de fonction externes OCaml.
- Les types des paramètres attendus par le constructeur JavaScript, avec la même syntaxe fléchée que les types de classes OCaml prenant des paramètres.
- La description de chaque méthode, de façon similaire celle des types de classes OCaml, et éventuellement le nom de la méthode en JavaScript s'il est différent avec la même syntaxe que pour le nom de classe.

Les types utilisables dans ces définitions d'interfaces (en termes plus concrets : les types des valeurs qui sont autorisées à transiter entre le monde JavaScript et le monde OCaml) sont un sous-ensemble des types dont la conversion est supportée : types de base, tableaux, objets déjà interfacés et fonctions.

Utilisation Une fois les interfaces écrites le programme est expansé par le pré-processeur vers des définitions d'objets OCaml, utilisant des fonctions externes JavaScript définies dans un fichier JavaScript auxiliaire, lui aussi produit par le pré-processeur. Le programmeur peut alors de façon transparente en utilisant la couche objet d'OCaml :
- Construire des objets JavaScript avec le mécanisme d'objets d'OCaml, et les utiliser comme des objets OCaml. Les objets ainsi construits restant directement utilisables depuis JavaScript.
- Appeler des méthodes sur ces objets avec l'appel de méthode d'OCaml. Les paramètres qu'il passe sont des valeurs OCaml, des types de base ou d'autres types objets définis dans l'interface, qui seront automatiquement converties en valeurs JavaScript correspondantes.
- Récupérer via les valeurs de retour de ces appels de méthodes des valeurs et objets JavaScript, qui sont automatiquement convertis en valeurs et objets OCaml.
- Étendre les classes OCaml générées par l'interface. Dans ce cas, la liaison tardive (*late-binding*) d'OCaml est utilisée de façon transparente depuis un appel JavaScript. En d'autres termes, lors d'un appel de méthode en JavaScript sur l'objet, si celle-ci a été redéfinie dans la classe dérivée en OCaml ayant construit l'objet, alors c'est le code OCaml qui est utilisé, les paramètres et valeurs de retour étant convertis automatiquement.

De plus, le mécanisme est suffisamment souple pour permettre :
- À la partie JavaScript de renvoyer des objets dont la classe n'est pas exactement celle interfacée, mais qui comportent les mêmes méthodes avec les mêmes types. Ce point permet d'une part d'embrasser le typage canard de JavaScript, et d'autre part de supporter l'héritage côté JavaScript dans un modèle à classe simulé. Dans ce dernier cas, un objet de classe C' dérivée en JavaScript depuis une classe C interfacée sera vu comme un objet de type C en OCaml, et les méthodes appelées depuis OCaml seront bien celles éventuellement redéfinies par C'.
- À la partie OCaml de passer des objets qui ont exactement le type d'une classe interfacée, mais n'en dérivent pas. Ceux-ci seront convertis en objets JavaScript dont les méthodes appelleront les méthodes OCaml. Ce point permet d'embrasser le modèle de typage structurel des objets OCaml, en éliminant les erreurs à l'exécution dans le cas où un objet du bon type, mais pas de la bonne hiérarchie est transmis à JavaScript.

5.3 Mise en œuvre

Le système repose sur les mécanismes fondamentaux suivants :
- La conversion de types simples, structurés et fonctionnels d'un monde à l'autre.
- L'appel de méthode JavaScript depuis OCaml, et réciproquement.
- La conversion d'objets interfacés d'un monde à l'autre.
- La résolution de méthode à travers les langages, en particulier en cas de liaison tardive.

Le premier point est assez facile et systématique en utilisant la FFI d'OBrowser déjà présentée. Pour le deuxième point, nous avons vu que le module JSOO définit l'appel de méthode JavaScript en OCaml, et pour la réciproque, la FFI définit la primitive callback_method(o,n,a). C'est en fait dans les deux derniers points, qui sont intimement liés, que réside toute l'intelligence du système.

Conversion d'objets L'idée générale est de considérer chaque objet interfacé comme un couple (objet JavaScript, objet OCaml), et non comme un seul objet d'un des deux mondes. La conversion est alors triviale : il suffit de prendre l'autre élément du couple. En pratique, on utilise un champ dans chacun des objets référençant sa contrepartie. La résolution et l'appel de méthode quand à eux sont le fruit d'un dialogue entre les deux parties, l'objet OCaml pourra déléguer dans certains cas à l'objet JavaScript et réciproquement.

On distingue trois cas possibles pour un objet candidat à transiter entre les deux mondes répondant à une interface de classe C, qui correspondant à trois façons de construire un tel objet, et induisent trois méthodes d'appel différentes.

1. L'objet a été créé en JavaScript, via le constructeur original de C, ou tout autre moyen qui fabrique un objet dont le type est compatible avec C.

   Résolution : L'objet OCaml lié va se contenter de déléguer toutes ses méthodes.

   Conversion : L'objet OCaml lié n'existe pas à la création. Le couple est formé à la demande, lorsque l'objet JavaScript est remonté dans le monde OCaml.

2. L'objet a été créé en OCaml, via le constructeur OCaml de C généré (ou celui d'une sous-classe), qui a à son tour appelé le constructeur JavaScript de C.

   Résolution : La délégation peut se produire dans les deux sens : par défaut d'OCaml vers JavaScript, et le contraire en cas de redéfinition de méthode en OCaml.

   Conversion : Ici, le couple est formé à la création.

3. L'objet a été créé en OCaml, sans passer par le constructeur de C. Il a le même type, donc peut être transmis comme un objet de type C.

   Résolution : C'est le cas inverse du premier, toutes les méthodes de l'objet JavaScript associé sont déléguées à l'objet OCaml.

   Conversion : L'objet JavaScript lié n'existe pas à la création. Le couple est formé à la demande, lorsque l'objet OCaml descend dans le monde JavaScript.

Médiateurs JavaScript Dans le schéma de génération, nous avons fait le choix de laisser la majeure partie du travail à JavaScript, à savoir les conversions de types et la résolution de l'appel de méthodes inter-langages.

On va pour cela faire intervenir dans le couple un troisième objet JavaScript que l'on appellera *médiateur*. Trois classes de médiateurs sont générés pour une même classe C, pour chacun des trois cas. Chacun prend en charge les appels de méthodes inter-langages en effectuant les délégations spécifiques au cas qu'il traite, ainsi que les conversions de types associées. Les rôles de ces trois classes PROXY, EXTEND et REVERSE seront décrits en détails dans la suite.

Le médiateur va devoir savoir recevoir les appels de méthodes depuis JavaScript et depuis OCaml. Pour différencier la provenance, on utilise une astuce de nommage : pour chaque méthode m de C, le médiateur définit :

1. Une méthode m pour les appels en provenance de JavaScript. Les paramètres et valeurs de retour sont des valeurs JavaScript.
2. Une méthode m_{ML} pour les appels en provenance d'OCaml. Les paramètres et valeurs de retour sont des valeurs OCaml.

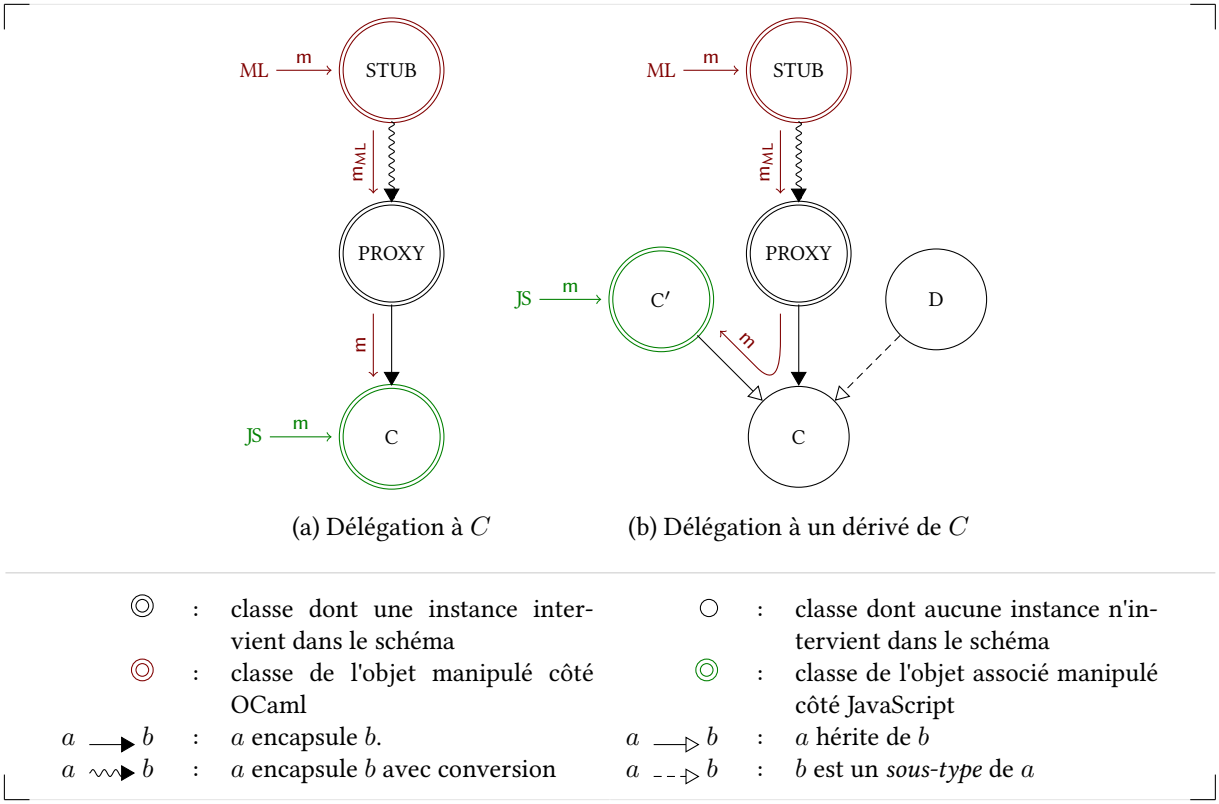
Relais OCaml Pour implanter la délégation d'OCaml à JavaScript, on génère une classe relais STUB par classe C, qui ne fait que déléguer les appels à chaque méthode m à la méthode m_{ML} correspondante du médiateur de son couple. L'appel utilise les primitives de JSOO, et passe et reçoit directement des valeurs OCaml. Comme expliqué, c'est le médiateur qui va convertir ces valeurs avant d'effectuer la délégation à la méthode JavaScript, et la valeur de retour avant de la restituer au STUB.

Médiateur PROXY Ce médiateur est utilisé pour lier un objet JavaScript indépendant. Un tel objet peut, par exemple, avoir été créé directement par le programmeur en JavaScript, ou bien encore lors de l'exécution d'une méthode JavaScript appelée depuis OCaml.

Dans le schéma d'interaction, on a donc un objet JavaScript indépendant, un STUB OCaml, et un objet médiateur PROXY indépendant, qui transmet les appels d'OCaml à l'objet existant, en effectuant les conversions.

La figure 5.1 décrit dans son diagramme (a) le cheminement d'un appel de la méthode m, depuis OCaml et depuis JavaScript. Le second diagramme (b) montre que ce médiateur permet par construction l'utilisation du typage canard (et l'héritage simulé) côté JavaScript, de façon transparente du point de vue OCaml. La sémantique des éléments graphiques est donnée en dessous des diagrammes, et sera réutilisée dans les figures suivantes.

Le STUB et le PROXY sont tous les deux créés à la demande, lors du passage d'une telle valeur du monde JavaScript au monde OCaml.



(a) Délégation à C (b) Délégation à un dérivé de C

FIGURE 5.1: Appels via l'adaptateur PROXY

Médiateur EXTEND En plus du STUB de C, on génère coté OCaml une classe C_{ML} qui en dérive, c'est elle qui permet de construire des objets C depuis OCaml, et de dériver des sous-classes de C en OCaml. Le médiateur EXTEND intervient dans ces objets construits par C_{ML} ou une de ses sous classes.

Concrètement C_{ML} hérite de toutes les méthodes de STUB sans les redéfinir, elle délègue donc systématiquement les appels au médiateur. Elle redéfinit par contre l'initialisation de la classe : elle prend les mêmes paramètres que C, construit son médiateur EXTEND associé (via un appel externe), qui a son tour convertit ces paramètres, et les utilise pour construire l'objet C associé.

Comme avec le médiateur PROXY, les appels en provenance d'OCaml sont systématiquement redirigés vers les méthodes de C, ce qui nous donne la délégation dans un sens.

L'autre sens de délégation est un peu plus difficile. D'une part, il faut que le médiateur sache si OCaml redéfinit ou non chaque méthode, pour choisir s'il doit passer la main à la définition originale de C ou à une redéfinition en OCaml. D'autre part, il faut que les appels depuis JavaScript dans le corps des méthodes de C soient automatiquement redirigés vers le médiateur pour qu'ils bénéricient de la redéfinition en OCaml.

La solution est de remplacer la relation de délégation utilisée pour le proxy par une relation d'héritage : EXTEND dérive de la classe C, l'objet JavaScript et son médiateur sons alors confondus. Les appels à l'intérieur de l'objet bénéficient donc directement de la médiation par construction.

Pour permettre la délégation de JavaScript à OCaml, la solution est la suivante :

- chaque méthode m_{ML} du médiateur-objet appelle la méthode originale m de la classe C,
- chaque méthode m du médiateur-objet appelle la méthode m de l'objet ML.

Ainsi, comme on peut le voir dans la figure 5.2, un appel de JavaScript à une méthode m redéfinie en OCaml appelle directement cette méthode, permettant la délégation JavaScript vers OCaml. D'autre part, si l'objet OCaml ne redéfinit pas la méthode m, alors c'est la méthode m du STUB qui est appelée, qui a son tour appelera m_{ML} du médiateur-objet, aboutissant de façon correcte à appeler la méthode m originale de C.

Concrètement, au niveau JavaScript, l'héritage est implanté par une copie de prototype, et l'utilisation du constructeur original de C, ainsi que les appels aux fonctions originales lors de la délégation depuis OCaml sont implantés grâce à la possibilité offerte par JavaScript de choisir le receveur d'un appel de méthode, via la primitive apply.

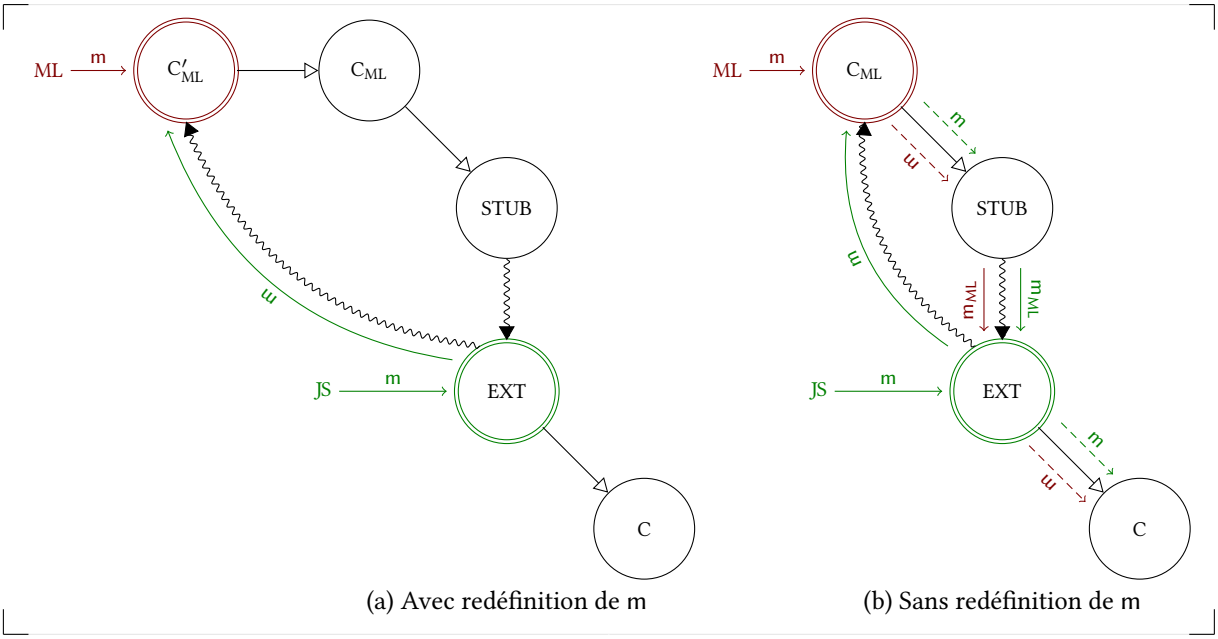
Une limitation importante du modèle tel que présenté est l'impossibilité d'utiliser l'héritage multiple sur plusieurs objets interfacés. Une erreur dynamique (qui pourrait, bien que plus difficilement, être statique) est soulevée dans un tel cas. Précisons cependant que le problème a peut être une solution, nous ne l'avons simplement pas encore investigué.

Médiateur REVERSE Ce médiateur est utilisé pour lier un objet OCaml existant, d'un type compatible avec STUB, mais n'en héritant pas, comme c'est possible avec le système de types d'OCaml. Cet objet n'est donc pas lié à sa création avec sa moitié JavaScript ou son médiateur. Comme pour le cas précédent, on le confond avec son médiateur, puisque l'objet JavaScript n'a pas de comportement propre, et son rôle est uniquement de de déléguer systématiquement ses appels et de convertir les données. Comme pour le PROXY, ce médiateur-objet est créé à la demande, lors du passage du monde OCaml au monde JavaScript.

La figure 5.3 récapitule les relations entre objets et les appels dans ce schéma de couple. La relation importante est celle entre REVERSE et C. Pour qu'un tel schéma fonctionne, il faut absolument qu'elle soit respectée, et que REVERSE soit exactement substituable à C dans le monde JavaScript, ce qui revient à dire que l'interface donnée par le programmeur décrive exactement l'ensemble des méthodes de C. La précision est importante, car pour les cas EXTEND et PROXY, il suffit que le programmeur donne un sous ensemble correct des méthodes de C pour que le système fonctionne.

5.4 Exemple

Le programmeur souhaite utiliser en OCaml les classes JavaScript Point et PointStore suivantes, car il en est très fier, et qu'il désire s'épargner un effort de réécriture en OCaml.



(a) Avec redéfinition de m (b) Sans redéfinition de m

FIGURE 5.2: Appels via l'adaptateur EXTEND

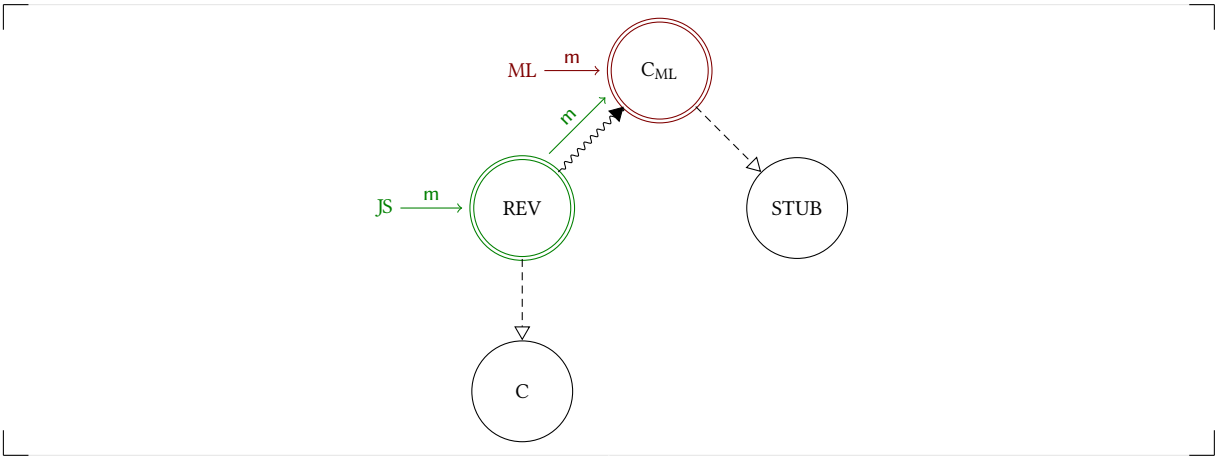


FIGURE 5.3: Appels via l'adaptateur REVERSE

```
1  function Point(x,y) {
2    /* setup private field _x */
3    this._x = x;
4    /* setup private field _y */
5    this._y = y;
6  }
7
8  /* private method _log */
9  Point.prototype._log = function (s) {
10   if (console)
11     console.debug ("Point :" + s);
12 }
13
14 /* public method toString */
15 Point.prototype.toString = function () {
16   return "(" + this.getX() + ","
17              + this.getY() + ")";
18 }
19
20 /* setter/getter for private field _x */
21 Point.prototype.getX = function () {
22   return this._x;
23 }
24 Point.prototype.setX = function (x) {
25    this._log ("field _x set");
26    this._x = x;
27 }
28
29 /* setter/getter for private field _y */
30 Point.prototype.getY = function () {
31   return this._y;
32 }
33 Point.prototype.setY = function (y) {
34   this._log ("field _y set");
35   this._y = y;
36 }
37
38 /* public Point cloning and identity */
39 Point.prototype.dupl = function () {
40   this._log ("object duplicated");
41   return new Point (this.getX (), this.getY());
42 }
43 Point.prototype.myself = function () {
44   return this;
45 }
```

```
function PointStore() {
  /* setup private collection _vec */
  this._vec = [];
}

/* public point storing methods */
PointStore.prototype.fill = function (n) {
  for (var i = 0;i < n;i++)
    this.push (new Point(Math.random (),
                         Math.random ()));
}
```

```
/* public insertion at the end */
PointStore.prototype.push = function (p) {
  this._vec.push(p);
}

/* public collection iterator*/
PointStore.prototype.iter = function (f) {
  for (var i = 0;i<this._vec.length;i++) {
    f(this._vec[i]);
  }
}
```

Il définit alors l'interface de ces classes avec l'IDL à l'intérieur de son code OCaml. Il utilise les noms point et point_store comme les classes ne peuvent commencer par une majuscule en OCaml, et il donne un aspect plus OCaml à ses classes en séparant des mots plutôt que d'utiliser la casse chameau (*Camel-Case*). Il choisit de ne pas faire apparaître la méthode privée, car il est sûr qu'elle n'est effectivement pas utilisée en dehors de l'objet, et qu'il ne brisera donc pas le schéma REVERSE s'il définit ses propres points en OCaml sans hériter de point.

```
external class point
  : float -> float ->
    object
       method to_string : string = "toString"
       method set_x : float -> unit = "setX"
       method get_x : float = "getX"
       method set_y : float -> unit = "setY"
       method get_y : float = "getY"
       method dupl : unit -> point
       method myself : unit -> point
     end
  = "Point" ;;
```

```
external class point_store
  : object
      method push : point -> unit
      method iter : (point -> unit) -> unit
      method fill : int -> unit
    end
  = "PointStore" ;;
```

Le programmeur peut alors utiliser toute la puissance d'OCaml pour définir une nouvelle classe scaled_point héritant de point, et constater que tout se passe bien, en particulier la méthode toString utilise bien la redéfinition en OCaml, et il peut stocker cet objet dans un point_store sans problème. De même, le programmeur arrive à stocker dans le même point_store un objet immédiat OCaml.

```
class scaled_point scale x y =
object
  inherit point x y as mom
  method get_x = mom#get_x *. scale
  method get_y = mom#get_y *. scale
end ;;

(* EXTEND test *)
let s = new point_store ;;
s#push (new point 2. 2.) ;;
s#push (new scaled_point 3. 2. 2.) ;;

(* PROXY test *)
s#fill 2 ;;

(* REVERSE test *)
let o = object (self)
  method to_string =
    "ML ( "
    ^ string_of_float self#get_x
    ^ " x "
    ^ string_of_float self#get_y
    ^ ")"
  method get_x = 2.
  method set_x = failwith "set_x"
  method get_y = 2.
  method set_y = failwith "set_y"
  method dupl () = Oo.copy self
  method myself () = self
end ;;
s#push o ;;
s#push (o#dupl ());;
s#push (o#myself ()) ;;

(* display to check *)
let cpt = ref 0 ;;
s#iter (fun p ->
  incr cpt ;
  print_string
    (sprintf "[%d] %s\n" !cpt p#to_string))
;;
```

Le résultat de l'exécution est donné ci-dessous, où on peut constater le fonctionnement du médiateur EXTEND grâce aux sorties [1] et [2], du médiateur PROXY grâce à [3] et [4] résultant d'objets créés indépendamment en JavaScript par la fonction fill, et du médiateur REVERSE grâce à [5], [6] et [7] où le "ML" montre que la méthode des objets immédiats du programme OCaml est bien appelée par la

méthode JavaScript iter de la classe

```
[1] (2,2)                  [5] ML ( 2. x 2. )
[2] (6,6)                  [6] ML ( 2. x 2. )
[3] (0.4995,0.3448)        [7] ML ( 2. x 2. )
[4] (0.1867,0.4262)
```

5.5 Extensions

Le système présenté jusqu'ici est correct dans le cadre du binding d'une bibliothèque bien conçue et dont les objets encapsulent correctement leurs données. Ce n'est clairement pas le cas de l'API JavaScript d'accès à l'environnement du navigateur, en particulier au document. Cette section propose des améliorations possibles (et effectivement implantées, mais nous donnons aussi des variations possibles) pour améliorer le système afin de pouvoir accéder à cet environnement, ainsi que quelques optimisations.

Optimisations Parmi les optimisations possibles, les deux plus importantes sont les suivantes :

1. La suppression de la boucle dans la résolution d'un appel JavaScript d'une méthode non redéfinie en OCaml avec le médiateur EXTEND. Cette optimisation est importante, car non seulement on effectue des appels inutiles, mais en plus ces appels sont assortis d'une double conversion des paramètres et valeurs de retour complètement inutile. Grâce à la représentation des objets OCaml, il est cependant facile de regarder la table des méthodes de l'objet OCaml, et de ne pas effectuer la boucle si la méthode à appeler est identique à celle du STUB.
2. La résolution des méthodes entre les deux mondes dans la technique présentée se fait en utilisant directement le nom des méthodes sous forme de chaînes de caractères. C'est d'autant plus problématique que ces chaînes sont converties à chaque fois. cependant, en réalité, l'implantation des objets en OBrowser utilise déjà un cache de conversions et de hachage des identifiants de méthodes, qui rend déjà optimisée la résolution dans le sens JavaScript vers OCaml. Pour le sens inverse, l'optimisation est triviale : il suffit de remplacer les noms m_{ML} par des identifiants uniques entiers.

Attributs Dans le DOM, l'accès à une large majorité des données n'est pas encapsulées derrière des méthodes d'accès. Le programmeur doit directement lire et modifier les attributs des objets JavaScript.

Une solution pour permettre d'accéder à ces attributs serait d'utiliser les variables d'instance des objets OCaml. D'une part, cette solution serait difficile car il n'est pas du tout évident d'intercepter leurs affectations pour les déléguer à JavaScript dans le cas d'un objet PROXY ou EXTEND, d'autre part car les variables d'instance sont privées en OCaml, or le but avoué ici est d'accéder à des données depuis l'extérieur des objets.

La solution la plus simple est donc de transposer en méthodes les accès et modifications, en précisant dans l'interface de la méthode OCaml qu'il s'agit d'un accès ou d'une modification, plutôt que d'un appel de méthode. On peut par exemple étendre la syntaxe comme suit :

```
ext-method  +=  method method-name : ext-class-type = get string-litteral
              | method method-name : ext-class-type -> unit = set string-litteral
```

Il y a cependant deux limites notables à cette méthode :

1. la redéfinition d'un getter ne sera pas prise en compte si l'attribut est lu depuis JavaScript,
2. le médiateur REVERSE ne peut pas être utilisé, pour la même raison.

La morale est qu'il vaut mieux bien convevoir (ou choisir) ses bibliothèques JavaScript, en encapsulant correctement les données, et éviter la redéfinition si ce n'est pas le cas.

```
external class element : string -> object
  method tag : string = getter "tagName"
  method children : element array = getter "children"
  method append : element -> unit = "appendChild"
  (* ... *)
end = fun "document.createElement" ;;

external class type document = object
  method createElement : string -> node
  method createTextNode : string -> node
  method body : element = getter "body"
  method get_by_id : string -> element = "getElementById"
  (* ... *)
end ;;

let window = external object "window" : object
  method document : document = getter "document"
  method change_page : string -> unit = setter "location"
  (* ... *)
end ;;
```

FIGURE 5.4: Interface au DOM via les objets OCaml

Extension aux objets immédiats JavaScript De nombreux objets JavaScript dans l'environnement du navigateur sont des objets uniques ou dont le constructeur ne peut être appelé par le programmeur et doivent être créés avec une fonction spécifique.

Il y a deux niveaux possibles pour intégrer au système ces objets :

1. Il serait possible facilement d'ajouter une syntaxe external class type, ne définissant pas de constructeur, mais permettant d'utiliser un objet de ce type comme paramètre ou valeur de retour d'une méthode d'un autre objet. Il serait possible de la même façon d'ajouter une syntaxe pour récupérer l'objet résultat de l'évaluation d'une expression JavaScript sous forme d'objet PROXY.

```
                definition  +=  ext-class-type-definition
ext-class-type-definition   ::=  external class type ext-class-type-binding { and ext-class-type-binding }
   ext-class-type-binding   ::=  class-name = object [ext-method ]*

                      expr  +=  external object string-litteral : ext-object-type
          ext-object-type   ::=  class-name
                            |  object [ext-method]* end
```

2. Pour les objets dont la fabrication est faite via une fonction prédéfinie, et non par un constructeur d'objet. On aimerait pouvoir simplement spécifier qu'une classe OCaml ne correspondant pas à une classe JavaScript, mais à une fonction de construction, par exemple avec la syntaxe suivante :

```
ext-class-binding += class-name : [ ext-class-type -> ]* object [ext-method]* end = fun string-litteral
```

   Le problème est que si on n'a pas accès à la classe JavaScript permettant de créer l'objet, il est impossible d'en dériver une classe d'objet-médiateurs EXTEND, et donc impossible de redéfinir des méthodes en OCaml. Une solution possible est d'ajouter un quatrième type de médiateur PATCH, qui modifie directement les méthodes de l'objet créé (et non celles de sa classe) afin qu'elles se comportent de la même manière qu'un EXTEND. Une telle solution permettrait notamment de dériver des éléments de page avec des fonctions de rappels d'événement spécialisées simplement par héritage avec redéfinition de méthode en OCaml.

Exemple Le code OCaml de la figure 5.4 montre comment le mécanisme, enrichi des extensions que nous venons de proposer permet d'interfacer le DOM pour l'utiliser à travers les objets OCaml.

5.6 Conclusion

Ce chapitre a décrit un système pour l'inter-opérabilité de haut niveau entre les objets d'OCaml et ceux de JavaScript, basée sur la génération automatique de code d'inter-opération à partir d'une description d'interface.

La technique présentée permet de prendre en charge les bibliothèques JavaScript tierces structurées en classes, celles utilisant le typage canard ainsi que les objets prédéfinis du navigateur, et ce en utilisant les objets d'OCaml, en conservant la liaison tardive et le typage structurel.

Travail futur : évaluation en pratique Le projet Ocsigen utilisant maintenant le compilateur js_of_ocaml à la place d'OBrowser, le binding de la bibliothèque Google Closure évoqué en introduction a finalement été réalisé avec l'interface avec JavaScript plus simple de js_of_ocaml, présentée au chapitre suivant, qui ne prend en particulier pas en charge la liaison tardive et les conversions automatiques. Il serait intéressant de reprendre l'expérience, pour voir en pratique l'intérêt des traits avancés disponibles avec cette technique.

Travail futur : typage de la grammaire D'autre part, nous avons vu que la technique autorise l'utilisation des objets du navigateur, en particulier du DOM, en forçant l'utilisation correcte de ces objets grâce au typage statique d'OCaml. Mais comme nous l'avons déjà expliqué en introduction et au chapitre 2, l'API du DOM est trop générique pour imposer le respect de la grammaire du document. Les types des objets OCaml reflétant exactement cette API, le typage statique ne peut donc malheureusement pas assurer la correction de l'utilisation des objets vis-à-vis de la grammaire. Il serait alors intéressant d'étudier la possibilité de mixer cette interface avec les techniques de manipulation bien typée du document développées dans la partie II, nous présenterons des pistes pour cela dans la conclusion de cette thèse.

6 Conclusion, travaux connexes et perspectives

Dans ce chapitre, nous faisons le point sur cette partie sur la programmation des navigateurs avec OBrowser, et abordons les possibilités de travaux futurs.

Comme nous l'avons expliqué au cours de cette partie, le développement d'OBrowser a été centré sur la fiabilité, la compatibilité et l'expressivité plus que sur les performances, pour les raisons respectives d'obtenir une plate-forme d'expérimentation rapidement, permettant la programmation client/serveur, et ouverte à toutes sortes d'expérimentations. Mais les navigateurs ont évolué depuis, et si on réussit à dompter les interprètes JavaScript modernes, il est possible d'exécuter de vrais programmes dans un navigateur. Malheureusement, comme ce sera détaillé section 6.1.2, ces améliorations sont peu susceptibles d'améliorer les performances d'OBrowser de façon optimale à cause de son architecture.

Nous commençons donc ce chapitre par présenter les possibilités d'évolution d'OBrowser, à titre de plate-forme pédagogique et de support de programmation client. En particulier, nous présentons des solutions pour améliorer les performances, sans pour autant sacrifier les capacités d'expressivité et de compatibilité avec la version standard d'OCaml, qui constituent la force et l'originalité d'OBrowser par rapport aux autres solutions se pliant au modèle du navigateur.

Dans une seconde partie, nous donnerons les autre projets permettant d'exécuter du code OCaml dans un interprète JavaScript et donnons leurs caractéristiques, notamment le sous-ensemble d'OCaml qu'ils prennent en charge, les éventuelles extensions qu'ils fournissent, et leurs performances.

Finalement, nous conclurons sur ce chapitre ainsi que toute cette partie I.

6.1 Directions possibles pour OBrowser

Il y a trois pistes principales, non mutuellement exclusives, pour le futur d'OBrowser. Une première piste que nous allons présenter est de détourner en partie le but actuel de la machine pour la transformer en plate-forme pédagogique. Les deux autres pistes concernent le cas où on veut conserver OBrowser comme plate-forme d'exécution de code OCaml dans les navigateurs, la section 6.1.2 présente les améliorations possibles de l'implantation actuelle, la section 6.1.3 présente la possibilité de transformer OBrowser en greffon et les difficultés que cette approche implique.

6.1.1 Plate-forme pédagogique

La version actuelle d'OBrowser est particulièrement proche de la spécification de la machine virtuelle OCaml, et pourrait être utilisée dans le cadre d'enseignements sur la compilation et les machines virtuelles.

En particulier lors de la mise au point d'OBrowser, une interface d'exécution pas-à-pas avait été mise en place (cf. figure 2.3), permettant d'observer les structures de la machine, d'examiner à la demande les structures des valeurs qu'elles contiennent, et de surveiller les entrées/sorties. Tout cela est possible grâce à la facilité d'écrire de petites interfaces interactives dans le navigateur.

Là encore, d'un point de vue pédagogique, il pourrait être intéressant d'ajouter des interfaces d'observations, par exemple pour surveiller l'exécution de programmes multi-threads, tracer les utilisations des verrous, observer ou intervenir sur l'algorithme d'ordonnancement, etc.

Il serait intéressant aussi de porter d'autres machines virtuelles comme la JVM ou la CLR (*Common Language Runtime (machine virtuelle .Net)*), et dans l'idéal l'écriture d'un cadre permettant de définir des machines virtuelles dans un navigateur et de les observer.

6.1.2 Optimisations d'OBrowser en JavaScript

Dans cette section, nous allons commencer par expliquer les avancées des interprètes JavaScript des navigateurs, en expliquant pourquoi OBrowser dans son état actuel ne peut pas profiter des plus importantes. Nous donnerons ensuite plusieurs pistes pour optimiser la machine virtuelle à ces nouveaux interprètes.

Un point sur les performances des navigateurs Comme nous l'avons déjà évoqué en introduction, ainsi que dans l'historique de JavaScript de l'annexe A, ces dernières années, le monde des navigateurs a énormément évolué, et en particulier, les performances des interprètes JavaScript.

Au sein des interprètes, ces améliorations se sont faites sur tout le langage.

1. Sur les performances brutes des interprètes, la principale optimisation est la spécialisation JIT en utilisation des traces d'exécution d'un interprète plus classique [📖15]. L'idée est de repérer dans un premier temps les chemins d'exécution les plus souvent pris, de vérifier que le type des valeurs dans ces chemins ne varient pas, et de générer un code optimisé et monomorphe. En pratique, les programmeurs JavaScript utilisant raisonnablement le polymorphisme, les gains sont impressionnants, et dans les comparatifs, JavaScript rivalise avec OCaml (code-octet) ou Java.

   Malheureusement, cette optimisation ne fonctionne pas pour OBrowser : le polymorphisme de JavaScript est la base de la représentation uniforme des données utilisée dans OBrowser, le flot de contrôle est complètement masqué par la structure de machine virtuelle, dont les interprètes d'instructions sont pour la plupart intrinsèquement polymorphes.
2. Les composants de la bibliothèque standard ont aussi été optimisés par les implantations de JavaScript. Les comparatifs ont pendant longtemps été composés de tests mesurant indépendamment ces composants. Ainsi, les interprètes bénéficient tous de composants de base (chaînes, nombres, expressions rationnelles, etc.) très optimisés.

   Ce n'était pas le cas lorsque l'expérience OBrowser a débuté, et même si certains interprètes avaient déjà des composants optimisés, il valait mieux utiliser des représentations personnalisées afin que les performances ne s'effondrent pas sur les autres interprètes.

Une amélioration très importante à noter aussi est l'uniformisation des performances, qui n'oblige plus à ruser et utiliser du code peu optimal pour qu'il fonctionne partout. C'est principalement vrai depuis que Microsoft a repris le développement d'Internet Explorer et l'abandon de sa version 6.

Compilation JIT Pour la distribution standard d'OCaml l'expérience de la compilation juste-à-temps de code-octet OCaml a déjà été tentée avec OCamlJIT [📖30] (puis plus récemment OCamlJIT 2 [📖58]). Le schéma de compilation était très simple : pour un segment de code-octet donné, ocamljit mettait bout-à-bout le code assembleur exécutant chaque instruction, avec très peu voir pas d'optimisations au niveau de la séquence. Un tel schéma donnait pourtant un gain non négligeable en performance.

Dans l'interprète JavaScript, une telle technique permettrait de ne pas partager le code des interprètes d'instructions entre deux codes OCaml ne travaillant pas sur les mêmes types de valeurs, et pourrait donc permettre l'optimisation par traces, menant très probablement à un gain de performances, pour une difficulté d'implantation pas trop élevée.

Avec un peu plus de travail, mais sans changer fondamentalement le mécanisme, il serait possible de compacter et rendre plus efficace certaines séquences courtes et très fréquentes. Par exemple, le JIT pourrait optimiser les appels externes, en particulier celles de threads, ou celles du module JSOO pour alléger le coût de l'interface OCaml/JavaScript (et par conséquent de l'interface objet du chapitre 5).

Un avantage important d'une forme aussi simple de JIT n'utilisant pas les structures de contrôle de flot et de données de JavaScript, est qu'elle permettrait de conserver simplement l'exécution incrémentale et les threads, soit en découpant le code avec une granularité assez fine, soit en permettant d'interrompre un segment par une rupture de contrôle JavaScript, quitte à repasser en interprétation ou à relancer une passe de JIT à partir de ce point de rupture, lors de la reprise de l'exécution.

Pré-traitement du code-octet L'établissement de la machine virtuelle, en particulier le chargement et le décodage du code-octet et la dé-sérialisation des globales, sont des tâches qui prennent du temps au démarrage de la page Web, et ce de façon systématique. Il serait possible assez facilement d'analyser le fichier de code-octet à l'avance, et le projeter vers une structure JavaScript pré-calculée, afin d'accéler le chargement, et de profiter d'un éventuel cache de l'interprète JavaScript.

Un autre problème du fichier de code-octet est que l'absence d'élimination du code mort du compilateur ocamlc produit des exécutables beaucoup plus gros que nécessaire. Concrètement, si le programmeur utilise une seule fonction du module List de la bibliothèque standard, alors l'intégralité de ce module est inclus [1]. Le coût est de plus répercuté à l'exécution, où l'on paye le temps d'initialisation de ces fonctions inutilisées. Une optimisation intéressante serait de supprimer cette forme de code mort détectable de façon raisonnablement simple du code-octet,

La suite d'outils OCAPIC [📖33], qui implante une machine virtuelle OCaml pour microcontrôleurs PIC, effectue ces pré-traitements, et le gain en taille et en performances est vraiment intéressant. D'autre part, l'outil ocamlclean, inclus permet de produire un exécutable en code-octet OCaml standard après la phase d'élimination de code mort, et pourrait être réutilisé tel-quel. Il effectue aussi une petite analyse de flot de données au sein des fonctions qui supprime certaines créations de blocs inutiles introduites par ocamlc, qui peut aussi s'avérer intéressante.

De même, les techniques décrites plus haut pour le JIT pourraient être adaptées en compilation AOT (*Ahead Of Time*), et les segments identifiables compilés à l'avance.

Représentation des valeurs Au niveau de la bibliothèque, l'implantation des valeurs pourrait être largement améliorée pour utiliser les primitives optimisées de JavaScript. Nous avons évoqué au cours de cette partie l'utilité d'une optimisation des conversions de chaînes. Les chaînes de JavaScript n'étant pas mutables, les conversions restent indispensables, mais une possibilité est d'utiliser un objet intermédiaire effectuant les conversions à la demande, avec si besoin une invalidation en cas de mutation. js_of_ocaml utilise une représentation optimisée des chaînes, qui pourrait être reprise sans difficulté.

Une autre optimisation serait de ne plus utiliser une représentation personnalisée des blocs, mais les tableaux optimisés de JavaScript. Le travail serait un peu plus important que pour les chaînes car il supprimerait la possibilité de simuler l'arithmétique des pointeurs, et il faudrait choisir des représentations différentes pour certains types de blocs, s'éloignant ainsi de la représentation des valeurs d'OCaml.

6.1.3 OBrowser dans un greffon

Il s'agit là d'embarquer la version standard d'OCaml dans un greffon. C'est un sujet sur lequel nous avons commencé à expérimenter, principalement pour en cerner les problématiques, mais qui n'a pas encore abouti.

Choix d'architecture Les greffons Java et .Net pour navigateurs, utilisent tous une technique de machine virtuelle téléchargeant classiquement du code octet. Mais il y a d'autres choix possibles pour implanter un greffon de langage. Pour OBrowser, voici plusieurs choix possibles permettant de réutiliser l'implantation d'OCaml existante .

1. La situation est encore pire si on utilise une bibliothèque utilisant l'option -pack d'OCaml, auquel cas tous les modules de la bibliothèque sont inclus systématiquement.

1. Embarquer la machine virtuelle OCamlocamlrun est la solution la moins difficile. Même si la compilation vers JavaScript donne des résultats qui concurrencent ocamlrun (ce sera présenté à la section suivante), il reste des traits (par exemple les exceptions), qui restent plus performants. D'autre part, c'est probablement la solution la plus susceptible d'aboutir, qui permettrait de démontrer que JavaScript n'est pas la solution ultime pour programmer les navigateurs.
2. Un problème posé par cette solution est que la machine virtuelle d'OCaml, pour des raisons de performances, n'effectue aucune vérification dynamique, et ne fonctionne donc correctement qu'avec du code-octet issu d'un programme bien typé. Concrètement, un programme incorrect peut provoquer une corruption ou un arrêt brutal de la machine. Dans un greffon, cette fonctionnalité se transforme en faille de sécurité, qu'il est possible de combler de deux façons principales, soit (1) en ajoutant des tests dynamiques, soit (2) en écrivant un vérificateur de code-octet, comme le fait la JVM.
3. Embarquer des programmes pré-compilés par ocamlopt donnerait effectivement des performances bien meilleures que JavaScript. Un problème est que l'API des greffons lance les greffons dans le même espace mémoire, et il n'est pas possible de lancer plusieurs exécutables dans un même programme, mais l'expérience OCaml for MultiCore (OC4MC) [📖55] cherche à y remédier, et alternativement on peut les lancer dans des threads.
4. La solution précédente pose des problèmes de sécurité évidents. Embarquer ocamlopt serait alors une possibilité, permettant de compiler les sources sur le client, assurant le bon comportement des programmes par le typage dans une bibliothèque standard spécialisée. Cette solution pourrait aussi être utiliser comme troisième alternative dans le cas de l'embarquement de la machine virtuelle.

Implantation Il y a deux principales difficultés dans l'implantation d'un tel greffon, causés par le fait que l'architecture de l'API des greffons (NPAPI/npruntime) n'a pas été conçue au départ pour des interactions complexes avec le document, mais plutôt pour embarquer un programme indépendant dans la fenêtre du navigateur, avec une interaction limitée avec la page :

1. L'API récente npruntime permet d'utiliser en C les objets JavaScript, et impose un modèle mémoire à comptage de références, dans lequel le greffon doit enregistrer un objet tant qu'il l'utilise, et le relâcher ensuite. Pour ne pas générer de fuites mémoire, il faut faire cohabiter ce modèle avec celui d'OCaml.

   Il y a alors plusieurs difficultés :
   - Le code OCaml pouvant manipuler les éléments de la page, il est possible qu'il construise des cycles et casse le comptage de références.
   - Plus grave, le programmeur peut mettre des valeurs OCaml dans les champs des objets JavaScript, les rendant invisibles pour le ramasse-miettes d'OCaml, à moins d'utiliser les racines globales, ce qui reviendrait à un comptage de références externes et impliquerait encore des fuites.
   - La réciproque du problème précédent est aussi problématique : les objets OCaml stockés dans les champs des objets JavaScript étant opaques pour le ramasse-miettes de JavaScript.

   Un solution possible, mais non triviale, serait de faire croire à JavaScript que tous les éléments de page sont vivants, et modifier le parcours des racines du ramasse-miettes d'OCaml pour qu'il parcoure les objets JavaScript, et qu'il déclare les objets morts comme tels auprès de JavaScript pour qu'ils soient nettoyés au prochain tour de son ramasse-miettes. La solution est valable dans le cas où le seul programme s'exécutant est le programme OCaml, le cas d'autres scripts ou de bibliothèques JavaScript externes reste un problème ouvert.
2. L'API des greffons est intrinsèquement événementielle et asynchrone, et les traitements effectués dans les fonctions de rappels associées en C sont exécutés dans le même espace mémoire et temporel que le code JavaScript sur la page. En particulier, si une telle fonction dure trop longtemps,

elle bloque l'interaction. Pour les greffons classiques n'interagissant pas avec la page, ce n'est pas un problème, il suffit de lancer le code dans un thread, voire un processus séparé. Dans le cas d'OBrowser, il faudrait soit modifier la machine virtuelle ou le schéma de compilation pour autoriser une exécution incrémentale comme dans l'implantation JavaScript, soit utiliser des threads et du passage de messages, mais cela risquerait de ternir les performances d'un programme effectuant beaucoup de manipulations de document.

6.2 Travaux connexes

Cette section décrit les deux alternatives pour exécuter un programme OCaml que sont OcamlJS, le compilateur OCaml modifié pour ciblé JavaScript de Jake Donham [⚓97] et js_of_ocaml, le compilateur de code-octet OCaml vers JavaScript [🕮53] de Jérôme Vouillon.

6.2.1 Compilation AOT du code-octet : js_of_ocaml

Le compilateur js_of_ocaml prend en entrée un exécutable en code-octet OCaml, et produit un programme JavaScript. Cette technique est assimilable à la compilation AOT des machines virtuelles modernes, ou aux techniques plus classiques de compilation reciblante, dans la mesure ou il s'agit de changer de plate-forme d'exécution un binaire déjà compilé, et de dé-compilation puisque cette nouvelle cible est un langage de haut niveau.

Schéma de compilation La première étape de js_of_ocaml est la dé-compilation du code-octet pour en reconstruire le graphe de contrôle. Puis, le compilateur effectue plusieurs passes, pour rendre le graphe compilable avec les structures de contrôle disponibles en JavaScript, et d'optimisation. Il optimise les appels terminaux sur la fonction elle-même en les remplaçant par une boucle dans le graphe de contrôle, la récursion terminale générale n'est par contre pas gérée. Une phase d'analyse de flot de données est aussi effectuée, pour supprimer les créations de blocs locaux inutiles, assez fréquemment laissées, voire introduites, par le compilateur ocamlc. js_of_ocaml effectue aussi une passe d'*inlining* de fonctions et d'élimination de code mort, qui ne sont pas faites par ocamlc.

La dernière étape est la projection vers du code JavaScript, dans laquelle js_of_ocaml utilise les mécanismes de branchement et d'appels de JavaScript pour transposer ceux du langage intermédiaire, et utilise les mécanismes de portée statiques de JavaScript pour compiler l'environnement des fermetures OCaml. Les exceptions sont projetées vers des exceptions JavaScript. Ces dernières ne sont pas très efficaces, si bien que l'utilisation d'un style de programmation intrinsèquement par exceptions n'est pas recommandé, mais celles-ci restent raisonnables pour ne pas avoir à être évitées complètement pour autant.

Les résultats, présentés dans [🕮53] sont très bons, et, à part pour certains cas (comme les programmes très fonctionnels, ceux utilisant la récursion non terminale générale, les chaînes mutables à outrance ou les exceptions), les temps de js_of_ocaml tournent autour de ceux de la machine virtuelle standard ocamlrun, en général même un peu mieux. Ces temps sont toutefois à relativiser, puisque certaines optimisations importantes manquantes à ocamlc sont effectuées par js_of_ocaml et ne profitent donc pas à ocamlrun.

Valeurs js_of_ocaml projette toutes les valeurs numériques vers le type générique Number de JavaScript, mais il utilise les mêmes ruses qu'OBrowser pour conserver la représentation interne entière lors des opérations entières. Il implante de plus les entiers 64 bits en les simulant avec un type objet encapsulant trois entiers 32 bits. Les blocs sont représentés par des tableaux JavaScript, dont la première valeur est l'étiquette.

Modèles objet js_of_ocaml définit deux modèles objets. Tout d'abord, il implante les objets d'OCaml, avec une représentation similaire à celle d'OCaml, avec des fonctions JavaScript implantant la résolution de méthodes. De façon orthogonale, il permet d'utiliser les objets de JavaScript de façon typée, en utilisant le système de types des objets d'OCaml via des types fantômes. Le programmeur utilise pour cela une extension de syntaxe ajoutant un opérateur d'appel de méthode ##, ainsi que des syntaxes pour les accesseurs, qui sont projetés vers des mécanismes bas niveau, à la façon de JSOO dans OBrowser. Ces appels bas-niveau sont spécifiquement reconnus et optimisés par le compilateur pour optimiser les performances. Ce second mécanisme objet permet d'utiliser de façon bien typée les objets de JavaScript, mais ne permet pas d'en définir ou d'en étendre en OCaml, et les deux modèles objets ne sont pas compatibles entre eux.

Modèle de concurrence js_of_ocaml conserve le modèle événementiel et asynchrone du navigateur, et préconise l'utilisation de la bibliothèque de threads coopératifs Lwt, pour laquelle il fournit des primitives permettant de rendre la main au navigateur sleep et yield. Même si elle impose un modèle de concurrence proche de celui du navigateur, le caractère fonctionnel d'OCaml, et la possibilité d'extension de syntaxe d'Lwt, rend plus agréable la programmation de ce type de concurrence qu'en JavaScript.

Aspect client/serveur Pour les communications, comme pour OBrowser, l'idée dans Ocsigen est de pouvoir utiliser la sérialisation. js_of_ocaml implante donc le même format de sérialisation qu'OCaml, légèrement limité à cause du fait qu'il est impossible de faire la différence entre les entiers et les flottants dans la représentation des valeurs de js_of_ocaml.

6.2.2 Compilation d'OCaml : OcamlJS

Le compilateur OcamlJS est conçu comme un *back-end* au compilateur OCaml, se greffant sur le *lambda-code* (le langage intermédiaire comportant encore les structures fonctionnelles, mais dont les types et les structures de haut niveau ont déjà disparu).

Schéma de compilation Comme pour js_of_ocaml, La traduction transpose les structures de contrôles d'OCaml vers celles de JavaScript, y compris les fonctions et les exceptions. Il n'utilise cependant pas l'application de JavaScript mais un mécanisme personnalisé permettant l'application partielle, et la récursion terminale générale via un trampoline.

Valeurs Pour les types simples et les structures de base (tableaux, etc.), OcamlJS utilise au maximum les équivalents JavaScript des types OCaml, rendant ainsi l'inter-opérabilité particulièrement simple. Il change cependant légèrement la sémantique d'OCaml, en projetant toutes les valeurs numériques vers le type générique Number de JavaScript. Une autre différence est faite pour les chaînes, pour lesquelles OcamlJS donne un choix de sémantique futé, légèrement différent de celui standard en OCaml : les littéraux de chaînes du programme sont non mutables, et les valeurs créées par les fonctions du type String.make sont des objets mutables, disposant d'une méthode toString permettant qu'elles soient utilisables en JavaScript de façon transparente.

Performances Lors de la conception d'OBrowser, les performances entre ce dernier et OcamlJS n'étaient pas très éloignées, certains programmes étant plus rapide d'un facteur 2 environ avec OcamlJS, les programmes très fonctionnels ou avec des exceptions au contraire beaucoup plus rapides avec OBrowser. Depuis les optimisations de traces, et les versions plus récentes d'OcamlJS, la différence s'est creusée. OcamlJS reste cependant globalement moins rapide que js_of_ocaml, le trampoline qui permet la récursion terminale générale que n'a pas js_of_ocaml étant la raison principale.

Modèle objet Jake Donham a essayé plusieurs méthodes pour les objets d'OcamlJS. Dans la dernière version, les objets sont inter-opérables avec ceux de JavaScript dans les deux sens, bien qu'il ne soit pas possible d'étendre les objets JavaScript en OCaml, et que certains cas de passages non gérés d'objets d'un monde à l'autre ne soient pas détectés statiquement et lèvent des exceptions. Étant tout seul à développer le compilateur sur son temps libre, le développement d'OcamlJS se fait petit à petit, et ces problèmes seront sûrement réglés dans une future version.

OcamlJS utilise aussi des ruses de nommage un peu comme celles de js_of_ocaml pour le polymorphisme, mais aussi pour les accesseurs et modificateurs où une méthode OCaml de la forme _get_x et (resp. _set_x) est projetée vers l'accès au (resp. la modification du) champ x.

Modèle de concurrence Comme js_of_ocaml, OcamlJS n'abstrait pas le modèle de concurrence, et il est recommandé de s'adapter au modèle événementiel avec les threads coopératifs de Lwt dont une implantation est fournie, avec les primitives supplémentaires yield permettant de rendre la main au navigateur, sleep effectuant la même chose mais avec un délai, et http_request pour effectuer des communications HTTP.

Aspect client/serveur OcamlJS ne fait pas partie d'un projet de cadre client/serveur unique, mais il est tout de même utilisé avec des serveurs écrits en OCaml. Il est associé pour cela une bibliothèque ORPC, qui transforme un IDL d'appels distants via HTTP en modules et fonctions OCaml pour chaque partie.

6.3 Conclusion

Dans cette partie I, nous avons présenté OBrowser, notre plate-forme expérimentale de support de programmation côté client. Cette expérience se démarque des travaux connexes en montrant qu'il est possible de programmer le client dans un langage, un système de types, un modèle objet et un modèle de concurrence différents de ceux imposés par le navigateur au programmeur JavaScript. Nous avons de plus présenté différentes méthodes implantées dans OBrowser permettant l'inter-opérabilité entre OCaml et JavaScript, pour utiliser l'environnement du navigateur ainsi que les bibliothèques tierces existantes tout en conservant le modèle d'exécution et le typage du langage de haut niveau.

Nous avons vu dans ce chapitre que certains choix dans l'architecture d'OBrowser font que les performances des autres solutions pour exécuter du code OCaml dans le navigateur sont bien meilleures. Nous avons pour cela entamé des travaux, d'une part pour mettre à jour le code de l'implantation JavaScript d'OBrowser, et d'autre part en vue de l'écriture d'un greffon intégrant la machine virtuelle OCaml, comme évoqué dans ce chapitre. Dans les deux cas, si le but est de chercher à améliorer les performances, nous conservons comme priorité l'abstraction de JavaScript, et la compatibilité avec le modèle d'OCaml.

Outre le problème des performances que nous venons d'évoquer, qui est principalement un problème d'optimisation de l'implantation et non un défaut de l'approche, il reste à l'issue de cette partie un point non satisfaisant dans la programmation du client en OCaml avec OBrowser : les manipulations du document. Comme nous l'avons évoqué en introduction, et plusieurs fois au cours de cette partie, l'API de manipulation du document donnée par le navigateur, le DOM, est bas-niveau et ne prend pas en compte la grammaire du document. Elle est donc particulièrement désagréable à utiliser dans un contexte de langage statiquement et fortement typé, car incohérente avec le reste du langage et de la bibliothèque, et source d'erreurs à l'exécution. Dans la partie II, nous chercherons à régler ce problème, en formalisant une API de manipulations du document sûres du point de vue des types. Bien sûr, utiliser le langage OCaml complet pour formaliser la solution aurait été inutilement complexe, nous avons donc décrit pour l'occasion un langage proche mais plus simple. Nous donnerons alors dans la partie III des

solutions pour intégrer les manipulations sûres du document de la partie II au sein d'OCaml, en utilisant les mécanismes d'inter-opérabilité présentés dans cette partie.

II Deuxième partie
Modifications du document

7 Problématique

Un document Web est essentiellement formé du contenu textuel et visuel de la page ainsi que de sa composition hiérarchique au sein d'éléments de plus haut niveau selon une structure d'arbre. Les nœuds du document peuvent alors être des nœuds de texte, ou des nœuds éléments liant une séquence d'autres nœuds. Les nœuds éléments portent une étiquette permettant d'en différencier plusieurs types, et les nœuds peuvent être paramétrés par des attributs nommés. Le moteur de rendu du document utilise ces étiquettes et ces attributs pour déduire la sémantique visuelle à donner au document.

Pour le stockage et la transmission, cette structure est en général aplatie au format XML. Dans le navigateur, après être lu depuis le format XML, le document est représenté à l'exécution par une structure en mémoire conforme à l'API DOM. Si le DOM initial d'une page Web reflète exactement l'arbre reçu sous forme XML, ce DOM peut être modifié au cours de la vie du document, par exemple depuis JavaScript, OBrowser, ou un greffon natif (cf. chapitre 4). Les changements effectués via le DOM sont répercutés dynamiquement sur le rendu visuel par le navigateur.

Pour que le moteur de rendu puisse attribuer une sémantique visuelle au document, celui-ci doit obéir à certaines règles de construction. Il s'agit principalement d'une vérification de typage par rapport à une grammaire d'arbres réguliers spécifique au format de document en question (HTML, XHTML, SVG, etc.). Des solutions existent pour vérifier la conformité du document initial, nous en donnons un aperçu à la section 7.1, cependant, l'API de modification du document via le DOM peut casser ces propriétés dynamiquement.

Dans cette partie, nous proposons une solution pour que le document reste conforme au cours des manipulations qu'il subit via son DOM. Nous décrivons par l'exemple la problématique à la section 7.2, passons en revue à la section 7.3 comment les autres environnements de programmation Web se comportent vis-à-vis de ce problème, et finalement décrivons à la section 7.4 la solution proposée que nous détaillons au cours des chapitres suivants.

7.1 Création de document bien typés

En pratique, les règles de formation du document sont définies au format DTD. Le consortium W3C définit les recommandations à suivre pour les différents types de documents que l'on peut trouver sur le Web et les publie selon ce format. Une fois écrite, une page HTML statique peut être validée en ligne via l'outil de validation du W3C [⚓69]. La plupart des bibliothèques de gestion du format XML intègrent en outre un vérificateur de conformité à une DTD. Il est donc possible de vérifier automatiquement qu'un site Web statique est valide, ou même vérifier à la volée que les pages Web générées et envoyées par des scripts serveur sont valides. Il reste bien entendu à savoir que faire si elles ne le sont pas. Dans cette section, nous présentons les solutions qui rendent cette question caduque en assurant que le programme ne peut générer que des pages valides.

7.1.1 Langages et systèmes de types pour XML

Le langage CDuce [🕮4] (et avant lui XDuce) offre un système de types spécialement adapté à la manipulation d'arbres et d'expressions réguliers, assorti d'une syntaxe proche d'XML. Le noyau fonctionnel et le filtrage des valeurs XML en profondeur en font un outil pratique pour manipuler des données arborescentes. Le système de types est assez expressif pour certifier que les programmes écrits en CDuce, s'ils sont bien typés, ne généreront que des document valides. L'inférence de types permet en outre

d'utiliser des étapes intermédiaires lors de la confection d'une page Web, et ce sans avoir à écrire manuellement les types des ces données de transition, ce qui est un vrai plus étant donné la verbosité des grammaire des documents Web.

Nous devons cependant citer une limitation importante de ce système de types, qui affecte les possibilités d'utilisation côté client, et s'approche du problème que nous essayons de résoudre dans ce travail. La grammaire HTML permet d'associer des identifiants aux nœuds via l'attribut *id*. Cependant, une contrainte annexe de la grammaire stipule qu'un même *id* ne peut apparaître plus d'une fois dans un document. Le système de types de CDuce ne permet pas une telle vérification. En pratique, une valeur XML comportant un *id* peut par exemple être insérée plusieurs fois dans un même parent, ou dans deux parents différents. Nous verrons concrètement pourquoi ce type de langages n'est pas directement utilisable pour travailler sur le DOM section 7.2.

7.1.2 Génération de XML dans les langages généralistes

Dans cette section, nous présentons les différentes solutions existantes permettant de programmer le Web avec une certaine sécurité de typage, tout en cherchant à conserver un langage généraliste existant comme langage hôte. Nous avons choisi de présenter ces solutions ensemble car elles s'inscrivent dans cet effort d'essayer d'apporter un peu de sûreté dans un langage généraliste. Mais pour rester honnête, ces solutions n'ont ni le même niveau de typage, ni le même succès commercial. Clairement, les *modèles* et les *widgets* sont déjà présents dans des solutions largement utilisées, mais elles n'apportent pas le même niveau de sûreté que les autres solutions présentées. Malheureusement, les solutions les plus sûres restent à ce jour très peu répandues.

Modèles L'approche la plus utilisée dans les environnements de programmation Web grand public est de découpler la mise en place du contenu grâce à un mécanisme de modèles (*templates*). Le programmeur va alors écrire des pages HTML statiques munies d'annotations définissant les emplacements où va être ajouté le contenu dynamique. Les pages statiques peuvent alors être vérifiées indépendemment du programme et apporter une certaine validité, mais limitent bien entendu fortement l'expressivité du modèle. Ce type de solution est en général couplé à une des solutions plus dynamiques présentées dans les paragraphes suivants.

Intégration d'un système de types spécifique Une possibilité pour générer du XML bien typé depuis un langage généraliste est d'y intégrer un DSL (*Domain Specific Language*) de manipulation de documents XML ainsi que son système de types. C'est ce que fait OCamlDuce [📖14]. Le langage OCaml est alors doté d'une extension de syntaxe permettant d'y insérer des morceaux de DSL et d'un type abstrait pour les morceaux de document XML. Une passe de compilation est effectuée afin de typer les fragments du DSL et les compiler vers OCaml. Lors de la phase de compilation par OCaml, les types XML n'apparaissent plus et sont remplacés par le type abstrait. Cette approche a l'avantage de pouvoir réutiliser le langage OCaml et ses outils, mais elle est plus limitée en termes d'expressivité que l'utilisation d'un langage dédié. En particulier, les termes issus du DSL étant abstraits vis-à-vis du système de types d'OCaml, il est impossible de les paramétrer, et il est impossible d'insérer des valeurs du langage au sein des valeurs XML.

Encodage du typage XML Une possibilité qui semble naturelle est d'exprimer les contraintes de bonne formation de la page Web directement dans le système de types du langage de manipulation. Étant donné l'insuffisante expressivité des systèmes de types communs, il est impossible en général d'exprimer toutes les contraintes d'une DTD donnée. Une approche possible est alors de restreindre énormément l'expressivité en encapsulant les manipulations et en exposant seulement quelques types de morceaux de

documents dont la combinaison est facile à typer. Cette approche sera détaillée un peu plus loin, au paragraphe sur les composants d'interface graphique. L'autre possibilité qui nous intéresse ici est d'exprimer un maximum de ces contraintes en poussant le système de types dans ses retranchements. C'est ce que fait [🕮12] pour SML Server [🕮11], [🕮50] pour Haskell ou le module OCaml XHTML.M [⚓79] utilisé par Ocsigen, qui est capable de prendre en charge en très grande partie la DTD d'HTML. Cette approche est parfaitement valable, mais demande une mise au point importante pour rendre l'API agréable et les erreurs de types lisibles. D'autre part, elle demande un système de types pour le langage hôte déjà beaucoup plus expressif que ce que permettent les langages grand public. XHTML.M, par exemple, exploite le sous-typage structurel introduit par les variants polymorphes d'OCaml pour encoder de façon concise le fait qu'une balise d'un certain type peut se retrouver comme fille de plusieurs types de balises, ce qui serait difficile et extrêmement verbeux avec le sous-typage des langages objets comme Java. Si cette approche est séduisante, elle souffre des même défauts, que nous verrons par la suite, qui font qu'elle n'est utilisable que pour la génération de XML mais pas directement pour typer le DOM.

Composants graphiques (*Widgets*) La plupart des environnement de développement Web pour les langages typés dynamiquement (Django [⚓83] pour Python, Symphony [⚓86] pour PHP, etc.), ainsi que pour les langages à objets généralistes (ASP.Net, etc.) s'affranchissent du code HTML en proposant des composants dont le code HTML est pré-programmé et bien testé.

En fournissant un ensemble de composants paramétrables et couvrant un maximum de fonctionnalités, il ne reste qu'à utiliser un mécanisme d'agrégation des composants assez souple pour que le programmeur soit complètement abstrait du code HTML. Le modèle généralement utilisé est le patron de conception de programmation par objets MVC, laissant apparaître une interface abstraite d'accès aux composants, et permettant de rendre ces composants inter-actifs grâce à une forme limitée et bien encapsulée de génération de code client.

Concrètement, cette approche est facilitée par l'élément *div* du HTML qui permet de contenir à peu près n'importe quel autre élément, et peut être placé dans un autre div. Les composants exposent alors un *div* principal contenant le reste de leur implantation, et les conteneurs proposent plusieurs emplacements pour des composants sous forme d'éléments *div*. En pratique, cela signifie souvent que le type d'interfaces ainsi générées est calqué sur la conception d'interface de bureau et s'éloigne du document inter-actif.

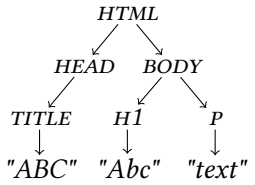
Grammaires abstraites de document À mi-chemin entre les deux solutions précédentes, il est possible de s'abstraire du typage compliqué du XML, tout en profitant de l'expressivité du système de type hôte, en définissant une grammaire intermédiaire de document. La bibliothèque Lambdoc [⚓88] en est un exemple : elle fournit un modèle de document simplement exprimé dans le système de types d'OCaml, et sait produire du code HTML bien typé à partir de n'importe quel document. Cette approche est valable est sûre de produire une API agréable, mais est limitée par le fait qu'elle est entièrement manuelle (il est bien sûr impossible d'extraire une sous grammaire simple et expressive automatiquement à partir d'une DTD).

7.2 Typage et modifications du document

Le DOM est une recommandation du W3C, définissant une API indépendante du langage, pour explorer, modifier et construire des parties de documents en mémoire. La spécification en langue anglaise ainsi que les types des primitives dans un IDL sont disponibles publiquement à l'adresse [⚓68]. Ces primitives sont en style impératif et de bas niveau.

La figure 7.1 présente une visualisation arborescente du DOM, le code XML correspondant, ainsi qu'une succession d'appels DOM en pseudo-JavaScript permettant de construire un tel document. Elle

présente les fonctions suivantes : *createElement* construit un nœud élément vide avec une étiquette donnée, *createTextNode* construit un nœud texte, et *appendChild* relie deux nœuds dans l'arbre.




```
<html>
  <head><title>ABC</title></head>
  <body>
    <h1>Abc</h1>
    <p>text</p>
  </body>
</html>
```

```
var title = createElement ("TITLE");
title.appendChild(createTextNode("ABC"));
var head = createElement ("HEAD");
head.appendChild (title);
var heading = createElement ("H1");
heading.appendChild(createTextNode("Abc"));
var par = createElement ("P");
par.appendChild(createTextNode("text"));
var body = createElement ("BODY");
body.appendChild (heading);
body.appendChild (par);
var html = createElement ("HTML");
html.appendChild (head);
html.appendChild (body);
return html;
```

FIGURE 7.1: Exemple de DOM et XML d'un document

Typage et effets de bord Comme nous avons vu à la section 7.1, il existe des moyens pour s'assurer que la génération d'XML est valide. La validité du document est essentielle pour que le comportement du moteur de rendu soit bien défini.

Le DOM, tout comme le format XML est conçu pour prendre en charge n'importe quel type de document arborescent. Les types des primitives du DOM sont complètement génériques par rapport à une éventuelle DTD. Cependant, pour être rendu correctement, un document doit satisfaire sa DTD tout au long de sa vie. Au final, les opérations DOM sont souvent trop génériques, permettant de modifier des documents et les rendre mal typés, et lorsque c'est le cas le navigateur fait ce qu'il peut pour rattraper la situation : parfois le moteur de rendu est assez souple pour continuer avec un arbre mal typé, parfois il répare l'arbre de façon arbitraire, ou bien il peut lancer un exception. Il est donc important de s'assurer que les opérations DOM ne cassent pas le typage si on veut conserver le déterminisme et la portabilité.

Effets de bord et structure d'arbre Les opérations DOM sont suffisamment de bas niveau pour permettre de produire des documents non arborescents, en introduisant du partage ou des cycles au sein de la structure. Afin d'assurer la forme d'arbre du document, les implantations du DOM utilisent une sémantique assez inhabituelle d'effets de bord non locaux. Par exemple, la figure 7.2 montre un document avec deux nœuds n_1 et n_2 fils du même nœud père p_1, et le résultat de l'affectation de n_2 à un nouveau parent p_2 (en JavaScript cela s'écrirait p_2.appendChild(n_2)). On peut voir que le nœud est automatiquement détaché de son parent précédent, afin de prévenir le partage.

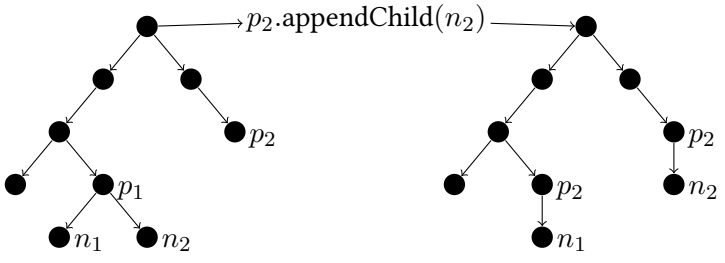


FIGURE 7.2: Déplacement implicite dans le DOM

Typage, effets de bord et structure d'arbre : le problème Nous avons vu comment il est possible, étant donné le caractère bas niveau des primitives du DOM, d'écrire des opérations pouvant casser explicitement le typage d'un nœud. Nous avons ensuite vu la solution pour que la forme d'arbre soit maintenue dans l'implantation du DOM par des déplacements implicites. Observons maintenant comment cette solution introduit une façon plus insidieuse de casser le typage du document.

La figure 7.3 présente un document HTML valide, un code JavaScript, et le document résultant après son exécution. Le document original est composé de deux listes (étiquette *ul*), la première comportant deux éléments (étiquette *li*), et la seconde un seul élément. Selon la DTD d'HTML, une liste doit comporter au moins un élément. Le code JavaScript prend une référence sur le fils unique de la seconde liste, et l'affecte en tant que fils de la première. Comme nous l'avons vu, l'élément est alors supprimé de la seconde liste pour se prémunir du partage. Le document résultat n'est alors plus du HTML valide.

```
1 <ul id="a">
2   <li>bli</li>
3   <li>bla</li>
4 </ul>
5 <ul id="b">
6   <li>blou</li>
7 </ul>
8 x
```

(a) Avant

```
1 <ul id="a">
2   <li>bli</li>
3   <li>bla</li>
4   <li>blou</li>
5 </ul>
6 <ul id="b">
7 </ul>
```

(b) Après

```
1 var a = document.getElementById ("a");
2 var b = document.getElementById ("b");
3 a.appendChild (b.firstChild);
```

(c) Code JavaScript

FIGURE 7.3: Opération DOM mal typée

Ce qu'il faut bien noter ici, est que la partie du document qui est devenue mal formée n'est pas la cible de l'affectation (la liste a), mais le parent de la valeur qui lui est affectée (la liste b). En d'autres termes, une valeur a été correctement construite, puis est devenue mal typée à cause d'un effet de bord sur une autre valeur. Ce qui montre qu'un système de types pour la génération d'XML n'est pas directement utilisable dans le monde dynamique du DOM.

Et en fonctionnel ? En première lecture, on peut se dire qu'il *suffit* de ne pas permettre les effets de bord, et de se limiter à des constructions fonctionnelles (en conjonction avec, par exemple, une forme limitée de mutations dans l'arbre comme des ajouts), quitte à limiter le coté impératif du langage.

Mais l'utilisation du DOM comme primitives de construction d'arbres casse aussi le typage des constructions fonctionnelles. Pour s'en convaincre, la figure 7.4 donne un exemple de construction d'arbre, dans une hypothétique implantation de CDuce utilisant les primitives de construction d'arbre du DOM, et explique son déroulement.

```
1 let a = <li>[ "Item A" ] ;;
2 let l1 = <ul>[ a ] ;;
3 let l2 = <ul>[ a ] ;;
```

(a) Code

→ On crée a et l1 de façon bien typée.
→ On crée l2 de façon bien typée.
→ Mais a est supprimé de l1 !
⇒ la création de l2 rend l1 mal typée.

(b) Exécution

FIGURE 7.4: Construction mal typée en CDuce/DOM

7.3 Solutions et contournements

Cette section présente les avantages et inconvénients de la gestion du document mise en place par les solutions existantes. Ces points seront finalement illustrés au travers d'un exemple commun présenté figure 7.5.

Dans OPA Pour la génération, OPA utilise une représentation du XML dans son propre système de types. Lors de nos tests, le typage XML ne semblait pas encore fort, mais le système de types est assez puissant pour y encoder un typage raisonnable, à peu près du niveau de celui de XHTML.M.

Point de vue manipulation DOM, la primitive proposée dans OPA [⚓89] solutionne simplement le problème de partage. En OPA, les nœuds déjà créés ne peuvent se trouver à droite d'une affectation, ce qui était la cause de la création du partage. La partie droite d'une affectation dans l'arbre n'est pas un nœud DOM mais est une valeur XML qui est transformée en DOM à la volée. Ainsi, si une valeur XML est affectée deux fois, sa valeur DOM sera recréée à chaque fois. D'autre part, puisque le programmeur ne peut obtenir de référence sur les nœuds et sous-nœuds ainsi créés, le mécanisme utilisé pour référencer des nœuds est principalement celui des *id*. On a par exemple la syntaxe `[#nid +<- xml]` pour ajouter du contenu XML au nœud dont l'id est *nid*.

Cette solution supprime simplement le problème du partage, mais elle induit deux problèmes principaux. Tout d'abord nous avons vu précédemment que l'utilisation des *id* est dangereuse et fastidieuse si le système de types ne sait pas gérer leur unicité, et ce n'est effectivement pas le cas de celui d'OPA. D'autre part, l'expressivité de manipulations du DOM en est fortement restreinte, on ne peut par exemple tout simplement pas déplacer un nœud existant.

Dans Links Links [🕮9] propose un mécanisme similaire à OPA en distinguant les types DOM et XML. Cependant, il permet aussi de manipuler directement le DOM au travers d'une interface bas niveau et non typée. On peut donc avoir la même expressivité qu'OPA, mais en s'autorisant des manipulations non typées de bas niveau pour palier à ce manque d'expressivité lorsque nécessaire.

Dans Ocsigen La version 2.0 d'Ocsigen propose une implantation côté client du module XHTML.M, le même module qui sert à typer le XML côté serveur. Cette implantation construit directement des nœuds DOM, et il est possible d'appeler les primitives de manipulation directement sur les nœuds. Un mécanisme de re-liaison des références vers les nœuds permet en outre de faire correspondre des valeurs XML construites côté serveur à leurs réflexions côté client après la phase de rendu du XML initial.

Cette méthode a l'avantage important de rendre caduque le problème des *id*, et en même temps d'unifier la programmation du document côté client et côté serveur au travers d'une même API et du même type. Cependant, elle ne gère pas le problème du partage, et les modifications sont pour l'instant non typées.

Dans HOP Bien sûr, on ne parle pas de typage pour HOP, mais on peut quand même observer que HOP ne fait pas de distinction entre les nœuds DOM et les éléments XML, côté client comme serveur. Il offre un mécanisme similaire à celui présenté précédemment pour Ocsigen pour se passer des *id*. Cependant, il ne règle pas non plus le problème du partage. La figure 7.5 montre les problèmes d'*id* et de partage au sein d'un même exemple.

7.4 Solution proposée

Une solution correcte serait d'interdire les comportements non voulus à l'aide d'une analyse statique (resp. des vérifications dynamiques) des manipulations DOM. Par exemple, on pourrait concevoir un système de types assez expressif pour n'autoriser qu'une seule utilisation de chaque référence. Ainsi, il

```
(define-service (s)
   (<HTML>
      (define ctnr (<DIV>))
      ~(define d (<DIV> "test"))
      (<BODY>
         (<BUTTON> :onclick ~(dom-append-child! $ctnr d) "+")
         ctnr)))
```

(a) En HOP, *"test"* s'affiche une seule fois

```
client let d = pcdata "test"
let s = register_new_service ~path:["s"] ~get_params:unit
  (fun () () ->
    let ctnr = div [] in
    Lwt.return
      [ button ~a:[a_onclick {{%cntr##appendChild d ; Lwt.return ()}}] [pcdata "+"]
        ctnr ])
```

(b) En Ocsigen, *"test"* s'affiche une seule fois

```
server = one_page_server("test", ->
   d = (<div>test</div>)
   <button onclick={ _ -> exec([#ctnr +<- d]) }>+</button>
   <div id="ctnr"></div>)
```

(c) En OPA, *"test"* s'affiche plusieurs fois

FIGURE 7.5: Exemple de manipulation DOM en HOP, Ocsigen et OPA

serait possible statiquement (resp. dynamiquement) de refuser les codes présentés dans les exemples de ce chapitre aboutissant à des arbres mal formés.

D'un autre côté, nous pensons qu'il est naturel pour le programmeur, en particulier s'il est habitué aux constructions fonctionnelles, de construire ou récupérer des morceaux de document, puis de construire de nouveaux morceaux de document à partir des précédents, ou de les affecter, comme il le ferait avec des arbres et des références classiques. Nous pensons donc que les exemples présentés ici devraient être acceptés, et fonctionner avec une sémantique plus intuitive et fonctionnelle. De plus, sur le serveur, le programmeur a déjà cette vision fonctionnelle du document dans la plupart des solutions, où les représentations abstraites d'XML n'ont pas cette restriction sur le partage, et il est intéressant de vouloir harmoniser les sémantiques du client et du serveur. Bien entendu, nous voulons aussi nous passer de l'utilisation des *id*, en considérant les nœuds du document comme valeurs de première classe et directement mutables.

Une sémantique alternative copiante Afin d'implanter ce comportement, nous proposons une sémantique alternative pour le DOM, dans laquelle les morceaux de document sont dupliqués lorsque qu'ils disparaîtraient avec la sémantique originale. La figure 7.6 révise la figure 7.3 en conséquence.

Avec une telle sémantique, les systèmes de types pour XML peuvent être adaptés, puisque des nœuds déjà construits de façon bien typée ne peuvent pas devenir mal typés. Il reste bien sûr à typer les opérations de mutation des nœuds, mais ceci est désormais une tâche raisonnable puisque seule la cible explicite de l'effet de bord est à prendre en compte.

Documents inter-actifs et le DOM À l'exécution, une page Web ne contient pas que le document lui-même, mais aussi, lorsqu'il est inter-actif, des scripts et les données manipulés par ces scripts. Pour rendre des parties du document inter-actives, ce code et ces données sont accrochés directement sur les nœuds de l'arbre. La figure 7.7 montre un document assorti d'objets ne faisant pas partie de l'arbre. Nous représentons les nœuds document en noir, et les objets en blanc (le code est embarqué dans des fermetures qui sont aussi des objets et donc des nœuds blancs). Cette notation sera reprise dans les prochains chapitres. On peut remarquer que les objets ne sont pas soumis aux mêmes règles que les nœuds document. Le code et les données peuvent être partagés et définis de façon mutuellement récursive.

```
1 <ul id="a">
2   <li>bli</li>
3   <li>bla</li>
4 </ul>
5 <ul id="b">
6   <li>blou</li>
7 </ul>
```

(a) Avant

```
1 <ul id="a">
2   <li>bli</li>
3   <li>bla</li>
4   <li>blou</li> <!-- copie -->
5 </ul>
6 <ul id="b">
7   <li>blou</li> <!-- original -->
8 </ul>
```

(b) Après

```
1 let a = get_element_by_id "a" in
2 let b = get_element_by_id "b" in
3   append a (List.hd (children b))
```

(c) Code JavaScript

FIGURE 7.6: Opération DOM bien typés, grâce à la sémantique alternative copiante

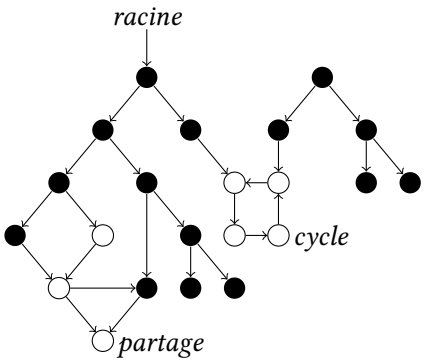


FIGURE 7.7: Exemple de document avec des objets attachés

La copie d'un nœud est alors rendue non triviale, puisqu'on voudrait évidemment dupliquer les comportements inter-actifs locaux, et donc les objets en relation avec les nœuds copiés, et la présence de cycles et de partage rend l'appartenance d'un objet à un nœud une décision non triviale.

La solution que nous proposons introduit une construction syntaxique spéciale au langage définissant l'ensemble des valeurs à attacher à un nœud document. Concrètement, on étend à l'exécution la notion de portée lexicale statique des valeurs pour obtenir l'appartenance aux nœuds document.

7.5 Travaux connexes

À notre connaissance, aucun autre travail ne propose une telle solution par mécanisme de copie implicite. Cependant, d'autres travaux cherchent à formaliser le DOM, en particulier pour rejeter les programmes effectuant des déplacement non voulus.

Formalisation du DOM Philippa A. Gardner et al. proposent une formalisation de l'API du DOM [📖16]. Ils définissent un sous-ensemble minimal de primitives du DOM, qu'ils spécifient en logique de Hoare. Puis ils introduisent un micro langage impératif formel, et l'utilisent pour implanter un plus grand ensemble de l'API du DOM, à partir du jeu minimal. De la composabilité de leur modélisation, ils peuvent alors déduire de façon automatique le comportement des fonctions plus complexes de l'API. Ce travail se rapporte à la formalisation que nous proposons au chapitre 8, mais l'approche que nous utilisons, et qui est nécessaire à la suite de notre travail, se différencie par le fait que nous cherchons à fournir un modèle simplifié et plus abstrait, et non à formaliser l'API complète du DOM.

Typage des modifications Notre solution principale utilisant des copies implicites est intéressante et novatrice, car elle permet d'assurer dans un même cadre le typage de la création et des modifications du document. En particulier, nous avons vu qu'elle permet d'écrire des constructions de document en style fonctionnel, sans se préoccuper du caractère impératif du modèle de document sous-jacent.

Cependant, nous verrons que la technique n'est pas triviale à implanter, et qu'elle implique un surcoût à l'exécution. Aussi, nous proposerons, en marge de notre solution principale, une version de notre modèle formel de document utilisant des vérification dynamiques, et produisant une erreur dynamique de typage dans le cas où une copie implicite aurait été nécessaire pour assurer la correction. L'intérêt pratique est de pourvoir conserver la même API sur toutes les parties d'une application, y compris dans le cas où l'une d'entre elles ne disposerait pas du mécanisme de copie, en assurant la sûreté par les erreurs dynamiques.

Les travaux de Peter Thiemann vont plus loin dans cette approche, en proposant une solution pour rejeter statiquement les programmes effectuant des déplacement implicites, en détectant ces déplacements au typage [📖31]. Pour ceci, il définit un système de types linéaires, interdisant l'ajout d'un nœud s'il a déjà été attaché, qu'il utilise sous forme d'annotations pour une API du DOM dans un langage généraliste. Ces annotations peuvent être écrites à la main, où synthétisées par analyse statique.

On peut aussi citer [📖18], où les auteurs utilisent un mécanisme génération automatique de tests pour rejeter des programmes réalisant des déplacements implicites.

Si ces travaux sont intéressants, et permettent d'assurer le typage du document au cours du programme, ils sont clairement différents de la solution que nous proposons dans le sens où nous avons pour but de rendre corrects et d'accepter les programmes qu'ils considèrent comme faux et rejettent.

7.6 Plan de cette partie

Dans cette partie, nous développons progressivement la solution présentée rapidement à la section précédente. Nous commençons par dompter puis assainir les bases existantes sur lesquelles nous pourrons alors ensuite construire le langage proposé.

Au chapitre 8, nous définissons ^{f}DOM, une version formelle simplifiée du document, sous forme d'API indépendante du langage, incluant les comportements impératifs que nous venons de voir.

Au chapitre 9, nous étendons ^{f}DOM en ^{c}DOM, pour y ajouter ces informations de portée et arriver à une sémantique copiante comme nous venons de présenter. Nous proposons aussi $^{\varepsilon}DOM$, une sémantique plus simple, qui empêche dynamiquement les déplacements.

Au chapitre 10, nous présentons FidoML un dialecte de ML, incluant les traits fonctionnel et impératif et les définitions de types personnalisés, et muni d'opérations de création et mutation de nœuds.

Aux chapitres 11 et 12, nous donnons un système de types statique pour FidoML, incluant le typage des nœuds, mais sans prise en compte d'une DTD. Nous donnons aussi la sémantique opérationnelle de FidoML dans laquelle les les effets sont délégués à ^{c}DOM.

Au chapitre 13, nous étendons le système de types de FidoML pour introduire le typage de la grammaire du document en fonction d'une DTD. Pour cela, nous donnons un mécanisme général permettant de générer des primitives de construction de document bien typées à partir d'une DTD dans un langage généraliste, et appliquons cette méthode à FidoML.

8 *fDOM*, modèle du document impératif

Dans ce chapitre, nous proposons *fDOM*, une vision formelle du document et de ses manipulations au sein du navigateur via le DOM. Cette formalisation est volontairement simplificatrice par rapport à l'API du DOM, mais n'en sacrifie pas l'expressivité. La recommandation DOM couvrant d'autres aspects, nous nous restreignons cependant à la manipulation de document.

Nous voulons également pouvoir utiliser *fDOM* comme spécification pour l'interface avec un DOM autre que celui des navigateurs, ou en implanter un nouveau. Par exemple il peut être intéressant de proposer la même interface de programmation de document côté serveur, ou de proposer une interface de manipulation à distance entre les parties cohérente avec l'interface de manipulation locale.

Dans cette optique, nous avons fait le choix de proposer une sémantique découplant les environnements d'implantation et de manipulation du document. Pour ceci, nous définissons *fDOM* comme un jeu de primitives aux entrées et sorties bien spécifiées, qui sera l'interface entre les deux environnements (une API). L'appel et la composition de ces primitives n'entrent pas dans le cadre de cette spécification et sont laissés à l'environnement hôte.

Avant de donner dans la section 8.2 leurs définitions formelles, nous commençons dans la section 8.1 par introduire de façon intuitive les termes et concepts généraux utilisés.

L'annexe C montre la validité pratique de l'approche au travers de deux implantations, l'une étant une implantation complète en OCaml, l'autre se basant sur le DOM du navigateur.

8.1 Définitions et terminologie du document impératif

Le concept que nous spécifions dans ce chapitre est celui de *document impératif*. Dans cette section introduisons progressivement les différents concepts utilisés.

Document L'élément essentiel de la page Web est le *document*. C'est une description arborescente de son contenu. Les nœuds de cet arbre servent à décrire à la fois le contenu textuel et l'imbrication graphique ou logique des éléments de la page. On peut parler de représentation uniforme, car les nœuds sont la seule entité présente dans le document, et ils sont tous de la même forme. Une telle définition des documents structurés peut s'appliquer à d'autres contextes que la page Web.

Nœud La brique de base pour constituer un document est donc le *nœud*. Un nœud peut contenir d'autres nœuds *enfants*, décrivant ainsi la structure d'arbre. Les enfants d'un nœud sont ordonnés. Un nœud ne doit être contenu que dans un seul *parent*, le document est un arbre et non un graphe (ni cycle, ni partage).

Un nœud possède une étiquette, définissant son rôle dans le document. Au sein d'un même document, plusieurs nœuds peuvent avoir la même étiquette. L'ensemble des étiquettes possibles est fini. Ce rôle n'est pas intrinsèque au document, il est donné par le programme qui l'interprète. Par exemple, un navigateur Web interprétera comme une liste à puces un nœud portant l'étiquette *ul*, en interprétant les enfants de ce nœud comme les éléments de la liste.

Comme nous avons vu au chapitre précédent, en général, le programme interprétant ce document demande à ce que l'arbre respecte des règles de formation. Par exemple, à l'intérieur d'un *ul*, le navigateur s'attend à trouver des nœuds d'étiquette *li*. Ce problème est bien découpé dans notre traitement formel du document, cette problématique ne sera introduite qu'au chapitre 13. À ce niveau de définition du document, un nœud peut contenir des nœuds de n'importe quelle étiquette.

Propriété L'imbrication des nœuds, avec les étiquettes, permet de décrire la structure hiérarchique du document, et de donner un sens à chaque sous-arbre. Mais la définition d'un document nécessite souvent de préciser plus en détail le rôle de chaque nœud. En particulier, dans un document textuel comme une page Web, il faut bien que le texte soit représenté quelque part dans le document.

Pour ceci, Chaque nœud, en plus de son étiquette et de ses enfants, possède un ensemble d'associations de *clefs* à des *valeurs* qu'on appelle *propriétés*.

La encore, il est de coutume que les nœuds d'une même étiquette aient les mêmes propriétés, et là encore, à ce niveau de spécification aucune hypothèse n'est faite sur l'ensemble des propriétés d'un nœud. Nous introduirons cette possibilité au chapitre 10.

Valeurs et données annexes Les *valeurs* des propriétés peuvent être des valeurs *immédiates* (nombres, chaînes, booléens, etc.), mais aussi des valeurs *structurées* (listes, etc.), contenant à leur tour des valeurs immédiates ou structurées. Les définitions de styles d'affichage graphique dans les nœuds d'une page Web sont un exemple de valeurs structurées.

On emploiera parfois le terme de *données annexes* pour définir l'ensemble de ces valeurs structurées qui font partie du document, mais pas directement de l'arbre.

Document impératif, état Le *document impératif* est défini par un *état* et un ensemble de *primitives*, qui peuvent être lui appliquées pour modifier son état.

L'*état* est un document, tel qu'il a été défini précédemment, c'est-à-dire un ensemble de nœuds étiquetés et leur structure d'arbre, et des données annexes.

Dans l'état du document impératif, on ajoute que les valeurs peuvent aussi être des nœuds. C'est le cas par exemple dans un navigateur, où les rattrapeurs d'évènement peuvent référencer d'autres nœuds pour agir dessus.

Environnement d'implantation, environnement hôte Dans notre spécification, nous avons choisi de découpler l'environnement (comprenant le langage, les bibliothèques, une éventuelle architecture matérielle spécifique, etc.) dans lequel est implanté le document, de celui dans lequel sont décrites les manipulations. Nous parlerons alors respectivement d'environnement et de langage d'implantation, et d'environnement et de langage hôte. Éventuellement, la spécification n'exclue pas qu'ils soient confondus.

Primitives Les *primitives* agissent sur le document impératif pour le transformer et modifier son état. Elles peuvent modifier l'ensemble des nœuds, leur structure d'arbre, et les données associées.

Les primitives sont identifiées par un nom. Elles prennent des paramètres, et renvoient un résultat. Concrètement, ces primitives seront implantées suivant le langage hôte comme des fonctions, méthodes, procédures, etc.

8.2 Définition formelle de *fDOM*

Puisqu'il s'agit d'une interface entre deux environnement non fixés, certains points de *fDOM* ne peuvent être spécifiés universellement. Nous commençons donc par identifier ces paramètres de *fDOM* à la section 8.2.2, en précisant les propriétés que doivent vérifier leur instanciation au sein d'une implantation.

Puis, section 8.2.2, nous définissons la spécification d'un état du document, que nous appellerons *état formel*, par opposition à l'*état concret* de l'implantation. Ensuite, section 8.2.3, nous définissons l'ensemble de primitives, et donnons les règles sémantiques décrivant les effets de ces primitives sur l'état formel.

Structure formelle, éléments de correction Pour qu'une implantation de *fDOM* soit correcte, il faut être capable de donner, pour chaque état concret pouvant exister dans l'implantation, l'état formel de *fDOM* auquel il correspond. De même, il faut s'assurer que les transformations de l'état concret opérées dans l'implantation de chaque primitive sont cohérentes avec les transformations de l'état formel de sa spécification.

Pour faciliter la vérification de la correction des implantations, plutôt que d'utiliser une structure mathématique complexe auto-suffisante, incluant en particulier la propriété d'arbre, nous avons choisi de découpler les éléments constitutifs de l'état de ses propriétés structurelles. L'état formel est donc décrit comme une structure ensembliste simple, insuffisante pour exprimer toutes les propriétés d'un document impératif. Nous définissons ensuite le sous-ensemble des états formels *valides* en ajoutant un prédicat de validité.

Si cette forme découplée peut être discutable dans une optique de définition de modèle théorique minimal, elle a deux avantages principaux dans une optique de spécification. D'une part une structure ensembliste simple peut, dans la plupart des langages, être directement interprétée comme structure de données d'une implantation. D'autre part, nous verrons que le prédicat de validité n'a pas besoin d'être interprété concrètement, à part pour le cas trivial d'un document initial vide. La correction d'une implantation se limite alors au lien entre état l'état concret et la structure ensembliste de l'état formel, ainsi qu'à la conformité de chaque primitive par rapport à sa spécification (indépendemment des autres). Pour ceci, nous montrerons que n'importe quelle séquence de primitives, si elle est appliquée à un état formel valide aboutit à un état formel valide.

8.2.1 Paramètres de *fDOM*

Comme dit plus haut, certains points de *fDOM* sont laissés en paramètres, à définir judicieusement lors de l'implantation. Ces paramètres servent d'une part à personnaliser la sémantique, et d'autre part à faire l'interface entre l'environnement d'implantation et l'environnement hôte. Voici la liste de ces paramètres, avec les conditions qu'ils doivent respecter, ainsi que des exemples d'implantations.

- ***Tag*** : les étiquettes possibles des nœuds.
  fDOM ne pose pas de condition sur ces étiquettes à part d'être un ensemble fini.
  Dans un navigateur, cet ensemble sera limité aux étiquettes définies dans la grammaire associée au document.
- ***Imm*** : l'ensemble des valeurs immédiates.
  En JavaScript, ce sera l'union des chaînes, des nombres, de la valeur nulle et des fonctions.
- ***Key*** : l'ensemble des noms de propriétés des objets. Il doit simplement être dénombrable et totalement ordonné, puisqu'on veut pouvoir effectuer des recherches par nom de propriété. En pratique, les valeurs doivent être non mutables, ou passées par copie, sinon la validité pourrait être cassée entre deux appels de primitive.
  En JavaScript, ce seront les chaînes de caractères (même si les interprètes JavaScript effectuent des optimisations si celles-ci sont assimilables à des entiers).
- ***nil*** : une valeur spéciale représentant un état non défini.
  Ce sera undefined en JavaScript, None ou () en OCaml suivant si le résultat/paramètre peut alternativement être défini ou s'il est toujours inutile. On utilisera $\{nil\}$ pour dénoter le type de *nil*.
- ***Int*** : la représentation des entiers positifs.
  En pratique, il faut juste s'assurer que le nombre d'enfants d'un nœud n'excède pas une éventuelle valeur maximale de l'implantation pour que la spécification soit respectée.
- ***Enum*** : la représentation des *ensembles énumérables finis*, utilisée pour les primitives renvoyant plusieurs valeurs. Cette représentation est nécessaire afin d'expliciter la conversion entre un ensemble de valeurs de la spécification et sa représentation manipulable par l'environnement hôte. Concrètement, il s'agit simplement d'une structure de données contenant plusieurs valeurs, et qui

peut être parcourue (Collection en Java ou List en OCaml).
Il faut donc définir $Enum(E)$ la représentation des ensembles énumérables de valeurs de E pour toutes les valeurs de E utilisées dans la spécification, et $enum : S \to Enum(S)$ la fonction calculant la représentation d'un ensemble S. Dans le cas où la sémantique est utilisée au sein d'une autre spécification formelle, comme c'est le cas au chapitre 12, on pourra simplement identifier $Enum(S)$ à $\mathcal{P}(S)$ (*enum* étant la fonction identité) et utiliser les opérations ensemblistes.

8.2.2 État du document

Un état formel de *fDOM* spécifie l'ensemble des nœuds étiquetés du document, leur structure arborescente ainsi que leurs propriétés, y compris les données annexes. Il prend la forme d'un quadruplet (H, L, T, P) comme suit :

$$fDOM \supseteq \left\{ (H, L, T, P) \;\middle|\; \begin{array}{rcl} H & \subseteq & Loc \times \{\bullet, \circ\} \\ L & \subseteq & H^\bullet \times Tag \\ T & \subseteq & H^\bullet \times List(H^\bullet) \\ P & \subseteq & H \times Key \times (H \cup Imm) \end{array} \right\}$$

- ***H*** (mnémonique pour *heap*), représente le domaine des *objets* présents dans la structure. On regroupe sous le terme *objet* les nœuds et les données annexes structurées. En termes d'implantation les objets sont les valeurs allouées dans le *tas*.
  Un élément de H est un couple. La première composante est issue de l'ensemble dénombrable *Loc* représentant les pointeurs eux-mêmes. La seconde est une couleur associée au pointeur, différenciant les données annexes des nœuds. On reprend les notation du chapitre chapitre 7 : la couleur noir dénote les nœuds étiquetés du document, la couleur blanc les données annexes. Un pointeur ne peut pas apparaître avec deux couleurs différentes ($\forall (l, c) \in H, \forall (l', c') \in H, l = l' \Rightarrow c = c'$).

  Notations : Afin d'améliorer la lisibilité des formules, on définit les notations suivantes sur H et ses éléments :
  - On utilise le glyphe $\bullet$ (resp. $\circ$) pour désigner un objet noir (resp. blanc).
  - Dans une formule mettant en jeu un seul élément de H $(l, \bullet)$ (resp. $(l, \circ)$), on désignera ce nœud simplement par $\bullet$ (resp. $\circ$).
  - Dans une formule mettant en jeu plusieurs éléments $(l_i, \bullet)$ (resp. $(l_i, \circ)$), on utilisera la notation $\bullet_i$ (resp. $\circ_i$).
  - Si la couleur d'un objet n'a pas d'importance dans une formule, on utilisera la notation $\bullet$ (ou $\bullet_i$ s'il y en a plusieurs).
  - On note $H^\bullet$ (resp. H°) l'ensemble des éléments de H dont la couleur est noir (resp. blanc). On pourra alors aussi noter H comme $H^\bullet$. On utilisera aussi cette notation en exposant pour d'autres ensembles faisant intervenir une couleur de la même façon.

- ***L*** (mnémonique pour *labels*), décrit les étiquettes des nœuds, en mettant en relation des éléments de $H^\bullet$ et de *Tag*.
  Chaque nœud a une et une seule étiquette, L a donc une forme d'application, on utilisera cette propriété pour obtenir l'étiquette d'un nœud en écrivant $L(\bullet)$.
  Comme nous l'avons expliqué, nous conservons dans la spécification de l'état une forme d'ensemble de couples, plutôt que d'utiliser une fonction. Cette structure est en effet plus simple, et directement disponible et manipulable dans les langages généralistes, contrairement aux applications. C'est donc le prédicat de validité qui encodera la forme d'application.
- ***T*** (mnémonique pour *tree*), décrit la structure d'arbre du document, en mettant en relation les

nœuds (éléments de $H^\bullet$) et les listes de leurs enfants. Nous utilisons des listes pour implanter le caractère ordonné, la définition des listes que nous utilisons est donnée figure 8.2, à la fin de ce chapitre.
Là encore, T associe une et une seule liste d'enfants à chaque nœud, et a donc une forme d'application, on utilisera donc $T(\bullet)$ pour obtenir les enfants d'un nœud.
Cette structure formelle permettant d'encoder des graphes plus généraux que les arbres, nous vérifierons la forme d'arbre dans le prédicat de validité.

– ***P*** (mnémonique pour *properties*) modélise les propriétés des objets sous forme de triplets (objet,clef,valeur). Les valeurs peuvent être des objets ou des valeurs immédiates, il n'y a pas de structure d'arbre à maintenir dans P, comme nous l'avons vu au chapitre 7.

Validité La définition présentée ci-dessus contient l'ensemble des entités présentes dans le document, mais sa définition est trop lâche pour exprimer toutes les propriétés voulues. On restreint donc *fDOM* aux quadruplets *valides*, $fDOM = \{(H, L, T, P) \text{ valide}\}$, avec la définition de validité suivante :

Définition 8.2.1 *Un quadruplet $d = (H, L, T, P)$ est valide si et seulement si :*

1. *L est une application de $H^\bullet$ dans Tag :*
$$\begin{cases} \forall \bullet \in H^\bullet, \exists t, (\bullet, t) \in L \\ \forall (\bullet_p, t) \in L, \forall (\bullet_{p'}, t') \in L, \bullet_p = \bullet_{p'} \Rightarrow t = t' \end{cases}$$
2. *T est une forêt :*
   (a) *T est une application de $H^\bullet$ dans $List(H^\bullet)$:*
   $$\begin{cases} \forall \bullet \in H^\bullet, \exists l, (\bullet, l) \in T \\ \forall (\bullet_p, l) \in T, \forall (\bullet_{p'}, l') \in T, \bullet_p = \bullet_{p'} \Rightarrow l = l' \\ \forall (\bullet, l) \in T, T(l) \subseteq H^\bullet \end{cases}$$
   (b) *Pas d'enfant partagé :*
   $\forall (\bullet_p, l) \in T, \forall \bullet_n \in l, \forall (\bullet_{p'}, l') \in T, \forall \bullet_{n'} \in l', \bullet_n = \bullet_{n'} \Rightarrow \bullet_p = \bullet_{p'}$
   (c) *Pas d'enfants doublons :*
   $\forall (\bullet_p, l) \in T$, *unique-list*$(l)$
   (d) *Pas de cycle :*
   $$\forall \bullet_x \in H^\bullet, \nexists \bullet_i, 0 \leqslant i \leqslant n, \begin{cases} \bullet_0 \in T(\bullet_x) \\ \bullet_{i+1} \in T(\bullet_i) \text{ pour } 0 \leqslant i \leq n \\ \bullet_x \in T(\bullet_n) \end{cases}$$
3. (a) *P ne porte que sur des pointeurs valides :*
   $\forall (\circ, k, v) \in P, \circ \in H$
   (b) *P ne référence que des pointeurs valides :*
   $\forall (\circ, k, v) \in P, v \in H \cup Imm$
   (c) *P associe une seule valeur à chaque couple objet/clef :*
   $\forall (\circ, k, v) \in P, \forall (\circ', k', v') \in P, (\circ, k) = (\circ', k') \Rightarrow v = v'$

8.2.3 Forme des primitives et règles

Nous donnons dans cette section l'ensemble des primitives, tout d'abord les primitives permettant d'accéder au document depuis le langage hôte, puis celles permettant de le modifier. Nous donnons pour chaque primitive une description textuelle de son utilité, le type de ses arguments et celui de son résultat, ainsi que les règles sémantiques décrivant les comportements possibles de chaque primitive. L'ensemble des primitives, leurs types et les règles associées seront récapitulés dans le tableau de la figure 8.1 à la fin de cette section.

Les types sont donnés sous la forme $T_{arg_0} \times \cdots \to T_{arg_n} \to T_{ret}$, où les T_{arg_i} sont les types des arguments, et T_{ret} le type de retour. Puisque ces paramètres et valeurs de retour sont à l'interface entre l'environnement d'implantation et l'environnement hôte, les T sont des paramètres de la sémantique ou des compositions de ceux-ci.

Chaque règle est de la forme $(\textsc{Règle})\frac{\textit{conditions}}{S \vdash \texttt{prim}(a_0,\cdots,a_n) = r, S'}$, se lisant : si les *conditions* sont vérifiées, alors la primitive `prim` s'exécute selon la règle (Règle), prenant en arguments les valeurs $(a_0, \cdots, a_n)$ et renvoyant le résultat r. Lors de cette exécution, l'état précédent S est transformé en le nouvel état S'.

Le comportement d'une primitive peut être défini par plusieurs règles. Dans ce cas, les conditions des règles d'une même primitive s'excluent mutuellement, rendant déterministe le choix de la règle à appliquer dans un contexte donné. Il n'y a pas de règle correspondant aux cas d'erreurs, on ne s'occupe que des appels *bien définis* selon la définition qui suit.

Définition 8.2.2 *Un appel de primitive p est dit bien défini dans un état du document S s'il est possible de trouver une (unique) règle pour p dont les prémisses sont vraies dans S.*

Définition 8.2.3 *Une séquence de primitives s est dite bien définie dans un état du document S si la première primitive de s est bien définie dans S et que chacune des primitives suivantes est bien définie dans l'état résultant de l'application de la précédente.*

En pratique, l'implantation utilisera les mécanismes à sa disposition pour gérer les cas d'erreur, par exemple (1) un schéma de compilation ou un typage ne permettant que l'appel dans des conditions correctes, (2) une vérification dynamique des prémisses, provoquant l'arrêt du programme en cas d'erreur, ou encore (3) une vérification dynamique et une levée d'exception, sans effectuer aucune modification du document afin que l'état reste valide.

8.2.4 Primitives d'accès

Ces primitives n'ont pas d'effets sur l'état, elles permettent simplement de parcourir le document et les données associées depuis le langage hôte.

Enfants On accède aux enfants d'un nœud via les primitives `children` et `child`.

`children` donne le nombre de nœuds enfants du nœud $\bullet$ qui lui est passé en paramètre. Pour ceci, elle accède à la liste des enfants de $\bullet$ dans T et en renvoie la longueur. Cette primitive, si elle est bien appelée avec un nœud, s'exécute toujours selon la règle (Children).

$$\texttt{children} : H^\bullet \to \mathit{Int}$$

$$(\textsc{Children})\frac{\bullet \in H^\bullet}{(H, L, T, P) \vdash \texttt{children}(\bullet) = \mathit{length}(T(\bullet)), (H, L, T, P)}$$

`child` renvoie un nœud enfant en fonction de sa position. Cette primitive s'exécute selon la règle (Child) si tout va bien, et selon la règle (Child-Unbound), renvoyant *nil*, si elle a reçu un indice négatif ou trop grand.

$$\texttt{child} : H^\bullet \times \mathit{Int} \to H^\bullet \cup \{\mathit{nil}\}$$

$$(\textsc{Child})\frac{\bullet \in H^\bullet \qquad 0 \leqslant i < \mathit{length}(T(\bullet))}{(H, L, T, P) \vdash \texttt{child}(\bullet, i) = \mathit{nth}(T(\bullet), i), (H, L, T, P)}$$

$$(\textsc{Child-Unbound})\frac{\bullet \in H^\bullet \qquad \neg(0 \leqslant i < \mathit{length}(T(\bullet)))}{(H, L, T, P) \ \vdash \ \texttt{child}(\bullet, i) = \mathit{nil}, (H, L, T, P)}$$

Racines L'ensemble des racines de la forêt peut être obtenu par la primitive `roots`. Pour ceci, on recherche tous les nœuds de H n'ayant pas de parent dans T. Suivant le modèle mémoire choisi, cette primitive pourrait ne pas être disponible, ou n'être utilisée que lors de la spécification du collecteur.

$$\texttt{roots} : \{\mathit{nil}\} \ \rightarrow \ \mathit{Enum}(H^\bullet)$$

$$(\textsc{Roots})\frac{}{(H, L, T, P) \ \vdash \ \texttt{roots}(\mathit{nil}) = \mathit{enum}(\{\bullet_n | \forall (\bullet_p, l) \in T, \bullet_n \notin l\}), (H, L, T, P)}$$

Propriétés On accède aux propriétés des objets par les primitives `properties` et `get`.

`properties` calcule le domaine des propriétés d'un nœud donné, c'est-à-dire la composante *Key* de tous les triplets de P dont la composante H est le nœud demandé.

$$\texttt{properties} : \ H \ \rightarrow \ \mathit{Enum}(\mathit{Key})$$

$$(\textsc{Properties})\frac{\circ \in H}{(H, L, T, P) \ \vdash \ \texttt{properties}(\circ) = \mathit{enum}(\{k | (\circ, k, v) \in P\}), (H, L, T, P)}$$

`get` peut s'exécuter selon la règle (Get) et renvoyer la valeur associée à un nœud et une clef dans P si elle existe. Si une telle valeur n'existe pas, `get` s'exécute selon la règle (Get-Unbound), et renvoie *nil*.

$$\texttt{get} : \ H \times \mathit{Key} \ \rightarrow \ (H \cup \mathit{Imm} \cup \{\mathit{nil}\})$$

$$(\textsc{Get})\frac{\exists (\circ, k, v) \in P}{(H, L, T, P) \ \vdash \ \texttt{get}(\circ, k) = v, (H, L, T, P)}$$

$$(\textsc{Get-Unbound})\frac{\nexists (\circ, k, v) \in P}{(H, L, T, P) \ \vdash \ \texttt{get}(\circ, k) = \mathit{nil}, (H, L, T, P)}$$

Étiquettes L'étiquette d'un nœud peut être récupérée par la primitive `tag`. Il n'y a pas de primitive pour modifier l'étiquette d'un nœud.

$$\texttt{tag} : \ H^\bullet \ \rightarrow \ \mathit{Tag}$$

$$(\textsc{Tag})\frac{(\bullet, t) \in L}{(H, L, T, P) \ \vdash \ \texttt{tag}(\bullet) = t, (H, L, T, P)}$$

8.2.5 Primitives à effet

Les primitives suivantes permettent de créer de nouveaux nœuds, de modifier la structure d'arbre et les propriétés.

Création d'objet Les objets noirs (resp. blancs) sont créés par $\texttt{create}^\bullet$ (resp. $\texttt{create}^\circ$).

$\texttt{create}^\bullet$ permet de créer un nœud, elle renvoie un nœud frais, c'est-à-dire une valeur de $\mathit{Loc} \times \{\bullet\}$ n'existant pas dans $H^\bullet$, et elle l'ajoute à la composante H de l'état de sortie. Elle met à jour les autres

composantes en conséquence : elle ajoute une liste d'enfants vide à T et l'étiquette demandée dans L. Elle crée implicitement une nouvelle racine.

$$\texttt{create}^\bullet : \mathit{Tag} \to H^\bullet$$

$$(\textsc{Create}^\bullet)\frac{t \in \mathit{Tag} \qquad \bullet \notin H}{(H, L, T, P) \vdash \texttt{create}^\bullet(t) = \bullet, (H \cup \{\bullet\}, L \cup \{(\bullet, t)\}, T \cup \{(\bullet, [\,])\}, P)}$$

$\texttt{create}^\circ$ permet de créer un objet blanc.

$$\texttt{create}^\circ : \{\mathit{nil}\} \to H^\circ$$

$$(\textsc{Create}^\circ)\frac{\circ \notin H}{(H, L, T, P) \vdash \texttt{create}^\bullet(\mathit{nil}) = \circ, (H \cup \{\circ\}, L, T, P)}$$

Structure d'arbre Les primitives `detach` et `bind` permettent respectivement de supprimer et rajouter un lien de parenté dans la structure d'arbre du document.

`detach` permet de supprimer le lien entre un nœud et son parent. Elle s'exécute selon la règle (Detach-1) en supprimant effectivement ce lien de T s'il existait, et s'exécute en laissant l'état intouché selon la règle (Detach-2) sinon.

$$\texttt{detach} : H^\bullet \to \{\mathit{nil}\}$$

$$(\textsc{Detach-1})\frac{\exists \bullet_p \in H^\bullet, \bullet_n \in T(\bullet_p)}{(H, L, T, P) \vdash \texttt{detach}(\bullet_n) = \mathit{nil}, (H, L, T \backslash \{(\bullet_p, l)\} \cup \{(\bullet_p, l - \bullet_n)\}, P)}$$

$$(\textsc{Detach-2})\frac{\nexists \bullet_p \in H^\bullet, \bullet_n \in T(\bullet_p)}{(H, L, T, P) \vdash \texttt{detach}(\bullet_n) = \mathit{nil}, (H, L, T, P)}$$

`bind` permet de lier à un nœud parent un nouvel enfant. Son comportement est différent si le nœud était déjà attaché ou non, comme expliqué au chapitre 7.

Si le nœud avait un parent, il en est détaché implicitement avant d'être lié à son nouveau parent afin de conserver la propriété d'arbre. Ce cas est décrit par la règle (Move). S'il n'avait pas de parent, il est simplement lié à son nouveau parent selon la règle (Attach). Dans tous les cas, le nœud est ajouté en tête de la liste d'enfants de son nouveau parent.

Cette distinction de cas permet d'empêcher le partage des nœuds dans T. La formation des cycles, quand à elle, est empêchée par la dernière prémisse de chacune des règles, qui empêche la liaison si elle entraîne la création d'un cycle.

$$\texttt{bind} : H^\bullet \times H^\bullet \to \{\mathit{nil}\}$$

$$(\textsc{Attach})\frac{\begin{array}{c}\bullet_p \in H^\bullet \qquad \bullet_n \in H^\bullet \\ \bullet_{p'} \in H^\bullet \Rightarrow \bullet_n \notin T(\bullet_{p'}) \\ \nexists \bullet_{l_i}, 0 \leqslant i \leqslant m, \bullet_{l_0} \in T(\bullet_n), \wedge \cdots \wedge \bullet_{l_{i+1}} \in T(\bullet_{l_i}), \wedge \cdots \wedge \bullet_{l_m} \in T(\bullet_p)\end{array}}{(H, L, T, P) \vdash \texttt{bind}(\bullet_p, \bullet_n) = \mathit{nil}, (H, L, T \backslash \{\bullet_p, T(\bullet_p)\} \cup \{\bullet_p, \bullet_n :: T(\bullet_p)\}, P)}$$

$$(\textsc{Move})\frac{\begin{array}{c}\bullet_p \in H^\bullet \qquad \bullet_n \in H^\bullet \\ \exists \bullet_{p'}, \bullet_n \in T(\bullet_{p'}) \\ \nexists \bullet_{l_i}, 0 \leqslant i \leqslant m, \bullet_{l_0} \in T(\bullet_n), \wedge \cdots \wedge \bullet_{l_{i+1}} \in T(\bullet_{l_i}), \wedge \cdots \wedge \bullet_{l_m} \in T(\bullet_p)\end{array}}{(H, L, T, P) \vdash \texttt{bind}(\bullet_p, \bullet_n) = \mathit{nil}, (H, L, T', P)}$$

$$\text{où } T' = T \backslash \{(\bullet'_p, T(\bullet'_p)), (\bullet_p, T(\bullet_p))\} \cup \{(\bullet'_p, T(\bullet'_p) - \bullet_n), (\bullet_p, \bullet_n :: T(\bullet_p))\}$$

Propriétés On modifie les propriétés (domaine et valeurs) des objets avec `set` et `unset`.

`set` permet d'affecter une propriété dans P, c'est-à-dire d'en ajouter une selon là règle (Set) ou bien d'en modifier une existante selon la règle (modify).

$$\texttt{set} : H \rightarrow \mathit{Key} \rightarrow (H \cup \mathit{Imm}) \rightarrow \{\mathit{nil}\}$$

$$(\text{Set})\frac{v \in H \cup \mathit{Imm} \quad \bullet \in H \quad \nexists v', (\bullet, k, v') \in P}{(H, L, T, P) \vdash \texttt{set}(\bullet, k, v) = \mathit{nil}, (H, L, T, P \cup (\bullet, k, v))}$$

$$(\text{Modify})\frac{v \in H \cup \mathit{Imm} \quad \exists v' \, (\bullet, k, v') \in P}{(H, L, T, P) \vdash \texttt{set}(\bullet, k, v) = \mathit{nil}, (H, L, T, P \backslash \{(\bullet, k, v')\} \cup (\bullet, k, v))}$$

`unset` supprime la liaison d'un nœud/clef à une valeur dans P. Elle s'exécute selon la règle (Unset-1) en supprimant effectivement cette liaison de P si elle existait, et s'exécute en laissant l'état intouché selon la règle (Unset-2) sinon.

$$\texttt{unset} : H \rightarrow \mathit{Key} \rightarrow \{\mathit{nil}\}$$

$$(\text{Unset-1})\frac{\exists \, (\bullet, k, v) \in P}{(H, L, T, P) \vdash \texttt{unset}(\bullet, k, v) = \mathit{nil}, (H, L, T, P \backslash \{(\bullet, k, v)\})}$$

$$(\text{Unset-2})\frac{\nexists \, (\bullet, k, v) \in P}{(H, L, T, P) \vdash \texttt{unset}(\bullet, k, v) = \mathit{nil}, (H, L, T, P)}$$

`Primitive`	Type des paramètres $\rightarrow$ Type de retour	Règles
`children`	$H^\bullet \rightarrow \mathit{Int}$	Children
`child`	$H^\bullet \times \mathit{Int} \rightarrow H^\bullet \cup \{\mathit{nil}\}$	Child, Child-Unbound
`roots`	$\{\mathit{nil}\} \rightarrow \mathit{Enum}(H^\bullet)$	Roots
`properties`	$H \rightarrow \mathit{Enum}(\mathit{Key})$	Properties
`get`	$H \times \mathit{Key} \rightarrow H \cup \mathit{Imm} \cup \{\mathit{nil}\}$	Get, Get-Unbound
`tag`	$H^\bullet \rightarrow \mathit{Tag}$	Tag
`create`$^\bullet$	$\mathit{Tag} \rightarrow H^\bullet$	Create$^\bullet$
`create`$^\circ$	$\{\mathit{nil}\} \rightarrow H^\circ$	Create$^\circ$
`detach`	$H^\bullet \rightarrow \{\mathit{nil}\}$	Detach-1, Detach-2
`bind`	$H^\bullet \times H^\bullet \rightarrow \{\mathit{nil}\}$	Attach, Move
`set`	$H \times \mathit{Key} \times H \cup \mathit{Imm} \rightarrow \{\mathit{nil}\}$	Set, Modify
`unset`	$H \times \mathit{Key} \rightarrow \{\mathit{nil}\}$	Set, Modify

Figure 8.1: *fDOM* : tableau récapitulatif

8.2.6 Conservation de la validité

En pratique, nous voulons être sûrs qu'à tout instant au cours de l'exécution d'un programme manipulant un document, l'état de ce dernier reste valide.

Nous avons expliqué alors que l'état formel de *fDOM* est défini comme une structure ensembliste simple, et avons encodé les propriétés à conserver au sein d'une notion de validité, vérifiée par un prédicat sur ces structures.

L'idée de ce découplage est de n'avoir à faire le lien entre l'état concret de l'implantation et l'état formel qu'au niveau de la structure, et de conserver implicitement la validité.

Pour ceci, on cherche à montrer que chaque règle conserve la validité, et par induction que la validité est conservée au cours du programme. Bien entendu, cette proposition n'est correcte que dans le cas ou la séquence est bien définie (c'est-à-dire que chaque appel de primitive est bien défini dans l'état du document avant son appel).

Cette conservation de la validité s'exprime et se prouve donc comme suit :

Théorème 8.2.1 *Toute séquence de primitives, si elle est bien définie et appliquée à un état de fDOM valide, aboutit à un état de fDOM valide.*

Preuve Par induction sur la séquence, la proposition est vraie si et seulement si chaque appel de primitive conserve la validité. Chaque appel étant décrit par une unique règle, on procède par cas sur l'ensemble des règles. Chaque appel bien défini conserve la validité si chaque règle conserve la validité. On prouve donc que chaque règle conserve les trois points de la définition de validité : (1) forme d'application de L, (2) forme d'arbre de T, et (3) bonne définition de P.

- `child`, `children`, `roots`, `tag`, `get`, `properties` et `detach`/DETACH-2 ne modifiant pas l'état, elle conservent trivialement sa validité.
- `set` et `unset` ne modifient pas l'arbre, elles doivent donc seulement vérifier (3), ce qui est direct si leurs prémisses sont satisfaites.
- `create`$^\circ$ ne touche qu'à H°
- `create`$^\bullet$ modifie $H^\bullet$, et modifie bien L en conséquence. Pour T, l'ajout d'une racine sans enfant permet de vérifier (2).
- `detach`/DETACH-1 supprime un lien dans T. Elle ne change pas la propriété vérifiant l'absence de cycle (2d), ni celles sur le partage (2b, 2c). Elle conserve la forme d'application (2a) puisque l'enfant n'est pas touché dans T, et que le père y est toujours présent avec une liste d'enfants diminuée.
- `bind`/ATTACH La dernière prémisse empêche que le nouveau lien créé ne fabrique de cycle, assurant (2d) et (2c). La vérification est faite que le nœud n'avait pas de parent, permettant d'assurer que le lien créé n'ajoutera pas de partage (2b). On conserve donc bien (2), et les autres trivialement.
- `bind`/MOVE Ici, il faut noter, en plus des arguments précédents, qu'on vérifie qu'un unique lien existe, et que celui-ci est supprimé avant d'ajouter le nouveau, assurant qu'aucun partage n'est introduit. □

8.3 Récapitulatif

Dans ce chapitre, nous avons donné une définition formelle du document impératif, sous forme d'un ensemble restreint de primitives. L'annexe C montre que ces primitives sont faciles à implanter, en partant de zéro comme en utilisant un DOM existant au travers de deux implantations.

Au final, nous obtenons une abstraction du document impératif se comportant selon le modèle du DOM des navigateurs et permettant d'exprimer l'essentiel des manipulations, tout en conservant une taille restreinte, et une bonne facilité d'implantation.

On peut déjà voir apparaître plusieurs points intéressants de cette formalisation pour la conception d'un langage de programmation Web :

1. Elle facilite la cohérence du langage, par un modèle homogène sur les différentes parties.
2. Elle permet de réfléchir formellement sur le document et son traitement par le langage.

3. Elle définit une interface claire et indépendante du langage et de la plate-forme sous-jacente, ce qui peut être utilisé pour rendre plus prévisibles les programmes et faciliter le déverminage.
4. Elle est raisonnablement simple pour servir de base à des expérimentations de plus haut niveau.

Ce dernier point sera mis en pratique dès le chapitre suivant, où nous proposons notre sémantique alternative pour le document impératif, sous forme d'une variante de *fDOM*.

Notations :

- *List*(S) désigne l'ensemble des listes d'éléments d'un ensemble S,
- $[\,]$ est la liste vide,
- $h :: t$ la liste dont le premier élément est h et la suite de la liste t,
- en résumé, *List*$(S) = \{[\,]\} \cup \{h :: t \mid h \in S \wedge t \in \textit{List}(S)\}$.

Opérations :

- Prédicat de vacuité :
  $\begin{cases} \textit{empty}([\,]) = T \\ \neg\textit{empty}(h :: t) = F \end{cases}$.
- Prédicat d'appartenance :
  $\begin{cases} e \notin [\,] \\ e \in h :: t \Leftrightarrow e = h \vee e \in t \end{cases}$
- Prédicat d'unicité des éléments :
  $\begin{cases} \textit{unique-list}([\,]) \\ \textit{unique-list}(h :: t) \text{ si } h \notin t \wedge \textit{unique-list}(t) \end{cases}$
- Suppression d'un élément :
  $\begin{cases} [\,] - e &= [\,] \\ (h :: t) - e &= t - e \text{ si } h = e \\ (h :: t) - e &= h :: (t - e) \text{ sinon} \end{cases}$
- Longueur :
  $\begin{cases} \textit{length}([\,]) &= 0 \\ \textit{length}(h :: t) &= 1 + \textit{length}(t) \end{cases}$
- Accès au $n^{\text{ième}}$ élément :
  $\begin{cases} \textit{nth}(h :: t, 0) &= h \\ \textit{nth}(h :: t, n > 0) &= \textit{nth}(t, n-1) \end{cases}$
- Itérateurs :
  $\begin{cases} \textit{map}(f, [\,]) &= [\,] \\ \textit{map}(f, h :: t) &= f(h) :: \textit{map}(f, t) \end{cases}$
  $\begin{cases} \textit{foldl}(f, r, [\,]) &= r \\ \textit{foldl}(f, r, h :: t) &= \textit{foldl}(f, f(r, h), t) \end{cases}$
  $\begin{cases} \textit{foldr}(f, [\,], r) &= r \\ \textit{foldr}(f, h :: t, r) &= f(h, \textit{foldr}(f, t, r)) \end{cases}$

Listes associatives :

- Une liste $l \in \textit{List}(K \times V)$ associe des éléments de K (appelés clefs ou noms) à des éléments de V (appelés valeurs).
- La fonction *find*(k, l) renvoie la valeur associée à la première apparition de k dans l
  $\begin{cases} \textit{find}(k, (k, v) :: t) &= v \\ \textit{find}(k, (k', v) :: t) &= \textit{find}(k, t) \end{cases}$
- On utilisera la notation $l(x)$ pour *find*(x, l).
- Le domaine d'une liste associative est l'ensemble de ses clefs : *dom*$(l) = \{k \mid (k, v) \in l\}$.

FIGURE 8.2: Définition des listes

9 ^{c}*DOM*, modèle alternatif du document impératif

Comme décrit de façon grossière au chapitre 7, nous proposons de définir une sémantique alternative, par copie, aux opérations sur le DOM. Le problème est alors de savoir quelle sous-partie du DOM est à copier en même temps qu'un nœud du document.

Le chapitre précédent modélisait le DOM en présentant une définition plus simple de document impératif et sa sémantique.

Dans ce chapitre, nous définissons ^{c}*DOM*, un modèle alternatif copiant du document impératif. Il s'agit d'une extension de *f*^{f}*DOM* permettant de rattacher des données annexes à un nœud, et disposant d'un mécanisme de copie, utilisé implicitement pour empêcher les déplacements implicites, et utilisable explicitement par le programmeur.

Nous commençons par proposer $^{\varepsilon}$*DOM*, une seconde sémantique alternative, plus simple et plus portable, qui interdit les opérations cassant le typage avec des exceptions.

9.1 $^{\varepsilon}$*DOM*, un DOM avec sémantique par exceptions

Dans cette variante de *f*^{f}*DOM*, `bind` refusera tout simplement d'attacher un nœud s'il avait déjà un parent. La primitive indique si elle a réussi en renvoyant un booléen $\in$ *Bool* $= \{T, F\}$, et le langage hôte pourra utiliser son mécanisme d'exceptions dans le cas où la primitive a échoué.

On modifie seulement la primitive `bind`, en supprimant la règle (Move), et en ajoutant les règles (Attach-Fail).

$$\texttt{bind} : H^{\bullet} \times H^{\bullet} \rightarrow \mathit{Bool}$$

$$(\textsc{Attach})\frac{\begin{array}{c}\bullet_p \in H \qquad \bullet_n \in H \\ \bullet_{p'} \in H^{\bullet} \Rightarrow \bullet_n \notin T(\bullet_{p'}) \\ \nexists \bullet_{l_i}, 0 \leqslant i \leqslant m, \bullet_{l_0} \in T(\bullet_n), \wedge \cdots \wedge \bullet_{l_{i+1}} \in T(\bullet_{l_i}), \wedge \cdots \wedge \bullet_{l_m} \in T(\bullet_p)\end{array}}{(H, L, T, P) \vdash \texttt{bind}(\bullet_p, \bullet_n) = T, (H, L, T \backslash \{(\bullet_p, T(\bullet_p)\} \cup \{(\bullet_p, \bullet_n :: T(\bullet_p)\}, P)}$$

$$(\textsc{Attach-Fail-1})\frac{\begin{array}{c}\bullet_p \in H \qquad \bullet_n \in H \\ \exists \bullet_{p'}, \bullet_n \in T(\bullet_{p'})\end{array}}{(H, L, T, P) \vdash \texttt{bind}(\bullet_p, \bullet_n) = F, (H, L, T, P)}$$

$$(\textsc{Attach-Fail-2})\frac{\begin{array}{c}\bullet_p \in H \qquad \bullet_n \in H \\ \exists \bullet_{l_i}, 0 \leqslant i \leqslant m, \bullet_{l_0} \in T(\bullet_n), \wedge \cdots \wedge \bullet_{l_{i+1}} \in T(\bullet_{l_i}), \wedge \cdots \wedge \bullet_{l_m} \in T(\bullet_p)\end{array}}{(H, L, T, P) \vdash \texttt{bind}(\bullet_p, \bullet_n) = F, (H, L, T, P)}$$

^{c}*DOM* versus $^{\varepsilon}$*DOM* En pratique, le changement introduit par $^{\varepsilon}$*DOM* est très simple, et permet de conserver le modèle actuel du DOM des navigateurs actuel tout en ajoutant un mécanisme de détection d'erreur dynamique. C'est une idée simple mais qui permet déjà de faciliter la mise au point en signalant les erreurs au plus tôt lors de l'exécution. Comme nous avons vu au chapitre 7, d'autres travaux vont plus loin en proposant une approche statique à l'interdiction de déplacements implicites.

Cependant, nous avons vu au chapitre 7 que le déplacement implicite cassait aussi certains programmes effectuant des constructions fonctionnelles de document, que nous voulons conserver.

Le modèle avec copies implicite ^{c}*DOM* permet de conserver la création fonctionnelle bien typée et sans surprise de documents. D'autre part nous verrons, dans ce chapitre et surtout à la définition du langage du chapitre 10, que cette copie implicite est compréhensible et adaptable par le programmeur.

9.2 ^{c}*DOM*, un DOM avec sémantique par copie

Pour cette variante, nous introduisons, comme expliqué au chapitre 7, une notion de portée de nœud. Pour rappel, la problématique de base est la suivante : lors le la copie en profondeur d'un nœud, on veut aussi copier les données annexes associées à ce nœud. La solution proposée est de définir dans le langage hôte une structure syntaxique délimitée pour les nœuds. On considère alors que les données annexes à copier sont celles construites au sein de la portée lexicale du nœud.

Pour implanter cette solution, nous avons donc besoin de prendre en compte dans la spécification ces informations d'appartenance des objets à une portée. Nous étendons donc dans un premier temps l'état en conséquence.

Mais il faut alors aussi faire le lien entre les informations de portée de la spécification du document et la structure syntaxique du langage hôte, ainsi qu'introduire le mécanisme de copie utilisant ces informations. Dans un second temps, nous ajoutons donc au jeu de primitives de *fDOM* de nouvelles primitives dédiées à la gestion de ces informations de portée et à la copie. Nous adaptons bien entendu au passage les primitives de *fDOM* en conséquence.

9.2.1 État du document

Le quadruplet (H, L, T, P) de *fDOM* est étendu comme suit :

$$
{}^{c}DOM \supseteq \left\{ (H, L, T, P, S, s) \;\middle|\; \begin{array}{l} H = H^{\bullet} \cup H^{\circ} \subseteq Loc \times \{\bullet, \circ\} \\ L \subseteq H^{\bullet} \times Tag \\ T \subseteq H^{\bullet} \times List(H^{\bullet}) \\ P \subseteq H \times Key \times (H \cup Imm) \\ S \subseteq H^{\bullet} \times H \\ s \in List(H^{\bullet}) \end{array} \right\}
$$

On rajoute S, l'ensemble des informations de portée, et s la pile des portées ouvertes.

- S (mnémonique pour *scopes*) lie chaque nœud à l'ensemble des objets créés dans sa portée. Puisque la composante S représente une information de portée lexicale, elle a une forme de forêt. En effet, un objet ne peut se retrouver dans la portée de deux nœuds dont les définitions ne sont pas imbriquées, empêchant la possibilité de partage. De même un cycle dans S n'aurait pas de sens en terme de portée lexicale.
- s (mnémonique pour *stack*) est la pile des nœuds dont les portées sont ouvertes. Elle sert, lors de la création d'un nouvel objet, à connaître le nœud dans la portée duquel il se trouve.

Intuitivement, le principe de fonctionnement est le suivant.

1. Quand un nouveau nœud est créé, sa portée dynamique est ouverte. Concrètement, le nœud est pour cela empilé sur s.
2. Quand un objet (blanc ou noir) est créé, il est attaché dans S au nœud dont la portée a été ouverte le plus récemment dans s.
3. La portée d'un nœud est explicitement fermée par une nouvelle primitive dédiée (qui dépile S).
4. La portée d'un nœud peut être ouverte (puis fermée) à nouveau, pour ajouter des éléments à copier après la création du nœud.

	Primitive	Type des paramètres $\rightarrow$ Type de retour	Règles
	children	$H^\bullet \rightarrow Int$	Children
	child	$H^\bullet \times Int \rightarrow H^\bullet \cup \{nil\}$	Child, Child-Unbound
	roots	$\{nil\} \rightarrow Enum(H^\bullet)$	Roots
	properties	$H \rightarrow Enum(Key)$	Properties
	get	$H \times Key \rightarrow H \cup Prim \cup \{nil\}$	Get, Get-Unbound
	tag	$H \rightarrow Tag$	Tag
*	create$^\bullet$	$Tag \rightarrow H^\bullet$	Create$^\bullet$
*	create$^\circ$	$\{nil\} \rightarrow H^\circ$	Create$^\circ$
*	close	$\{nil\} \rightarrow \{nil\}$	Close-Scope
*	reopen	$H^\bullet \rightarrow \{nil\}$	Reopen-Scope
	detach	$H^\bullet \rightarrow \{nil\}$	Detach-1, Detach-2
*	bind	$H^\bullet \times H^\bullet \rightarrow \{nil\}$	Attach, Move
	set	$H \times Key \times H \cup Prim \rightarrow \{nil\}$	Set, Modify
	unset	$H \times Key \rightarrow \{nil\}$	Set, Modify

Figure 9.1: ^{c}DOM : tableau récapitulatif

Ainsi, dans le langage hôte, on pourra définir une construction syntaxique représentant la portée lexicale d'un nœud, et S reportera cette information de portée statique à l'exécution. La ré-ouverture d'un nœud sera utilisable pour faire en sorte que les entités de code associées à un nœud et appelables depuis l'extérieur définies par le langage (appels de méthodes, rattrapeurs d'exceptions, etc.) soient considérées à l'intérieur de la portée.

Validité Comme pour ^{f}DOM, on définit ^{c}DOM comme l'ensemble des sextuplets de l'ensemble défini ci-dessus valides, $^{c}DOM = \{(H, L, T, P, S, s) \text{ valide}\}$ où la définition de validité est étendue comme suit.

Définition 9.2.1 *Un sextuplet (H, L, T, P, S, s) est dit valide, si et seulement si :*

1. *(H, L, T, P) est valide selon la définition de ^{f}DOM*
2. *S est une forêt :*
   - *(a) Pas de partage :*
     $\forall (\bullet_p, \bullet_c) \in S, (\bullet'_p, \bullet'_c) \in S, \bullet_c = \bullet'_c \Rightarrow \bullet_p = \bullet'_p$
   - *(b) Pas de cycle :*
     $\forall s \in \mathbb{N}, \forall \{(\bullet_{p_i}, \bullet_{c_i}) | 0 \leqslant i \leqslant s\} \subseteq S$ *tel que* $\forall i, 0 \leqslant i < s, \bullet_{c_i} = \bullet_{p_{i+1}}$ *on a* $\bullet_{c_s} \neq \bullet_{p_0}$

9.2.2 Primitives et règles sémantiques

Cette partie introduit les nouvelles primitives liées à la copie, et ajuste les règles des autres primitives en conséquence.

On ne répétera pas les règles qui sont identiques à celles de ^{f}DOM, aux éléments de l'état inchangés supplémentaires près. L'ensemble est récapitulé dans le tableau de la figure 9.1, les primitives ajoutées ou ayant changé sont marquées d'une étoile.

D'autre part, on se permettra dans la suite d'étendre implicitement ^{f}DOM et $^{\varepsilon}DOM$ de ces nouvelles primitives (qui ne feront rien), pour que les trois modèles aient la même interface. Ainsi on pourra remplacer une sémantique par une autre suivant le besoin.

Création Les règles de création changent et sont maintenant différentes pour les objets noirs et blancs.

`create`$^\bullet$ crée un nœud noir, et ouvre sa portée. Si la portée d'un nœud était ouverte, le nouveau nœud est placé dans sa portée selon la règle (Create$^\bullet$). Sinon, le nœud n'est mis sous la portée d'aucun selon la règle (Create-Root$^\bullet$).

$$\texttt{create}^\bullet : \ \mathit{Tag} \ \rightarrow \ H^\bullet$$

$$(\textsc{Create}^\bullet)\frac{\bullet_n \notin H}{(H, L, T, P, S, \bullet_p :: s) \ \vdash \ \texttt{create}^\bullet(\mathit{nil}) = \bullet_n, (H \cup \{\bullet_n\}, L, T, P, S \cup \{(\bullet_p, \bullet_n)\}, \bullet_n :: \bullet_p :: s)}$$

$$(\textsc{Create-Root}^\bullet)\frac{\bullet_n \notin H}{(H, L, T, P, S, [\,]) \ \vdash \ \texttt{create}^\bullet(\mathit{nil}) = \bullet_n, (H \cup \{\bullet_n\}, L, T, P, S, \bullet_n :: [\,])}$$

`create`$^\circ$ crée un nœud blanc qui peut peut là aussi être mis ou non sous la portée d'un nœud noir selon la règle (Create$^\circ$) ou (Create-Root$^\circ$).

$$\texttt{create}^\circ : \ \mathit{Tag} \ \rightarrow \ H^\circ$$

$$(\textsc{Create}^\circ)\frac{\circ_n \notin H}{(H, L, T, P, S, \bullet_p :: s) \ \vdash \ \texttt{create}^\circ(\mathit{nil}) = \circ_n, (H \cup \{\circ_n\}, L, T, P, S \cup \{(\bullet_p, \circ_n)\}, \bullet_p :: s)}$$

$$(\textsc{Create-Root}^\circ)\frac{\circ_n \notin H}{(H, L, T, P, S, [\,]) \ \vdash \ \texttt{create}^\circ(\mathit{nil}) = \circ_n, (H \cup \{\circ_n\}, L, T, P, S, [\,])}$$

La nouvelle primitive `close` permet de sortir de la dernière portée ouverte.

$$\texttt{close} : \ \{\mathit{nil}\} \ \rightarrow \ \{\mathit{nil}\}$$

$$(\textsc{Close-Scope})\frac{}{(H, L, T, P, S, \bullet_p :: s) \ \vdash \ \texttt{close}(\mathit{nil}) = \mathit{nil}, (H, L, T, P, S, s)}$$

Réouverture On ajoute une primitive `reopen` permettant d'exécuter les primitives suivantes dans la portée d'un nœud existant, comme expliqué dans la présentation de la composante S de l'état.

$$\texttt{reopen} : \ H^\bullet \ \rightarrow \ \{\mathit{nil}\}$$

$$(\textsc{Reopen-Scope})\frac{\bullet_p \in H^\bullet}{(H, L, T, P, S, s) \ \vdash \ \texttt{reopen}(\bullet_p) = \mathit{nil}, (H, L, T, P, S, \bullet_p :: s)}$$

Opération de copie La nouvelle primitive `copy` permet de copier explicitement un nœud en profondeur. Elle est définie formellement ici de façon déclarative, on donnera section 9.2.5 un algorithme de copie respectant cette spécification dans la section sur l'implantation.

La définition de la règle (Copy) utilise une association $C^{\bullet}$ entre les nœuds originaux et les nœuds copiés qui sera définie juste après. Elle ajoute à H les nouveaux nœuds, et met à jour les composantes T et P, en reliant les enfants et les valeurs des propriétés vers les copies si elles existent, ou vers les nœuds et données annexes originales dans H si ceux-ci n'ont pas été copiés.

$$\texttt{copy} : \ H^\bullet \ \rightarrow \ H^\bullet$$

$$\frac{\begin{array}{c}H' = \{\bullet | (\cdot, \bullet) \in C^\bullet\} \cup \{\circ | (\cdot, \circ) \in C^\circ\} \cup H \\ L' = \{(\bullet', l) | (\bullet, \bullet') \in C^\bullet, (\bullet, l) \in L\} \cup L \\ T' = \{(\bullet', l') | (\bullet', \bullet) \in C^\bullet, l = T(\bullet), l' = \mathit{map}(\mathit{rebind}, l)\} \cup T \\ P' = \{(\bullet', k, v') | (\bullet', \bullet) \in C^\bullet, v = P(\bullet, k), v' = \mathit{rebind}(v)\} \cup P\end{array}}{(H, L, T, P, S, s) \vdash \mathtt{copy}(\bullet_n) = A^\bullet(\bullet_n), (H', L', T', P', S', s)} \text{(Copy)}$$

Où la fonction de re-liaison *rebind* se comportant comme une association sur son domaine de définition, et une identité sur son dual dans H est :

$$\mathit{rebind}(\bullet_o \in H) = \begin{cases} \bullet_c \text{ si } (\bullet_o, \bullet_c) \in C^\bullet \\ \bullet_o \text{ sinon} \end{cases}$$

Et la spécification des copies $C^\bullet$ est décomposée en deux temps, comme suit. Intuitivement, il s'agit de collecter d'abord tous les objets à copier, puis d'en extraire le sous-ensemble atteignable depuis la racine à copier, afin de réaliser exactement les copies nécessaires. La collecte initiale se fait en considérant que les objets dans la portée de chaque nœud à copier ainsi que ses enfants dans l'arbre sont aussi à copier.

1. $D^\bullet = D^\bullet \cup D^\circ$ représente l'ensemble des nœuds sous la portée du nœud à copier. On donne une écriture sous forme de collecte récursive, en parcourant l'arbre sous la portée, et en ajoutant à chaque fois les nœuds noirs dans la portée des nœuds déjà collectés.

$$\begin{array}{l} D^\bullet = \bigcup_{n \in \mathbb{N}} \left(D^\bullet_n = \left\{\bullet_c \middle| (\bullet_p, \bullet_c) \in S \vee \bullet_c \in T(\bullet_p), \bullet_p \in D^\bullet_{n-1}\right\}\right) \text{ avec } D^\bullet_0 = \{\bullet_n\} \\ D^\circ = \{\circ_c | (\bullet_p, \circ_c) \in S, \bullet_p \in D^\bullet\} \end{array}$$

2. $A^\bullet = A^\bullet \cup A^\circ$ représente le sous-ensemble atteignable, de $D^\bullet$, par l'arbre ou les propriétés. On donne là encore une écriture sous forme de parcours de l'arbre et des propriétés, tant que les chemins parcourus restent dans la portée.

$$A^\bullet = \bigcup_{n \in \mathbb{N}} \left(A^\bullet_n = \left\{\bullet_c \middle| (\bullet_p, k, \bullet_c) \in P \vee \bullet_c \in T(\bullet_p), \bullet_p \in A^\bullet_{n-1}, \bullet_c \in D^\bullet\right\}\right) \text{ avec } A^\bullet_0 = \{\bullet_n\}$$

Finalement, $C^\bullet = C^\bullet \cup C^\circ$ donne les associations entre ces nœuds et des copies fraîches.

$$\begin{array}{l} C^\bullet = \{(\bullet, \bullet') | \bullet \in A^\bullet, \bullet' \notin H^\bullet \text{ (nœud frais)}\} \\ C^\circ = \{(\circ, \circ') | \circ \in A^\circ, \circ' \notin H^\circ \text{ (objet frais)}\} \end{array}$$

Copies implicites On utilise maintenant la primitive de copie lors de l'appel à `bind`, lorsque le nœud était déjà attaché à un parent. Ainsi un nœud ne disparaîtra plus d'un endroit de l'arbre lorsqu'on l'ajoutera ailleurs. Notons néanmoins que ce comportement est toujours possible à programmer grâce à la présence de la primitive `detach`, mais maintenant il ne peut se produire implicitement.

$$\mathtt{bind} : H^\bullet \times H^\bullet \to \{\mathit{nil}\}$$

$$\frac{\begin{array}{c}\bullet_p \in H^\bullet \qquad \bullet_n \in H^\bullet \\ \bullet_{p'} \in H^\bullet \Rightarrow \bullet_n \notin T(\bullet_{p'}) \\ \nexists \bullet_{l_i}, 0 \leqslant i \leqslant m, \bullet_{l_0} \in T(\bullet_n), \wedge \cdots \wedge \bullet_{l_{i+1}} \in T(\bullet_{l_i}), \wedge \cdots \wedge \bullet_{l_m} \in T(\bullet_p)\end{array}}{(H, L, T, P, S, s) \vdash \mathtt{bind}(\bullet_p, \bullet_n) = \mathit{nil}, (H, L, T\left[\bullet_p \to \bullet_n :: T(\bullet_p)\right], P)} \text{(Attach)}$$

$$\frac{\begin{array}{c}\bullet_p \in H^\bullet \qquad \bullet_n \in H^\bullet \\ \exists \bullet_{p'}, \bullet_n \in T(\bullet_{p'}) \\ (H, L, T, P, S, s) \vdash \mathtt{copy}(\bullet_n) = \bullet_{n'}, (H', T', P', S', s)\end{array}}{(H, L, T, P, S, s) \vdash \mathtt{bind}(\bullet_p, \bullet_n) = \mathit{nil}, (H', L', T'\left[\bullet_p \to \bullet_{n'} :: T'(\bullet_p)\right], P', S', s)} \text{(Attach-Copy-1)}$$

$$\text{(Attach-Copy-2)}\ \frac{\begin{array}{c}\bullet_p \in H^\bullet \quad \bullet_n \in H^\bullet \\ \exists \bullet_{l_i}, 0 \leqslant i \leqslant m, \bullet_{l_0} \in T(\bullet_n), \wedge \cdots \wedge \bullet_{l_{i+1}} \in T(\bullet_{l_i}), \wedge \cdots \wedge \bullet_{l_m} \in T(\bullet_p) \\ (H, L, T, P, S, s) \ \vdash\ \mathtt{copy}(\bullet_n) = \bullet_{n'}, (H', T', P', S', s)\end{array}}{(H, L, T, P, S, s) \ \vdash\ \mathtt{bind}(\bullet_p, \bullet_n) = \mathit{nil}, (H', L', T'\left[\bullet_p \to \bullet_{n'} :: T'(\bullet_p)\right], P', S', s)}$$

9.2.3 Conservation de la validité

On définit la notion de structure suivante, que l'on utilisera dans la preuve de conservation de validité, et qui participera aussi à la correction du système de types.

Définition 9.2.2 *On dit que deux nœuds ont la même structure, si ils ont :*

1. *les mêmes étiquettes,*
2. *le même nombre de fils,*
3. *et les mêmes propriétés,*
4. *et que pour chaque propriété (resp. numéro de fils), la valeur associée (resp. le nœud fils lié) ont récursivement la même structure.*

On veut alors assurer le lemme suivante :

Lemme 9.2.1 *La copie d'un nœud, effectuée via la primitive* copy *en respecte la structure.*

Preuve Cette preuve découle directement de la définition de la règle (Copy) :

1. L' est calculée en associant aux copies les étiquettes des nœuds originaux associés dans L.
2. De la même façon, dans T', la liste des enfants de chaque copie est obtenue avec l'appel de *map*, qui conserve la longueur, sur la liste originale.
3. De même, la définition du calcul de P' énumère tous les noms de propriétés (par la variable d'énumération k) de l'objet original associé à chaque objet copié.
4. Les nœuds et données annexes sont tous traités de la même façon, donc les trois propriétés précédentes sont vraies pour tout nœud. La récursion se montre alors simplement.
   Pour chaque nœud copié, puis pour chaque nœud enfant et objets associés à une propriété, il y a deux possibilités :
   - soit c'est un original, auquel cas il a bien la même structure que lui-même,
   - soit c'est une copie, auquel cas il à la même structure que son original par induction. □

Théorème 9.2.1 *(identique à 8.2.1, pour cDOM au lieu de fDOM) Toute composition séquentielle de primitives, si elle est bien définie et appliquée à un état de cDOM valide, aboutit à un état de cDOM valide.*

Preuve Sur chacune des propriétés :

1. (état de *fDOM* valide) Sur toutes les primitives sauf copy et bind, celles-ci sont soit inchangées par rapport à *fDOM*, soit ne modifient pas le sous ensemble correspondant à *fDOM*. bind est similaire au bind de *fDOM* à l'appel de copy près, il reste donc seulement à montrer (1) dans le cas copy pour montrer (1) pour toutes les primitives.
2. (S est une forêt) Là encore, nous laissons le cas de copy pour la fin. Pour le reste des primitives, sur la couleur du nœud :
   - Les objets blancs ne peuvent se trouver qu'à droite, il n'y a donc pas de possibilité de créer de cycle avec, montrant (2b). Ils sont introduits dans S uniquement par create$^\circ$/Create$^\circ$, qui y ajoute un objet frais. Pour qu'il y ait partage, il faudrait ajouter un objet déjà ajouté précédemment, ce qui est impossible par définition, montrant (2a).

- De la même façon, Les nœuds sont introduits dans S uniquement par `create`$^{\bullet}$/Create$^{\bullet}$, interdisant le partage.
- Montrons maintenant qu'une chaîne cyclique noire ne peut exister dans S, par induction sur la taille de la chaîne :
  - Toute chaîne cyclique de taille minimale 2 a la forme $\{(\bullet_p, \bullet_c), (\bullet_c, \bullet_p)\}$.
    Le second couple est de la forme $(\bullet_c, \bullet_?)$, un couple de cette forme ne peut être ajouté par la règle (Create$^{\bullet}$), où $\bullet_c$ est le sommet de s. Or, le sommet de s n'est changé que par les règles (Create$^{\bullet}$), (Create-Root$^{\bullet}$) et (Reopen), et dans chacune de ces règles, le nouveau sommet est soit frais, et ajouté en même temps à $H^{\bullet}$, soit déjà existant dans $H^{\bullet}$. Puisque l'ajout de l'élément $\bullet_c$ à $H^{\bullet}$ est fait uniquement par (create$^{\bullet}$), qui crée en même temps un unique couple $(\bullet_?, \bullet_c)$, on peut dire qu'un couple de la forme $(\bullet_c, \bullet_?)$ ne peut être ajouté qu'*après* un couple de la forme $(\bullet_?, \bullet_c)$. Il en résulte que si ces deux couples apparaissent dans S, le second couple doit avoir été ajouté avant le premier.
    De façon symétrique, sur $\bullet_p$, le premier couple doit avoir été ajouté avant le second. Il est donc impossible causalement que S contienne une chaîne de taille 2.
  - On montrerait de la même façon pour pour toute chaîne non cyclique de taille n, ajouter le dernier maillon pour former un cycle de taille $n+1$ cause une impossibilité causale .

Il reste alors à montrer ces trois propriétés pour copy.

On voit dans la définition de copy que les éléments X' (où $X \in \{H, L, T, P, S\}$) de l'état de sortie sont tous créés sous la forme $X' = X^{+} \cup X$, où X est l'élément original. Pour montrer la validité de cet état de sortie, on procède en deux étapes.

1. On montre d'abord que la validité est stable par union disjointe. On considère alors l'état virtuel $(H^{+}, L^{+}, T^{+}, \emptyset, S^{+}, [\,])$ et on montre qu'il est valide. On obtient alors un état intermédiaire, valide car résultat de l'union de deux états valides.
2. Puis on ajoute $P+$, ce qui nous donne l'état de sortie attendu, et en montrant que cet ajout ne change pas la validité, on a montré que copy conserve la validité.

La stabilité par union disjointe se montre facilement : l'union de deux forêts reste une forêt, donc $T \cup T^{+}$ et $S \cup S^{+}$ le sont et les forêts résultantes sont bien construites d'objets valides puisqu'on a fait l'union des tas $H \cup H^{+}$. Le premier point se montre alors en observant que H^{+}, L^{+}, T^{+} et S^{+} sont des copies disjointe d'une sous partie de l'état original, en en respectant la structure (avec la proposition 9.2.1), donc la validité. Le second point est plus facile à montrer, puisque l'élément P n'a pas de restriction de structure, il y a là encore seulement à observer que P' ne porte bien que sur des éléments de H'. □

9.2.4 Implantation de $^{\varepsilon}DOM$

L'implantation de $^{\varepsilon}DOM$ est plutôt simple, le listing suivant modifie l'implantation OCaml de ^{f}DOM donnée dans l'annexe C.

```
1 (* redéfinition de bind *)
2 let bind dom p n =
3   match p with
4     | Black (_,_, links) ->
5         begin match find_parent dom n with
6           | None ->
7               trace "bind" "attach";
8               links := n :: !links ;
9               true
10          | Some (Black (_, _, links') as p') ->
11            trace "bind" "attach-fail";
12            false
13          | Some (White _) -> assert false
14        end
15    | White _ -> raise Bad_args
```

La figure 9.2 redonne l'exemple donné pour ^{f}DOM à la figure C.1 et la trace d'exécution avec cette nouvelle implantation.

```
1 Prim: create dom          Rule: create dom
2 Prim: create black        Rule: create black
3 Prim: create black        Rule: create black
4 Prim: create black        Rule: create black
5 Prim: set                 Rule: set
6 Prim: bind                Rule: attach
7 Prim: bind                Rule: attach
8 Prim: bind                Rule: attach-fail
```

(b) Trace d'exécution

```
1 let d = create_dom () ;;
2 let p = create_black d (Element "P") ;;
3 let n1 = create_black d (Element "N") ;;
4 let n2 = create_black d (Element "N") ;;
5 set d p (String "test") (Imm (Int 12)) ;;
6 ignore (bind d p n1) ;
7 ignore (bind d p n2) ;;
8 ignore (bind d n1 n2) ;;
```

(c) Code d'exemple

FIGURE 9.2: Implantation de $^{\epsilon}DOM$ et même exemple que pour ^{f}DOM

L'implantation pour la version JavaScript/OCaml pour OBrowser serait tout aussi simple, le DOM du navigateur permettant d'accéder au parent d'un nœud via la propriété parentNode.

9.2.5 Implantation de ^{c}DOM

De part le découplage de ^{c}DOM et du langage hôte réalisant la composition de ses primitives, il n'est pas possible, au niveau de l'implantation de ^{c}DOM de prévoir statiquement les informations de portée. En outre, pour calculer à l'avance ces informations, il faudrait soit que le langage hôte soit simpliste au niveau du contrôle de flot, soit une analyse statique poussée. Dès lors, il n'est pas possible d'effacer l'information de portée dans l'implantation, celle-ci doit être présente et calculée à l'exécution.

Stockage des informations de portée Il y a deux possibilités principales pour stocker ces informations :

1. On peut stocker dans chaque objet alloué une référence faible [1] vers le nœud dont la portée est ouverte. Cette méthode peut être implantée par réécriture du programme, en encapsulant chaque allocation. Elle peut aussi être implantée, de façon plus optimale mais, en modifiant les primitives de gestion mémoire.
2. On peut aussi stocker dans chaque nœud une liste (ou un tableau extensible par blocs) de références faibles des objets sous sa portée.

Exemples d'implantation des informations Suivant l'environnement, il n'est pas évident de choisir laquelle des solutions implanter. Voici deux exemples pour chaque, l'un en OCaml, car c'est un exemple où il est possible de modifier la bibliothèque d'exécution (même si cela requiert des connaissances techniques non triviales), l'autre en JavaScript où, au contraire, on a très peu de prise sur l'implantation :

1. Pointeur chaque objet vers le nœud :
   - En OCaml les valeurs allouées ont déjà un mot machine supplémentaire pris par des informations d'en-tête ; il serait possible de prendre un mot de plus pour stocker un pointeur vers le nœud, qui ne serait pas pris en compte dans le calcul des valeurs vivantes par le ramasse-miettes. Cette solution est simple à mettre en place, mais nécessite un sur-coût systématique, même dans les parties calculatoires du programme n'utilisant pas du tout ^{c}DOM.
   - En JavaScript, il n'y a pas de notion de référence faible, une telle implantation pourrait donc causer des fuites mémoire. Cependant, les objets abusivement conservés seraient les nœuds document, dont on peut dire raisonnablement qu'ils sont en moyenne des valeurs moins volatiles que les valeurs du langage. D'autre part, dans le cas d'une sur-couche à JavaScript, comme

1. Une référence faible est une référence qui n'intervient pas dans le calcul des valeurs vivantes lors d'un nettoyage de la mémoire. Ainsi, son déréférencement peut échouer et renvoyer une valeur nulle (via un type optionnel en OCaml).

OBrowser, il serait possible d'effectuer un parcours des valeurs du langage de temps en temps pour supprimer les arcs arrière vers des nœuds non atteints, et ainsi restaurer l'exactitude du modèle mémoire. Grâce au côté dynamique de JavaScript, il serait possible de changer l'allocateur lors du premier nœud rencontré, et le restaurer lors de la fermeture de ce nœud, et de ne stocker les liens que sur demande, permettant un sur-coût en mémoire et en temps nul dans les parties calculatoires. Cette optimisation nécessite cependant d'utiliser une fonction d'allocation personnalisée et non constante, et peut donc diminuer les performances en cassant certaines optimisations de l'interprète JavaScript.

2. Liste des objets dans le nœud :
   - En OCaml, il serait facilement possible d'avoir dans le type du nœud un tableau de références faibles extensible. Avec cette solution, il n'y a pas de sur-coût systématique en mémoire. Il reste par contre nécessaire de modifier l'allocateur. Éventuellement, il serait possible de changer l'allocateur au premier nœud rencontré afin d'avoir un sur-coût nul dans les parties calculatoires du programme.
   - En JavaScript, cette solution implique que toutes les allocations faites sous le nœud ne soient jamais nettoyées, ou qu'il faille le faire à la main. À moins d'avoir une sur-couche qui permette d'aider le ramasse miette en éliminant certains liens, de façon similaire à celle indiquée pour la première méthode, cette méthode est à proscrire. Bien sûr, il serait possible de la même façon de changer l'allocateur en cours de route.

Suivant l'environnement, on choisira donc l'une ou l'autre des solutions (ici, plutôt la première pour JavaScript et plutôt la seconde pour OCaml).

Copie En pratique, on peut utiliser l'algorithme suivant pour effectuer la copie :

1. Pointeur dans chaque objet vers le nœud :
   (a) On parcourt l'arbre en profondeur en tenant à jour une liste des objets copiés (en association avec les objets originaux pour retrouver le lien), en commençant par la racine, que l'on copie.
   (b) On recopie les objets dont le parent a été copié, et on recopie les pointeurs externes tels-quels.
   (c) On parcourt à nouveau l'arbre en mettant à jour les pointeurs externes si finalement ils ont été copiés (et on leur applique l'algorithme de copie récursivement).
   (d) on itère jusqu'au point fixe.
2. Liste des objets dans le nœud :
   On peut appliquer un algorithme proche de la spécification. On calcule transitivement l'ensemble des objets dans la portée, puis on parcourt l'arbre en s'arrêtant de copier lorsqu'on arrive sur un objet non présent dans l'ensemble précédemment calculé.

La terminaison de ces algorithmes s'obtient directement de la définition de la validité d'un état de ^{c}*DOM*.

10 Un langage pour manipuler le document : FidoML

Dans ce chapitre, nous présentons FidoML (pour *Functional and Imperative Document Operations ML*), un langage à la ML, fonctionnel et impératif à évaluation stricte, muni de primitives de manipulation du document. Nous allons présenter la grammaire concrète du langage, et en décrire les principaux traits au travers d'explications introductives et d'exemples. La présentation formelle sera faite au chapitre suivant, dans lequel nous donnerons sa sémantique opérationnelle et le système de types.

Dans ces deux chapitres, nous cherchons à fournir des primitives de manipulations du document bien typées et au comportement prévisible, en utilisant le mécanisme de copie implicite défini au chapitre précédent. Cependant, et même si le typage et les vérifications que nous allons introduire en sont des pré-requis, nous n'introduirons pas encore le typage de l'imbrication des nœuds vis-à-vis d'une grammaire de document. Ce sujet sera traité indépendamment au chapitre 13.

Le langage étant basé sur ML une large partie de ce chapitre est constituée de rappels permettant au lecteur de se replacer dans le contexte du langage ML, ainsi que de fixer la variante que nous utilisons. Le lecteur déjà familier avec ML et ses dérivés pourra se concentrer sur les parties spécifiques à FidoML, qui seront mises en valeur de la même façon que ce paragraphe. À l'opposé, Le lecteur novice pourra utiliser les ouvrages [📖67] (resp. [📖59]) pour une initiation plus pédagogique à Caml Light (resp. OCaml).

Pour faciliter la présentation, nous allons partir de la grammaire du langage ML de base pour l'enrichir petit à petit en introduisant les concepts, et en expliquant comment nous allons les traiter formellement au chapitre suivant. La syntaxe complète du langage est néanmoins récapitulée dans la figure 10.1, à la fin de ce chapitre.

10.1 Rappels sur ML et généralités

Un programme est une suite de phrases séparées par des doubles points-virgules, une phrase pouvant simplement être une expression à évaluer, ou une liaison globale d'une expression à un nom pour le reste du programme. Le programmeur peut annoter les définitions du type associé pour documenter son code.

```
prog   ::=  phrase [ ;; phrase ]*
phrase ::=  expr | let id [ : type ]? = expr | let rec id [ : type ]? = rec-expr
```

Dans notre variante du langage ML, l'évaluation est *stricte*, c'est-à-dire que les phrases et les expressions sont évaluées dans l'ordre décrit par la syntaxe (en particulier, les arguments d'une fonction sont évalués strictement avant le corps lors d'un appel). Chaque phrase-expression est évaluée immédiatement, et une liaison associe à un nom le résultat de l'évaluation immédiate de l'expression associée.

Il est possible de masquer une liaison par une nouvelle liaison avec le même nom. La liaison précédente existe toujours et sa valeur n'est pas modifiée, elle n'est simplement plus accessible après la nouvelle définition. L'erreur commune du débutant est de penser que `let x = x + 1 ;;` est une modification, alors qu'il s'agit de définir une nouvelle liaison en utilisant la liaison précédente.

Liaisons récursives Les liaisons ne sont pas récursives par défaut en FidoML, il faut ajouter explicitement le mot clef rec. Dans ce cas, le nom de la liaison peut être utilisé dans sa propre définition et

désigne bien la liaison en cours de construction et pas une liaison antérieure.

Comme dans la plupart des ML à évaluation stricte, seule une partie des expressions est acceptée dans une définition récursive. Dans FidoML tel qu'il est spécifié au chapitre prochain, les seules récursions acceptées sont les définitions de fonctions récursives et l'auto-référencement au sein d'un nœud, que nous introduirons plus tard dans ce chapitre.

Expressions FidoML fournit les constructions d'expressions classiques du langage ML :

```
expr    ::=  let rec id [ : type ]? = expr in rec-expr
           | let id [ : type ]? = expr in expr
           | expr expr  |  id  |  ( expr )
           | expr :: expr
           | ( expr [ , expr ]* )
           | string  |  integer
           |  true  |  false  |  ()  |  []
           | if expr then expr else expr end
           | expr infix expr
           | expr ; expr

rec-expr ::=  fun id -> expr

id      ::=  [a-zA-Z0-9_]+  |  ( infix )

infix   ::=  [-+*/.:=!@]+
```

- Les liaisons locales, dont la portée est restreinte à une sous-expression. Il n'y a qu'un espace de noms pour les liaisons locales et globales, et il est possible de masquer une liaison (globale ou locale) existante par une nouvelle (globale ou locale) avec le même nom.
- La séquence séparant deux expressions par un point-virgule. Utilisée en général lorsque le l'intérêt de la première expression réside dans ses effets de bord et non dans son résultat.
- La définition de fonction avec le mot clef fun, et l'application en juxtaposant la fonction et son argument (comme en λ-calcul). Une exception est faite pour les opérateurs binaires qui peuvent être appliqués en notation infixe ou utilisés normalement s'ils sont entourés de parenthèses.
- La construction de n-uplets en notation parenthésée.
- La construction de listes avec la liste vide notée [] et la construction par le constructeur infixe ::.
- Les valeurs booléennes de base, ainsi que l'alternative if et les opérations classiques (&) et (|).
- Les entiers et les opérations classiques (+), (-), (*), (/), (=), (>), etc.
- Les chaînes notées entre guillemets, et l'opération de concaténation (^).

Fitrage FidoML implante le filtrage par motifs en profondeur, qui permet d'effectuer un branchement conditionnel en fonction du contenu d'une valeur, permettant en même temps de nommer les sous-valeurs qui la composent pour les réutiliser.

Concrètement, il s'agit de tester si une valeur a une structure similaire à un des motifs présents dans le filtrage. Lorsqu'un motif échoue, on teste le suivant. Le premier motif reconnaissant la valeur voit sa branche associée évaluée pour donner le résultat final. Le motif attrappe-tout, qui accepte n'importe quelle valeur s'écrit _. L'utilisation d'un nom à la place permet en plus de réutiliser cette sous-valeur dans le corps de la branche.

```
expr   +=  match expr with [ | pat -> expr ]+
pat    ::=  id  |  _
          |  ( pat as id )
          |  string  |  integer  |  true  |  false  |  ()  |  []
          |  pat :: pat
          |  ( pat [ , pat ]* )
```

Typage La syntaxe des phrases et expressions n'utilise pas de types. En effet, en ML, les types sont synthétisés par le compilateur plutôt que donnés par le programmeur. Cependant, le programmeur peut

ajouter des annotations de types explicitement dans les expressions, et nous verrons juste après qu'il peut aussi définir de nouveaux types. Il faut donc une syntaxe concrète pour les types. Cette syntaxe est aussi utilisée pour que le typeur puisse afficher les types de façon lisible.

```
    expr   +=   ( expr : type )
    type   ::=  int | bool | unit | string
              |  type-var
              |  type -> type
              |  type [ * type ]+
              |  id | type id | [( type [, type]+ ) id
type-var   ::=  '[a-zA-Z]+
```

- Les valeurs primitives sont d'un type de base int ou bool.
- Les expressions impératives dont le résultat est inutile renverront () de type unit. Ces le cas des fonctions prédéfinies d'entrées/sorties ou des affectations que nous verrons à la prochaine section.
- Le type d'un n-uplet est le produit cartésien des types de ses composantes.
- Le type d'une fonction prenant un argument de type t_1 et retournant une valeur de type t_2 est t_1 -> t_2. L'application d'une telle fonction est alors typée en vérifiant que l'argument est de type t_1.
- Les listes en ML sont homogènes. Le type des listes est paramétré par le type de ses éléments, une liste d'entiers sera par exemple notée int list.
- Lorsqu'une expression est générique (on dit aussi *polymorphe*), en d'autres termes qu'elle peut être évaluée sans avoir à connaître la structure de tout ou partie des valeurs qu'elle utilise, le typeur insérera une variable de type à chaque fois qu'il détecte de la généricité. Par exemple, la liste vide est de type 'a list , et une fonction inversant les composantes d'un couple sera de type ('a * 'b) -> ('b * 'a).
- Chaque expression devant avoir un type, le typage des branchements (if et match) se fait en vérifiant que toutes les branches ont bien le même type.
- De la même façon, les motifs d'un filtrage doivent être homogènes entre eux et compatibles avec le type de la valeur filtrée.

Le système de types de FidoML, y compris le polymorphisme, sera présenté en détails section11.2.

10.2 Types de données personnalisés

FidoML donne à l'utilisateur la possibilité de définir ses propres structures de données, via la définition de types de données algébriques :

```
   phrase   +=   type ( type-var [, type-var]+ ) id = type-def
              |  type type-var? id = type-def
type-def    ::=  type
              |  cstr [of type]? [ | cstr [of type]? ]*
              |  { mutable? id : type [ ; mutable? id : type ]* }
     expr   +=   { id = expr [ ; id = expr ]* }
              |  cstr expr?
     cstr   ::=  [A-Z][a-zA-Z0-9]*
```

Un type peut être paramétré, dans ce cas les parties polymorphes de la définition seront repérées par des variables de type. Toutes les variables présentes dans la définition doivent être reportées dans la liste des paramètres du type, à gauche du nom. Lors de la création d'une valeur d'un type polymorphe, ces variables sont instanciées en fonction des sous-expressions correspondantes, ou laissées polymorphes le cas échéant. C'est le même mécanisme que pour les n-uplets ou les listes du langage de base.

Les types définis peuvent être des types *somme*, des *enregistrements* ou des *alias* (nommage de types existants).

- Un type *somme* est un ensemble fini de *constructeurs*. On appelle parfois un tel type *type énuméré* ou *union discriminante* dans d'autres langages. Les constructeurs peuvent être associés à des types, afin de réunir les valeurs provenant de plusieurs types en un seul. L'exemple suivant donne une définition des listes simplement chaînées ainsi que quelques primitives associées dans la syntaxe de FidoML.

```
type 'a l =
 | Nil
 | Cons of ('a * 'a l) ;;

let rec map = fun f -> fun l ->
  match l with
    | Nil -> Nil
    | Cons (x, xs) -> Cons (f x, map f xs)
;;

let integers = fun max ->
  let rec aux = fun n ->
    if n = max then
      Nil
    else
      Cons (n, aux (n + 1))
  in
    aux 0
;;
```

  Lors du filtrage d'un type somme, le langage vérifie que tous les cas ont bien été traités, et refuse le programme si ce n'est pas le cas.
- Un type *enregistrement* permet d'encapsuler plusieurs valeurs au sein d'une même valeur construite en utilisant des *champs nommés*. On utilise plus souvent le terme anglais *record*.
  L'exemple suivant montre la syntaxe de définition, d'accès et de modification des enregistrements, au travers de la définition du classique type 'a ref et des primitives associées.
  Un champ d'enregistrement est modifiable seulement si il est marqué avec le mot clef mutable.
  La syntaxe d'accès et de modification utilisée dans cet exemple est décrite juste après.

```
(* type des références *)
type 'a ref = {
  mutable contents : 'a
} ;;

(* création d'une nouvelle référence *)
let ref = fun v ->
  { contents = v } ;;

(* affectation *)
let (:=) = fun r -> fun v ->
  r.(field contents) <- v ;;

(* déréférencement *)
let (!) = fun r ->
  r.(field contents) ;;
```

sélecteurs FidoML utilise une syntaxe commune pour la déconstruction des valeurs avec une notion de *sélecteurs*.

expr	+=	*expr* .(*sel*)	*accès*
	|	*expr* .(*sel*) <- *expr*	*modification*
sel	::=	field *id*	*champ d'enregistrement*
	|	proj *integer* / *integer*	i^e *projection d'un* n*-uplet*

Le sélecteur field prend en paramètre un nom de champ, la valeur dé-construite devant être du dernier type enregistrement défini avec un champ de ce nom. Le sélecteur proj prend deux constantes entières en paramètres la première étant l'indice de la composante à sélectionner, et la seconde donne le nombre de champs de la valeur à dé-construire (il n'y a pas de généricité sur la taille du n-uplet).

Si le langage supportait les tableaux, on pourrait ajouter par exemple un sélecteur cell prenant une expression de type int en paramètre. De la même façon, le programmeur pourrait définir des fonctions de dé-construction et d'affectation d'un type et les utiliser avec cette syntaxe. Ces extensions ne présentent pas de grande difficulté et ne sont pas traitées dans la spécification formelle.

Le filtrage est aussi enrichi de motifs permettant de dé-construire les valeurs de types personnalisés en profondeur :

FidoML utilise une syntaxe commune pour la dé-construction des valeurs avec une notion de *sélecteurs*.

```
pat   +=   { id = pat [ ; id = pat ]* }
       |   cstr pat?
```

Redéfinitions Il est interdit au programmeur de définir deux types avec le même nom dans un même programme FidoML. Par contre, il est autorisé de définir deux types enregistrement (resp. somme) partageant un même nom de champ (resp. constructeur). Si un tel cas se produit, le nom de champ (resp. constructeur) précédent est masqué par la nouvelle définition, et toute utilisation postérieure de ce nom se référera à la nouvelle définition de type.

10.3 Création et manipulation de documents

La syntaxe des expressions est enrichie d'une nouvelle construction de création de nœuds. La construction d'un nœud peut faire l'objet d'une définition récursive, mais des restrictions sont nécessaires afin qu'il ne puisse pas être utilisé de façon incorrecte durant sa construction. Ces restrictions sont implantées sous la forme d'une analyse au préalable du programme, présentée formellement section 11.1.

Un nœud est défini par une étiquette, la liste de ses enfants et l'ensemble de ses propriétés.

```
rec-expr   +=   node <tag> expr [ prop id = expr ]* end
     tag   ::=  [a-zA-Z0-9-]+
```

Afin d'être valide, la construction d'un nœud doit se conformer à une définition de type de nœud antérieure. Une telle définition décrit, pour une étiquette donnée, l'ensemble des propriétés et leur types. Une seule définition est autorisée par étiquette au cours du programme. Un nœud correctement construit pour une étiquette t sera de type <t> node. Ces types ne sont pas polymorphes.

```
phrase   +=   node type <tag> [ mutable? prop id : type ]* end
  type   +=   <tag> node
```

Deux nouveaux mots-clefs sont ajoutés pour obtenir la liste des enfants d'un nœuds déjà construit, et pour remplacer l'intégralité des enfants par une nouvelle liste.

```
expr   +=   children expr  |  replace expr expr
```

Sont aussi ajoutées la déconstruction par filtrage, et via un nouveau sélecteur prop. Le filtrage permet d'explorer en profondeur la liste des enfants et les propriétés.

```
pat   +=   node <tag> pat [ prop id = pat ]* end
sel   +=   prop id
```

Sémantique par copie Grâce à cette syntaxe bien délimitée, nous pourrons, chapitre 12, donner un schéma d'évaluation utilisant la notion de portée introduite dans notre sémantique du document avec copies implicites comme suit :

- La portée d'un nœud commence au mot-clef node et se termine au end associé.
- Toutes les valeurs construites dans cette portée et accessibles depuis une des propriétés, seront copiées lorsque le nœud sera copié.
- Lorsqu'une fonction définie au sein de cette portée et accessible depuis une de ses propriété est appelée, elle ré-ouvre la portée du nœud pour s'y exécuter.

Nous donnons plusieurs exemples à la section suivante permettant d'éclaircir ce comportement en pratique.

Généricité des nœuds Le système présenté jusqu'à maintenant n'est pas cohérent au niveau des types, car deux nœuds avec deux étiquettes différentes sont de types différents, or nous avons dit que la construction de nœud prenait une liste de nœuds, sans en spécifier l'étiquette, alors que les listes sont homogènes.

Pour ceci, nous introduisons un type node non étiqueté, *compatible* avec n'importe quel type étiqueté <*t*> node. Dès que le typeur devra synthétiser un type à partir de deux types étiquetés différemment, il utilisera ce type plutôt que de produire une erreur. Ainsi, on pourra obtenir une liste de nœuds, y compris s'ils ont des étiquettes différentes.

Bien sûr, les opérations possibles sur ce nœud sont restreintes. Il est possible d'utiliser les opérations children et replace. Nous introduisons aussi un nouveau sélecteur prop? ne renvoyant la valeur de la propriété demandée que si le nœud la définit.

```
type  +=  node
 sel  +=  prop? id
```

Afin d'assurer la correction du typage de ce sélecteur prop?, il faut la propriété que si deux définitions de types de nœuds partagent un même nom de propriété, alors les types associés doivent être égaux. Cette vérification est décrite dans les jugements de typage relatifs aux nœuds de la section 11.2.

Filtrage par étiquette Nous avons déjà vu que le filtrage permet de dé-construire une valeur de type <*t*> node en profondeur avec un motif de la même étiquette *t*.

Nous ajoutons alors la possibilité de filtrer une valeur de type générique node, et d'utiliser une étiquette différente dans chaque motif du filtrage. Le but est, d'une part, d'exécuter un code différent en fonction de l'étiquette effectivement portée par le nœud filtré, et, d'autre part, de revenir au type étiqueté correspondant pour lui appliquer des traitements plus spécifiques. Concrètement, si le motif étiqueté est nommé à l'aide du motif as, ce nom représentera le nœud, avec le type spécifique, dans la branche associée.

Exhaustivité du filtrage Étant donné que le type plus général node peut représenter tous les types spécifiques étiquetés, tout au long du programme, la vérification de l'exhaustivité d'un tel filtrage nécessite de connaître à l'avance l'ensemble des étiquettes de nœuds.

La solution la plus simple est d'interdire l'utilisation du filtrage sur les nœuds avant que tous les types de nœuds du programme aient été définis. Ainsi, le programmeur peut définir progressivement les types des nœuds, avec les types et fonctions auxiliaires définissant le format de son document, mais il ne pourra écrire les fonctions de parcours du document qu'une fois son format complètement connu, ce qui paraît être une solution raisonnable.

10.4 Exemples

Opérations arithmétiques Le listing 10.1 montre la syntaxe pour définir les types de nœuds au travers d'un format de document représentant des expressions arithmétiques. On peut y remarquer que la propriété name a bien le même type dans chacun des types de nœuds où elle apparaît. Cet exemple montre aussi la construction des nœuds au travers de la définition d'une expression arithmétique.

On peut observer de plus le parcours récursif bien typé du document permis par le filtrage spécialisant dans les fonctions to_string et to_xml.

```
(* type des feuilles constantes *)
node type<const>
  prop val : int
end ;;

(* type des opérations n-aires *)
node type<op>
  prop name : string
end ;;

(* type des variables *)
node type<var>
  prop name : string
end ;;



(* exemple *)
node<op>
  [ node<const> [] prop val = 8 end ;
    node<var> [] prop name = "y" end ;
    node<op>
      [ node<var> [] prop name = "z" end ;
        node<const> [] prop val = 3 end ]
      prop name = "-"
    end ]
  prop name = "+"
end ;;


(* afficheur infixe *)
let rec to_string = fun n ->
  match n with
    | node<const> _ prop val = i end ->
      string_of_int i
    | node<op> (e :: []) prop name = n end ->
      n ^ " " ^ (to_string e)
    | node<op> (e :: es) prop name = n end ->
      foldl
        (fun r -> fun e ->
           r ^ " " ^ n ^ " " ^ to_string e)
        (to_string e)
        es
    | node<var> _ prop name = n end ->
      n
;;

(* afficheur to_xml *)
let rec to_xml = fun n ->
  match n with
    | node<const> _ prop val = i end ->
      "<const val='" ^ string_of_int i ^ "'/>"
    | node<op> [e] prop name = n end ->
      "<op name='" ^ n ^ "'>" ^ to_xml e ^ "</op>"
    | node<op> l prop name = n end ->
      "<op name='" ^ n ^ "'>"
      ^ foldl (fun r -> fun e -> r ^ to_xml e) "" l
      ^ "</op>"
    | node<var> _ prop name = n end ->
      "<var name='" ^ n ^ "'/>"
;;
```

Listing 10.1: Expressions arithmétiques entières et parcours.

Parcours générique Le listing 10.2 montre un exemple de parcours récursif bien typé générique, avec utilisation du sélecteur prop?. Certains types de nœuds définis dans ce programme ont une propriété class de type string. La fonction nodes_by_class parcourt un document et renvoie la liste de tous les nœuds y apparaissant dont la propriété class correspond à une valeur demandée. Cet exemple montre que ce typage générique des nœuds, bien que simple, permet déjà d'exprimer des programmes intéressants sur les documents, en restant bien typé.

```
(* définition du format *)
node type <paragraph>
  prop class : string
end ;;
node type <section>
  prop class : string
end ;;
node type <float>
  prop class : string
  prop caption : string
end ;;

node type <text>
  prop content : string
end ;;


(* récupération des nœuds de la classe sc *)
let rec nodes_by_class = fun sc -> fun n ->
  match n.(kprop? class) with
    | Some c ->
        if c = sc then
          n :: map
                (nodes_by_class sc)
                (children n)
        else
          map
           (nodes_by_class sc)
           (children n)
    | None ->
        map
          (nodes_by_class sc)
          (children n)
;;
```

Listing 10.2: Exemple de parcours générique

Sémantique par copie Le listing 10.3 est une version FidoML des exemples donnés au chapitre 7 (figures 7.4 et 7.3). Si l'implantation du document sous-jacente choisie est conforme à ^{c}DOM, alors *bla* est bien ajouté deux fois comme fils de l_1, la seconde instance est une copie implicite. De même, *blou* sera bien copié (et non déplacé) implicitement de l_2 vers l_1 lors de l'opération replace. On obtient donc bien le comportement sémantique que nous cherchions à obtenir pour assurer la prévisibilité du langage et qui est nécessaire à la correction du typage du document que nous introduirons au chapitre 13.

```
(* nœuds texte *)
node type <text>
  prop textContent : string
end ;;

let text = fun s ->
  node<text>
    prop textContent = s
  end ;;

(* types pour les listes *)
node type <li> end ;;
node type <ul> end ;;

(* afficheur de listes *)
let rec print = fun n ->
  match n with
    | node<li> _ end
      iter print (children n)
    | node<ul> cl end ->
      iter print cl
    | node<text> end as t ->
      print_string t.(prop textContent) ;;

(* définition des listes *)
let l1 =
  let bla = node<li> [ text "bla" ] end in
    node<ul> [ bla ; bla ] end ;;

let l2 =
  node<ul>
    [node<li> [ text "blou" ] end ]
  end ;;



(* affichage avant et après l'effet *)
print_string "l1 -> " ; print l1 ;;
print_string "l2 -> " ; print l2 ;;

replace
  l1
  (children l1 @ children l2) ;;

print_string "l1 -> " ; print l1 ;;
print_string "l2 -> " ; print l2 ;;
```

Listing 10.3: Exemple de sémantique par copie.

Sémantique par copie et mécanisme de portée Le listing 10.5 montre comment cette notion de portée couplée à la syntaxe délimitée permet de prédire facilement le comportement du programme au niveau source, et donne un moyen classique pour le programmeur pour gérer la localité des données.

```
let with_shared_counter =
  let cpt = ref 0 in
  let rec self =
    node <a>
      [ node <text> content = "incr" end ]
      prop on_click = fun () ->
        cpt := !cpt + 1 ;
        replace self [ node <text> content = string_of_int !cpt end ]
    end
  in self
;;

let with_copied_counter =
  let rec self =
    node <a>
    let cpt = ref 0 in
      [ node <text> content = "incr" end ]
      prop on_click = fun () ->
        cpt := !cpt + 1 ;
        replace self [ node <text> content = string_of_int !cpt end ]
    end
  in self
;;
```

Listing 10.4: Exemple d'utilisation du mécanisme de portée.

Gestion de la sémantique par copie Le listing 10.5 montre comment le programmeur peut maîtriser les copies au niveau des nœuds. Il donne un exemple de *glisser-déposer*, ou un nœud doit être enlevé de son parent, puis ajouté à un autre. Pour ceci, le programmeur doit détacher le nœud de son emplacement explicitement, puis le rattacher à son nouvel emplacement. On se place dans le contexte d'une bibliothèque d'interface graphique fictive, dont la structure serait représentée par un document impératif, les nœuds en étant les composants (on utilise ici des conteneurs verticaux VBOX, des étiquettes texte LABEL et une fenêtre WINDOW, les ellipses dans le code matérialisent qu'on ne définit que les propriétés qui nous intéressent pour notre exemple). Concrètement, on définit deux conteneurs verticaux, et un item du conteneur vbox_from peut être glissé et déposé dans la deuxième boite vbox_to.

```
let rec vbox_from =
  node <vbox>
  let item = fun s ->
    let rec me =
      node <label> []
        prop text = "Item " ^ s
        prop on_mouse_drag = fun () ->
          (* on se supprime de la boite from *)
          replace
            vbox_from
            (filter ((<>) me) (children vbox_from)) ;
          (* on s'attache à la souris *)
          cursor := me
         (* ...  *)
      end
    in me
  in
    [ item "A" ; item "B" ; item "C" ]
    (* ... *)
  end ;;

let rec vbox_to =
  node <vbox>
    [ ]
    prop on_mouse_release = fun () ->
      replace
        vbox_to
        (!cursor :: (children vbox_to)) ;
      cursor := standard_cursor
    (* ... *)
  end
;;

run
 (node <window>
    [ vbox_from ; vbox_to ]
    (* ... *)
  end) ;;
```

Listing 10.5: Glisser-déposer sans copie.

```
prog     ::=  phrase [ ;; phrase ]*
phrase   ::=  expr
           |  let id [ : type ]? = expr
           |  let rec id [ : type ]? = rec-expr
           |  type ( type-var [, type-var]+ ) id = type-def
           |  type type-var? id = type-def
           |  node type <tag>
                [ mutable? prop id : type ]*
              end

expr     ::=  rec-expr
           |  let id [ : type ]? = expr in expr
           |  let rec id [ : type ]? = expr in rec-expr
           |  if expr then expr else expr end
           |  expr expr  |  expr infix expr
           |  expr :: expr
           |  expr ; expr
           |  ( expr [ , expr ]* )
           |  id  |  ( expr )
           |  string  |  integer
           |  true  |  false  |  ()  |  []
           |  match expr with [ | pat -> expr ]+
           |  ( expr : type )
           |  { id = expr [ ; id = expr ]* }
           |  cstr expr?
           |  expr .( sel )  |  expr .( sel ) <- expr
           |  children expr  |  replace expr expr

id       ::=  [a-zA-Z0-9_]+  |  ( infix )
infix    ::=  [_-+*/.:=!@_]+
cstr     ::=  [A-Z][a-zA-Z0-9]*

rec-expr ::=  fun id -> expr
           |  node <tag> expr [ prop id = expr ]* end

type-def ::=  type
           |  cstr [of type]? [ | cstr [of type]? ]*
           |  { mutable? id : type [ ; mutable? id : type ]* }

type     ::=  int  |  bool  |  unit  |  string
           |  type-var
           |  type -> type
           |  type [ * type ]+
           |  id  |  type id  |  [( type [, type]+ ) id
           |  node  |  <tag> node

pat      ::=  id  |  _
           |  ( pat as id )
           |  string  |  integer
           |  true  |  false  |  ()  |  []
           |  pat :: pat
           |  ( pat [ , pat ]* )
           |  { id = pat [ ; id = pat ]* }
           |  cstr pat?
           |  node <tag> pat [ prop id = pat ]* end

sel      ::=  field id
           |  proj integer / integer
           |  prop id  |  prop? id

type-var ::=  '[a-zA-Z]+
tag      ::=  [a-zA-Z0-9-]+
```

FIGURE 10.1: Syntaxe complète de FidoML

11 Sémantique statique de FidoML

Dans les deux chapitres qui viennent, nous donnons une spécification formelle du langage FidoML présenté au chapitre précédent.

Ce chapitre présente les vérifications statiques de programme. Tout d'abord, la section 11.1 définit la vérification de la bonne construction des définitions de nœuds et des filtrages par motifs dans le programme. Puis la section 11.2 décrit le système de types.

Le chapitre 12 donnera la sémantique opérationnelle décrivant l'évaluation des programmes considérés valides par ces vérifications statiques, ainsi que des exemples d'implantation possibles.

Cette formalisation a volontairement la forme d'une spécification complète en vue d'implantation, et non celle d'un modèle théorique minimal. Elle prend donc en compte l'ensemble du langage présenté au chapitre précédent. Pour faciliter la lecture, comme au chapitre précédent, les points importants de ces deux chapitres sont marqués d'un trait dans la marge.

11.1 Analyses de programme

Nous avons évoqué au chapitre précédent des restrictions nécessaires du langage pour assurer sa correction. Il s'agit de la limitation de l'utilisation d'un nœud lors de sa définition récursive, ainsi que la vérification de l'exhaustivité du filtrage des nœuds. Ces deux restrictions sont présentées dans cette section sous forme d'analyses statiques, la correction du typage et de l'évaluation n'étant assurée que pour un programme ayant été accepté par ces analyses.

11.1.1 Vérification des définitions de nœuds

Cette première analyse est nécessaire pour assurer la correction de la construction des nœuds. Concrètement, on veut pouvoir considérer la construction d'un nœud comme une construction fonctionnelle, en faisant abstraction du caractère impératif des primitives sous-jacentes. Pour ceci, on cherche à restreindre les programmes acceptés à ceux n'effectuant pas d'actions impératives qui pourraient interférer avec celles implantant la construction. Le but est de pouvoir utiliser des techniques de typage classiques pour la construction, ne faisant pas intervenir ce caractère impératif.

En se plaçant au niveau ^{c}DOM, il s'agit alors d'assurer la propriété suivante de bonne construction des nœuds. Celle-ci assure que les utilisations du nœud ne peuvent se faire qu'une fois la construction terminée, et toutes les propriétés correctement initialisées (pour fixer le vocabulaire, on utilise le terme de construction au niveau du nœud complet, et initialisation au niveau d'une propriété, pour désigner son affectation initiale).

Proposition 11.1.1 *Une construction de nœud FidoML est correcte si elle répond aux critères suivants.*

1. *la copie implicite du nœud par ^{c}DOM ne peut pas arriver avant la fin de sa construction,*
2. *les propriétés du nœud ne peuvent être accédées avant leur initialisation,*
3. *les propriétés du nœud ne peuvent être affectées autrement que par leur initialisation.*

Restriction des liaisons récursives On constate qu'un programme qui ne respecte pas un des points précédents contient forcément un appel de primitive impérative sur le nœud lui-même pendant sa

construction. En FidoML, ceci implique que la définition du nœud soit récursive.

Une solution simple serait de ne pas autoriser du tout la liaison récursive de nœuds. Mais le listing 11.1 montre comment elle demanderait au programmeur des ruses programmatoires trop inesthétiques et au comportement moins prévisible, pour obtenir des comportements classiques et largement utilisés en programmation Web, que nous souhaitons permettre.

```
(* simulation de musée avec rec *)
let rec n =
  node<museum>
    [ node<person> []
        name = "the guardian"
      end ]
    prop on_enter =
      fun p ->
        replace n (p :: children n)
  end ;;
```

```
(* simulation de musée sans rec *)
let np : <museum> node option ref = ref None ;;
let n =
  node<museum>
    [ node<person> []
        name = "the guardian"
      end ]
    prop on_enter =
      fun p ->
        let n = unoption !np in
        replace n (p :: children n)
  end ;;
np := Some n ;;
```

Listing 11.1: Modification au clic avec et sans liaison récursive des nœuds.

Puisqu'une définition récursive est nécessaire pour enfreindre une des propriétés précédentes, nous cherchons alors à identifier les liaisons récursives invalides, afin de rejeter les programmes en contenant. Pour donner une intuition des définitions problématiques, la figure 11.2 en donne quelques exemples concrets.

```
(* utilisation directe du nœud *)
let rec n : <t> node =
  node <t>
    [ n ]
  end ;;
```

```
(* utilisation indirecte du nœud *)
let r = ref node <t> end
let set = fun x -> r := x
let rec n : <t> node =
  node <t>
  let u = set n in
    [ !r ]
  end ;;
```

```
(* utilisation des champs *)
let rec n : <t> node =
  node <t>
    prop p = n.(prop p)
             (* initialisé ? *)
  end ;;
```

```
(* affectation des chamops *)
let rec n : <t> node =
  node <t>
  let u = n.(prop p) <- 3 in
    prop p = 2
  end ;;
  (* valeur de p ? *)
```

Listing 11.2: Exemples de constructions récursives dangereuses

Au niveau de FidoML, on veut alors assurer la propriété suivante, qui implique la propriété 11.1.1, en empêchant les copies implicites et accès aux propriétés prématurés. Cette propriété rejette potentiellement plus de programmes que nécessaire, mais elle a l'avantage d'être systématique et facile à comprendre.

Proposition 11.1.2 *Une construction de nœud FidoML est correcte si aucun nom faisant référence (directement ou indirectement) à un nœud en cours de construction ne peut*

1. *se trouver dans la liste des enfants d'une construction de nœud,*
2. *se trouver dans la liste des nœuds d'une opération replace,*
3. *apparaître dans une expression évaluée durant la construction du nœud et contenant une lecture ou modification de propriété ou de liste d'enfants.*

$$\textit{nodeCheck}(\text{let } n = e) = \textit{nodeCheck}_{\emptyset}(e)$$
$$\textit{nodeCheck}(\text{let rec } n = e) \text{ où } e \text{ est de la forme node } \cdots \text{ end} = \textit{nodeCheck}_{\{n\}}(e)$$
$$\textit{nodeCheck}(\text{let rec } n = e) = \textit{nodeCheck}_{\emptyset}(e)$$

$$\textit{nodeCheck}_N(\text{let rec } n = e_n \text{ in } e) \text{ où } e_n \text{ est de la forme node } \cdots \text{ end} =$$
$$\textit{nodeCheck}_{N\cup\{n\}}(e_n) \wedge \begin{cases} \textit{nodeCheck}_{N\setminus\{n\}}(e) \text{ si } \mathcal{F}(e_n) \cap N = \emptyset & \textit{(réhabilitation par masquage)} \\ \textit{nodeCheck}_{N\cup\{n\}}(e) \text{ sinon} & \textit{(propagation)} \end{cases}$$
(n est dangereux dans sa définition, et dans le corps si sa définition est dangereuse)

$$\textit{nodeCheck}_N(\text{let rec } n = e_n \text{ in } e) =$$
$$\begin{cases} \textit{nodeCheck}_{N\setminus\{n\}}(e_n) \wedge \textit{nodeCheck}_{N\setminus\{n\}}(e) \text{ si } \mathcal{F}(e_n) \cap N = \emptyset & \textit{(réhabilitation par masquage)} \\ \textit{nodeCheck}_{N\cup\{n\}}(e_n) \wedge \textit{nodeCheck}_{N\cup\{n\}}(e) \text{ sinon} & \textit{(propagation)} \end{cases}$$
(n peut être réhabilité dans sa propre définition et dans le corps)

$$\textit{nodeCheck}_N(\text{let } n = e_n \text{ in } e) =$$
$$\textit{nodeCheck}_N(e_n) \wedge \begin{cases} \textit{nodeCheck}_{N\setminus\{n\}}(e) \text{ si } \mathcal{F}(e_n) \cap N = \emptyset \\ \textit{nodeCheck}_{N\cup\{n\}}(e) \text{ sinon} \end{cases}$$
(n est réhabilité dans le corps si sa nouvelle définition n'est pas dangereuse)

$$\textit{nodeCheck}_N(\text{node } e \text{ prop } l_0 = e_0 \cdots \text{ prop } l_n = e_n \text{ end}) =$$
$$(\mathcal{F}(e) \cap N = \emptyset) \wedge (\textstyle\bigwedge_i \textit{nodeCheck}_N(e_i))$$
(noms dangereux totalement interdits dans les enfants)

$$\textit{nodeCheck}_N(\text{match } e \text{ with } l_0 \text{ -> } e_0 \text{ | } \cdots \text{ | prop } l_n \text{ -> } e_n) =$$
$$\textit{nodeCheck}_N(e) \wedge \begin{cases} \left(\bigwedge_i \textit{nodeCheck}_{N\setminus\mathcal{V}(p_i)}(e_i)\right) \text{ si } \mathcal{F}(e) \cap N = \emptyset \\ \left(\bigwedge_i \textit{nodeCheck}_{N\cup\mathcal{V}(p_i)}(e_i)\right) \text{ sinon} \end{cases}$$

$$\textit{nodeCheck}_N(n) =$$
$$T$$

$$\textit{nodeCheck}_N(\textit{cstr } e) =$$
$$\textstyle\bigwedge_i \textit{nodeCheck}_N(e)$$

$$\textit{nodeCheck}_N((e_0, \cdots, e_n)) = \textit{nodeCheck}_N(\textit{cstr } e_0) =$$
$$\textit{nodeCheck}_N(\{\ l_0 = e_0\ ;\ \cdots\ ;\ l_n = e_n\ \}) =$$
$$\textstyle\bigwedge_i \textit{nodeCheck}_N(e_i)$$

$$\textit{nodeCheck}_N(\text{if } e \text{ then } e_t \text{ else } e_f) =$$
$$\textit{nodeCheck}_N(e) \wedge \textit{nodeCheck}_N(e_t) \wedge \textit{nodeCheck}_N(e_f)$$

$$\textit{nodeCheck}_N(e_1\ e_2) = \textit{nodeCheck}_N(e_1\ ;\ e_2) = \textit{nodeCheck}_N(e_1.(s\ e_2)) =$$
$$\textit{nodeCheck}_N(e_1.(s\ e_2) \text{ <- } e_3) = \textit{nodeCheck}_N(\text{replace } e_2) = \textit{nodeCheck}_N(\text{children } e_2) =$$
$$T \text{ si } (\textstyle\bigcup_i \mathcal{F}(e_i)) \cap N = \emptyset, F \text{ sinon}$$
(expressions interdites si elles contiennent des liaisons dangereuses)

FIGURE 11.1: Vérification de la construction de nœuds

Solution La solution que nous implantons est de limiter l'utilisation de cette récursion aux liaisons locales de *valeurs* (au sens de Wright [📖52] : noms, abstractions, constructions fonctionnelles), dans le nœud ou un de ses sous-nœuds.

De façon générale, le principe de cette analyse est de déclarer les noms des liaisons récursives de nœuds comme dangereuses, puis de propager le danger aux liaisons accédant à un nom impliqué dans une liaison dangereuse, et à itérer jusqu'au point fixe. Il faut alors rejeter les programmes contenant des opérations impératives faisant intervenir une liaison dangereuse dans la portée d'un nœud. Dans un langage muni d'un contrôle plus complexe, ou si on voulait une analyse plus fine des références, une telle analyse pourrait nécessiter une itération non triviale. Dans notre cas un simple parcours en profondeur des expressions paraît suffisant pour accepter les idiomes que nous cherchons à autoriser.

Pour implanter cette analyse, on utilise alors la fonction *nodeCheck* donnée figure 11.1 (dans laquelle $\mathcal{F}$ est la collecte de variables libres dans les expressions définie figure 11.2, page 138), qui renvoie T si le programme est considéré correct, et F s'il doit être rejeté. Il s'agit d'une définition inductive sur la

structure syntaxique. Les trois premières règles définissent le comportement sur les phrases du programme, la suite des règles spécifie la vérification des expressions par induction structurelle sur ses sous-expressions.

L'analyse fonctionne en collectant les liaisons dangereuses dans un ensemble de noms transmis et adapté lors du parcours en profondeur de l'expression. Cet ensemble est agrandi lors de la définition récursive d'un nouveau nœud, ainsi qu'à chaque liaison dont l'expression contient un nom libre déjà présent dans l'ensemble. L'ensemble peut éventuellement être réduit par masquage d'une liaison dangereuse, réhabilitant le nom dans le corps de la définition. Finalement l'analyse accepte les expressions qui sont des valeurs, et rejette toutes les expressions impératives faisant intervenir une liaison dangereuse. Elle rejette aussi les définitions de nœuds dont la liste des enfants fait intervenir une liaison dangereuse.

$$
\begin{aligned}
\mathcal{F}(\ \texttt{let}\ n\ \texttt{=}\ e_d\ \texttt{in}\ e_b\ ) &= \mathcal{F}(e_d) \cup (\mathcal{F}(e_b) \backslash \{n\}) \\
\mathcal{F}(\ \texttt{let rec}\ n\ \texttt{=}\ e_d\ \texttt{in}\ e_b\ ) &= (\mathcal{F}(e_d) \backslash \{n\}) \cup (\mathcal{F}(e_b) \backslash \{n\}) \\
\mathcal{F}(\ \texttt{fun}\ n\ \texttt{->}\ e\ ) &= \mathcal{F}(e) \backslash \{n\} \\
\mathcal{F}(\ \texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f\ ) &= \mathcal{F}(e) \cup \mathcal{F}(e_t) \cup \mathcal{F}(e_f) \\
\mathcal{F}(e_1\ \mathit{op}\ e_2) = \mathcal{F}(e_1\ e_2) = \mathcal{F}(e_1\ \texttt{;}\ e_2) = \mathcal{F}(e_1 :: e_2) &= \mathcal{F}(e_1) \cup \mathcal{F}(e_2) \\
\mathcal{F}((e_0, \cdots, e_n)) = \mathcal{F}(\{\ l_0\ \texttt{=}\ e_0\ \texttt{;}\ \cdots\ \texttt{;}\ l_n\ \texttt{=}\ e_n\ \}) &= \textstyle\bigcup_{0 \leqslant i \leqslant n} \mathcal{F}(e_i) \\
\mathcal{F}(\ e\texttt{.(}\mathit{sel}\ \ e_f\texttt{)}\ ) &= \mathcal{F}(e) \cup \mathcal{F}(e_f) \\
\mathcal{F}(\ e\texttt{.(}\mathit{sel}\ \ e_f\texttt{)}\ \texttt{<-}\ e_v\ ) &= \mathcal{F}(e) \cup \mathcal{F}(e_f) \cup \mathcal{F}(e_v) \\
\mathcal{F}(\ n \in V\ ) &= \{n\} \\
\mathcal{F}(\ \mathit{imm}\ ) &= \emptyset \\
\mathcal{F}(\texttt{children}\ e) = \mathcal{F}(\texttt{replace}\ e) = \mathcal{F}((e : \tau)) = \mathcal{F}(\mathit{cstr}\ e_a) &= \mathcal{F}(e) \\
\mathcal{F}(\ \texttt{node <}\mathit{tag}\texttt{>}\ p\ \texttt{prop}\ l_0\ \texttt{=}\ p_0\ \cdots\ \texttt{prop}\ l_n\ \texttt{=}\ p_n\ \texttt{end}) &= \mathcal{F}(e) \cup \textstyle\bigcup_{0 \leqslant i \leqslant n} \mathcal{F}(e_i) \\
\mathcal{F}(\ \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ p_0\ \texttt{->}\ e_0\ \cdots\ \texttt{|}\ p_n\ \texttt{->}\ e_n\ \texttt{end}) &= \mathcal{F}(e) \cup \textstyle\bigcup_{0 \leqslant i \leqslant n} (\mathcal{F}(e_i) \backslash \mathcal{V}_p(p_i))
\end{aligned}
$$

$\mathcal{V}$ est la collecte de liaisons dans les motifs :

$$
\begin{aligned}
\mathcal{V}(\ \mathit{imm}\ ) &= \emptyset \\
\mathcal{V}(\ n\ ) &= \{n\} \\
\mathcal{V}((p_0, \cdots, p_n)) = \mathcal{V}(\{\ l_0\ \texttt{=}\ p_0\ \texttt{;}\ \cdots\ \texttt{;}\ l_n\ \texttt{=}\ p_n\ \}) &= \textstyle\bigcup_{0 \leqslant i \leqslant n} \mathcal{V}(p_i) \\
\mathcal{V}(\ \mathit{cstr}\ p\ ) &= \mathcal{V}(p) \\
\mathcal{V}(\ \texttt{node <}\mathit{tag}\texttt{>}\ p\ \texttt{prop}\ l_0\ \texttt{=}\ p_0\ \cdots\ \texttt{prop}\ l_n\ \texttt{=}\ p_n\ \texttt{end}) &= \mathcal{V}(p) \cup \textstyle\bigcup_{0 \leqslant i \leqslant n} \mathcal{V}(p_i)
\end{aligned}
$$

FIGURE 11.2: Collecte de variables libres dans les expressions.

11.1.2 Exhaustivité du filtrage

Lors du filtrage d'une valeur d'un type somme, d'une liste ou d'une constante (booléen, entier, chaîne), tous les cas doivent apparaître dans les motifs pour que l'évaluation soit bien définie. Une implantation d'ML ne doit accepter le programme que lorsque ce n'est pas le cas, ou donner un avertissement et lever une erreur dynamique si l'implantation dispose d'un tel mécanisme. En pratique, cette vérification est souvent faite au moment de la compilation et de l'optimisation, comme décrit dans [🕮1] ou plus récemment [🕮13].

Dans FidoML, le langage ne disposant pas d'exceptions ou de mécanisme similaire, nous devons refuser les programmes n'assurant pas l'exhaustivité du filtrage. Pour vérifier cette exhaustivité, il faut aussi traiter les filtres de nœuds, puisque nous permettons le filtrage d'une valeur de type node avec des filtres de différentes étiquettes. Nous ne donnons que la vérification du filtrage et non son optimisation, il ne s'agit pas d'une transformation de programme.

$$\mathit{matchProgCheck}(p) \;=\; \mathit{matchProgCheck}_{(F,\emptyset,\emptyset)}(p)$$

$$\begin{array}{l}
\mathit{matchProgCheck}_{(v,E_N,E_C)}(\epsilon) \;=\; T \\
\mathit{matchProgCheck}_{(F,E_N,E_C)}(\texttt{node type <}t\texttt{> } \cdots \texttt{ end ;; } p) \;=\; \mathit{matchProgCheck}_{(F,E_N\cup\{t\},E_C)}(p) \\
\mathit{matchProgCheck}_{(T,E_N,E_C)}(\texttt{node type <}t\texttt{> } \cdots \texttt{ end ;; } p) \;=\; F \\
\mathit{matchProgCheck}_{(F,E_N,E_C)}(\texttt{type (}\cdots\texttt{) t = } c_0 \texttt{ of } \tau_0 \texttt{ | } \cdots \texttt{ | } c_n \texttt{ of } \tau_n \texttt{ ;; } p) \;= \\
\qquad \mathit{matchProgCheck}_{(F,E_N,(c_0,C_d)::\cdots::(c_n,C_d)::E_C)}(p) \;\text{ où } C_d = \bigcup_i \{c_i\} \\
\mathit{matchProgCheck}_{(v,E_N,E_C)}(\texttt{let [rec] } n \texttt{ =}e\texttt{ ;; p}) \;= \\
\qquad \mathit{matchExprCheck}_{(E_N,E_C)}(e) \wedge \mathit{matchProgCheck}_{(v',E_N,E_C)} \\
\qquad \text{où } v' = \begin{cases} T \text{ si } e \text{ contient un filtrage de nœuds} \\ v \text{ sinon} \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathit{matchExprCheck}_{(E_N,E_C)}(\texttt{match } e \texttt{ with } p_0 \texttt{ -> } e_0 \texttt{ | } \cdots \texttt{ | } p_n \texttt{ -> } e_n) \;= \\
\qquad \mathit{matchExprCheck}_{(E_N,E_C)}(e) \wedge \bigwedge_i \mathit{matchExprCheck}_{(E_N,E_C)}(e_i) \\
\qquad \wedge \mathit{matchCheck}_{(E_N,E_C)}(\bigcup_i \{p_i\}) \\
\mathit{matchExprCheck}_{(E_N,E_C)}(e) \;= \\
\qquad \bigwedge_{e\in E} \mathit{matchExprCheck}_{(N,C)}(e) \text{ avec } E \text{ l'ensemble des sous-expression de } e
\end{array}$$

$$\begin{array}{l}
\mathit{matchCheck}_{(E_N,E_C)}(\{n,\cdots\}) \;=\; \mathit{matchCheck}_{(E_N,E_C)}(\{_,\cdots\}) \;= T \\
\mathit{matchCheck}_{(E_N,E_C)}(\{\mathsf{true},\mathsf{false}\}) \;=\; T \\
\mathit{matchCheck}_{(E_N,E_C)}(\{\texttt{[]}, h_0 :: t_0, \cdots, h_n :: t_n\}) \;= \\
\qquad \mathit{matchCheck}_{(E_N,E_C)}(\bigcup_{0\leqslant i\leqslant n}\{h_i\}) \wedge \mathit{matchCheck}_{(E_N,E_C)}(\bigcup_{0\leqslant i\leqslant n}\{t_i\}) \\
\mathit{matchCheck}_{(E_N,E_C)}(P = \{(p_i^0,\cdots,p_i^n) \mid i \in I\}) \;= \\
\qquad \bigwedge_{0\leqslant j\leqslant n} \mathit{matchCheck}_{(E_N,E_C)}(\{t_i^j \mid i \in I\}) \\
\mathit{matchCheck}_{(E_N,E_C)}(P = \{c_0\ p_0, \cdots, c_n\ p_n\}) \;= \\
\qquad (\mathit{find}(c_0,E_C) = \cdots = \mathit{find}(c_n,E_n) = C) \qquad \textit{(tous les constructeurs sont présents)} \\
\qquad \wedge \bigwedge_{g\in G} \mathit{matchCheck}_{(E_N,E_C)}(g) \qquad \textit{(vérification des sous-motifs)} \\
\qquad \text{où } C = \{c \mid c\ p \in P\} \qquad \textit{(constructeurs apparaissant)} \\
\qquad \text{où } G = \{\{p \mid c\ p \in P\} \mid c \in C\} \qquad \textit{(motifs groupés par constructeur)} \\
\mathit{matchCheck}_{(E_N,E_C)}(P = \{\texttt{node <}t_i\texttt{> } p_i \texttt{ prop } l^0 \texttt{ = } p_i^0 \cdots \texttt{ prop } l^n \texttt{ = } p_i^n \mid i \in I\}) \;= \\
\qquad \bigcup_i \{t_i\} = E_N \qquad \textit{(toutes les étiquettes sont présentes)} \\
\qquad \wedge \bigwedge_{g\in G} \mathit{matchCheck}_{(E_N,E_C)}(g) \qquad \textit{(vérification des sous-motifs)} \\
\qquad \wedge \bigwedge_{g_j\in G_j, 0\leqslant j\leqslant n} \mathit{matchCheck}_{(E_N,E_C)}(g_j) \\
\qquad \text{où } E = \{e \mid \texttt{node <}e\texttt{> } \cdots \texttt{ end} \in P\} \qquad \textit{(étiquettes apparaissant)} \\
\qquad \text{où } G = \{\{p \mid \texttt{node <}e\texttt{> p } \cdots \texttt{ end} \in P\} \mid e \in E\} \qquad \textit{(motifs de liste groupés par constructeur)} \\
\qquad \text{où } G_j = \{\{p^j \mid \texttt{node <}e\texttt{> } \cdots \texttt{ prop } l^j \texttt{ = } p^j \cdots \texttt{ end} \in P\} \mid e \in E\} \text{ où } 0 \leqslant i \leqslant n \\
\qquad\qquad \textit{(motifs de chaque propriété groupés par constructeur)} \\
\mathit{matchCheck}_{(E_N,E_C)}(\text{autres cas}) \;= F
\end{array}$$

FIGURE 11.3: Vérification d'exhaustivité du filtrage.

Définition de l'analyse La vérification est donnée dans la figure 11.3.

- La fonction principale *matchProgCheck* vérifie qu'aucun filtrage n'intervient avant que l'ensemble des types de nœuds soit fixé. Son premier paramètre booléen, initialement à F, passe à T dès qu'un filtrage de nœuds a été effectué. Le programme est refusé si ce paramètre vaut T et que l'analyse rencontre une nouvelle définition de type de nœud.
- Pour effectuer l'exhaustivité du filtrage, et vérifier qu'une étiquette n'est pas déjà utilisée, elle accumule l'ensemble des étiquettes de nœuds dans un paramètre $E_N \subseteq (\mathit{tag})$.
- Elle collecte aussi les définitions de constructeurs afin de vérifier l'exhaustivité sur les types sommes dans un paramètre accumulateur $E_C \subseteq (\mathit{cstr} \times \mathit{id})$, qui mémorise pour chaque construc-

teur le nom du type associé, pour prendre en compte le masquage de constructeurs.
- Puis *matchExprCheck* vérifie que tous les filtrages sont exhaustifs au sein d'une expression.
- Cette dernière utilise *matchCheck* qui vérifie qu'un ensemble de motifs traite bien tous les cas possibles. Pour simplifier. *matchCheck* n'est définie que sur les motifs bien typés.

11.2 Système de types

Dans un langage comme ML, chaque expression du langage doit pouvoir se voir attribuer un type pour que le programme soit accepté. On dit alors qu'il est *bien typé*. La légende [📖47] dit même qu'un programme bien typé se comporte correctement (nous donnerons des éléments de correction section 12.6). Le système de types présenté ici n'est cependant pas auto-suffisant, et la sûreté d'exécution n'est assurée que si les analyses statiques de la section précédente ont elles-aussi accepté le programme. Concrètement, il est possible d'appliquer les analyses avant ou après la phase de typage. Une implantation pourrait aussi effectuer à la demande les vérifications en même temps que l'inférence de types.

Le système de types est celui classique défini par Hindley et Milner [📖47] pour le langage ML, auquel a été ajouté la gestion des nœuds présentée en introduction. La correction du polymorphisme en présence d'effets de bord est assurée par la *value restriction* de Wright [📖52].

Nous présentons d'abord à la section 11.2.1 la forme des types et les environnements de typage. Nous donnons aussi le traitement des définitions de types de données personnalisés et des annotations de types en tenant compte des différents espaces de noms, toujours dans l'esprit de fournir une spécification d'implantation complète. Puis, à la section 11.2.2 nous donnons l'ensemble des règles de typage du cœur du langage, et l'étendons avec les règles spécifiques aux nœuds.

11.2.1 Types et définitions de types

L'ensemble des types sur lesquels travaille le système est directement engendré par la grammaire suivante.

τ	::=	int | bool | unit | string	*types prédéfinis*
	|	$\tau \to \tau$	*type fonctionnel*
	|	$\tau \times \cdots \times \tau$	*type produit*
	|	α	*variable de type*
	|	node	*type générique des nœuds*
	|	<*tag*> node	*nœuds étiquetés*
	|	$(\tau, \cdots, \tau)$ *id*	*instance de type personnalisé paramétré*

Cette grammaire est exactement la grammaire concrète des types du langage, dans laquelle on a simplement substitué les notations concrètes par celles usuelles des systèmes de types : (*type* $\equiv \tau$), (*type-var* $\equiv \alpha$), (-> $\equiv \to$) et (* $\equiv \times$), et où on regroupe les cas des types à 0, 1 ou n paramètres. La conversion étant sans aucune ambiguïté, on se permettra dans la suite de confondre la syntaxe concrète et la syntaxe formelle.

Environnement de définitions Les définitions de types et de types de nœuds enrichissent un environnement global D :

$$D = (D_A, \overbrace{D_R\ ,\ D_F}^{\text{enregistrements}}\ , \overbrace{D_N\ ,\ D_P}^{\text{nœuds}}, \overbrace{D_S\ ,\ D_C}^{\text{sommes}})$$

Les composantes D_X servent à encoder les différents espaces de noms. Chaque composante est encodée soit par une structure de fonction partielle lorsque l'espace de nom ne permet pas la redéfinition (types, propriétés et étiquettes), soit par une liste associative (selon la définition de la figure 8.2) si l'espace de nom permet la redéfinition, l'ordre induit par la structure de liste étant alors utilisé pour représenter le masquage.

- $D_A \subset (id \times (List(var) \times \tau))$ est l'ensemble des alias de types (types qui ne définissent pas de nouvelles structures de données) définis par le programmeur. Il associe les noms des alias aux paramètres et aux types associés. Contrairement aux définitions de types algébriques suivantes, cet ensemble ne peut contenir de définitions cycliques.
- $D_R \subset (id \times (List(var) \times \mathcal{P}(id)))$ associe chaque nom de type enregistrement à ses paramètres et à l'ensemble des noms de ses champs.
- $D_F \in List(id \times (id \times \tau \times \{T, F\}))$ représente l'espace de noms des champs d'enregistrement et associe à chaque nom de champ le nom du type produit auquel il appartient, son propre type, et s'il est mutable ou non. L'ordre qu'implique la structure de liste représente le masquage des noms de champs.
- $D_N \subset (tag \times \mathcal{P}(id))$ associe à chaque étiquette de nœud l'ensemble de ses propriétés.
- $D_P \subset (id \times (\tau \times \{T, F\}))$ associe à chaque nom de propriété le type associé et la mutabilité. Contrairement aux champs d'enregistrement, il n'y a pas de masquage. Si deux types de nœuds définissent deux propriétés avec le même nom, alors elles doivent avoir le même type et la même mutabilité.
- $D_S \subset (id \times (List(var) \times \mathcal{P}(cstr)))$ associe chaque nom de type somme à ses paramètres et à l'ensemble des noms de ses constructeurs.
- $D_C \in List(id \times (id \times (\tau \cup \bot)))$ représente l'espace de noms des constructeurs et associe à chaque constructeur le nom du type somme auquel il appartient et son propre type, s'il en a un. Comme pour les champs d'enregistrements, l'ordre qu'implique la structure de liste représente le masquage (le premier constructeur dans la liste est le dernier défini).

Notations Afin de simplifier et uniformiser les écritures, on utilise les notations suivantes, qui utilisent soit celles usuelles des fonctions, soit celles que nous avons défini pour les listes associatives :

- $dom(D_X)$ représente l'ensemble des noms définis dans l'espace de noms représenté par D_X.
- $D_X(n)$, définie sur $dom(D_X)$, renvoie la définition associée à n dans cet de noms. Si l'espace de noms permet la redéfinition, il s'agit de la dernière définition.
- On définit aussi le prédicat d'appartenance d'un type à l'environnement de définitions selon la notation suivante.

$$
\begin{array}{rcl}
(\alpha_0, \cdots, \alpha_n)\ t \in dom(D_A) & \equiv & (t, (\alpha_0 :: \cdots :: \alpha_n :: [\,], \tau)) \in D_A \\
(\alpha_0, \cdots, \alpha_n)\ t \in dom(D_R) & \equiv & (t, (\alpha_0 :: \cdots :: \alpha_n :: [\,], F)) \in D_R \\
(\alpha_0, \cdots, \alpha_n)\ t \in dom(D_S) & \equiv & (t, (\alpha_0 :: \cdots :: \alpha_n :: [\,], C)) \in D_S \\
(\alpha_0, \cdots, \alpha_n)\ t \in dom(D) & \equiv & (\alpha_0, \cdots, \alpha_n)\ t \in dom(D_A) \\
 & \vee & (\alpha_0, \cdots, \alpha_n)\ t \in dom(D_R) \\
 & \vee & (\alpha_0, \cdots, \alpha_n)\ t \in dom(D_S)
\end{array}
$$

Définitions de types Lors des définitions de types (ainsi que lors de la vérification des annotations de types dans les expressions) il faut s'assurer que le type donné par le programmeur a du sens dans l'environnement de typage courant. La fonction $wd(\tau, D, V)$ donnée figure 11.4 vérifie que τ est correctement défini dans l'environnement D. Elle est définie par induction sur la structure des types, s'assurant que

chaque type nommé apparaissant est bien défini dans D, et avec le bon nombre de paramètres le cas échéant. La fonction prend aussi l'ensemble des variables de types autorisées à apparaître.

$$
\begin{array}{rcl}
wd(\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{string}, D, V) & = & T \\
wd(\mathsf{node}, D, V) & = & T \\
wd(\alpha, D, V) & = & T \text{ si } \alpha \in V, F \text{ sinon} \\
wd(\tau_0 \times \cdots \times \tau_n, D, V) & = & wd(\tau_0, D, V) \wedge \cdots \wedge wd(\tau_n, D, V) \\
wd(\tau_a \to \tau_b, D, V) & = & wd(\tau_a, D, V) \wedge wd(\tau_b, D, V) \\
wd((\tau_0, \cdots, \tau_n)\ t, D, V) & = & (\alpha_0, \cdots, \alpha_n)\ t \in dom(D) \\
 & & \wedge wd(\tau_0, D, V) \wedge \cdots \wedge wd(\tau_n, D, V) \\
wd(t, D, V) & = & t \in dom(D) \\
wd(\text{<}t\text{>}\ \mathsf{node}, D, V) & = & T \textit{(utilisation avant définition possible)}
\end{array}
$$

FIGURE 11.4: Vérification de type dans un environnement de définitions.

L'environnement peut alors être enrichi de façon correcte par la fonction *def* de la figure 11.5 en utilisant *wd*.

$$
\begin{array}{l}
def(D, \mathsf{type}\ (\alpha_0, \cdots, \alpha_n)\ \mathtt{t}\ \mathtt{=}\ \{\ l_0\ :\ e_{m_0}\ \tau_0\ ;\ \cdots\ ;\ l_m\ :\ e_{m_m}\ \tau_m\ \}) = \\
\quad (D_A, D'_R, D'_F, D_N, D_P, D_S, D_C) = D' \\
\qquad \text{où } D'_F = (l_0, (\tau_0, m_0)) :: \cdots :: (l_m, (\tau_m, m_m)) :: D_F \\
\qquad \text{où } m_i = T \text{ si } e_{m_i} = \mathsf{mutable}, F \text{ sinon}, 0 \leqslant i \leqslant m \\
\qquad \text{et } D'_R = D_R \cup \{(t, (\alpha_0 :: \cdots :: \alpha_n :: [], \{l_i \mid 0 \leqslant i \leqslant m\}))\} \\
\qquad \text{si } \forall i, \forall j, l_i \neq l_j \text{ et } \forall 0 \leqslant i \leqslant m, wd(\tau_i, D', \{\alpha_k, 0 \leqslant k \leqslant n\}) \\
def(D, \mathsf{type}\ (\alpha_0, \cdots, \alpha_n)\ \mathtt{t}\ \mathtt{=}\ c_0\ e_0\ \mathtt{|}\ \cdots\ \mathtt{|}\ c_m\ e_m) = \\
\quad (D_A, D_R, D_F, D_N, D_P, D'_S, D'_C) = D' \\
\qquad \text{où } \tau_i = \begin{cases} \tau_{e_i} \text{ si } e_i = \mathsf{of}\ \tau_{e_i} \\ \bot \text{ si } e_i = \epsilon \end{cases} \text{ pour } 0 \leqslant i \leqslant m \\
\qquad \text{où } D'_C = (c_0, \tau_0) :: \cdots :: (c_m, \tau_m) :: D_C \\
\qquad \text{et } D'_S = D_S \cup \{(t, (\alpha_0 :: \cdots :: \alpha_n :: [], \{c_i \mid 0 \leqslant i \leqslant m\}))\} \\
\qquad \text{si } \forall i, \forall j, c_i \neq c_j \text{ et } \forall 0 \leqslant i \leqslant m, wd(\tau_i, D', \{\alpha_k, 0 \leqslant k \leqslant n\}) \\
def(D, \mathsf{type}\ (\alpha_0, \cdots, \alpha_n)\ \mathtt{t}\ \mathtt{=}\ \tau) = \\
\quad (D'_A, D_R, D_F, D_N, D_P, D_S, D_C) \\
\qquad \text{où } D'_A = D_A \cup \{(t, (\alpha_0 :: \cdots :: \alpha_n :: [], \tau))\} \\
\qquad \text{si } wd(\tau, D, \{\alpha_k, 0 \leqslant k \leqslant n\}) \\
def(D, \mathsf{node\ type}\ \text{<}tag\text{>}\ \mathsf{prop}\ e_{m_0}\ p_0\ :\ \tau_0\ \cdots\ \mathsf{prop}\ e_{m_m}\ p_m\ :\ \tau_m\ \mathsf{end}) = \\
\quad (D_A, D_R, D_F, D'_N, D'_P, D_S, D_C) \\
\qquad \text{où } D'_N = D_N \cup \{(tag, \{p_i \mid 0 \leqslant i \leqslant m\})\} \\
\qquad \text{où } m_i = T \text{ si } e_{m_i} = \mathsf{mutable}, F \text{ sinon}, 0 \leqslant i \leqslant m \\
\qquad \text{et } D'_P = D_P \cup \{(p_i, (\tau_i, m_i)) \mid 0 \leqslant i \leqslant m\} \\
\qquad \text{si } t \notin dom(D_N) \text{ et } \forall 0 \leqslant i \leqslant m, l_i \in dom(D_P) \Rightarrow D_P(l_i) = \tau_i, m_i \\
\qquad \text{et } \forall i, \forall j, p_i \neq p_j \text{ et } \forall 0 \leqslant i \leqslant m, wd(\tau_i, D', \emptyset)
\end{array}
$$

FIGURE 11.5: Enrichissement d'un environnement de définitions.

Un type somme (resp. enregistrement) est bien défini si ses constructeurs (resp. champs) sont tous

différents et associés à un type bien défini. Comme en Caml [1], le système fournit définitions iso-récursives, mais pas équi-récursives. En clair, la récursion doit être gardée par une construction, les définitions de types sommes et enregistrements sont donc implicitement récursives, mais les alias ne le sont pas.

11.2.2 Typage ML

Le système de types de FidoML est classiquement décrit comme un ensemble de *règles d'inférence* de la forme $(\text{Règle})\frac{\textit{prémisses}}{\Gamma, D \vdash \textit{expr} : \tau}$ se lisant : τ est un type correct pour une expression de la forme *expr* si les prémisses sont vérifiées dans un environnement de typage Γ et un environnement de définitions D. La formule $\Gamma, D \vdash \textit{expr} : \tau$ est ce qu'on appelle *jugement de typage*. L'environnement D a déjà été présenté, l'environnement Γ, que nous présenterons juste après, contient les variables, globales et locales, définies à cet endroit du programme, et les associe à leurs types. Les *prémisses* sont un ensemble de jugements de typage et conditions devant tous être vérifiés pour que la règle soit applicable. En général, *expr* ne représente pas une expression unique, mais est définie de façon symbolique, tout ou partie de ses sous-expressions étant des variables. Les prémisses sont donc définies en fonction de ces variables, de même que le type τ.

On dit alors qu'une expression e est *typable* dans un environnement Γ, D si il est possible de *dériver* un jugement de typage $\Gamma, D \vdash e : \tau$, depuis une des règles du système. Pour qu'une telle dérivation soit possible, il faut bien sûr trouver une instanciation des symboles au sein de la règle telle que l'expression symbolique soit instanciée en e, mais il faut aussi que les prémisses ainsi instanciées soient des formules logiques correctes. En particulier, si les prémisses contiennent des jugements de typage à vérifier (ce qui est en général le cas pour les expressions composées), il faut aussi dériver une règle du système pour chacune de ces prémisses, et ainsi de suite récursivement (de façon finie). On parle alors d'*arbre de dérivation* ou d'*arbre d'inférence*.

Pour ce système, nous avons choisi un ensemble de règles *dirigés par la syntaxe*, c'est-à-dire qu'il n'y a qu'un choix de règle possible pour une expression donnée de la syntaxe concrète. Cette approche utilise des règles un peu plus compliquées que si on permettait le choix de plusieurs règles pour une même expression, mais elle est intéressante car elle facilite la définition d'un algorithme d'inférence de types. Nous donnons d'abord l'ensemble de jugements de typage correspondant au Hindley-Milner classique. Nous l'enrichirons ensuite progressivement avec les définitions de types personnalisés, puis les règles spécifiques auxnœuds.

Cœur du langage Les règles de typage pour les constantes n'ont pas de prémisses, ce sont, avec les variables, les feuilles de l'arbre d'inférence.

$$(\text{Int})\frac{}{\Gamma, D \vdash i : \mathsf{int}} \qquad (\text{Unit})\frac{}{\Gamma, D \vdash \texttt{()} : \mathsf{unit}}$$

$$(\text{True})\frac{}{\Gamma, D \vdash \mathsf{true} : \mathsf{bool}} \qquad (\text{False})\frac{}{\Gamma, D \vdash \mathsf{false} : \mathsf{bool}}$$

La règle de typage de la construction de liste introduit l'homogénéité de listes d'ML en reliant par un même symbole τ le jugement de typage de la sous-expression correspondant à la tête et celui correspondant à la queue. De même, le type de l'expression est lié par le même symbole τ faisant le lien entre le type de l'expression et les types des sous-expressions. Bien sûr, cette homogénéité va se propager dans les deux sous-arbres d'inférence des sous-expressions. En particulier, la liste vide peut être considérée comme une liste d'éléments de n'importe quelle type τ, mais ce symbole τ devra être cohérent avec le reste de l'arbre d'inférence.

1. Le compilateur `ocaml` peut permettre l'équi-récursion via l'option `-rectypes`.

$$(\textsc{List-2})\frac{\Gamma, D \vdash e \ : \ \tau \qquad \Gamma, D \vdash l \ : \ \tau \ \mathsf{list}}{\Gamma, D \vdash e \ :: \ l \ : \ \tau \ \mathsf{list}} \qquad (\textsc{List-1})\frac{}{\Gamma, D \vdash \texttt{[]} \ : \ \tau \ \mathsf{list}}$$

Pour les n-uplets, la règle de typage introduit un symbole différent par composante, qui sont indépendants comme voulu. Les mêmes symboles se retrouvent dans le type global du n-uplet, faisant le lien entre les composantes du type produit et les types des composantes du produit.

Pour la projection de composante d'un n-uplet, il faut vérifier que l'expression à projeter a bien un type de n-uplet, que la constante de taille donnée par le programmeur est bien ce n, et que la projection i demandée est bien dans l'intervalle $[0, n-1]$. Le type de l'expression d'accès est alors lié symboliquement à la i^{e} composante du type du n-uplet.

$$(\textsc{Tuple})\frac{\Gamma, D \vdash e_0 \ : \ \tau_0 \qquad \cdots \qquad \Gamma, D \vdash e_n \ : \ \tau_n}{\Gamma, D \vdash (e_0, \cdots, e_n) \ : \ \tau_0 \times \cdots \times \tau_n}$$

$$(\textsc{Proj})\frac{\Gamma, D \vdash e \ : \ \tau_0 \times \cdots \times \tau_n \qquad 0 \leqslant i \leqslant n}{\Gamma, D \vdash e.(\mathsf{proj} \ i \ / \ n) \ : \ \tau_i}$$

Pour qu'une expression if soit typable et soit de type τ, il faut pouvoir donner ce même type τ aux deux branches.

$$(\textsc{IfThenElse})\frac{\Gamma, D \vdash e \ : \ \mathsf{bool} \qquad \Gamma, D \vdash e_t \ : \ \tau \qquad \Gamma, D \vdash e_f \ : \ \tau}{\Gamma, D \vdash \mathsf{if} \ e \ \mathsf{then} \ e_t \ \mathsf{else} \ e_f \ : \ \tau}$$

$$(\textsc{IfThen})\frac{\Gamma, D \vdash e \ : \ \mathsf{bool} \qquad \Gamma, D \vdash e_t \ : \ \mathsf{unit}}{\Gamma, D \vdash \mathsf{if} \ e \ \mathsf{then} \ e_t \ : \ \mathsf{unit}}$$

Une définition de fonction est typable si son corps est typable dans un environnement où le nom du paramètre est lié au type de son argument. Réciproquement, une application est bien typée si l'expression gauche est bien d'un type flèche, dont la partie gauche correspond au type de l'argument.

$$(\textsc{App})\frac{\Gamma, D \vdash e_f \ : \ \tau_1 \to \tau_2 \qquad \Gamma, D \vdash e_a \ : \ \tau_1}{\Gamma, D \vdash e_f \ e_a \ : \ \tau_2} \qquad (\textsc{Abs})\frac{(n, \tau_1) \ :: \ \Gamma, D \vdash \ e \ : \ \tau_2}{\Gamma, D \vdash \mathsf{fun} \ n \ \texttt{->} \ e \ : \ \tau_1 \ \to \ \tau_2}$$

Polymorphisme Le sous-ensemble du système que nous avons présenté jusqu'ici ne suffit par pour prendre en charge le polymorphisme à la ML. Plus précisément, le mécanisme simple de liaison symbolique des types des expressions et de leurs sous-expressions dans l'arbre d'inférence fait qu'il est impossible de donner un type générique (contenant une variable de type) à une sous-expression intrinsèquement générique, si celle-ci est utilisée par ailleurs sous un type moins générique. C'est exactement ce que nous avons vu avec le type de la liste vide, dont le paramètre doit être le même que celui du type de l'expression de liste qui la contient, alors que la liste vide est intrinsèquement polymorphe.

Nous voulons donc permettre de donner un type générique, contenant des variables, à une sous-expression, et l'utiliser par ailleurs dans l'arbre avec différentes valeurs de ces variables. Pour ceci, on introduit une notion de *schéma de types*, qui est un type dont les variables (ou éventuellement seulement une partie) sont placées sous quantification universelle. Dans Hindley-Milner, on introduit seulement les quantifications en position prénexe. Concrètement, nous utilisons donc la notation σ définie ci-dessous pour les schémas de types.

$$\sigma \ ::= \ \forall \alpha.\sigma \ \mid \ \tau$$

Il faut alors introduire dans le système ces schémas de types par deux mécanismes permettant l'introduction et l'élimination des quantifications, la *généralisation* et l'*instanciation*.

Généralisation et instanciation La généralisation construit un type polymorphe à partir d'un type contenant des variables, en quantifiant tout ou partie de ses variables, de façon appropriée suivant le contexte (nous verrons qu'il n'est pas toujours possible de généraliser toutes les variables). Sa réciproque, l'instanciation, construit un type spécialisé à partir d'un type polymorphe en remplaçant des variables et leurs quantifications par des types là encore cohérents par rapport au contexte.

Dans un système de règles non dirigé par la syntaxe, il est possible d'introduire une règle (Inst) et une règle (Gen) pouvant s'insérer n'importe-où dans l'arbre. Classiquement, dans notre système à la Hindley-Milner dirigé par la syntaxe, et comme c'est le cas dans l'algorithme d'inférence induit, les schémas de types n'apparaissent pas directement dans les jugements de typage, mais sont introduits dans le système via l'environnement de typage. La généralisation est donc faite explicitement à l'aide d'une fonction $gen(\tau, \Gamma)$ lors de la règle (Let), qui enrichit l'environnement d'un schéma de type. L'instanciation est alors faite à l'aide d'une relation $\leqslant$ au niveau des variables, en spécialisant le type associé dans l'environnement.

L'environnement de typage Γ est donc une liste associative de noms à des schémas de types :

$$\Gamma \in List(id \times \sigma)$$

La fonction de généralisation $gen(\tau, \Gamma)$ quantifie les variables de types $\alpha_0, \cdots, \alpha_n$ présentes dans τ et n'apparaissant pas libres dans Γ pour constituer le schéma de type $\forall\alpha_0.\cdots\forall\alpha_n.\tau$. La définition formelle est donnée figure 11.6.

$$\begin{aligned}
gen(\tau, \Gamma) &= \forall\alpha_0.\cdots\forall\alpha_n.\tau \text{ où } \{\alpha_i \mid 0 \leqslant i \leqslant n\} = \mathcal{F}(\tau)\backslash\mathcal{F}(\Gamma) \\
&\text{avec } \mathcal{F} \text{ la collecte des variables libres dans les schémas :} \\
\mathcal{F}(\forall\alpha.\sigma) &= \mathcal{F}(\sigma)\backslash\{\alpha\} \\
\mathcal{F}(\alpha) &= \{\alpha\} \\
\mathcal{F}(\mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{string}) &= \emptyset \\
\mathcal{F}(\mathsf{node} \mid \mathsf{<}t\mathsf{>\ node}) &= \emptyset \\
\mathcal{F}(\tau_a \to \tau_b) &= \mathcal{F}(\tau_a) \cup \mathcal{F}(\tau_b) \\
\mathcal{F}(\tau_0 \times \cdots \times \tau_n) &= \mathcal{F}(\tau_0) \cup \cdots \cup \mathcal{F}(\tau_n) \\
\mathcal{F}((\tau_0, \cdots, \tau_n)\ t) &= \mathcal{F}(\tau_0) \cup \cdots \cup \mathcal{F}(\tau_n) \\
&\text{et dans les environnements :} \\
\mathcal{F}([\,]) &= \emptyset \\
\mathcal{F}((n, \tau) :: t) &= \mathcal{F}(\tau) \cup \mathcal{F}(t)
\end{aligned}$$

Figure 11.6: Fonction de généralisation.

La relation d'instanciation est alors notée $\sigma \leqslant \tau$ et définie comme suit.

$$\exists\tau_i, 0 \leqslant i \leqslant n, \tau = \tau'[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant n] \Rightarrow \forall\alpha_0.\cdots\forall\alpha_n.\tau' \leqslant \tau$$

Généralisation en présence d'effets de bord Nous utilisons la *value restriction* de Wright [📖52], qui montre qu'en présence d'effets de bord, la généralisation au let est possible, si l'expression liée est dite *non expansive* selon la définition syntaxique de la figure 11.7. Nous étendons trivialement la définition aux nœuds, puisque ceux-ci ne sont pas polymorphes.

Liaisons locales Avec ces définitions, on peut finalement donner le typage des définitions et utilisations de variables. Si on lie une expression non expansive, celle-ci devient polymorphe dans l'environnement de typage du corps de la définition. S'il s'agit d'une expression expansive, les éventuelles inconnues dans son type se retrouveront tel quel (non généralisées) dans l'environnement du corps de la définition.

$$(\textsc{Var})\frac{\Gamma(x) = \sigma \qquad \sigma \leqslant \tau}{\Gamma, D \vdash x \ : \ \tau}$$

$$(\textsc{LetIn-1})\frac{\neg \mathit{expansive}(e_d) \qquad \Gamma, D \vdash e_d \ : \ \tau_d \qquad (n, \mathit{gen}(\tau_d, \Gamma)) :: \Gamma, D \vdash e_b \ : \ \tau_b}{\Gamma, D \vdash \mathsf{let} \ n \ \mathsf{=} \ e_d \ \mathsf{in} \ e_b \ : \ \tau_b} \qquad (\textsc{LetIn-2})\frac{\mathit{expansive}(e_d) \qquad \Gamma, D \vdash e_d \ : \ \tau_d \qquad (n, \tau_d) :: \Gamma, D \vdash e_b \ : \ \tau_b}{\Gamma, D \vdash \mathsf{let} \ n \ \mathsf{=} \ e_d \ \mathsf{in} \ e_b \ : \ \tau_b}$$

$$(\textsc{LetRecIn-1})\frac{\neg \mathit{expansive}(e_d) \qquad (n, \tau_d) :: \Gamma, D \vdash e_d \ : \ \tau_d \qquad (n, \mathit{gen}(\tau_d, \Gamma)) :: \Gamma, D \vdash e_b \ : \ \tau_b}{\Gamma, D \vdash \mathsf{let\ rec} \ n \ \mathsf{=} \ e_d \ \mathsf{in} \ e_b \ : \ \tau_b} \qquad (\textsc{LetRecIn-2})\frac{\mathit{expansive}(e_d) \qquad (n, \tau_d) :: \Gamma, D \vdash e_d \ : \ \tau_d \qquad (n, \tau_d) :: \Gamma, D \vdash e_b \ : \ \tau_b}{\Gamma, D \vdash \mathsf{let\ rec} \ n \ \mathsf{=} \ e_d \ \mathsf{in} \ e_b \ : \ \tau_b}$$

Liaisons globales Pour le typage des phrases du programme, on introduit une nouvelle forme de règles $(\textsc{Règle})\frac{\textit{prémisses}}{\Gamma, D \vdash p \ : \ \Gamma', D'}$. En d'autres termes, on considère que le type d'un programme est l'ensemble des liaisons globales avec leurs types, ainsi que l'ensemble des définitions de types personnalisés.

De façon similaire aux liaison locales, si on lie une expression non expansive, celle-ci devient polymorphe dans l'environnement de typage de la suite du programme. Par contre, dans FidoML, on n'autorise pas la liaison globale d'expression de type non généralisable.

$$(\textsc{T-Let})\frac{\neg \mathit{expansive}(e) \qquad \Gamma, D \vdash e \ : \ \tau \qquad \Gamma' = (n, \mathit{gen}(\tau, \Gamma)) :: \Gamma}{\Gamma, D \vdash \mathsf{let} \ n \ \mathsf{=} \ e \ : \ \Gamma', D} \qquad (\textsc{T-LetRec})\frac{\neg \mathit{expansive}(e) \qquad (n, \tau) :: \Gamma, D \vdash e \ : \ \tau \qquad \Gamma' = (n, \mathit{gen}(\tau, \Gamma)) :: \Gamma}{\Gamma, D \vdash \mathsf{let\ rec} \ n \ \mathsf{=} \ e \ : \ \Gamma', D}$$

$$(\textsc{T-Expr})\frac{\Gamma, D \vdash e \ : \ \tau}{\Gamma, D \vdash e \ : \ \Gamma, D} \qquad (\textsc{T-Seq})\frac{\Gamma, D \vdash p_1 \ : \ \Gamma', D' \qquad \Gamma', D' \vdash p_2 \ : \ \Gamma'', D''}{\Gamma, D \vdash p_1 \ ; ; \ p_2 \ : \ \Gamma'', D''}$$

$$\begin{aligned}
\mathit{expansive}(\mathsf{fun} \ e_a \ \mathsf{->} \ e_b) &= F \\
\mathit{expansive}(e_1 \ ; \ e_2) &= \mathit{expansive}(e_2) \\
\mathit{expansive}(\mathsf{let\ n} \ \mathsf{=} \ e_1 \ \mathsf{in} \ e_2) &= \mathit{expansive}(e_1) \vee \mathit{expansive}(e_2) \\
\mathit{expansive}(n) &= F \\
\mathit{expansive}(\mathit{cstr} \ e) &= \mathit{expansive}(e) \\
\mathit{expansive}((e_0, \cdots, e_n)) &= \mathit{expansive}(e_0) \vee \cdots \vee \mathit{expansive}(e_n) \\
\mathit{expansive}(\{ \ l_0 \ \mathsf{=} \ e_0 \ ; \ \cdots \ ; \ e_n \ \}) &= \mathit{expansive}(e_0) \vee \cdots \vee \mathit{expansive}(e_n) \\
\mathit{expansive}(\mathsf{node\ <}t\mathsf{>} \ \cdots \ \mathsf{end}) &= F \\
\mathit{expansive}(\text{autres cas}) &= T
\end{aligned}$$

FIGURE 11.7: Expressions expansives.

11.2.3 Types personnalisés

Il y a deux possibilités principales pour typer les structures de données définies par le programmeur. Une approche possible est d'enrichir l'environnement de typage de fonctions virtuelles de création,

d'accès et de modification. Les jugements de typage se contentent alors d'ajouter un étage faisant la traduction entre les structures de la syntaxe concrète et des applications virtuelles des fonctions décrites précédemment. Ce type d'encodage est en général plutôt utilisé lorsqu'on cherche à décrire un modèle théorique minimal du système de types, sur lequel on cherche à montrer des propriétés et où on ne se soucie pas les problèmes pratiques comme le masquage, l'exhaustivité des champs, etc.

Dans notre optique de donner une spécification en vue d'implantation, nous avons préféré ajouter les définitions et utilisations de types personnalisés sans les transfigurer. Nous proposons alors des règles un peu plus nombreuses et complexes, mais moins artificielles, ne laissant pas de côté des détails non centraux mais néanmoins nécessaires à résoudre en vue d'une implantation.

Nous pouvons alors étendre l'ensemble des règles de typage au niveau programme pour prendre en compte les définitions de types. Ces règles utilisent la fonction *def* définie plus tôt pour enrichir l'environnement D pour le typage de la suite du programme, et par extension pour le type du programme lui-même.

$$(\text{T-Def})\frac{D' = \mathit{def}(\texttt{type}\ \ (\alpha_0, \cdots, \alpha_n)\ \ \texttt{t = } d_t, D)}{\Gamma, D \vdash \texttt{type}\ \ (\alpha_0, \cdots, \alpha_n)\ \ \texttt{t = } d_t\ \texttt{ : }\ \Gamma, D'}$$

$$(\text{T-DefNode})\frac{D' = \Gamma, \mathit{def}(\texttt{node type <}\mathit{tag}\texttt{> } \cdots \texttt{ end}, D)}{\Gamma, D \vdash \texttt{node type <}\mathit{tag}\texttt{> } \cdots \texttt{ end}\ \texttt{ : }\ \Gamma, D'}$$

Enregistrements Pour qu'une création d'environnement il faut, dans l'ordre des prémisses :

- vérifier que les champs présents dans l'expression appartiennent tous à un même type t enregistrement, en les recherchant dans la composante D_F de l'environnement de définitions D,
- vérifier que l'ensemble de ces types est exhaustif, en le comparant à l'ensemble des étiquettes associé à t dans D_R
- trouver une instanciation $(\tau_0, \cdots, \tau_m)$ des paramètres $(\alpha_0, \cdots, \alpha_m)$ de t, telle que l'expression donnée pour chaque champ l_i est typable dans l'environnement, et que son type est bien l'instanciation de celui donné lors de la définition.

$$(\text{Record})\frac{\begin{array}{c} D_F(l_i) = (t_{l_i}, \tau_{l_i}) \text{ où } 0 \leqslant i \leqslant n \qquad t_{l_0} = \cdots = t_{l_n} \\ (\alpha_0, \cdots, \alpha_m)\ t \in \mathit{dom}(D_R) \qquad D_R(t) = \{l_i \mid 0 \leqslant i \leqslant n\} \\ \tau_{e_i} = \tau_{l_i}[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant m] \qquad \Gamma, D \vdash e_i\ \texttt{ : }\ \tau_{e_i} \text{ où } 0 \leqslant i \leqslant n \end{array}}{\Gamma, D \vdash \texttt{\{ } l_0\ \texttt{ = } e_0\ \texttt{ ; } \cdots \texttt{ ; } l_n\ \texttt{ = } e_n\ \texttt{ \}}\ \texttt{ : }\ (\tau_0, \cdots, \tau_m)\ t}$$

L'accès à un champ, étant donné une expression correctement typée comme un enregistrement, donne le type défini lors de la déclaration pour le champ sélectionné, dans lequel les variables ont été correctement instanciées par rapport aux paramètres du type de l'enregistrement.

$$(\text{Field})\frac{\begin{array}{c} D_F(l) = (t, \tau_l, b) \qquad (\alpha_0, \cdots, \alpha_m)\ t \in \mathit{dom}(D_R) \\ \Gamma, D \vdash e\ \texttt{ : }\ (\tau_0, \cdots, \tau_m)\ t \qquad \tau = \tau_l[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant m] \end{array}}{\Gamma, D \vdash e\texttt{.(field } l\texttt{)}\ \texttt{ : }\ \tau}$$

$$(\text{Field-Set})\frac{\begin{array}{c} D_F(l) = (t, \tau_l, T) \qquad (\alpha_0, \cdots, \alpha_m)\ t \in \mathit{dom}(D_R) \\ \Gamma, D \vdash e\ \texttt{ : }\ (\tau_0, \cdots, \tau_m)\ t \qquad \Gamma, D \vdash e_v\ \texttt{ : }\ \tau \qquad \tau = \tau_l[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant m] \end{array}}{\Gamma, D \vdash e\texttt{.(field } l\texttt{)}\ \texttt{ <- }\ e_v\ \texttt{ : unit}}$$

Constructeurs La création de valeur construite d'un type somme suit la même logique que la construction d'enregistrement. Elle fait le lien entre le type de l'expression encapsulée et celui demandé pour

ce constructeur dans la définition grâce à une substitution par les paramètres effectifs du type résultant. Puisque tous les paramètres de type n'apparaissent pas forcément tous dans le type encapsulé par chaque constructeur, cette règle introduit potentiellement des contraintes sur une partie seulement des paramètres, introduisant du polymorphisme. De même, la règle des constructeurs constants n'introduit aucune contrainte sur les paramètres de type.

$$\text{(Cstr-1)}\frac{(\alpha_0, \cdots, \alpha_m)\ t \in D_S \qquad D_C(c) = (t, \tau_c) \qquad \Gamma, D \vdash e\ :\ \tau \qquad \tau = \tau_c[\alpha_i \leftarrow \tau_i \text{ où } 0 \leqslant i \leqslant m]}{\Gamma, D \vdash c\ e\ :\ (\tau_0, \cdots, \tau_m)\ t}$$

$$\text{(Cstr-2)}\frac{(\alpha_0, \cdots, \alpha_m)\ t \in D_S \qquad D_C(c) = (t, \bot)}{\Gamma, D \vdash c\ :\ (\tau_0, \cdots, \tau_m)\ t}$$

11.2.4 Filtrage

Là encore dans un modèle théorique minimal, on préférerait ne traiter qu'une expression plus simple du type case, et décrire éventuellement l'équivalence d'expressivité avec le filtrage en profondeur. Ici, notre approche spécification nous amène à typer le filtrage sans le transfigurer. Nous ajoutons donc une règle de typage du match, ainsi qu'un ensemble de règles permettant de typer les motifs.

On introduit une nouvelle forme de règles $\text{(Règle)}\frac{\textit{prémisses}}{\Gamma, D, \tau \vdash p}$, se lisant : le motif p est un filtre bien typé pour le type τ, dans les environnements Γ et D.

Typage du match Le typage du filtrage en profondeur se fait alors en vérifiant que la valeur filtrée est typable, d'un type τ_e, que toutes les branches sont du même type τ, et que chaque motif p est bien typé par rapport à τ_e.

Le typage des liaisons introduites par chaque motif est fait en utilisant le même environnement, enrichi de ces liaisons, dans les jugements de typage de motif et la branche associée.

$$\text{(Match)}\frac{\begin{array}{c}(v_0^i, \cdots, v_n^i) = \mathcal{V}(p_i) \\ \Gamma, D \vdash e\ :\ \tau_e \qquad (v_0^i, \tau_0^i) :: \cdots :: (v_n^i, \tau_n^i) :: \Gamma, D \vdash e_i\ :\ \tau \quad \text{où } 0 \leqslant i \leqslant n \\ (v_0^i, \tau_0^i) :: \cdots :: (v_n^i, \tau_n^i) :: \Gamma, D, \tau_e \vdash p_i\end{array}}{\Gamma, D \vdash \mathsf{match}\ e\ \mathsf{with}\ p_0\ \texttt{->}\ e_0\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{->}\ e_n\ :\ \tau}$$

Typage des motifs Comme pour les expressions, les motifs simples n'ont pas de prémisse et constituent les feuilles de l'arbre d'inférence.

$$\text{(P-String)}\frac{}{\Gamma, D, \mathsf{string} \vdash s} \qquad \text{(P-Int)}\frac{}{\Gamma, D, \mathsf{int} \vdash i}$$

$$\text{(P-Bool-1)}\frac{}{\Gamma, D, \mathsf{bool} \vdash \mathsf{true}} \qquad \text{(P-Bool-2)}\frac{}{\Gamma, D, \mathsf{bool} \vdash \mathsf{false}}$$

$$\text{(P-Unit)}\frac{}{\Gamma, D, \mathsf{unit} \vdash \texttt{()}} \qquad \text{(P-List-1)}\frac{}{\Gamma, D, \tau\ \mathsf{list} \vdash \texttt{[]}}$$

$$\text{(P-Tuple)}\frac{\Gamma, D, \tau_0 \vdash p_0 \cdots \Gamma, D, \tau_n \vdash p_n}{\Gamma, D, (\tau_0, \cdots, \tau_n) \vdash (p_0, \cdots, p_n)} \qquad \text{(P-List-2)}\frac{\Gamma, D, \tau \vdash p_h \qquad \Gamma, D, \tau\ \mathsf{list} \vdash p_t}{\Gamma, D, \tau\ \mathsf{list} \vdash p_h\ \texttt{::}\ p_t}$$

Le motif *attrape-tout* est un filtre bien typé pour n'importe quel type. Le motif *variable* est bien typé si son type associé dans l'environnement est bien celui à filtrer, ce qui signifie par définition de la règle

de typage du match que le nom est utilisé de façon cohérente dans la branche associée. Le motif as est traité de façon similaire, en vérifiant toutefois que le sous-motif qu'il nomme est aussi bien typé.

$$\text{(P-Var)}\frac{n \in \mathit{dom}(\Gamma) \qquad \Gamma(n) = \tau}{\Gamma, D, \tau \vdash n} \qquad \text{(P-Catch)}\frac{}{\Gamma, D, \tau \vdash \_}$$

$$\text{(P-As)}\frac{n \in \mathit{dom}(\Gamma) \qquad \Gamma(n) = \tau \qquad \Gamma, D, \tau \vdash p}{\Gamma, D, \tau \vdash (p \texttt{ as } n)}$$

Les motifs pour les types personnalisés vérifient que le type à filtrer est bien cohérent moyennant substitution avec les définitions associées dans D, à la manière des règles de typage des expressions de construction de valeurs de types personnalisés. Une différence est que le filtrage des enregistrements permet cependant de ne spécifier qu'un sous-ensemble des champs.

$$\text{(P-Cstr)}\frac{\begin{array}{c}(\alpha_0, \cdots, \alpha_n)\ t \in \mathit{dom}(D_S) \qquad D_C(c) = (t, \tau_c) \\ \Gamma, D, \tau_p \vdash p \qquad \tau_p = \tau_c[\alpha_i \leftarrow \tau_i]\end{array}}{\Gamma, D, (\tau_0, \cdots, \tau_n)\ t \vdash c\ p}$$

$$\text{(P-Record)}\frac{\begin{array}{c}D_F(l_i) = (t_{l_i}, \tau_{l_i}) \text{ où } 0 \leqslant i \leqslant n \qquad t_{l_0} = \cdots = t_{l_n} = t \\ (\alpha_0, \cdots, \alpha_m)\ t \in \mathit{dom}(D_R) \qquad D_R(t) \supseteq \{l_i \mid 0 \leqslant i \leqslant n\} \\ \Gamma, D, \tau_{l_i}[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant m] \vdash p_i \text{ où } 0 \leqslant i \leqslant n\end{array}}{\Gamma, D, (\tau_0, \cdots, \tau_m)\ t \vdash \{\ l_0 = p_0\ ;\ \cdots\ ;\ l_n = p_n\ \}}$$

11.2.5 Typage des nœuds

Nous introduisons maintenant le typage des constructions du langage en rapport avec la construction et la manipulation des nœuds. Pour cela nous utilisons une famille de types <*t*> node, $t \in \mathit{dom}(D_N)$, chaque type étant le type spécifique induit par la définition de type de nœud associée.

On introduit alors un polymorphisme supplémentaire, afin de permettre à certaines manipulations intrinsèquement génériques des nœuds d'être utilisables sur un nœud de n'importe quelle étiquette. On introduit pour cela le type générique node, l'intuition étant qu'il représente l'union de tous les types de nœuds étiquetés.

Sous-typage On définit donc la relation de sous-typage $\lessdot$ définie principalement, comme nous l'avons expliqué informellement, par $\forall t \in \mathit{dom}(D_N)$, <*t*> node$\lessdot$node. La définition complète de cette relation étendue aux types composés de façon classique [📖64] est donnée figure 11.8. Selon cette définition, un nœud d'un type étiqueté peut être considéré comme un nœud générique, une liste contenant des nœuds d'une certaine étiquette peut être considérée comme une liste générique, etc. La partie gauche (resp. droite) d'un type flèche évolue de façon *contravariante* (resp. *covariante*). La relation est bien sur réflexive et transitive. On n'introduit pas le sous-typage des types personnalisés, dont les paramètres doivent être invariants, pour des raisons de simplicité.

$$\begin{array}{rcll}
\text{<}t\text{> node} & \lessdot & \text{node} & \\
\tau^a & \lessdot & \tau^b & \text{si} \quad \tau^a = \tau^b \\
\tau_l^a \ \to\ \tau_r^a & \lessdot & \tau_l^b \ \to\ \tau_r^b & \text{si} \quad \tau_l^b \lessdot \tau_l^a \wedge \tau_r^a \lessdot \tau_r^b \\
\tau_0^a \times \cdots \times \tau_n^a & \lessdot & \tau_0^b \times \cdots \times \tau_n^b & \text{si} \quad \forall i, \tau_i^a \lessdot \tau_i^b \\
\tau^a \text{ list} & \lessdot & \tau^b \text{ list} & \text{si} \quad \tau^a \lessdot \tau^b
\end{array}$$

FIGURE 11.8: Relation de sous-typage.

Coercition Pour que le système reste dirigé par la syntaxe, on n'introduit pas de relation de subsomption implicite. À la place, on étend la relation d'instanciation pour permettre d'utiliser une valeur d'un type spécifique comme une valeur plus générique :

$$\tau_1 \leqslant: \tau_2 \Rightarrow \tau_1 \leqslant \tau_2$$

Pour obtenir une valeur de type node, le programmeur devra utiliser l'instanciation d'une liaison effectuée par un let précédent, ou demander la coercition explicite (*upcast*) via la syntaxe `(`*expr*`:`τ`)`. On insère tout de même une coercition implicite à la création d'un nœud, permettant de considérer directement le résultat d'une construction de nœud comme étant de type node. Concrètement, ce comportement permet d'écrire une liste de nœuds hétérogènes avec le type node list sans demander la coercition explicite de chacun des nœuds.

$$(\textsc{Node})\frac{\begin{array}{c}D_N(\mathit{tag}) = \bigcup_i \{l_i\} \\ \Gamma, D \vdash e \ :\ \texttt{node list} \qquad \Gamma, D \vdash e_i \ :\ D_P(l_i) \text{ où } 0 \leqslant i \leqslant n \qquad \texttt{<}\mathit{tag}\texttt{> node} \leqslant: \tau\end{array}}{\Gamma, D \vdash \texttt{node <tag> } e \texttt{ prop } l_0 \texttt{ = } e_0 \cdots \texttt{ prop } l_n \texttt{ = } e_n \texttt{ end} \ :\ \tau}$$

$$(\textsc{Annot})\frac{\Gamma, D \vdash e \ :\ \tau_e \qquad \tau_e \leqslant: \tau}{\Gamma, D \vdash \texttt{(}e \ :\ \tau\texttt{)} \ :\ \tau}$$

Spécialisation Le traitement opposé, c'est-à-dire le passage d'un nœud de type générique à un nœud de type étiqueté, se fait lors du filtrage. L'idée est qu'un motif de nœud étiqueté accepte de filtrer un nœud de type générique. En utilisant un motif as, le programmeur pourra lier la valeur reconnue par ce filtre, et le type associé dans l'environnement pourra être ce type spécifique. C'est ainsi que, comme nous l'avons vu dans les exemples du chapitre 10, Le filtrage sert alors d'opération de spécialisation, le programmeur filtrant un nœud de type générique pour lui appliquer dans les branches du filtrage les opérations spécialisées appropriées pour son étiquette.

$$(\text{P-}\textsc{Node})\frac{\begin{array}{c}\bigcup_i \{l_i\} \subseteq D_N(\mathit{tag}) \\ \Gamma, D, \texttt{node list} \vdash p \qquad \Gamma, D, D_P(l_i) \vdash p_i \text{ où } 0 \leqslant i \leqslant n \qquad \texttt{<}\mathit{tag}\texttt{> node} \leqslant: \tau\end{array}}{\Gamma, D, \tau \vdash \texttt{node <tag> } p \texttt{ prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end}}$$

Opérations spécialisées Pour un nœud de type étiqueté, les opérations bien typées sont l'accès et la modification des propriétés définies par ce type de nœud, via le sélecteur prop, ainsi que celles valables pour les nœuds génériques que nous donnons juste après.

$$(\textsc{Prop})\frac{\Gamma, D \vdash e \ :\ \texttt{<}t\texttt{> node} \qquad l \in D_N(t) \qquad \tau = D_P(l)}{\Gamma, D \vdash e\texttt{.(prop } l\texttt{)} \ :\ \tau}$$

$$(\textsc{Prop-Set})\frac{\Gamma, D \vdash e \ :\ \texttt{<}t\texttt{> node} \qquad l \in D_N(t) \qquad D_P(l) = (\tau_l, T) \qquad \Gamma, D \vdash e_v \ :\ \tau_l}{\Gamma, D \vdash e\texttt{.(prop } l\texttt{) <- } e_v \ :\ \texttt{unit}}$$

Opérations génériques Les opérations disponibles sur tous les nœuds sont les primitives children et replace, pour manipuler les enfants, et l'accès et l'affectation via le sélecteur prop ?. Ces primitives utilisent le type prédéfini list, ainsi que le type option habituel en ML pour définir des fonctions partielles, dont nous donnons la définition ci-dessous, et qui est présent dans l'environnement de typage initial.

```
type 'a option = None | Some of 'a
```

$$(\textsc{Children})\frac{\Gamma, D \vdash e \ : \ \tau \qquad \tau \lessdot \mathsf{node}}{\Gamma, D, \tau \vdash \mathsf{children}\ \ e \ : \ \mathsf{node\ list}} \qquad (\textsc{Replace})\frac{\Gamma, D \vdash e \ : \ \tau \qquad \tau \lessdot \mathsf{node} \quad \Gamma, D \vdash e_l \ : \ \tau_l \qquad \tau_l \lessdot \mathsf{node\ list}}{\Gamma, D, \tau \vdash \mathsf{replace}\ \ e\ \ e_l \ : \ \mathsf{unit}}$$

$$(\textsc{Prop ?})\frac{\Gamma, D \vdash e \ : \ \tau_e \qquad \tau_e \lessdot \mathsf{node} \quad l \in D_N(t) \qquad \tau = D_P(l)}{\Gamma, D \vdash e.(\mathsf{prop?}\ \ l) \ : \ \tau\ \mathsf{option}} \qquad (\textsc{Prop ?-Set})\frac{\Gamma, D \vdash e \ : \ \tau \qquad \tau \lessdot \mathsf{node} \quad l \in D_N(t) \qquad D_P(l) = (\tau_l, T) \quad \Gamma, D \vdash e_v \ : \ \tau_l}{\Gamma, D \vdash e.(\mathsf{prop?}\ \ l)\ \texttt{<-}\ \ e_v \ : \ \mathsf{unit}}$$

11.3 Conclusion

Nous avons donné dans ce chapitre les vérifications statiques nécessaire à la sûreté d'exécution de FidoML. Nous avons pu observer que le typage statique des nœuds présenté informellement au chapitre 10 s'intègre effectivement correctement à un système de types classique pour ML avec définitions de types personnalisés.

Avec ces vérifications statiques, les constructions et modifications de nœuds se font en respectant les types déclarés par le programmeur. En particulier, la copie implicite d'un nœud ne peut arriver que lors d'une opération replace, et uniquement sur des nœuds complètement construits. La grammaire du document, c'est-à-dire le typage de l'imbrication des nœuds, n'est cependant pas traitée, le chapitre 13 sera dédié à ce problème, en donnant une solution plus générale puis en l'intégrant dans le système de types présenté dans ce chapitre.

Le typage des nœuds présenté, dans lequel chaque nœud peut être considéré soit comme de son type étiqueté propre soit comme un type complètement générique, est suffisamment simple pour être facilement implanté, et permet déjà d'exprimer des comportements complexes, comme nous l'avons vu au chapitre 10. Mais son principal avantage est de pouvoir être ré-utilisé dans la plupart des systèmes de types des langages généralistes modernes. Si on ne s'attarde pas sur la verbosité, on peut par exemple facilement imaginer des encodages utilisant les variants polymorphes d'OCaml ou l'héritage de Java.

Extension Dans le cas d'une implantation indépendante de FidoML, n'ayant pas besoin d'être restreinte aux possibilités d'un système de types existant, il pourrait être rendu un peu plus souple, sans pour autant en augmenter outre mesure l'expressivité. L'idée est de rester proche des utilisations des nœuds dans la programmation Web, et donc conserver le caractère monomorphe des nœuds et des propriétés, mais de permettre d'étiqueter un type par un ensemble d'étiquettes, représentant les différents types de nœuds possibles pour une valeur de ce type.

$$\textit{type}\ \texttt{+=}\ \texttt{<}\ \textit{tag}\ [\ \texttt{,}\ \textit{tag}\ ]^*\ \texttt{>}\ \mathsf{node}$$

La sémantique sous-jacente d'un tel sous-typage est facile à comprendre pour le programmeur : l'ensemble de valeurs d'un type <$t_0, \cdots, t_n$> node est l'union des ensemble de valeurs des types <t_i> node, et l'ensemble de ses propriétés est l'intersection des ensemble de propriétés des types <t_i> node.

Un avantage important par rapport au système simple présenté précédemment est la possibilité de construire des listes hétérogènes, regroupant des nœuds portant des étiquettes d'un ensemble donné. Ce motif est particulièrement utile pour le typage des grammaires usuelles du Web, comme nous le verrons au chapitre 13.

12 Sémantique opérationnelle de FidoML

Nous donnons dans ce chapitre la spécification de l'évaluation de FidoML, sous forme d'une sémantique opérationnelle à grand pas, dans laquelle les effets sont délégués à une sémantique du document. Nous commençons par décrire les valeurs manipulées par le langage, et le lien avec les valeurs du modèle du document (les paramètres et résultats des primitives). Puis nous donnons l'ensemble de règles, d'abord sur l'évaluation du programme, puis progressivement les expressions : expressions de base, valeurs construites, filtrage et valeurs fonctionnelles. Enfin nous étendons l'ensemble de règles pour ajouter les manipulations du document.

12.1 Domaine sémantique

L'originalité de cette sémantique est de déléguer entièrement les effets à un modèle de document (f*DOM*, ^{c}*DOM* ou $^\varepsilon$*DOM*). Elle utilise les primitives du document non seulement pour gérer les opérations spécifiques à ce dernier, mais aussi pour construire, accéder et modifier les valeurs construites du langage. Ces dernières sont comme on s'y attend les *objets blancs* de la sémantique du document, et l'accès à leur contenu ne peut donc être fait qu'en utilisant les primitives associées. L'état de la sémantique du document reste opaque du point de vue de la sémantique du langage. Le point original, et intéressant pour notre approche de spécification proche de l'implantation, qui en découle est donc que certains traits du langage, sont décrits de façon beaucoup plus opérationnelle et précise que dans une sémantique traditionnelle.

Valeurs Une valeur est le résultat de l'évaluation d'une expression. Les valeurs sont spécifiées dans la sémantique par l'ensemble *value* suivant :

$$\begin{array}{rcl} \mathit{Val} & = & \mathit{ObjVal} \cup \mathit{ImmVal} \\ \mathit{ObjVal} & = & H \\ \mathit{ImmVal} & = & \{\mathsf{true}, \mathsf{false}, ()\} \cup \mathit{integer} \cup \mathit{string} \cup \mathit{cstr} \end{array}$$

Cet ensemble regroupe :

- Les objets provenant de la sémantique du document, blancs pour les valeurs construites et noirs pour les nœuds du document.

  > **Rappel :** on note $\bullet$ un objet noir (ou de façon indicée $\bullet_n$ s'il faut en distinguer plusieurs), de la même façon $\circ_{(n)}$ un objet blanc et $\bullet_{(n)}$ un objet si sa couleur n'a pas d'importance. Ces objets proviennent de la composante H (qu'on appelera parfois tas) de l'état du document, avec $H = H^\bullet = H^\bullet \cup H^\circ$ l'union des ensembles des objets noirs et blanc.

- Les valeurs immédiates du langage : entiers (*int*), booléens, chaînes et constructeurs. Pour la sémantique, les constantes du langage coïncident classiquement avec les valeurs immédiates qui résultent de l'évaluation. La projection vers des valeurs plus concrètes est laissée à l'implantation. On laisse aussi à l'implantation le choix d'une éventuelle projection optimisée des constructeurs, par exemple vers des valeurs entières de la machine d'implantation.

Interface avec la sémantique de document Avant de pouvoir utiliser une des sémantiques du document, il faut en instancier les paramètres. De façon cohérente avec les valeurs décrites précédemment, soit $\mathcal{I}$ cette instanciation :

- $\mathcal{I}(Imm) = ImmVal \cup expr$
  Ce paramètre représente les valeurs associables aux propriétés des objets qui ne sont pas des objets, c'est donc l'ensemble des valeurs immédiates du langage. Puisqu'on peut avoir des valeurs fonctionnelles, on ajoute les expressions du langage. Une implantation pourrait bien sûr utiliser une représentation plus concrête comme un pointeur de code.
- $\mathcal{I}(Key) = int \cup id$
  Les noms de propriétés coïncident avec les identificateurs du langage pour la spécification des accès via des champs nommés, et les entiers pour les accès indexés.
- $\mathcal{I}(nil) = nil$
  La valeur ne sera pas visible à l'extérieur de la définition des règles de sémantique et peut donc rester abstraite.
- $\mathcal{I}(Tag) = tag$
  Les étiquettes du document coïncident avec celles du langage.
- $\mathcal{I}(Int) = int$
  On utilise le même type pour les entiers du langage et ceux du document.
- $\mathcal{I}(Enum(E)) = \mathcal{P}(\mathcal{I}(E))$
  On conserve une représentation abstraite des ensembles, qui sera concrétisée explicitement en valeur FidoML si besoin.

Environnement d'évaluation L'évaluation utilise un environnement $\Gamma \in List(id \times \text{value})$ qui associe des noms à des valeurs. Il une forme de liste associative, selon la définition de la figure 8.2.

Puisque l'évaluation a des effets sur le document, celle-ci fait intervenir, en plus de l'environnement Γ, un état S du document. Cet état peut être conforme à ^{f}DOM, ^{c}DOM ou $^{\epsilon}DOM$. Nous donnerons à la fin de ce chapitre des interprétations pratiques de la sémantique du langage, selon si elle utilise une sémantique du document classique, par exceptions ou par copie.

12.2 Évaluation du cœur de langage

Les *règles* ou *schémas d'évaluation* de cette sémantique opérationnelle à grand pas, sont de la forme $(\text{Règle})\frac{\textit{prémisses et conditions}}{E \vdash e \rightsquigarrow r}$, se lisant : si les *prémisses* et *conditions* sont vérifiables, une expression de la forme e s'évalue dans l'environnement E, pour donner le résultat r.

On appellera la partie inférieure le *jugement d'évaluation*. Les prémisses peuvent être des formules logiques à vérifier, des jugements d'évaluation, ou dans notre cas des appels de primitives de la sémantique du document. Bien sûr, les prémisses portent en général sur l'environnement E et/ou les sous-expressions de e, et le résultat r est en général fonction des résultats des jugements d'évaluation des sous-expressions.

Les domaines de l'environnement E et du résultat r dépendent de la forme de l'expression à évaluer (phrase, expression ou motif). Pour faciliter la lecture, les règles suivent la convention de nommage suivante : les règles des expressions sont préfixées par E, celles des phrases par T (pour *top-level*) et celles des motifs par P (pour *pattern*). De même, les règles menant à un échec d'évaluation (qui ne doivent pas arriver si le programme est bien typé) sont suffixées par F (pour *failure*), et sont indiquées en rouge foncé.

Évaluation du programme Pour décrire l'évaluation des phrases, on utilise des règles de la forme $(\text{T-Règle})\frac{\textit{prémisses et conditions}}{\Gamma,S \vdash p \overset{\perp}{\rightarrow} \Gamma',S'}$. Un programme p peut ajouter des liaisons globales dans Γ pour aboutir à un nouvel environnement Γ', et transformer le document de l'état S à l'état S'. Les règles d'échec ont la

forme $(\text{T-Règle-F})\frac{\textit{prémisses et conditions}}{\Gamma,S \vdash p \overset{\text{I}}{\rightsquigarrow} \bot,S'}$. L'environnement d'évaluation n'a plus de sens à l'issue d'une telle évaluation, mais le document a tout de même été transformé.

Les règles pour les phrases-expressions se contentent de déléguer l'évaluation aux règles d'évaluation des expressions. Le programme échoue dès qu'une expression échoue (la règle (T-Expr-F) peut préter à confusion, il faut lire le haut comme l'échec de l'évaluation d'une expression vers une valeur, et le bas vers l'échec de l'évaluation d'une phrase-expression vers un environnement).

$$(\text{T-Expr})\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e \overset{\text{I}}{\rightsquigarrow} \Gamma, S'} \qquad (\text{T-Expr-F})\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash e \overset{\text{I}}{\rightsquigarrow} \bot, S'}$$

Les règles des liaisons associent le résultat de l'évaluation de l'expression au nom dans l'environnement résultant. L'échec de l'évaluation du corps d'une liaison entraine l'échec du programme. Les liaisons récursives n'étant définies que sur un sous-ensemble des expressions et demandant un traitement spécifique, elles seront présentées plus tard.

$$(\text{T-Let})\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} v, S'}{\Gamma, S \vdash \text{ let } n = e \overset{\text{I}}{\rightsquigarrow} (n, v) :: \Gamma, S'} \qquad (\text{T-Let-F})\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash \text{ let } n = e \overset{\text{I}}{\rightsquigarrow} \bot, S''}$$

Un programme est évalué en évaluant ses phrases dans l'ordre, et échoue dès qu'une phrase échoue. Pour la spécification de l'évaluation, les définitions de types sont simplement ignorées. Une implantation pourrait cependant les interpréter, par exemple pour projeter les constructeurs, étiquettes et champs vers une implantation optimisée lors de leur définition.

$$(\text{T-Seq})\frac{\Gamma, S \vdash p_1 \overset{\text{I}}{\rightsquigarrow} \Gamma_1, S_1 \quad \Gamma_1, S_1 \vdash p_2 \overset{\text{I}}{\rightsquigarrow} \Gamma_2, S_2}{\Gamma, S \vdash p_1 \text{ ;; } p_2 \overset{\text{I}}{\rightsquigarrow} \Gamma_2, S_2} \qquad (\text{T-Seq-F})\frac{\Gamma, S \vdash p_1 \overset{\text{I}}{\rightsquigarrow} \bot, S_1}{\Gamma, S \vdash p_1 \text{ ;; } p_2 \overset{\text{I}}{\rightsquigarrow} \bot, S_1}$$

$$(\text{T-TypeDef})\frac{}{\Gamma, S \vdash \text{type } = \tau \overset{\text{I}}{\rightsquigarrow} \Gamma, S} \qquad (\text{T-NodeDef})\frac{}{\Gamma, S \vdash \text{node type } = \overset{\text{I}}{\rightsquigarrow} \Gamma, S}$$

Dans les règles qui précèdent comme dans la suite du chapitre, seules les règles d'échec nécessaires (et suffisantes) à la spécification de la propagation des erreurs sont données. Ici, par exemple, la règle d'échec de la partie droite d'une séquence pourrait être ajoutée, mais elle n'est pas nécessaire car si la partie droite échoue, alors son résultat d'échec $\bot$ sera aussi celui de la séquence. La règle (T-Seq-F) est par contre nécessaire, car c'est elle qui précise que la partie droite de la séquence n'est pas évaluée en cas d'échec de la partie gauche.

Expressions Les règles décrivant l'évaluation des expressions sont de la forme $(\text{E-Règle})\frac{\textit{prémisses et conditions}}{\Gamma,S \vdash e \overset{\text{E}}{\rightsquigarrow} v,S'}$. L'évaluation d'une expression e produit une valeur v et transforme l'état du document initial S en S'. Les règles d'échec sont de la forme $(\text{E-Règle-F})\frac{\textit{prémisses et conditions}}{\Gamma,S \vdash e \overset{\text{E}}{\rightsquigarrow} \bot,S'}$. L'évaluation d'une variable effectue un accès à l'environnement, l'échec ne devrait pas arriver dans un programme bien typé. Les valeurs immédiates étant confondues avec les constantes du langage, la règle est triviale. Les annotations de types sont ignorées au même titre que les définitions.

$$(\text{E-Var})\frac{n \in \textit{dom}(\Gamma)}{\Gamma, S \vdash n \in \textit{var} \overset{\text{E}}{\rightsquigarrow} \Gamma(n), S} \qquad (\text{E-Var-F})\frac{n \notin \textit{dom}(\Gamma)}{\Gamma, S \vdash n \in \textit{var} \overset{\text{E}}{\rightsquigarrow} \bot, S}$$

$$(\text{E-Imm})\frac{}{\Gamma, S \vdash i \in \textit{ImmVal} \overset{\text{E}}{\rightsquigarrow} i, S} \qquad (\text{E-Annot})\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} v, S'}{\Gamma, S \vdash (\ e\ :\ t\ ) \overset{\text{E}}{\rightsquigarrow} v, S'}$$

Les règles des liaisons locales et des séquences forcent l'ordre d'évaluation. En particulier, les règles d'échec spécifient l'arrêt de l'évaluation au plus tôt.

$$\text{(E-Let)}\frac{\Gamma, S \vdash e_d \overset{E}{\rightsquigarrow} v_d, S' \qquad (n, v_d) :: \Gamma, S' \vdash e_b \overset{E}{\rightsquigarrow} v_b, S''}{\Gamma, S \vdash \mathsf{let}\ \ n\ \ \mathsf{=}\ \ e_d\ \ \mathsf{in}\ \ e_b \overset{E}{\rightsquigarrow} v_b, S''}$$

$$\text{(E-Let-F)}\frac{\Gamma, S \vdash e_d \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash \mathsf{let}\ \ n\ \ \mathsf{=}\ \ e_d\ \ \mathsf{in}\ \ e_b \overset{E}{\rightsquigarrow} \bot, S'}$$

$$\text{(E-Seq)}\frac{\Gamma, S \vdash e_1 \overset{E}{\rightsquigarrow} v_1, S' \qquad \Gamma, S' \vdash e_2 \overset{E}{\rightsquigarrow} v_2, S''}{\Gamma, S \vdash e_1\ ;\ \ e_2 \overset{E}{\rightsquigarrow} v_2, S''}$$

$$\text{(E-Seq-F)}\frac{\Gamma, S \vdash e_1 \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash e_1\ ;\ \ e_2 \overset{E}{\rightsquigarrow} \bot, S'}$$

L'alternative est classiquement définie par deux règles selon la valeur du résultat de l'évaluation de la condition. Si la condition s'évalue vers $\bot$, ou une valeur non booléenne (ce qui ne devrait pas arriver si le programme est bien typé), aucune des branches n'est évaluée.

$$\text{(E-Alt-Left)}\frac{\Gamma, S \vdash e \overset{E}{\rightsquigarrow} \mathsf{true}, S' \qquad \Gamma, S' \vdash e_t \overset{E}{\rightsquigarrow} v_t, S''}{\Gamma, S \vdash \mathsf{if}\ \ e\ \ \mathsf{then}\ \ e_t\ \ \mathsf{else}\ \ e_f \overset{E}{\rightsquigarrow} v_t, S''}$$

$$\text{(E-Alt-Right)}\frac{\Gamma, S \vdash e \overset{E}{\rightsquigarrow} \mathsf{false}, S' \qquad \Gamma, S' \vdash e_f \overset{E}{\rightsquigarrow} v_f, S''}{\Gamma, S \vdash \mathsf{if}\ \ e\ \ \mathsf{then}\ \ e_t\ \ \mathsf{else}\ \ e_f \overset{E}{\rightsquigarrow} vft, S''}$$

$$\text{(E-Alt-F)}\frac{\Gamma, S \vdash e \overset{E}{\rightsquigarrow} v \notin \mathsf{bool}, S'}{\Gamma, S \vdash \mathsf{if}\ \ e\ \ \mathsf{then}\ \ e_t\ \ \mathsf{else}\ \ e_f \overset{E}{\rightsquigarrow} \bot, S''}$$

Valeurs construites (accès) Les champs des valeurs construites sont accédés et affectés grâce aux primitives du document. Pour référence, la table des primitives avec leurs types est donnée figure 9.1 (page 117).

Afin de factoriser la définition des accès aux différents types d'objets (qui au final sont tous représentés par des objets de la sémantique du document), on commence par définir un sélecteur virtuel Σ, dont le paramètre un nom de propriété.

$$\text{(E-Access)}\frac{\begin{array}{c}\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} \bullet, S_b \\ S_v\ \vdash\ \mathtt{get}(\bullet, k) = v, S'\end{array}}{\Gamma, S \vdash e_b.(\Sigma\ \ n) \overset{E}{\rightsquigarrow} v, S'}$$

$$\text{(E-Access-F-1)}\frac{\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} v \notin H, S_b}{\Gamma, S \vdash e_b.(\Sigma\ \ n) \overset{E}{\rightsquigarrow} \bot, S_b} \qquad \text{(E-Access-F-2)}\frac{\begin{array}{c}\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} \bullet, S_b \\ S_v\ \vdash\ \mathtt{get}(\bullet, k) = \mathit{nil}, S'\end{array}}{\Gamma, S \vdash e_b.(\Sigma\ \ n) \overset{E}{\rightsquigarrow} \bot, S'}$$

Pour toutes les expressions impératives, on s'efforce de préserver une sémantique cohérente avec l'évaluation stricte : si l'évaluation des paramètres de l'affectation échoue, aucune modification n'est effectuée dans le document. Ici, cette propriété est triviale à assurer puisqu'il est nécessaire d'avoir le résultat avant de le passer à la primitive `set`.

$$\text{(E-Assign)}\frac{\begin{array}{c}\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} \bullet, S_b \qquad \Gamma, S_b \vdash e_v \overset{E}{\rightsquigarrow} v_v, S_v \\ S_v\ \vdash\ \mathtt{set}(\bullet, k, v_v) = \mathit{nil}, S'\end{array}}{\Gamma, S \vdash e_b.(\Sigma\ \ n)\ \texttt{<-}\ \ e_v \overset{E}{\rightsquigarrow} (), S'}$$

$$\text{(E-Assign-F-1)}\frac{\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} v \notin H, S'}{\Gamma, S \vdash e_b.(\Sigma\ \ n)\ \texttt{<-}\ \ e_v \overset{E}{\rightsquigarrow} \bot, S'}$$

$$\text{(E-Assign-F-2)}\frac{\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} \bullet, S_b \qquad \Gamma, S_b \vdash e_v \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash e_b\,.\,(\Sigma\ n)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} \bot, S'}$$

On peut alors définir les accès via les sélecteurs du langage en fonction du sélecteur virtuel. La valeur résultat v peut valoir $\bot$ si l'évaluation de l'accès par le sélecteur virtuel s'est faite par une règle d'échec, c'est pourquoi les cas d'erreur n'apparaissent pas explicitement ici.

$$\text{(E-Field)}\frac{\Gamma, S \vdash e_b\,.\,(\Sigma\ n) \overset{E}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{field}\ n) \overset{E}{\rightsquigarrow} v, S'} \qquad \text{(E-Field-Set)}\frac{\Gamma, S \vdash e_b\,.\,(\Sigma\ n)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S'}{\Gamma, S \vdash\ e_b\,.\,(\texttt{field}\ n)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S'}$$

$$\text{(E-Proj)}\frac{\Gamma, S \vdash e_b\,.\,(\Sigma\ i) \overset{E}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{proj}\ i\ /\ j) \overset{E}{\rightsquigarrow} v, S'} \qquad \text{(E-Proj-Set)}\frac{\Gamma, S \vdash e_b\,.\,(\Sigma\ i)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{proj}\ i\ /\ j)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S'}$$

Ce sélecteur virtuel permet en outre d'étendre plus facilement les sélecteurs, et de se concentrer sur les cas d'erreurs inhérents au sélecteur concret. Par exemple, un sélecteur cell pour les tableaux pourrait être spécifié comme suit.

$$\text{(E-Cell)}\frac{\Gamma, S \vdash e_i \overset{E}{\rightsquigarrow} i, S' \quad \Gamma, S' \vdash e_b\,.\,(\Sigma\ i) \overset{E}{\rightsquigarrow} v, S''}{\Gamma, S \vdash e_b\,.\,(\texttt{cell}\ e_i) \overset{E}{\rightsquigarrow} v, S''} \qquad \text{(E-Cell-Set)}\frac{\Gamma, S \vdash e_i \overset{E}{\rightsquigarrow} i, S' \quad \Gamma, S' \vdash e_b\,.\,(\Sigma\ i)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S''}{\Gamma, S \vdash e_b\,.\,(\texttt{cell}\ e_i)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S''}$$

$$\text{(E-Cell-F)}\frac{\Gamma, S \vdash e_i \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{cell}\ e_i) \overset{E}{\rightsquigarrow} \bot, S''} \qquad \text{(E-Cell-Set-F)}\frac{\Gamma, S \vdash e_i \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{cell}\ e_i)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} \bot, S''}$$

Valeurs construites (construction) Pour allouer une valeur construite, on utilise la primitive de création de nœud blanc, puis on initialise chacune de ses propriétés. Pour les n-uplets et les enregistrements, les noms de propriétés sont directement les entiers et les noms de champs du langage. Si une des sous-expressions échoue, ses effets et ceux des expressiosn à sa gauche sont observables dans l'état résultant, mais les expressions suivantes ne sont pas évaluées et le bloc n'est pas créé.

$$\text{(E-Tuple)}\frac{\begin{array}{c}\Gamma, S_i \vdash e_i \overset{E}{\rightsquigarrow} v_i, S_{i+1} \text{ où } 0 \leqslant i \leqslant n \\ S_{n+1} \vdash \texttt{create}^\circ() = \circ, S'_0 \\ S'_i \vdash \texttt{set}^\circ(\circ, i, v_i) = \mathit{nil}, S'_{i+1} \text{ où } 0 \leqslant i \leqslant n\end{array}}{\Gamma, S_0 \vdash (e_0, \cdots, e_n) \overset{E}{\rightsquigarrow} \circ, S'_{n+1}}$$

$$\text{(E-Tuple-F)}\frac{\Gamma, S_i \vdash e_i \overset{E}{\rightsquigarrow} v_i, S_{i+1} \text{ où } 0 \leqslant i \leqslant m < n \qquad \Gamma, S_m \vdash e_m \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S_0 \vdash (e_0, \cdots, e_n) \overset{E}{\rightsquigarrow} \bot, S'}$$

$$\text{(E-Record)}\frac{\begin{array}{c}\Gamma, S_i \vdash e_i \overset{E}{\rightsquigarrow} v_i, S_{i+1} \text{ où } 0 \leqslant i \leqslant n \\ S_{n+1} \vdash \texttt{create}^\circ() = \circ, S'_0 \\ S'_i \vdash \texttt{set}^\circ(\circ, l_i, v_i) = \mathit{nil}, S'_{i+1} \text{ où } 0 \leqslant i \leqslant n\end{array}}{\Gamma, S \vdash \{\ l_0\ \texttt{=}\ e_0\ \texttt{;}\ \cdots\ \texttt{;}\ l_0\ \texttt{=}\ e_0\ \} \overset{E}{\rightsquigarrow} \circ, S'_{n+1}}$$

$$\text{(E-Record-F)}\frac{\Gamma, S_i \vdash e_i \overset{E}{\rightsquigarrow} v_i, S_{i+1} \text{ où } 0 \leqslant i \leqslant m < n \qquad \Gamma, S_m \vdash e_m \overset{E}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash \{\ l_0\ \texttt{=}\ e_0\ \texttt{;}\ \cdots\ \texttt{;}\ l_0\ \texttt{=}\ e_0\ \} \overset{E}{\rightsquigarrow} \bot, S''}$$

Pour les constructeurs (hormis les constructeurs constants qui ont déjà été traités par la règle (E-Imm)), le nom de constructeur est stocké dans la propriété *tag*, et la valeur encapsulée dans la pro-

priété *val*.

$$\text{(E-Cstr)}\frac{\begin{array}{c}\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} v, S_e \\ S_e \vdash \texttt{create}^\circ() = \circ, S_n \\ S_n \vdash \texttt{set}^\circ(\circ, \texttt{val}, v) = \mathit{nil}, S' \\ S' \vdash \texttt{set}^\circ(\circ, \texttt{tag}, c) = \mathit{nil}, S''\end{array}}{\Gamma, S \vdash c\ e \overset{\text{E}}{\rightsquigarrow} \circ, S''} \qquad \text{(E-Cstr-F)}\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash c\ e \overset{\text{E}}{\rightsquigarrow} \bot, S'}$$

12.3 Filtrage

Pour décrire l'évaluation du filtrage, on utilise un nouveau type de règles pour spécifier l'évaluation des motifs, la règle d'évaluation du filtrage par motifs faisant appel à ce nouvel ensemble de règles.

La règle pour un filtre f est de la forme $\text{(P-Règle)}\frac{\textit{prémisses et conditions}}{\Gamma, S, v \vdash f \overset{\text{P}}{\rightsquigarrow} \Gamma', S'}$, où le contexte contient en plus la valeur v à discriminer, et le résultat principal est l'environnement Γ' contenant les liaisons de sous-valeurs effectuées par les sous-motif de liaison. En ML, le filtrage est une structure passive du point de vue des effets, l'état du document ne change donc pas. Il est cependant conservé dans le résultat des règles, pour permettre le filtrage *actif* des enfants d'un nœud, comme nous le verrons plus loin.

La règle décrivant un filtre f qui refuse une valeur est de la forme $\text{(P-Règle-F)}\frac{\text{prémisses et conditions}}{\Gamma, S, v \vdash f \overset{\text{P}}{\rightsquigarrow} \bot, S'}$. La valeur $\bot$ renvoyée par l'évaluation d'un filtre décrit l'arrêt de son évaluation, mais ne dénote pas ici une erreur du programme, puisqu'elle peut être rattrapée par l'évaluation du filtrage pour passer la main au filtre suivant.

L'évaluation du filtrage se fait donc comme suit, en évaluant chaque filtre dans l'ordre, jusqu'à ce qu'un filtre accepte la valeur, auquel cas le résultat est l'évaluation de la branche, dans l'environnement résultant de l'évaluation du filtre, et les autres motifs ne sont pas évalués. Le filtrage échoue si toutes les branches ont échoué.

$$\text{(E-Match)}\frac{\begin{array}{c}\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} v, S' \\ \Gamma, S', v \vdash p_i \overset{\text{P}}{\rightsquigarrow} \bot \text{ où } 0 \leqslant i < j \leqslant n \qquad \Gamma, S', v \vdash p_j \overset{\text{P}}{\rightsquigarrow} \Gamma_j \\ \Gamma_j, S' \vdash e_j \overset{\text{E}}{\rightsquigarrow} v', S''\end{array}}{\Gamma, S \vdash \texttt{match}\ e\ \texttt{with}\ p_0\ \texttt{->}\ e_0\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{->}\ e_n \overset{\text{E}}{\rightsquigarrow} v', S''}$$

$$\text{(E-Match-F-1)}\frac{\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} \bot, S'}{\Gamma, S \vdash \texttt{match}\ e\ \texttt{with}\ p_0\ \texttt{->}\ e_0\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{->}\ e_n \overset{\text{E}}{\rightsquigarrow} \bot, S''}$$

$$\text{(E-Match-F-2)}\frac{\begin{array}{c}\Gamma, S \vdash e \overset{\text{E}}{\rightsquigarrow} v, S' \\ \Gamma, S', v \vdash p_i \overset{\text{P}}{\rightsquigarrow} \bot \text{ où } 0 \leqslant i \leqslant n\end{array}}{\Gamma, S \vdash \texttt{match}\ e\ \texttt{with}\ p_0\ \texttt{->}\ e_0\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{->}\ e_n \overset{\text{E}}{\rightsquigarrow} \bot, S''}$$

La variable et l'attrappe-tout réussissent systématiquement, la variable enrichit l'environnement au passage. Le as réussit si le motif qu'il englobe réussit, et enrichit l'environnement. Il est aussi possible d'utiliser des immédiats dans les motifs, dont le succès est défini par comparaison à la valeur.

$$\text{(P-Bind)}\frac{}{\Gamma, S, v \vdash n \overset{\text{P}}{\rightsquigarrow} (n, v) :: \Gamma, S} \qquad \text{(P-Catch)}\frac{}{\Gamma, S, v \vdash \_ \overset{\text{P}}{\rightsquigarrow} \Gamma, S}$$

$$\text{(P-As)}\frac{\Gamma, S, v \vdash p \overset{\text{P}}{\rightsquigarrow} \Gamma', S'}{\Gamma, S, v \vdash p\ \texttt{as}\ n \overset{\text{P}}{\rightsquigarrow} (n, v) :: \Gamma', S'} \qquad \text{(P-As-F)}\frac{\Gamma, S, v \vdash p \overset{\text{P}}{\rightsquigarrow} \bot}{\Gamma, S, v \vdash p\ \texttt{as}\ n \overset{\text{P}}{\rightsquigarrow} \bot, S}$$

$$\text{(P-Imm-T)}\frac{v = i}{\Gamma, S, v \vdash i \in imm \overset{p}{\rightsquigarrow} (n, v) :: \Gamma, S} \qquad \text{(P-Imm-F)}\frac{v \neq i}{\Gamma, S, v \vdash i \in imm \overset{p}{\rightsquigarrow} \bot, S}$$

Les motifs n-uplet et enregistrement réussissent si tous leurs sous-motifs réussissent. L'environnement résultant est l'environnement initial enrichi successivement par l'évaluation des sous-motifs.

$$\text{(P-Tuple)}\frac{\begin{array}{c} S_i \vdash \texttt{get}(\circ, i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{p}{\rightsquigarrow} \Gamma_{i+1}, S_{i+1} \end{array} \text{ où } 0 \leqslant i \leqslant n}{\Gamma_0, S_0, \circ \vdash (p_0, \cdots, p_n) \overset{p}{\rightsquigarrow} \Gamma_{n+1}, S_{n+1}}$$

$$\text{(P-Tuple-F-1)}\frac{}{\Gamma, S, i \in \mathit{ImmVal} \vdash (p_0, \cdots, p_n) \overset{p}{\rightsquigarrow} \bot, S}$$

$$\text{(P-Tuple-F-2)}\frac{\begin{array}{c} S_i \vdash \texttt{get}(\circ, i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{p}{\rightsquigarrow} \Gamma_{i+1} \end{array} \quad S_m \vdash \texttt{get}(\circ, m) = \mathit{nil}, S_m \text{ où } 0 \leqslant i < m \leqslant n}{\Gamma_0, S_0, \circ \vdash (p_0, \cdots, p_n) \overset{p}{\rightsquigarrow} \bot, S_m}$$

$$\text{(P-Tuple-F-3)}\frac{\begin{array}{c} S_i \vdash \texttt{get}(\circ, i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{p}{\rightsquigarrow} \Gamma_{i+1} \end{array} \quad S_m \vdash \texttt{get}(\circ, j) = v_m, S_m \quad \Gamma_m, S_m, v_m \vdash p_m \overset{p}{\rightsquigarrow} \bot, S_{m+1} \text{ où } 0 \leqslant i < m \leqslant n}{\Gamma_0, S_0, \circ \vdash (p_0, \cdots, p_n) \overset{p}{\rightsquigarrow} \bot, S_{m+1}}$$

De façon cohérente avec le typage, le filtrage sur les enregistrements fonctionne y compris si le programmeur ne filtre qu'une partie des champs.

$$\text{(P-Record)}\frac{\begin{array}{c} S_i \vdash \texttt{get}(\circ, l_i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{p}{\rightsquigarrow} \Gamma_{i+1}, S_{i+1} \end{array} \text{ où } 0 \leqslant i \leqslant n}{\Gamma_0, S_0, \circ \vdash \{l_0 \texttt{ = } p_0 \texttt{ ; } \cdots \texttt{ ; } l_n \texttt{ = } p_n \texttt{ }\} \overset{p}{\rightsquigarrow} \Gamma_{n+1}, S_{n+1}}$$

$$\text{(P-Record-F-1)}\frac{}{\Gamma, S, i \in \mathit{ImmVal} \vdash \{l_0 \texttt{ = } p_0 \texttt{ ; } \cdots \texttt{ ; } l_n \texttt{ = } p_n \texttt{ }\} \overset{p}{\rightsquigarrow} \bot, S}$$

$$\text{(P-Record-F-1)}\frac{\begin{array}{c} S_i \vdash \texttt{get}(\circ, l_i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{p}{\rightsquigarrow} \Gamma_{i+1} \end{array} \quad S_m \vdash \texttt{get}(\circ, l_m) = \mathit{nil}, S_m \text{ où } 0 \leqslant i < m \leqslant n}{\Gamma_0, S_0, \circ \vdash \{l_0 \texttt{ = } p_0 \texttt{ ; } \cdots \texttt{ ; } l_n \texttt{ = } p_n \texttt{ }\} \overset{p}{\rightsquigarrow} \bot, S_m}$$

$$\text{(P-Record-F-2)}\frac{\begin{array}{c} S_i \vdash \texttt{get}(\circ, l_i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{p}{\rightsquigarrow} \Gamma_{i+1} \end{array} \quad S_m \vdash \texttt{get}(\circ, l_j) = v_m, S_m \quad \Gamma_m, S_m, v_m \vdash p_m \overset{p}{\rightsquigarrow} \bot, S_{m+1} \text{ où } 0 \leqslant i < m \leqslant n}{\Gamma_0, S_0, \circ \vdash \{l_0 \texttt{ = } p_0 \texttt{ ; } \cdots \texttt{ ; } l_n \texttt{ = } p_n \texttt{ }\} \overset{p}{\rightsquigarrow} \bot, S_{m+1}}$$

Le filtrage d'un constructeur réussit si le nom de constructeur correspond, puis que le sous-motif filtre la valeur encapsulée.

$$\text{(P-Cstr)}\frac{\begin{array}{c} S \vdash \texttt{get}(\circ, \texttt{tag}) = c, S \\ S \vdash \texttt{get}(\circ, \texttt{val}) = v', S \\ \Gamma, S, v' \vdash p \overset{p}{\rightsquigarrow} \Gamma', S' \end{array}}{\Gamma, S, \circ \vdash c\ p \overset{p}{\rightsquigarrow} \Gamma', S'} \qquad \text{(P-Cstr-F-1)}\frac{}{\Gamma, S, i \in \mathit{ImmVal} \vdash c\ e \overset{p}{\rightsquigarrow} \bot}$$

$$\text{(P-Cstr-F-2)}\frac{S \vdash \texttt{get}(\circ, \text{tag}) = c', S \qquad c \neq c'}{\Gamma, S, \circ \vdash c\ p \overset{\text{P}}{\rightsquigarrow} \bot} \qquad \text{(P-Cstr-F-3)}\frac{S \vdash \texttt{get}(\circ, \text{tag}) = c, S \quad S \vdash \texttt{get}(\circ, \text{val}) = \mathit{nil}, S}{\Gamma, S, \circ \vdash c\ p \overset{\text{P}}{\rightsquigarrow} \bot}$$

Listes La syntaxe concrête des listes s'exprime en utilisant les règles déja définies pour les constructeurs où les listes sont projetées vers le type somme suivant :

```
type 'a list = Nil | Cons of 'a * 'a list;;
```

$$\text{(E-Cons)}\frac{\Gamma, S \vdash \texttt{Cons}\ (e_h, e_t) \overset{\text{E}}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e_h\ ::\ e_t \overset{\text{E}}{\rightsquigarrow} v, S'} \qquad \text{(E-Nil)}\frac{\Gamma, S \vdash \texttt{Nil} \overset{\text{E}}{\rightsquigarrow} v, S'}{\Gamma, S \vdash \texttt{[]} \overset{\text{E}}{\rightsquigarrow} v, S'}$$

$$\text{(P-Cons)}\frac{\Gamma, S, v \vdash \texttt{Cons}\ (p_h, p_t) \overset{\text{P}}{\rightsquigarrow} r}{\Gamma, S, v \vdash p_h\ ::\ p_t \overset{\text{P}}{\rightsquigarrow} r} \qquad \text{(P-Nil)}\frac{\Gamma, S, v \vdash \texttt{Nil} \overset{\text{P}}{\rightsquigarrow} r}{\Gamma, S, v \vdash \texttt{[]} \overset{\text{P}}{\rightsquigarrow} r}$$

12.4 Valeurs fonctionnelles et application

Pour représenter les fonctions à l'exécution, on utilise des *fermetures*. Une fermeture est une valeur construite dans laquelle sont capturés le code de la fonction et l'environnement d'évaluation lors de la définition de la fonction. La capture de l'environnement est nécessaire, car le code peut contenir des variables libres qui ne sont ni des globales ni des paramètres de la fonction, et qui font donc référence à des liaisons locales présentes dans l'environnement d'évaluation au moment de la définition.

Dans une sémantique plus classique où les valeurs construites sont directement manipulables, il est habituel, lors de la construction d'une fermeture, d'utiliser directement l'environnement d'évaluation Γ courant comme environnement de fermeture, et l'expression du corps de la fonction comme corps de la fermeture.

Mais dans notre sémantique, le seul moyen de confectionner une valeur construite est d'utiliser un objet blanc du document. Les immédiats du langage incluant les expressions, le code peut être encapsulé dans une valeur structurée sans encodage (une implantation pourrait utiliser un simple mécanisme de pointeurs de code). Par contre, l'environnement d'évaluation Γ est une spécification mathématique de l'état de l'évaluation, et ne peut en aucun cas être considéré comme une valeur et inséré dans un objet du document. Il faut donc encoder dans la fermeture la partie utile de l'environnement d'évaluation, en utilisant les primitives sur les objets blancs.

Il faut bien noter que la solution donnée ici n'est pas unique, le point important qui doit être retenu est que le caractère découplé de la sémantique oblige à spécifier des comportements habituellement laissés à l'implantation, et pourtant non triviaux.

Encodage de l'environnement La solution pour capturer l'environnement d'évaluation dans un objet blanc est de générer un code dont l'évaluation produit un tel objet, à partir de l'expression du corps de la fonction. On définit pour cela la méta-fonction $\mathcal{G}$ suivante. Le code généré suppose que l'objet devant contenir l'environnement est lié dans Γ sous le nom E, $\mathcal{G}$ calcule alors les variables libres du corps pour n'extraire que la partie utile de l'environnement, puis génère l'affectation de chaque champ n de E correspondant à une variable libre n par la valeur courante de la variable.

$$\begin{aligned} \mathcal{G}\,(\texttt{fun}\ n_a\ \texttt{->}\ e) &= \mathcal{G}'(\mathcal{F}(e)\backslash\{n_a\}) \\ \mathcal{G}'\,(\{n\} \cup V) &= \texttt{E.(}\Sigma\ \texttt{n)}\ \texttt{<-}\ \texttt{n;}\ \mathcal{G}'\,(V) \\ \mathcal{G}'\,(\emptyset) &= \texttt{()} \end{aligned}$$

Encodage du corps Pour que le code encapsulé dans la fermeture soit valable, il faut qu'il accède à l'environnement de fermeture plutôt qu'à l'environnement d'évaluation. Pour cela, on définit une seconde méta-fonction $\mathcal{E}$ réécrivant le corps pour que les accès aux variables libres soient remplacés par des accès à l'environnement. On utilise pour cela une astuce : plutôt que de réécrire chaque accès, on préfixe le code du corps non modifié des liaisons locales de chaque variable libre n au résultat de l'accès au champ n de l'environnement de fermeture E [1].

$$\begin{array}{rcl} \mathcal{E}\,(\texttt{fun}\ n\ \texttt{->}\ e) & = & \texttt{fun}\ n\ \texttt{->}\ \mathcal{E}'(e, \mathcal{F}(e)\backslash\{n\}) \\ \mathcal{E}'\,(e, \{n\} \cup V) & = & \texttt{let n = E.(}\Sigma\ \texttt{n) in}\ \mathcal{E}'\,(e, V) \\ \mathcal{E}'\,(e, \emptyset) & = & \texttt{e} \end{array}$$

Création de fermeture On peut alors décrire l'abstraction, créant une fermeture encapsulant l'environnement (resp. le code réécrit) dans sa propriété E (resp. C). Dans le cas d'une fermeture récursive, elle doit être ajoutée à son propre environnement.

$$\text{(E-Closure)}\ \frac{\begin{array}{c} S \vdash \texttt{create}^\circ() = \circ_e, S_1 \qquad S_1 \vdash \texttt{create}^\circ() = \circ_c, S_2 \\ (E, \circ_e) :: \Gamma, S_2 \vdash \mathcal{G}(\texttt{fun}\ n\ \texttt{->}\ e) \overset{\text{E}}{\rightsquigarrow} (), S_3 \\ S_3 \vdash \texttt{set}(\circ_c, C, \mathcal{E}(\texttt{fun}\ n\ \texttt{->}\ e)) = \mathit{nil}, S_4 \qquad S_4 \vdash \texttt{set}(\circ_c, E, \circ_e) = \mathit{nil}, S_5 \end{array}}{\Gamma, S \vdash \texttt{fun}\ n\ \texttt{->}\ e \overset{\text{E}}{\rightsquigarrow} \circ_c, S_5}$$

$$\text{(E-Closure-Rec)}\ \frac{\begin{array}{c} S \vdash \texttt{create}^\circ() = \circ_e, S_1 \qquad S_1 \vdash \texttt{create}^\circ() = \circ_c, S_2 \\ (E, \circ_e) :: \Gamma, S_2 \vdash \mathcal{G}(\texttt{fun}\ n\ \texttt{->}\ e) \overset{\text{E}}{\rightsquigarrow} (), S_3 \\ S_3 \vdash \texttt{set}(\circ_c, C, \mathcal{E}(\texttt{fun}\ n\ \texttt{->}\ e)) = \mathit{nil}, S_4 \qquad S_4 \vdash \texttt{set}(\circ_c, E, \circ_e) = \mathit{nil}, S_5 \\ S_5 \vdash \texttt{set}(\circ_e, f, \circ_c) = \mathit{nil}, S_6 \\ (f, \circ_e) :: \Gamma, S_6 \vdash e_n \overset{\text{E}}{\rightsquigarrow} v, S_7 \end{array}}{\Gamma, S \vdash \texttt{let rec}\ f\ \texttt{=}\ \texttt{fun}\ n\ \texttt{->}\ e\ \texttt{in}\ e_n \overset{\text{E}}{\rightsquigarrow}, S_7}$$

Dans ces règles, on peut penser que chaque évaluation d'une définition de fonction effectue la réécriture de code. Bien entendu, une implantation d'interprète pourrait calculer ces réécritures à l'avance, ou de façon paresseuse, et ces réécritures sont complètement syntaxiques et se prêtent facilement à la compilation.

Application L'application examine alors la fermeture et exécute son code, en ayant lié l'environnement encapsulé et l'argument dans l'environnement d'évaluation.

$$\text{(E-Apply)}\ \frac{\begin{array}{c} \Gamma, S \vdash e_f \overset{\text{E}}{\rightsquigarrow} v_b \in H^\circ, S' \qquad \Gamma, S' \vdash e_a \overset{\text{E}}{\rightsquigarrow} v_a, S'' \\ S'' \vdash \texttt{get}(v_b, C) = \texttt{fun}\ n\ \texttt{->}\ e, S'' \qquad S'' \vdash \texttt{get}(v_b, E) = v_e, S'' \\ (n, v_a) :: (E, v_e) :: [\,], S'' \vdash e \overset{\text{E}}{\rightsquigarrow} v_r, S''' \end{array}}{\Gamma, S \vdash e_f\ e_a \overset{\text{E}}{\rightsquigarrow} v_r, S'''}$$

$$\text{(E-Apply-F-1)}\ \frac{\Gamma, S \vdash e_f \overset{\text{E}}{\rightsquigarrow} v_b \in \{\bot\} \cup \mathit{ImmVal}, S'}{\Gamma, S \vdash e_f\ e_a \overset{\text{E}}{\rightsquigarrow} \bot, S'''} \qquad \text{(E-Apply-F-2)}\ \frac{\begin{array}{c} \Gamma, S \vdash e_f \overset{\text{E}}{\rightsquigarrow} v_b, S' \\ \Gamma, S \vdash e_a \overset{\text{E}}{\rightsquigarrow} \bot, S' \end{array}}{\Gamma, S \vdash e_f\ e_a \overset{\text{E}}{\rightsquigarrow} \bot, S'''}$$

1. C'est la même astuce que l'instruction RESTART de la machine virtuelle d'OCaml, qui pousse les valeurs dans l'environnement de fermeture sur la pile avant de passer au corps de la fonction, permettant de partager le code pour l'application totale et celui embarqué dans les fermetures.

$$\text{(E-Apply-F-3)}\ \frac{\Gamma, S \vdash e_f \overset{E}{\rightsquigarrow} v_b \in H^\circ, S' \quad \Gamma, S' \vdash e_a \overset{E}{\rightsquigarrow} v_a, S'' \quad S'' \vdash \texttt{get}(v_b, C) = v, S'' \quad v \neq \texttt{fun}\ n\ \texttt{->}\ e}{\Gamma, S \vdash e_f\ e_a \overset{E}{\rightsquigarrow} \bot, S'''}$$

$$\text{(E-Apply-F-4)}\ \frac{\Gamma, S \vdash e_f \overset{E}{\rightsquigarrow} v_b \in H^\circ, S' \quad \Gamma, S' \vdash e_a \overset{E}{\rightsquigarrow} v_a, S'' \quad S'' \vdash \texttt{get}(v_b, C) = \texttt{fun}\ n\ \texttt{->}\ e, S'' \quad S'' \vdash \texttt{get}(v_b, E) = \mathit{nil}, S''}{\Gamma, S \vdash e_f\ e_a \overset{E}{\rightsquigarrow} \bot, S'''}$$

12.5 Opérations sur les nœuds

La construction syntaxique de création de nœuds noirs utilise le mécanisme de portée du document défini à cet effet. L'évaluation de la construction utilise l'évaluation des autres opérations définies dans la suite pour l'affectation initiale des enfants et des propriétés.

$$\text{(E-Node)}\ \frac{S \vdash \texttt{create}^\bullet(\mathit{tag}) = \bullet, S' \quad (B, \bullet) :: \Gamma, S' \vdash \text{replace } B\ e \overset{E}{\rightsquigarrow} (), S_0 \quad (B, \bullet) :: \Gamma, S_i \vdash \texttt{B.(prop}\ l_i\texttt{)}\ \texttt{<-}\ e_i \overset{E}{\rightsquigarrow} (), S_{i+1} \text{ où } 0 \leqslant i < n \quad S_{i+1} \vdash \texttt{close}(\bullet) = \mathit{nil}, S''}{\Gamma, S \vdash \texttt{node <}\mathit{tag}\texttt{> e prop}\ l_0\ \texttt{=}\ e_0 \cdots \texttt{prop}\ l_n\ \texttt{=}\ e_n\ \texttt{end} \overset{E}{\rightsquigarrow} \bullet, S''}$$

$$\text{(E-Node-F-1)}\ \frac{S \vdash \texttt{create}^\bullet(\mathit{tag}) = \bullet, S' \quad (B, \bullet) :: \Gamma, S' \vdash \text{replace } B\ e \overset{E}{\rightsquigarrow} \bot, S_0 \quad S_0 \vdash \texttt{close}(\bullet) = \mathit{nil}, S''}{\Gamma, S \vdash \texttt{node <}\mathit{tag}\texttt{> e prop}\ l_0\ \texttt{=}\ e_0 \cdots \texttt{prop}\ l_n\ \texttt{=}\ e_n\ \texttt{end} \overset{E}{\rightsquigarrow} \bot, S''}$$

$$\text{(E-Node-F-2)}\ \frac{S \vdash \texttt{create}^\bullet(\mathit{tag}) = \bullet, S' \quad (B, \bullet) :: \Gamma, S' \vdash \text{replace } B\ e \overset{E}{\rightsquigarrow} (), S_0 \quad (B, \bullet) :: \Gamma, S_i \vdash \texttt{B.(prop}\ l_i\texttt{)}\ \texttt{<-}\ e_i \overset{E}{\rightsquigarrow} (), S_{i+1} \quad \text{où } 0 \leqslant i \leqslant j < n \quad (B, \bullet) :: \Gamma, S_j \vdash \texttt{B.(prop}\ l_j\texttt{)}\ \texttt{<-}\ e_j \overset{E}{\rightsquigarrow} \bot, S_{j+1} \quad S_{j+1} \vdash \texttt{close}(\bullet) = \mathit{nil}, S''}{\Gamma, S \vdash \texttt{node <}\mathit{tag}\texttt{> e prop}\ l_0\ \texttt{=}\ e_0 \cdots \texttt{prop}\ l_n\ \texttt{=}\ e_n\ \texttt{end} \overset{E}{\rightsquigarrow} \bot, S''}$$

La construction d'un nœud étant acceptée dans d'une liaison récursive, on décrit son exécution comme l'enrichissement de l'environnement pendant la création, juste après l'allocation.

$$\text{(E-Node-Rec)}\ \frac{S \vdash \texttt{create}^\bullet(\mathit{tag}) = \bullet, S' \quad (n, \bullet) :: \Gamma, S' \vdash \text{replace } n\ e \overset{E}{\rightsquigarrow} (), S_0 \quad (n, \bullet) :: \Gamma, S_i \vdash \texttt{n.(prop}\ l_i\texttt{)}\ \texttt{<-}\ e_i \overset{E}{\rightsquigarrow} (), S_{i+1} \text{ où } 0 \leqslant i < n \quad S_{i+1} \vdash \texttt{close}(\bullet) = \mathit{nil}, S'' \quad (n, \bullet) :: \Gamma, S'' \vdash e' \overset{E}{\rightsquigarrow} v, S'''}{\Gamma, S \vdash \left[\begin{array}{l}\texttt{let rec}\ n\ \texttt{=} \\ \quad \texttt{node <}\mathit{tag}\texttt{>}\ e\ \texttt{prop}\ l_0\ \texttt{=}\ e_0 \cdots \texttt{prop}\ l_n\ \texttt{=}\ e_n\ \texttt{end} \\ \texttt{in}\ e'\end{array}\right] \overset{E}{\rightsquigarrow} v, S'''}$$

$$\text{(T-Node-Rec)}\ \frac{S \vdash \texttt{create}^\bullet(\mathit{tag}) = \bullet, S' \quad (n, \bullet) :: \Gamma, S' \vdash \text{replace } n\ e \overset{E}{\rightsquigarrow} (), S_0 \quad (n, \bullet) :: \Gamma, S_i \vdash \texttt{n.(prop}\ l_i\texttt{)}\ \texttt{<-}\ e_i \overset{E}{\rightsquigarrow} (), S_{i+1} \text{ où } 0 \leqslant i < n \quad S_{i+1} \vdash \texttt{close}(\bullet) = \mathit{nil}, S'' \quad \Gamma' = (n, \bullet) :: \Gamma}{\Gamma, S \vdash \left[\begin{array}{l}\texttt{let rec}\ n\ \texttt{=} \\ \quad \texttt{node <}\mathit{tag}\texttt{>}\ e\ \texttt{prop}\ l_0\ \texttt{=}\ e_0 \cdots \texttt{prop}\ l_n\ \texttt{=}\ e_n\ \texttt{end}\end{array}\right] \overset{T}{\rightsquigarrow} \Gamma', S''}$$

Propriétés L'accès aux propriétés d'un nœud dont l'étiquette est connue, peut se faire via le sélecteur prop. Celui-ci échoue donc si la propriété n'est pas définie, c'est un simple alias au sélecteur virtuel Σ.

Le sélecteur prop? par contre, peut être utilisé sur n'importe quel nœud, renvoyant alors une valeur de type option.

```
type 'a option = None | Some of 'a
```

$$\text{(E-Prop)}\frac{\Gamma, S \vdash e_b\,.\,(\Sigma\ n) \overset{E}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{prop}\ n) \overset{E}{\rightsquigarrow} v, S'} \qquad \text{(E-Prop-Set)}\frac{\Gamma, S \vdash e_b\,.\,(\Sigma\ n)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{prop}\ n)\ \texttt{<-}\ e_v \overset{E}{\rightsquigarrow} v, S'}$$

$$\text{(E-Prop ?-1)}\frac{\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} \bullet, S' \qquad S' \vdash \texttt{get}(\bullet, n) = \mathit{nil}, S'}{\Gamma, S \vdash e_b\,.\,(\texttt{prop?}\ n) \overset{E}{\rightsquigarrow} \texttt{None}, S'}$$

$$\text{(E-Prop ?-2)}\frac{\Gamma, S \vdash e_b \overset{E}{\rightsquigarrow} \bullet, S' \qquad S' \vdash \texttt{get}(\bullet, n) = v, S' \qquad (V, v) :: \Gamma, S' \vdash \texttt{Some V} \overset{E}{\rightsquigarrow} v', S''}{\Gamma, S \vdash e_b\,.\,(\texttt{prop?}\ n) \overset{E}{\rightsquigarrow} v', S''}$$

Opération children La primitive children est définie comme suit : on demande à au document les enfants du nœud, puis on évalue la construction d'une liste les contenant.

$$\text{(E-Children)}\frac{\begin{array}{c}\Gamma, S \vdash e \overset{E}{\rightsquigarrow} \bullet, S' \qquad S' \vdash \texttt{children}(\bullet) = n, S' \qquad \Gamma, S' \vdash \texttt{[ ]} \overset{E}{\rightsquigarrow} l_0, S_0 \\ S_i \vdash \texttt{child}(\bullet, i) = \bullet_i, S_i \\ (E, \bullet_i) :: (L, l_i) :: \Gamma, S_i \vdash \texttt{E :: L} \overset{E}{\rightsquigarrow} l_{i+1}, S_{i+1}\end{array} \text{ où } 0 \leqslant i < n}{\Gamma, S \vdash \texttt{children}\ e \overset{E}{\rightsquigarrow} l_n, S_n}$$

$$\text{(E-Children-F)}\frac{\Gamma, S \vdash e \overset{E}{\rightsquigarrow} v \notin H^{\bullet}, S'}{\Gamma, S \vdash \texttt{children}\ e \overset{E}{\rightsquigarrow} \bot, S'}$$

Opération replace L'opération replace est définie de façon récursive, en dé-construisant la liste fournie pour ajouter un à un ses éléments comme fils du nœud. Un aspect important à noter est que le nœud est vidé avant d'être reconstruit, ainsi les fils déjà présents seront ajoutés à nouveau sans avoir à être copiés. La première règle d'évaluation E-Replace-Eval précise que le nœud et la liste doivent être évalués en premier. L'environnement est alors enrichi des noms B et V contenant les résultats de ces évaluations, et la main est passée à la règle E-Replace-Step, qui décrit le parcours de la liste récursivement, et décrit l'ajout des fils en récursion en queue. La dernière récursion est alors évaluée avec la règle E-Replace-Clear qui décrit le nettoyage du nœud parent grâce aux primitives `children`, `child` et `detach`.

$$\text{(E-Replace-Eval)}\frac{\Gamma, S \vdash e_n \overset{E}{\rightsquigarrow} \bullet, S' \qquad \Gamma, S' \vdash e_c \overset{E}{\rightsquigarrow} v, S'' \qquad (B, \bullet) :: (V, v) :: \Gamma, S'' \vdash \texttt{replace B V} \overset{E}{\rightsquigarrow} (), S'''}{\Gamma, S \vdash \texttt{replace}\ e_n\ e_c \overset{E}{\rightsquigarrow} (), S'''}$$

$$\text{(E-Replace-Step)}\frac{\begin{array}{c}\Gamma, S \vdash \texttt{V.(}\Sigma\ \texttt{val).(}\Sigma\ \texttt{0)} \overset{E}{\rightsquigarrow} \bullet_c, S \qquad \Gamma, S \vdash \texttt{V.(}\Sigma\ \texttt{val).(}\Sigma\ \texttt{1)} \overset{E}{\rightsquigarrow} t, S \\ (B, \bullet) :: (V, t) :: \Gamma, S \vdash \texttt{replace B V} \overset{E}{\rightsquigarrow} (), S' \qquad S' \vdash \texttt{bind}(\bullet, \bullet_c) = (), S''\end{array}}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \texttt{replace B V} \overset{E}{\rightsquigarrow} (), S''}$$

$$\text{(E-Replace-Clear)}\frac{\begin{array}{c}S \vdash \texttt{children}(\bullet) = n, S_0 \\ S_i \vdash \texttt{child}(\bullet, i) = \bullet_i, S'_i \\ S'_i \vdash \texttt{detach}(\bullet_i) = \mathit{nil}, S_{i+1}\end{array} \text{ où } 0 \leqslant i < n}{(B, \bullet) :: (V, Nil) :: \Gamma, S \vdash \texttt{replace B V} \overset{E}{\rightsquigarrow} (), S'_n}$$

$$\text{(E-Replace-Eval-F-1)}\frac{\Gamma, S \vdash e_n \overset{\mathsf{E}}{\rightsquigarrow} v \notin H^\bullet, S'}{\Gamma, S \vdash \mathsf{replace}\ \ e_n\ \ e_c \overset{\mathsf{E}}{\rightsquigarrow} \bot, S'}$$

$$\text{(E-Replace-Eval-F-2)}\frac{\Gamma, S \vdash e_n \overset{\mathsf{E}}{\rightsquigarrow} \bullet, S' \qquad \Gamma, S' \vdash e_c \overset{\mathsf{E}}{\rightsquigarrow} \bot, S''}{\Gamma, S \vdash \mathsf{replace}\ \ e_n\ \ e_c \overset{\mathsf{E}}{\rightsquigarrow} \bot, S''}$$

$$\text{(E-Replace-Step-F-1)}\frac{\Gamma, S \vdash \mathtt{V.(\Sigma\ val).(\Sigma\ 0)} \overset{\mathsf{E}}{\rightsquigarrow} \bot, S}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \mathsf{replace}\ \ \mathtt{B\ V} \overset{\mathsf{E}}{\rightsquigarrow} \bot, S''}$$

$$\text{(E-Replace-Step-F-2)}\frac{\Gamma, S \vdash \mathtt{V.(\Sigma\ val).(\Sigma\ 0)} \overset{\mathsf{E}}{\rightsquigarrow} v \notin H^\bullet, S}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \mathsf{replace}\ \ \mathtt{B\ V} \overset{\mathsf{E}}{\rightsquigarrow} \bot, S''}$$

$$\text{(E-Replace-Step-F-3)}\frac{\Gamma, S \vdash \mathtt{V.(\Sigma\ val).(\Sigma\ 0)} \overset{\mathsf{E}}{\rightsquigarrow} \bullet_c, S \qquad \Gamma, S \vdash \mathtt{V.(\Sigma\ val).(\Sigma\ 1)} \overset{\mathsf{E}}{\rightsquigarrow} \bot, S}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \mathsf{replace}\ \ \mathtt{B\ V} \overset{\mathsf{E}}{\rightsquigarrow} \bot, S''}$$

Cette évaluation pourrait être décrite autrement, par exemple en décrivant replace comme une fonction du langage et/ou en étendant la sémantique du document. Il est alors impératif que l'implantation de replace respecte les deux points suivants.

Proposition 12.5.1 *À l'issue de l'évaluation de la primitive replace, l'ensemble des enfants du nœud se trouve dans l'un des deux cas suivant :*

1. *inchangé (si la primitive a échoué),*
2. *complètement remplacé, par les nœuds (ou des copies de ceux-ci) apparaissant dans la liste passée en paramètre, et ce dans le même ordre.*

Cette première proposition décrit la gestion d'erreur, et est en particulier indispensable si une telle solution devait être adaptée à un langage avec exceptions.

Proposition 12.5.2 *Lors de l'évaluation de la primitive replace n l, chaque nœud apparaissant dans la liste l est ajouté comme suit :*

- *S'il n'était pas déjà enfant de n, et qu'il était enfant d'un autre nœud, une copie est ajoutée.*
- *S'il était déjà enfant, et qu'il n'apparaît qu'une seule fois dans l, alors il reste enfant sans être copié.*
- *S'il apparaît plusieurs fois dans l, alors la dernière occurrence sera le nœud lui-même, et les occurrences précédentes seront des copies distinctes.*

Cette seconde proposition décrit le comportement voulu de la primitive, permettant des opérations comme la suppression d'un nœud sans que tous les autres se voient implicitement copiés. Elle est en particulier nécessaire pour que l'exemple du glisser-déposer que nous avons vu à la section 10.4 fonctionne comme prévu.

Filtrage FidoML définit un filtre permettant de discriminer un nœud selon son étiquette, et permettant d'analyser ses propriétés en profondeur. Son évaluation est similaire aux autres types produits. Cependant, si on veut filtrer les enfants d'un nœud en donnant un filtre de liste, il convient de recréer (et donc d'allouer) la liste des enfants à l'aide de la primitive children, puisque cette liste est implantée de façon interne par le document, et pas comme une liste ML. Comme un tel filtrage *actif* n'est pas habituel pour ML, nous donnons les deux variantes.

1. **Variante sans imbrication** Cette variante conserve la propriété que l'état du document résultant est l'état initial.

$$\text{(P-Node)}\ \frac{\begin{array}{c} S \vdash \mathsf{tag}(\bullet) = t, S \quad t = tag \\ S \vdash \mathsf{get}(\bullet, l_i) = v_i, S \\ \Gamma_i, S, v_i \vdash p_i \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{i+1} \end{array} \text{ où } 0 \leqslant i \leqslant n}{\Gamma_0, S, \bullet \vdash \texttt{node <tag> prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{n+1}}$$

$$\text{(P-Node-F-1)}\ \frac{S \vdash \bullet(t) =, \quad t \neq tag}{\Gamma, S, \bullet \vdash \texttt{node <tag> prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \bot}$$

$$\text{(P-Node-F-2)}\ \frac{\begin{array}{c} S \vdash \bullet(t) =, \quad t \neq tag \\ S \vdash \mathsf{get}(\bullet, l_i) = v_i, S \\ \Gamma_i, S, v_i \vdash p_i \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{i+1} \\ \Gamma_j, S, v_j \vdash p_j \overset{\mathrm{P}}{\rightsquigarrow} \bot \end{array} \text{ où } 0 \leqslant i < j \leqslant n}{\Gamma, S, \bullet \vdash \texttt{node <tag> prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \bot}$$

2. **Variante avec imbrication** Dans cette variante au contraire, l'état en sortie peut être modifié. Par le jeu des liaisons au sein du filtrage, l'environnement peut en outre référencer les valeurs allouées par le filtre actif.

$$\text{(P-Node')}\ \frac{\begin{array}{c} S \vdash \mathsf{tag}(\bullet) = t, S \quad t = tag \\ (B, \bullet) :: \Gamma, S \vdash \mathsf{children}\ B \overset{\mathrm{E}}{\rightsquigarrow} v_l, S' \quad \Gamma_0, S', v_l \vdash p \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_0, S' \\ S \vdash \mathsf{get}(\bullet, l_i) = v_i, S \\ \Gamma_i, S', v_i \vdash p_i \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{i+1}, S'' \end{array} \text{ où } 0 \leqslant i \leqslant n}{\Gamma, S, \bullet \vdash \texttt{node <tag> } p \texttt{ prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{n+1}, S''}$$

$$\text{(P-Node'-F-1)}\ \frac{S \vdash \mathsf{tag}(\bullet) = t, S \quad t \neq tag}{\Gamma, S, \bullet \vdash \texttt{node <tag> } p \texttt{ prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{n+1}, S}$$

$$\text{(P-Node'-F-2)}\ \frac{\begin{array}{c} S \vdash \mathsf{tag}(\bullet) = t, S \quad t = tag \\ (B, \bullet) :: \Gamma, S \vdash \mathsf{children}\ B \overset{\mathrm{E}}{\rightsquigarrow} v_l, S' \quad \Gamma_0, S', v_l \vdash p \overset{\mathrm{P}}{\rightsquigarrow} \bot, S' \end{array}}{\Gamma, S, \bullet \vdash \texttt{node <tag> } p \texttt{ prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{n+1}, S'}$$

$$\text{(P-Node'-F-3)}\ \frac{\begin{array}{c} S \vdash \mathsf{tag}(\bullet) = t, S \quad t = tag \\ (B, \bullet) :: \Gamma, S \vdash \mathsf{children}\ B \overset{\mathrm{E}}{\rightsquigarrow} v_l, S' \quad \Gamma_0, S', v_l \vdash p \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_0, S_0 \\ S_i \vdash \mathsf{get}(\bullet, l_i) = v_i, S_i \\ \Gamma_i, S_i, v_i \vdash p_i \overset{\mathrm{P}}{\rightsquigarrow} \Gamma_{i+1}, S_{i+1} \\ \Gamma_j, S, v_j \vdash p_j \overset{\mathrm{P}}{\rightsquigarrow} \bot, S'' \end{array} \text{ où } 0 \leqslant i < j \leqslant n}{\Gamma, S, \bullet \vdash \texttt{node <tag> } p \texttt{ prop } l_0 \texttt{ = } p_0 \cdots \texttt{ prop } l_n \texttt{ = } p_n \texttt{ end} \overset{\mathrm{P}}{\rightsquigarrow} \bot, S''}$$

12.6 Correction

Notre présentation de FidoML était volontairement axée langage et spécification d'implantation plutôt que modèle théorique minimal. Il serait donc trop long et fastidieux de donner les preuves exhaustives et formelles de correction du système. Cette partie présente tout de même les lemmes principaux assurant la cohérence entre les sémantiques et le système de types, ainsi que les schémas généraux des preuves associées.

Lemme 12.6.1 (Interface) *Au sein de la sémantique du langage, un appel de primitive du document n'est effectué que si ses prémisses sont satisfaites.*

Schéma de preuve Par construction des règles. Lorsqu'une primitive doit être appelée, la sémantique définit toujours (1) une règle de réussite, appelant la primitive si les prémisses sont satisfaites, et (2) une règle d'échec, correspondant au cas où les prémisses ne sont pas satisfaites, dans laquelle l'appel n'est pas fait. □

Lemme 12.6.2 (Progression de l'évaluation) *Pour une expression du langage, il existe toujours une règle décrivant son évaluation.*

Schéma de preuve Par induction sur la syntaxe, pour chaque forme d'expression, on vérifie qu'il y a une règle de réussite, et que le dual des prémisses correspond exactement à l'union des prémisses des règles d'échec correspondantes. La propagation d'une erreur provoquée par l'évaluation d'une sous-expression à travers la règle de succès est transparente pour la progression. □

Lemme 12.6.3 (Conservation) *L'évaluation d'une expression e du langage jugée de type τ selon le système de types, si elle termine, produit une valeur de type τ.*

Schéma de preuve La preuve de ce lemme n'est pas complètement classique, y compris pour la partie ML, puisque dans notre sémantique, les valeurs structurées ne sont pas un sous-ensemble des termes, et qu'il n'est donc pas possible d'y appliquer directement les règles de typage. Nous donnons juste une structure de preuve possible, sans rentrer dans les détails.

Un pré-requis à une telle preuve est donc d'ajouter une notion de compatibilité entre les valeurs construites et les types du langage. On définirait donc, par induction sur la grammaire des types, les valeurs compatibles avec chaque type. Par exemple, une valeur est compatible avec le type $(\alpha \times \beta)$ si c'est un nœud blanc, avec deux propriétés de noms 0 et 1.

Pour la partie ML classique, la preuve de conservation se fait alors par induction sur la structure syntaxique, grâce à l'aspect dirigé par la syntaxe de la sémantique opérationnelle et du système de types.

1. Pour les constructions, il faut vérifier que le type de la valeur produite par la règle associée de la sémantique opérationnelle est compatible avec le type demandé par la règle de typage.
2. Pour les mutations, il faut s'assurer que le type de la valeur modifiée est conservé.

Et dans les deux cas, l'induction se traduit par le fait que les sous-expressions mises en œuvre, si elles terminent, s'évaluent en produisant des valeurs dont les types sont compatibles avec ceux décrits dans les prémisses de la règle de typage (et ne s'évaluent en particulier pas en $\bot$).

Pour les manipulations de document, la preuve se fait de façon similaire, en définissant une notion de typage pour les objets noirs du tas. Cependant, des difficultés supplémentaires sont introduites par les potentielles définitions récursives de nœuds, et les copies implicites.

On dit alors qu'un nœud est compatible avec le type générique node si ses enfants sont des nœuds bien typés, ses propriétés sont les bonnes par rapport à celles décrites dans le type de nœud correspondant à son étiquette, et les valeurs associées aux propriétés sont bien typées. Un nœud est compatible avec le type spécifique <*n*> node s'il est compatible avec le type node, et que son étiquette est n.

La difficulté par rapport aux autres constructions du langage est qu'à cause des définitions récursives de nœuds, on ne peut directement utiliser l'hypothèse d'induction pour dire que les nœuds utilisés pour la construction d'un nœud (où sa mutation par replace) sont bien typés. Ceci est problématique car on ne peut alors ni assurer que les enfants du nœud sont bien typés, ni que les éventuelles copies faites au cours de l'opération produisent des nœuds bien typés. C'est là qu'intervient l'analyse préalable de construction des nœuds. On utilise donc la proposition 11.1.2 pour appuyer l'induction, et montrer que les enfants sont bien typés, car il ne peut s'agir que de nœuds complètement construits. On utilise de plus la proposition 9.2.1 pour montrer que si un nœud est le résultat d'une copie implicite, alors il est bien typé, et son type est le même que le nœud.

Un cas particulier intéressant à observer est celui où un nœud est ajouté comme fils de lui-même par une opération replace. Dans ce cas, c'est la copie du nœud avant l'affection, qui était bien typée, qui lui est ajoutée. □

Théorème 12.6.1 (Correction) *L'évaluation d'une expression FidoML bien typée ne peut bloquer ou donner le résultat* $\bot$.

Preuve Avec le lemme de progression 12.6.2, on sait que l'évaluation ne peut bloquer à cause du langage, et avec le lemme d'interface 12.6.1, on sait qu'elle ne peut bloquer à cause d'un potentiel appel mal défini de primitive du document.

L'évalution peut alors (1) ne pas terminer, ou (2) terminer, auquel cas on sait grâce au lemme 12.6.3 qu'elle rend une valeur bien typée. Dans les deux cas, le programme ne peut s'évaluer en $\bot$. □

12.7 Interprétations pratiques

En résumé, nous avons :

1. Défini une sémantique formelle du document impératif décrivant (de façon plus abstraite) le comportement actuel des navigateurs.
2. Proposé et défini une sémantique alternative implantant un mécanisme de copie implicite pour palier aux comportements inhabituels de la sémantique précédente.
3. Puis nous avons proposé un langage, proposant une interface au document impératif, utilisant notre formalisation du document.
4. Nous avons alors spécifié l'évaluation de ce document sous forme d'une sémantique opérationnelle. Cette sémantique a pour particularité de déléguer ses accès et modifications aux valeurs allouées à la sémantique du document, de sorte qu'elle décrit l'évaluation de façon très précise et séquentielle.

Concrètement, la sémantique de FidoML varie suivant le modèle de document sous-jacent. Les opérations sur les valeurs FidoML autres que les nœuds se comportent de façon identique (et correcte et habituelle) dans les trois variantes. Au contraire, la sémantique du document sous-jacente influe considérablement sur le comportement des manipulations du document depuis FidoML.

- **Avec f_{DOM}** nous avons cherché à modéliser le comportement actuel des navigateurs. La sémantique de FidoML qui en découle conserve donc la majorité des problèmes dus aux manipulations impératives des enfants des nœuds. Cependant, le typage et la vérification statique des constructions apporte la sécurité du typage au niveau des attributs des nœuds, et évite les manipulations impératives intempestives du nœud durant sa construction.
- **Avec ε_{DOM}** on obtient une sémantique correcte pour le langage, dans laquelle toutes les opérations se comportent comme prévu. Le revers de la médaille est que cette propriété est assurée par le fait que le programme peut s'arrêter brutalement lors d'une opération sur un nœud. On obtient un langage qui paraît correct si on assume un modèle impératif de document, mais pas suffisante pour implanter un document fonctionnel.

– **Avec ^{c}DOM** FidoML répond à notre but initial en devenant un langage dans lequel toutes manipulations de document, impératives comme fonctionnelles, sont bien typées et ne peuvent échouer, grâce aux copies implicites.

Nous présentons maintenant rapidement plusieurs implantations possibles de ce langage, en donnant les intérêts et problématiques pratiques des différentes variantes. Nous reviendrons sur ces implantations et les inter-actions possibles entre elles dans la conclusion de cette thèse.

12.7.1 Dans un navigateur

Bien sûr, la première implantation de FidoML qui vient à l'esprit est dans un navigateur, à la façon d'OBrowser que nous avons décrit dans la première partie de cette thèse, pour permettre d'utiliser l'implantation du document du navigateur de façon plus saine.

Partie haute Il y a trois possibilités raisonnables d'implantation de FidoML dans ce cadre :

1. Compilation de FidoML vers le code-octet OCaml (ou vers OCaml puis passage à ocamlc), puis exécution du code-octet avec OBrowser (ou js_of_ocaml).
2. Écriture d'un compilateur de FidoML vers JavaScript.
3. Extension du langage OCaml pour ajouter la syntaxe node, et utilisation d'OBrowser, OcamlJS ou js_of_ocaml.

Les deux premières approches donnent le plus de souplesse, et permettent de respecter au mieux le schéma d'évaluation. D'autre part, FidoML est raisonnablement simple pour que celles-ci soient viables. La dernière version est cependant probablement plus ambitieuse, puisqu'elle permet de réutiliser tout le reste du langage OCaml. Cependant il faut alors s'assurer que l'extension est compatible avec le reste du langage (objets, modules, etc.), du point de vue typage comme évaluation.

Partie basse Dans une implantation utilisant JavaScript, on considère que les objets noirs sont les nœuds du document du navigateur, et les objets blancs sont tous les autres objets manipulés par JavaScript.

– **Avec ^{f}DOM et $^{\varepsilon}DOM$** , on peut considérer que les représentations des données OCaml pour les objets blancs sont correctes dans les trois implantations. Cependant, si on veut que la sémantique de FidoML soit respectée du point de vue des constructions de nœuds, il convient de ne pas permettre au programmeur d'accéder au document autrement que par une implantation des primitives de ^{f}DOM. En particulier si la solution est une extension d'OCaml, il faut restreindre les bibliothèques utilisées, ou les réécrire en conséquence.
– **Avec ^{c}DOM** , comme nous l'avons expliqué section 9.2.4, il faut embarquer les informations de portée à l'exécution. Avec OBrowser, cela signifie modifier les instructions et primitives de la bibliothèque d'exécution relatives à l'allocation. Pour un compilateur dédié il faut prendre en compte la mise en place de ces informations dans le schéma de compilation. De même, pour js_of_ocaml ou OcamlJS, cela signifierait adapter la représentation des données et l'émission de code.

Avec un greffon Une implantation possible de FidoML serait d'évaluer le code (avec un interprète, une machine virtuelle ou en le compilant à la volée) dans un greffon du navigateur.

Nous avons déjà présenté rapidement les API des greffons et les problématiques pour implanter un langage inter-agissant avec la page avec ces technologies dans le chapitre 6. Un problème majeur est l'accès aux valeurs allouées entre les deux mondes sans provoquer de fuite mémoire.

Il y a alors deux possibilités :

1. La possibilité la plus simple, mais pas la plus efficace, serait de reprendre un schéma de compilation vers JavaScript évoqué plus haut, et de déléguer toutes les opérations sur les valeurs structurées à l'API npruntime qui rend les objets JavaScript accessibles aux greffons. Avec un tel mécanisme, la gestion mémoire est transparente pour le greffon, qui n'a plus qu'à s'occuper du modèle d'exécution du langage.
2. Une autre possibilité est de conserver un allocateur séparé pour le langage. Pour cela, la séparation des nœuds et du reste des valeurs dans notre modèle de document, ainsi le fait que les opérations mémoire du langage soient entièrement déléguées avec les primitives est un bon point de départ pour permettre de formaliser l'inter-action mémoire.

   Cependant, il n'est pas possible en pratique de simplement considérer les objets noirs comme objets du navigateur et les objets blancs comme objets du langage. En effet, le navigateur interprète certaines propriétés du document qui ne sont pas des valeurs immédiates (par exemple, les styles), et donc ces valeurs ne peuvent pas être des valeurs de FidoML, opaques pour le navigateur. Une direction possible pour résoudre ce problème serait la scission de l'ensemble des objets blancs du document, par exemple en deux ensembles d'objets blancs et gris, et l'extension de FidoML pour gérer spécifiquement ces objets gris.

12.7.2 Sur le serveur

Sur le serveur, l'habitude est d'utiliser une représentation intermédiaire fonctionnelle du document, permettant le partage au sein de l'arbre. Ce partage est transformé en duplication lors de l'émission du XML correspondant, mécanisme introduisant au passage les problèmes de duplication des champs *id* que nous avons décrits évoqués en introduction de cette partie.

Même dans les représentations plus intégrées au langage comme celle d'Ocsigen, ou il est possible de se passer des *id*s en prenant des références natives du langage sur les sous-nœuds, la transformation du partage en duplication ne peut être faite correctement en profondeur. En particulier, lorsque l'arbre contient des fonctions de rappel à transmettre au client, qui contiennent des références aux nœuds de l'arbre, il n'est pas possible de réaliser une copie en profondeur sans information supplémentaire (sinon, FidoML n'aurait pas de raison d'être).

L'utilisation de FidoML apporte alors plusieurs avantages pour une utilisation côté serveur.

- Tout d'abord, FidoML apporte la possibilité de programmer dans un style impératif le document du côté serveur, avec la même API que côté client.
- Ensuite, elle permet d'implanter le document fonctionnel, et ce de façon plus sûre qu'actuellement en donnant une sémantique claire aux duplications citées plus haut, en utilisant les informations supplémentaires de portée, et en les faisant à la demande lors de la construction de l'arbre, plutôt que toutes à la fois lors de l'émission d'XML.
- Et enfin, elle permet de mélanger des deux styles de façon bien typée. En particulier, un problème actuellement est que si l'on veut définir une fonction affectant seulement une partie de l'arbre (par exemple pour ajouter une feuille de style, remplacer un lien par un autre, etc.), alors il est nécessaire de construire une copie modifiée de l'arbre. Cette copie est problématique puisque si l'arbre contenait des références vers ses nœuds, par exemple via l'environnement de fonctions de rappel, celles-ci référenceront encore l'arbre original dans la copie. Avec FidoML, il est possible de modifier localement l'arbre de façon impérative conservant les références au sein de l'arbre. Il est même possible de copier le document à la racine, et les références au sein de l'arbre seront correctement redirigées dans la copie, il suffira de modifier impérativement ensuite la copie, pour obtenir une copie modifiée correcte du point de vue des références internes.

Partie haute Comme pour le navigateur, il y a deux possibilités : soit implanter un compilateur de FidoML, éventuellement en compilant vers OCaml comme langage intermédiaire, soit étendre le langage

OCaml pour lui ajouter les constructions de nœuds.

L'implantation du document étant écrite spécifiquement pour FidoML, l'implantation sur le serveur est cependant plus facile puisqu'il n'y a pas à se soucier d'éventuelles bibliothèques pouvant manipuler le document en concurrence avec les constructions du langage.

Partie basse La seule implantation permettant de donner complètement la vision fonctionnelle du document est l'implantation au dessus de ^{c}DOM. Nous avons déjà présenté les possibilités d'implantation de ^{c}DOM section 9.2.4.

13 Grammaire du document impératif

Le système de types présenté pour FidoML au chapitre précédent, permet d'assurer le typage des nœuds en eux-mêmes, mais pas de leur imbrication dans l'arbre selon une grammaire de document. Dans ce chapitre, nous ajoutons le typage de cette imbrication.

Nous présentons pour cela une technique permettant de projeter une grammaire du document vers des définitions de types FidoML. Le système de types de FidoML n'etant pas aussi évolué que ceux des langages dédiés à la manipulation d'arbre en XML, d'aucuns pourraient argumenter, de façon recevable qu'il vaudrait mieux utiliser un système de types plus approprié pour typer l'imbrication dans FidoML. La technique que nous développons prend volontairement le contre-pied de cet argument.

Le principe de la technique de typage de grammaire du document que nous développons ici est d'adapter une grammaire DTD par un système d'annotations, afin de la plier à un langage et un système de types visés. Concrètement, le système est séparé en deux parties : un *front-end* permet l'adaptation automatique ou manuelle, et il produit la grammaire adapée en vérifiant que les annotations sont cohérentes. Puis un *back-end* spécifique au langage visé projette la grammaire adaptée vers les primitives de ce langage (si elle est effectivement rendue projetable par les adaptations).

Après quelques rappels sur les grammaires de document, nous présentons tout d'abord les motivations et le fonctionnement général de l'approche. Puis, nous détaillons la partie front-end en nous appuyant sur un exemple. Finalement, nous donnons le back-end FidoML, en donnant la projection de notre exemple, et présentons d'autres back-ends possibles.

Rappels Commençons par rappeler les notions et termes relatifs aux formats de documents du Web, et faire le lien avec les concepts liés du document impératif.

- **Format des spécifications** Les formats des documents du Web (HTML, SVG, etc.) sont spécifiés par le W3C, sous la forme de DTD décrivant la syntaxe des fichiers XML conformes, et de spécifications en langue anglaise décrivant la sémantique à leur donner.
- **DTD** Une DTD décrit un ensemble fichiers XML (que l'on dit *valides* par rapport à cette DTD). Elle spécifie les *éléments* et *attributs* pouvant y apparaitre.
- **Éléments** En terminologie XML, les *éléments* correpondent aux *types de nœuds* du document impératif. Un élément est caractérisé dans une DTD par un ensemble d'*attributs* et une expression décrivant les *séquences* d'enfants possibles pour cet élémént.
- **Attributs** Les attributs sont les *propriétés* du document impératif, à ceci près que les valeurs associées ne peuvent être que des chaînes. Cependant, dans les DTD, les valeurs de ces chaînes peuvent éventuellement être contraintes à une expression rationnelle. En pratique, il s'agit en général simplement de restreindre les valeurs des attributs à des chaînes représentant des valeurs numériques, ou des valeurs spécifiques choisies dans un ensemble fixé (énumérations). On peut donc souvent interpréter de façon légitime les valeurs des attributs étant d'un certain type, et non forcément comme des chaînes.
- **Séquences** Dans les DTD, les séquences d'enfants possibles pour un élément sont décrites par une expression rationnelle sur les éléments [1].

1. D'autres systèmes de grammaires pour XML, comme XMLSchema ou Relax NG ont des variantes d'expressivités variées, mais le format historique et le plus répandu et pris en charge reste DTD.

13.1 Motivations et présentation générale

Typage XML statique dans les solutions Web Pour comprendre un premier intérêt de cette approche très pragmatique du typage de la grammaire dans FidoML, rappelons-nous quel est l'état du typage de la grammaire du document dans les solutions Web existantes.

- Comme le montre OCamlDuce [📖14], l'intégration d'un système de types pour XML dans un langage applicatif est possible, mais non trivial.
- De ce premier point découle le constat que les solutions utilisées dans les langages Web les plus courants font simplement l'impasse sur le typage de la grammaire.
- Dans Ocsigen, qui est probablement la solution Web utilisable en pratique la plus avancée dans ce domaine, le typage de la grammaire du document existe. Mais là aussi, son implantation reste décevante puisqu'elle est encodée manuellement dans le système de types d'OCaml, avec le caractère fastidieux et les risques d'erreurs et d'oublis que cela implique.
- Pour renchérir sur ce dernier point, un tel encodage manuel, s'il est utilisable, est impossible à réutiliser sans réécriture dans un autre cadre (même proche comme SML ou F#, car il utilise des traits avancés spécifiques à OCaml comme les variants polymorphes), et nécessite un système de types d'expressivité au moins équivalente.

Le système de types pour les nœuds eux-mêmes introduit dans FidoML est suffisamment simple et adaptable pour pouvoir être incorporé à d'autres langages, voire encodé dans d'autres systèmes de types, sans trop de difficulté. Il nous a donc paru logique de continuer sur cette voie, et de typer la grammaire selon une technique simple, adaptable à d'autres langages de systèmes de types d'expressivités variées, et ce de façon mécanisée.

Les grammaires du Web Outre cette volonté d'obtenir une solution réutilisable et adaptable à des systèmes existants, il y a d'autres raisons pour lesquelles on ne veut pas utiliser directement les grammaires existantes du W3C pour typer les documents. En pratique, les DTD du Web ont un certain nombre de points négatifs, dont voici ceux sur lesquels notre solution se concentre.

- Certains formats de document, dans le but de faciliter l'écriture manuelle, permettent la construction de certains éléments de façons multiples. L'encodage sous forme d'expressions rationnelles est en général fastidieux à écrire et difficile à relire [2].
- En pratique, certaines structures de documents décrites dans les DTDs sont inutilisées, ou non supportées, et un document bien typé peut donc échouer car il utilise une telle structure.
- Certains éléments peuvent être construits de nombreuses façons différentes, sans réelle justification.
- Pire, ces façons différentes sont souvent encodées dans de larges expressions disjonctives difficilement lisibles.
- De façon générale, certaines séquences ou alternatives complexes sont parfois trop difficilement lisibles, car il n'est pas possible d'étiqueter ou nommer les sous-expressions. Le résultat est que certaines DTD (SVG est un exemple frappant [3]) abusent des attributs pour encoder le contenu du document là où l'utilisation d'éléments aurait pu être plus justifiée, principalement car ceux-ci sont identifiés par un nom et que le XML est donc plus lisible.
- Dans le but d'intégrer la grammaire à un langage de programmation généraliste, les noms utilisables pour les éléments et attributs sont un peu trop permissifs.

Notre proposition Nous proposons donc un système mécanisé, dans lequel le programmeur adapte à ses besoins une grammaire de document existante, sous forme de règles de réécriture.

2. Un exemple est la section *head* du format HTML, qui est décrite de façon alambiquée, pour que l'élément *title* y apparaisse une (et une seule) fois, et ce en n'importe quelle position. Concrètement, l'expression ressemble à `(title,X*)|(X+,title,X*)`.
3. Par exemple, en SVG, une transformation géométrique est encodée dans une chaîne associée à un attribut transform comme suit : `<g transform="translate(700 210) rotate(-30)">...</g>`.

1. Le système prend en entrée une DTD, et la transforme en une grammaire lisible par un humain. Le programmeur donne alors dans un fichier séparé des règles permettant de réécrire la grammaire. Le front-end peut générer aussi automatiquement certaines règles de réécriture qui lui semblent judicieuses, qui peuvent servir de base au programmeur.
2. Le système se charge alors automatiquement de vérifier que la grammaire adaptée décrit bien des documents correspondants à ceux de la grammaire originale, et produit une grammaire alternative, dans un format intermédiaire.
3. À ce point de la chaîne, le front-end passe la main au backend, qui va projeter automatiquement cette grammaire intermédiaire vers des primitives du langage cible.
4. Le système est bien séparé, de façon à pouvoir écrire la projection vers un nouveau langage cible sans toucher au reste du système. Bien sûr, cette séparation est à relativiser par le fait que les annotations données par le programmeur pour adapter la grammaire dépendent des possibilités du langage cible et du back-end.

Le choix d'utiliser des règles de réécriture de grammaire a été préféré à la réécriture manuelle dans le but de pouvoir plus facilement prendre en compte les changements mineurs entre les versions de la DTD. La figure 13.1 donne une vue d'ensemble du système, dont nous expliquerons les différentes parties dans la suite.

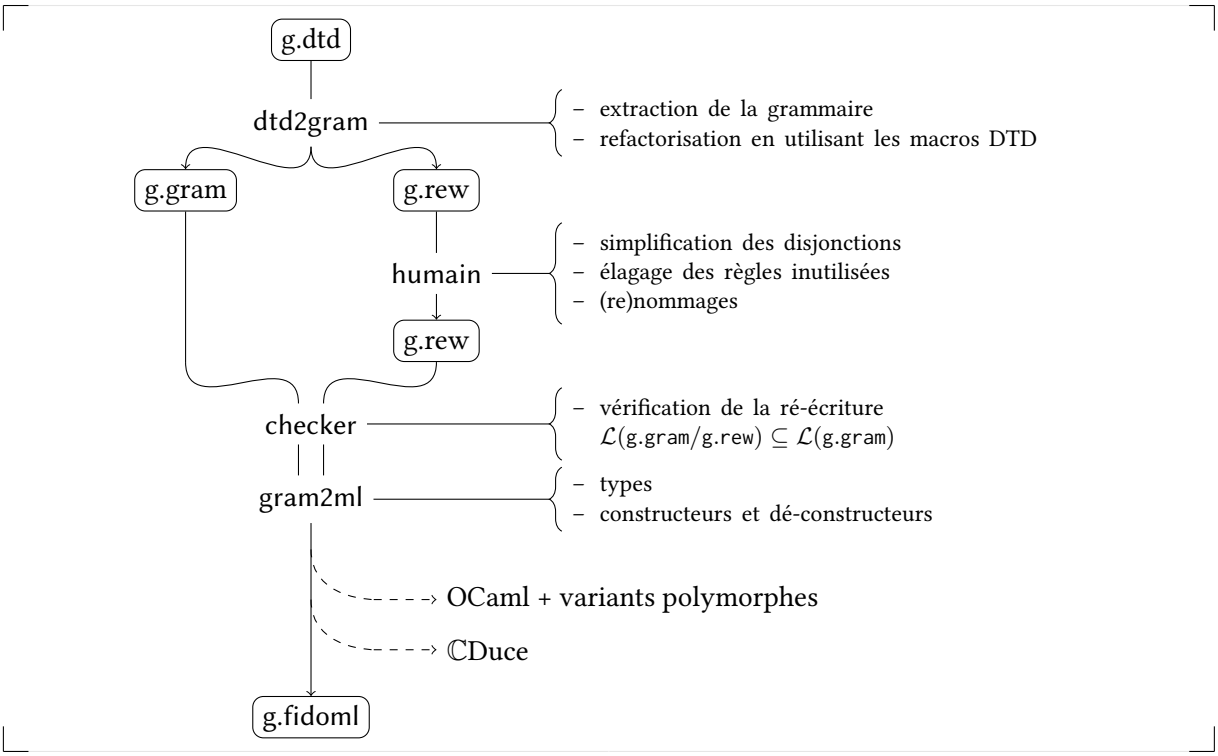


FIGURE 13.1: Schéma général du système.

13.2 Front-end

Le front-end est composé de deux outils dtd2gram et checkgram (et un humain), et manipule trois formats de description pour les grammaires, les annotations et les grammaires adaptées. L'outil dtd2gram transforme une DTD en un fichier de grammaire, et checkgram prend une grammaire et un fichier d'annotations pour produire une grammaire adaptée.

Grammaire initiale La première tâche du front-end est donc de produire une grammaire lisible par un humain à partir d'une DTD. Pour ceci, nous introduisons une syntaxe de description de grammaires, utilisant la terminologie du document impératif plutôt que celle d'XML, dont la BNF (*Bachus Naur Form*) est donnée ci-après.

- La syntaxe node représente une définition de type de nœud. Elle doit préciser l'ensemble des noms de propriétés, et l'expression rationnelle décrivant les séquence possibles d'enfants de ce type de nœuds.
- La syntaxe prop permet d'associer un nom de propriété à son type. Dans cette syntaxe description de grammaires, les propriétés portant le même nom doivent avoir le même type (comme en FidoML, cette décision a été prise car une majorité de DTD entrent dans cette restriction, et cela simplifie un certain nombre de problèmes).
- La syntaxe expr permet de nommer une expression rationnelles, utilisable pour factoriser et rendre plus lisibles les définitions de types de nœuds. Les définitions ne sont pas récursives (ni mutuellement récursives).

Dans le cas où la grammaire utilisée par le programmeur ne provient pas d'une source tierce, il est bien entendu possible de l'écrire directement dans le format, sans passer par une DTD.

```
grammar   ::=  [ phrase ]+
phrase    ::=  node id : [ [ id+ ] ]? expr
             |  prop id : prop-expr
             |  expr id = expr
expr      ::=  ( expr [ | expr ]+ ) suffix?
             |  ( expr [ , expr ]+ ) suffix?
             |  [ id | text | void ] suffix?
suffix    ::=  ? | + | *
prop-expr ::=  int | string | url | date | ··· | enum
enum      ::=  id [ | id ]*
```

Nous allons, dans cette section utiliser un exemple de grammaire pour un canevas de dessin vectoriel. Commençons donc par donner la grammaire de base de cet exemple avec la syntaxe que nous venons de décrire :

```
1  prop value : int
2  prop name : string
3  prop unit : px | pt | in
4
5  node dimen : [ value unit? ] void
6  node point : [ name? ] (dimen, dimen)
7
8  node rotation : (point, dimen)
9  node scale : (point, dimen, dimen)
10 node translation : (point, dimen, dimen)
11 expr transform = (rotation | scale | translation | transformgroup)
12 node transformgroup : transform*
13
14 node rect : [ name? ] (point, point) | (point, dimen, dimen)
15 node circle : [ name? ] (point, point) | (point, dimen)
16 node triangle : [ name? ] (point, point, point)
17 expr shape = (rect | circle | triangle | group)
18 node group : [ name? ] (transform?, shape*)
19
20 node title : text
21 node meta : [ name ] text
22 node head : (title, meta*) | (meta+, title, meta*)
23 node body : shape*
24
25 node canvas : (head, body)
```

Lecture d'une DTD Extraire une telle grammaire à partir d'une DTD n'est pas aussi trivial qu'on pourrait le croire. Un problème majeur du format DTD est que la factorisation et la paramétrisation sont faites uniquement par la définition d'*entités*, qui sont des substitutions de texte. Concrètement, il est possible de définir une entité `nnn` par la syntaxe `<!ENTITY % nnn "xxx">` dans la DTD, et toute apparition de `&nnn;` dans la suite de la DTD sera remplacée par `xxx`.

Il n'y a pas de condition sur le contenu du texte à substituer, en particulier, le texte d'une macro peut être mal formé ou mal parenthésé, du moment que l'expansion complète de toutes les macros donne au final une DTD bien formée. De plus, les entités peuvent être définies en utilisant l'expansion d'autres entités. Il en résulte que la seule solution, pour être sûr de considérer la bonne grammaire, est de partir de la DTD complètement expansée.

Il y a deux problèmes principaux si on part d'une DTD complètement expansée. D'une part, si certaines entités étaient utilisées pour la lisibilité de la grammaire par nommage de sous-expressions, alors cette information est perdue. Et puisqu'il ne s'agit que de remplacement de texte, on ne peut pas savoir de façon automatique, lors de la définition d'une entité, si celle-ci va être utilisée pour un nommage de sous-expression ou pour tout autre utilisation. D'autre part, les DTD utilisent souvent des DTD externes standard contenant des entités standard, ainsi que des formats d'attributs classiques (tels que les nombres, les URL, etc.).

Fonctionnement de dtd2gram Le prototype de dtd2gram que nous avons écrit utilise l'algorithme suivant pour extraire une grammaire :

1. dtd2gram maintient une table d'associations (*nom*,*remplacement*,*expression*), ou *nom* est un nom d'entité et *remplacement* est le texte de substitution associé déjà expansé s'il contenait d'autres entités. *expression* contient éventuellement l'expression de la grammaire de sortie correspondante, si elle existe.

   Cette table est initialisée avec une base connue, associant les noms et descriptions standard pour les types d'attributs connus aux types de base de la grammaire (`int`, `string`, etc.)
2. dtd2gram maintient aussi une séquence de règles correspondant aux règles de la grammaire de sortie, initialement vide.
3. La DTD est lue du début à la fin, en effectuant normalement les substitutions d'entités (le prototype utilise une bibliothèque XML existante pour cette tâche).
4. Lors de la définition d'une entité, un triplet est ajouté, avec une troisième composante vide, et une pseudo-règle est ajoutée à la fin de la séquence de sortie, dans le but de mémoriser à quel moment cette entité a été lue.
5. Lors de la définition d'un élément, dtd2gram ajoute une règle `node` à la séquence de sortie.

   Pour calculer l'expression associée, l'algorithme effectue l'analyse syntaxique de l'expression rationnelle de la DTD, pour obtenir un AST (*Abstract Syntax Tree (Arbre de Syntaxe Abstraite)*). Puis l'AST est parcouru, et pour chaque sous-expression, il vérifie si la sous-chaine correspondante existe comme deuxième composante d'un triplet de la table.

   (a) Si une substitution est reconnue, et que sa troisième composante n'est pas vide, alors la sous-expression dans l'AST est remplacée par une référence à cette substitution.

   (b) Sinon, dtd2gram applique le même algorithme au texte de remplacement de la substitution, et modifie la troisième composante dans la table par l'AST résultant.

   Pour calculer les propriétés, dtd2gram effectue l'analyse syntaxique de la définition des attributs dans la DTD, et pour chaque attribut, ajoute au début de la séquence de sortie une définition de propriété, si celle-ci n'existait pas déjà (et vérifie que le type est le même si elle existait déjà). Le type de chaque attribut est comparé avec les types primitifs présents dans la table. Si le type n'est pas un type simple, le prototype actuel de dtd2gram sait convertir automatiquement les types

énumérés simples. Sinon il échoue et le programmeur peut corriger le problème en effectuant la projection à la main et en l'ajoutant à la table initiale.

6. Finalement, dtd2gram imprime la séquence de sortie, en sautant les pseudo-règles correspondant aux entités auxquelles aucune signification n'a été donnée.

Au final, l'algorithme fonctionne bien en pratique, et retrouve et factorise correctement les nommages de sous-expressions. De plus, comme cette analyse est faite à la demande, l'algorithme n'essaye pas d'analyser les substitutions ne correspondant pas à des nommages. En plus d'éviter des erreurs inutiles, l'avantage de cette technique est qu'elle n'ajoute pas de bruit dans la grammaire de sortie pour les substitutions qui ressemblent syntaxiquement à des nommages de sous-expressions, mais ne sont pas utilisées comme telles.

Réécriture de la grammaire Concrètement, pour répondre aux problèmes présentés à la section précédente, nous cherchons à permettre les quatre sortes d'opérations suivantes.

1. **Élagage** Dans les alternatives, il faut pouvoir supprimer des branches inutilisées, ou trop compliquées à projeter vers le langage cible.
2. **Suppression** Il faut pouvoir supprimer un type de nœuds ou une propriété non supportée ou inutilisée.
3. **Nommage** Les branches des alternatives et les éléments des séquences doivent tous pouvoir être nommés. Cela peut être tout d'abord indispensable pour rendre projetable la grammaire dans le langage cible (ou au moins éviter une génération automatique de noms laids), mais aussi pour clarifier et documenter le rôle de certains éléments de la grammaire.
4. **Alias** Le programmeur doit pouvoir contrôler comment les noms de la grammaire sont projetés vers les identifiants du langage, dans le cas où les noms originaux ne sont pas utilisables directement.

Le fichier d'annotations répond à la BNF suivante :

```
rules   ::=  [ rule ]+
rule    ::=  remove [ prop | node ] id
          |  rename id to id
          |  rewrite node id to [ [ id+ ] ]? annot-expr
          |  rewrite expr id to expr
          |  expr id = expr
expr    ::=  ( expr [ | expr ]+ ) suffix? [ as id ]?
          |  ( expr [ , expr ]+ ) suffix? [ as id ]?
          |  [ id | text | void ] suffix? [ as id ]?
```

Le programmeur peut ainsi renommer n'importe quel mot de la grammaire avec rename, réécrire une règle de construction de nœuds, pour réduire sa liste de propriétés ou élaguer son expression, définir de nouveaux alias pour factoriser la grammaire, et nommer toutes les sous-expressions dans les expressions de nœuds et d'alias.

Par exemple, le listing suivant donne un ensemble de règles permettant au programmeur de rendre plus lisible notre exemple de grammaire en nommant les champs des objets graphiques et transformations, les cas possibles pour la construction de chaque objet. Le programmeur souhaite aussi enlever les noms, car il préfère utiliser les références de son langage plutôt que de rechercher les objets par leur nom. Il enlève aussi les triangles, car il les trouve trop pointus, et ne laisse la possibilité de créer des cercles que par leur centre et leur rayon. De même, le programmeur n'autorise le titre du document à se trouver qu'au début de l'en-tête, car il préfère avoir une API plus simple dans son langage cible que la possibilité de mettre son titre n'importe-où. Pour ne pas interférer avec le type prédéfini du langage, il renomme aussi unit.

```
rewrite node point to (dimen as x, dimen as y)
rewrite node rotation to (point as center, dimen as angle)
rewrite node scale to (point as center, dimen as sx, dimen as sy)
rewrite node translation to (point as center, dimen as dx, dimen as dy)

rewrite rect to
  ( (point as topleft, point as bottomright) as absolute
  | (point as center, dimen as width, dimen as height) as relative)
rewrite node circle to (point as center, dimen as radius)
rewrite expr shape to rect | circle | group
remove node triangle

rewrite head to (title, meta*)
rename unit to u
```

Grammaire adaptée L'outil checkgram vérifie alors que les expressions réécrites sont cohérentes avec les expressions originales (en vérifiant l'inclusion de langages). Il vérifie aussi que seuls les attributs optionnels sont enlevés dans les réécritures. Et bien sûr, il s'assure que les types de nœuds et propriétés supprimées ne sont pas utilisées dans le reste de la grammaire réécrite. La BNF décrivant les grammaires adaptées se déduit facilement des deux autres.

Puis la grammaire est réécrite vers la grammaire intermédiaire, simplement en remplaçant les expressions des types de nœuds et d'alias par les expressions réécrites avec rewrite, en insérant les nouveaux alias ajoutés avec expr avant leur première utilisation, en enlevant les éléments supprimés avec remove, et finalement en effectuant toutes les substitutions de noms demandées avec rename.

Au final, la grammaire adaptée par le programmeur devient :

```
prop value : int
prop name : string
prop u : px | pt | in

node dimen : [ value u? ] void
node point : (dimen as x, dimen as y)

node rotation : (point as center, dimen as angle)
node scale : (point as center, dimen as sx, dimen as sy)
node translation : (point as center, dimen as dx, dimen as dy)
expr transform = (rotation | scale | translation | transformgroup)
node transformgroup : transform*

node rect :
  ( (point as topleft, point as bottomright) as absolute
  | (point as center, dimen as width, dimen as height) as relative)
node circle : (point as center, dimen as radius)
expr shape = (rect | circle | group)
node group : [ name? ] (transform?, shape*)

node title : text
node meta : [ name ] text
node head : (title, meta*)
node body : shape*

node canvas : (head, body)
```

13.3 Back-end FidoML

Pour utiliser une grammaire de document en FidoML, nous étendons tout d'abord le langage pour restreindre les séquences enfants possibles d'un type de nœud, puis nous donnons le principe de projection d'une grammaire vers le langage ainsi étendu.

13.3.1 Extension de langage

L'idée de cette extension est de pouvoir contraindre la séquence des enfants des nœuds en utilisant le système de types existant du langage. L'extension pourrait d'ailleurs être utilisée de façon indépendante du front-end, et la grammaire définie directement avec les types du langages.

Avec la spécification de FidoML présentée au chapitre 11, lors de la construction ou l'affectation d'un nœud, le langage demande une valeur de type node list. La séquence des enfants est alors simplement composée des éléments de la liste, en respectant l'ordre.

Avec cette extension, au lieu de prendre une liste de nœuds, de type node list, la construction d'un nœud prend une valeur d'un type spécifique à son étiquette. Le développement de cette valeur en une séquence d'enfants est le résultat du parcours en profondeur de la valeur. Par exemple, un couple (a, b) de listes de nœuds sera développé en la séquence contenant d'abord les éléments de la liste a dans l'ordre, puis les éléments de la liste b, dans l'ordre. Une valeur de type node list sera développée comme dans le langage original. La primitive replace est aussi modifiée pour prendre une valeur de ce type plutôt qu'une liste. On ne change pas l'accès aux enfants, qui utilise toujours une liste de nœuds de type générique pour simplifier.

La syntaxe de définition des types de nœuds doit donc être modifiée pour ajouter ce type (le reste de la syntaxe n'a pas besoin d'être modifié).

```
phrase  -=  node type <tag> [ mutable? prop id : type ]* end
phrase  +=  node type <tag> type [ mutable? prop id : type ]* end
```

Terminaison Puisque les enfants du nœud sont ceux rencontrés au cours du parcours de la valeur, il y a un risque que l'opération ne termine pas si ce parcours est infini, ce qui arrive si la valeur contient un cycle. Il y a trois possibilités pour assurer la terminaison. (1) Laiser au programmeur la responsabilité de ne pas donner de valeur cyclique. (2) Restreindre les types acceptés dans la définition de type de nœud à un sous-ensemble pour lequel le langage ne peut construire de valeur cyclique. (3) Stopper le programme dynamiquement si un cycle est détecté.

Le choix le plus naturel pour FidoML est le second, puisque le langage ne dispose pas de liaisons récursives généralisées et donc la confection de valeur récursive doit passer par une mutation, et qu'une erreur signifie l'arrêt du programme puisque le langage n'a pas d'exceptions. Mais pour une adaptation à un langage comme OCaml, la troisième solution est probablement meilleure.

Modification du système de types Tout d'abord, il faut modifier la prise en charge des définitions de types de nœuds. Pour ceci, on ajoute une composante $D_G \subseteq (tag \times \tau)$ à l'environnement de définition D, que l'on enrichit avec la fonction *def*. Une définition n'est acceptée que si elle ne référence aucune structure mutable.

$$def(D, \text{node type <}tag\text{> } \tau \text{ prop } e_{m_0}\ p_0 : \tau_0 \cdots \text{prop } e_{m_m}\ p_m : \tau_m \text{ end}) = (D_A, D_R, D_F, D'_N, D'_P, D_S, D_C, D'_G)$$
$$\text{où } D'_G = D_G \cup (tag, \tau) \quad \textit{(autres composantes inchangées)}$$
$$\text{si } wd(\tau, D, \emptyset) \text{ et } developable(\tau, D) \quad \textit{(autres conditions inchangées)}$$

Où la fonction $developable(\tau, D)$ vérifie si le type τ (monomophe) ne désigne que des valeurs dont le parcours est possible et fini.

$$
\begin{array}{rcl}
\textit{developable}(\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{string}, D) & = & T \\
\textit{developable}(\mathsf{node} \mid \texttt{<}t\texttt{>}\ \mathsf{node}, D) & = & T \textit{(non parcourus en profondeur)} \\
\textit{developable}(\tau_a \to \tau_b, D) & = & F \\
\textit{developable}(\tau_0 \times \cdots \times \tau_n) & = & \textit{developable}(\tau_0, D) \wedge \cdots \wedge \textit{developable}(\tau_n, D) \\
\textit{developable}((\tau_0, \cdots, \tau_n)\ t, D) & \text{où} & (\alpha_0, \cdots, \alpha_n)\ t \in D_A \text{ et } D_A(t) = \tau \\
 & = & \textit{developable}(\tau[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant n]) \\
\textit{developable}((\tau_0, \cdots, \tau_n)\ t, D) & \text{où} & (\alpha_0, \cdots, \alpha_n)\ t \in D_R \text{ et } D_R(t) = \{l_0, \cdots, l_m\} \\
 & = & \forall 0 \leqslant k \leqslant m, (l_k, (t, \tau_k, F)) \in D_F \\
 & & \wedge\ \textit{developable}(\tau_k[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant n]) \\
\textit{developable}((\tau_0, \cdots, \tau_n)\ t, D) & \text{où} & (\alpha_0, \cdots, \alpha_n)\ t \in D_S \text{ et } D_S(t) = \{c_0, \cdots, c_m\} \\
 & = & \forall 0 \leqslant k \leqslant m, (c_k, (t, \tau_k)) \in D_C \\
 & & \wedge\ \textit{developable}(\tau_k[\alpha_i \leftarrow \tau_i, 0 \leqslant i \leqslant n])
\end{array}
$$

Puis il faut modifier le typage de la construction des nœuds, et celui de replace (qui n'est plus une opération générique) pour utiliser les informations de D_G.

$$
(\textsc{Node})\frac{D_N(\textit{tag}) = \bigcup_i \{l_i\} \qquad \Gamma, D \vdash e\ :\ D_G(\textit{tag}) \qquad \Gamma, D \vdash e_i\ :\ D_P(l_i) \text{ où } 0 \leqslant i \leqslant n \qquad \texttt{<}\textit{tag}\texttt{>}\ \mathsf{node} \lessdot \tau}{\Gamma, D \vdash \mathsf{node}\ \texttt{<tag>}\ e\ \mathsf{prop}\ l_0\ \texttt{=}\ e_0\ \cdots\ \mathsf{prop}\ l_n\ \texttt{=}\ e_n\ \mathsf{end}\ :\ \tau}
$$

$$
(\textsc{Replace})\frac{\Gamma, D \vdash e\ :\ \texttt{<}\textit{tag}\texttt{>}\ \mathsf{node} \qquad \Gamma, D \vdash e_c\ :\ D_G(\textit{tag})}{\Gamma, D, \tau \vdash \mathsf{replace}\ e\ e_c\ :\ \mathsf{unit}}
$$

Modification de la sémantique Les deux structures du langage dont la sémantique change sont la construction et la primitive replace. La sémantique de la construction étant définie en utilisant celle de replace, on modifie donc seulement l'évaluation de cette dernière, qui effectue un parcours en profondeur d'une valeur générique, plutôt qu'un parcours de liste. Pour ceci, de la même manière que pour le parcours de liste, une première règle (E-Replace-Eval) effectue les évaluations des sous-expressions, et vide le nœud si celles-ci ont réussi. Puis l'environnement est enrichi de noms virtuels B et V permettant de lier les résultats de ces évaluations, et la main est passée à plusieurs règles de parcours (E-Replace-Traverse). Ici, l'exploration utilise la représentation des données définie par les règles de constructions. On ne spécifie pas l'ordre de parcours exact des enregistrements, car la spécification de la représentation des valeurs ne spécifie pas l'ordre des champs (le choix est laissé à l'implantation, en pratique, dans les implantations de ML, les étiquettes sont souvent remplacées par des entiers représentant l'ordre dans la déclaration et partagent la même représentation que les n-uplets). Il est donc important que le choix soit fait en accord avec l'ordre utilisé dans la projection de la grammaire. Une alternative serait d'utiliser les informations de typage et de générer un parcours spécialisé pour chaque type de nœud.

$$
(\textsc{E-Replace-Eval})\frac{\begin{array}{c}\Gamma, S \vdash e_n \rightsquigarrow \bullet, S_n \qquad \Gamma, S_n \vdash e_c \rightsquigarrow v, S_c \\ S_c\ \vdash\ \mathtt{children}(\bullet) = m, S_0 \\ S_i\ \vdash\ \mathtt{child}(\bullet, i) = \bullet_i, S'_i \\ S'_i\ \vdash\ \mathtt{detach}(\bullet_i) = \textit{nil}, S_{i+1} \end{array} \text{ où } 0 \leqslant i < m \qquad \textit{(evaluation du parcours)} \qquad (B, \bullet) :: (V, v) :: \Gamma, S'_m \vdash \mathsf{replace}\ \mathtt{B}\ \mathtt{V} \rightsquigarrow (), S'}{\Gamma, S \vdash \mathsf{replace}\ e_n\ e_c \rightsquigarrow (), S'}
$$

$$\text{(E-Replace-Traverse-Node)}\frac{S \vdash \texttt{bind}(\bullet, \bullet_c) = \mathit{nil}, S'}{(B, \bullet) :: (V, \bullet_c) :: \Gamma, S \vdash \texttt{replace B V} \rightsquigarrow (), S'}$$

$$\text{(E-Replace-Traverse-Imm)}\frac{}{(B, \bullet) :: (V, i \in \mathit{ImmVal}) :: \Gamma, S \vdash \texttt{replace B V} \rightsquigarrow (), S}$$

$$\text{(E-Replace-Traverse-Cstr)}\frac{\begin{array}{c} S \vdash \texttt{properties}(\circ) = \{\texttt{tag}, \texttt{val}\}, S \\ S \vdash \texttt{get}(\circ, \texttt{val}) = v_i, S \\ (B, \bullet) :: (V, v_i) :: \Gamma, S \vdash \texttt{replace B V} \rightsquigarrow (), S' \end{array}}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \texttt{replace B V} \rightsquigarrow (), S'}$$

$$\text{(E-Replace-Traverse-Tuple)}\frac{\begin{array}{c} S \vdash \texttt{properties}(\circ) = \{0, \cdots, n\}, S_0 \\ S_i \vdash \texttt{get}(\circ, n-i) = v, S_i \\ (B, \bullet) :: (V, v) :: \Gamma, S_i \vdash \texttt{replace B V} \rightsquigarrow (), S_{i+1} \end{array} \quad \text{où } 0 \leqslant i \leqslant n}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \texttt{replace B V} \rightsquigarrow (), S_{n+1}}$$

$$\text{(E-Replace-Traverse-Record)}\frac{\begin{array}{c} S \vdash \texttt{properties}(\circ) = \{l_0, \cdots, l_n\}, S_0 \\ S_i \vdash \texttt{get}(\circ, l_{n-i}) = v, S_i \\ (B, \bullet) :: (V, v) :: \Gamma, S_i \vdash \texttt{replace B V} \rightsquigarrow (), S_{i+1} \end{array} \quad \text{où } 0 \leqslant i \leqslant n}{(B, \bullet) :: (V, \circ) :: \Gamma, S \vdash \texttt{replace B V} \rightsquigarrow (), S_{n+1}}$$

$$\text{(E-Replace-Eval-F-1)}\frac{\Gamma, S \vdash e_n \rightsquigarrow v \notin H^\bullet, S'}{\Gamma, S \vdash \texttt{replace } e_n \ e_c \rightsquigarrow \bot, S'}$$

$$\text{(E-Replace-Eval-F-2)}\frac{\Gamma, S \vdash e_n \rightsquigarrow \bullet, S' \qquad \Gamma, S' \vdash e_c \rightsquigarrow \bot, S''}{\Gamma, S \vdash \texttt{replace } e_n \ e_c \rightsquigarrow \bot, S''}$$

13.3.2 Projection de la grammaire

La projection d'une expression rationnelle vers des types FidoML se fait récursivement, suivant sa structure (séquence, alternative, terminal ou application d'opérateur suffixe).

- **Séquences** Les séquences sont directement projetables vers des n-uplets. Éventuellement, si tous les champs sont nommés, on peut projeter vers un enregistrement (ici, on considère que l'ordre des champs est celui de la déclaration). Si on veut aussi un nom intelligible pour le type enregistrement, il faut de même que l'alternative soit nommée.
- **Alternatives** Afin d'être projetables dans le système de types de FidoML, toutes les alternatives présentes dans les expressions doivent être nommées, puisque la seule possibilité est de les projeter vers un type somme dont chaque cas représente une alternative. Si on veut aussi un nom intelligible pour le type somme, il faut de même que l'alternative soit nommée.
  Avec l'extension du système de types que nous avons évoqué section 11.3, dans laquelle un type de nœud peut être multi-étiqueté, il serait cependant possible de projeter les alternatives dont les cas sont seulement des types de nœuds vers un seul type.
- **Terminal** Un terminal associé à une définition de type de nœud t donne tout simplement un type <*t*> node. S'il s'agit d'une expression nommé, c'est le type FidoML du même nom.
- **Suffixes** ?*+ Ces opérations sont facilement projetées, respectivement, vers les types FidoML 'a option, 'a list et 'a * 'a list.

Les propriétés sont ajoutées avec le type de base FidoML correspondant. S'il s'agit de propriétés utilisant un type énuméré, on génère un type somme FidoML.

Conflits de noms Même si les sous-expressions sont toutes correctement nommées dans la grammaire, il se peut que les noms générés soient en conflit. Par exemple, dans notre grammaire de canevas, plusieurs types de nœuds ont un nœud enfant nommé center. Pour ceci, tous les champs, et tous les constructeurs sont préfixés par le nom du type auquel ils appartiennent. Une solution alternative, plus concise mais moins systématique, serait de vérifier la nécessité de ce préfixe. Il serait aussi possible de refuser de projeter une grammaire avec des conflits de noms, et laisser le programmeur lever les ambigüités.

Exemple Avec un tel algorithme, notre exemple de grammaire de canevas géométrique serait donc projetée vers le code FidoML qui suit.

```
type u_prop = Px | Pt | In

node type <dimen> unit
  prop value : int
  prop u : u_prop option
end

type point_content = {
  point_x : <dimen> node ;
  point_y : <dimen> node
}
node type <point> point_content end

type rotation_content = {
  center : <point> node ;
  angle : <dimen> node
}
node type <rotation> rotation_content end

type scale_content = {
  scale_center : <point> node ;
  scale_sx : <dimen> node ;
  scale_sy : <dimen> node
}
node type <scale> scale_content end

type translation_content = {
  translation_center : <point> node ;
  translation_dx : <dimen> node ;
  translation_dy : <dimen> node
}
node type <translation> translation_content end

type transform =
 | Tranform_rotation of <rotation> node
 | Tranform_scale of <scale> node
 | Tranform_translation of <translation> node
 | Tranform_transformgroup of <transformgroup> node
node type <transformgroup> transform list end

type rect_absolute_content = {
  rect_absolute_topleft : <point> node ;
  rect_absolute_bottomright : <point> node
}
type rect_relative_content = {
  rect_relative_center : <point> node ;
  rect_relative_width : <dimen> node
  rect_relative_height : <dimen> node
}
type rect_content =
  | Rect_absolute of rect_absolute_content
  | Rect_relative of rect_relative_content

node type <rect> rect_content end

type circle_content = {
  circle_center : <point> node ;
  circle_radius : <dimen> node
}
node type <circle> circle_content end

type shape =
 | Shape_rect of <rect> node
 | Shape_circle of <circle> node
 | Shape_group of <group> node

node type <group> (transform option * shape list)
  prop name : string option
end

node type <title> (<text> node) end
node type <meta> (<text> node)
  prop name : string
end
node type <head>
  (<title> node * <meta> node list)
end
node type <body> (shape list) end

node type <canvas> (<head> node * <body> node) end
```

Nommage automatique Pour le programmeur, il peut être fastidieux de nommer tous les cas. Heureusement, pour obtenir des noms intelligibles en pratique, il suffit de nommer certaines sous-expressions clefs, et d'appliquer un nommage automatique tel que l'algorithme basique $\mathcal{E}$ qui suit ().

$$\begin{array}{rcl} \mathcal{N}((e_0,\cdots,e_n)) & = & \mathcal{N}(e_0)\_\text{and}\_\cdots\_\text{and}\_\mathcal{N}(e_n) \\ \mathcal{N}((e_0 \mid \cdots \mid e_n)) & = & \mathcal{N}(e_0)\_\text{or}\_\cdots\_\text{or}\_\mathcal{N}(e_n) \end{array}$$

$$\mathcal{N}((e?)) = \mathcal{N}(e)\_\text{opt} \qquad \mathcal{N}((e+)) = \mathcal{N}(e)\text{s} \qquad \mathcal{N}((e*)) = \mathcal{N}(e)\text{s\_opt}$$

Concrètement, avec le système de types de FidoML (ou d'OCaml que nous présentons après), les

sous-expressions à nommer absolument pour que le nommage automatique soit raisonnable et non ambigu sont principalement les alternatives très imbriquées (par exemple une alternative, à l'intérieur d'une séquence, à l'intérieur d'une alternative). Il faut aussi bien entendu renommer d'éventuels noms contenant `_and_`, `_or_`, etc.

13.3.3 Correction

En utilisant le typage des nœuds de FidoML muni de l'extension que nous venons de présenter, ainsi que le mécanisme de copie de *cDOM*, nous sommes à même d'exprimer le résultat principal de cette partie, que nous évoquions au chapitre 7.

Théorème 13.3.1 *Un programme FidoML, s'il utilise les définitions de types de nœuds générées depuis une DTD par* `gram2fidoml`*, ne peut construire que des documents valides par rapport à cette DTD, et ces documents resteront valides quelles que soient les manipulations effectuées par le programme.*

Preuve La preuve se décompose en trois points. (1) Une construction de nœud bien typée produit un nœud valide. (2) Un nœud modifié par une opération replace reste valide. (3) Les enfants d'un nœud peuvent être modifiés uniquement par une opération replace sur celui-ci. Le dernier point s'obtient directement du mécanisme de copies implicites, les deux premiers par composition des deux lemmes suivants. □

Lemme 13.3.1 *Le type projeté pour chaque expression ne décrit que des valeurs dont le parcours en profondeur produit une suite de nœuds dont les étiquettes forment une séquence correcte pour l'expression.*

Preuve Par induction sur la structure des expressions rationnelles, puis pour chaque cas, par définition de la règle de projection associée. □

Lemme 13.3.2 (conservation) *La construction (resp. la modification par replace) d'un nœud, à partir d'une valeur bien typée par rapport à la définition de type de nœud associée, produit un nœud bien typé (resp. conserve le typage du nœud).*

Preuve Comme pour le lemme 12.6.3 original de conservation de FidoML, il faut définir la notion de *nœud bien typé*, en ajoutant la prise en compte de l'extension du système de types. On dit alors naturellement qu'un nœud est bien typé si on peut construire une valeur du type spécifié dans la définition de type de nœud dont le parcours en profondeur produit exactement la suite des enfants du nœud.

Comme pour le lemme 12.6.3, on sait que la valeur v utilisée pour la construction (resp. donnée à replace) est bien typée, en utilisant les propositions 9.2.1 et 11.1.2 pour appuyer l'induction.

En utilisant à nouveau la proposition 9.2.1, on peut alors dire que n'importe quelle valeur v' dans laquelle certains nœuds sont remplacés par leurs copies est bien typée, et du même type que v. On choisit alors la valeur v' dans laquelle (1) les nœuds déjà attachés sont remplacés par des copies et (2) les occurrences multiples dans v d'un nœud non attaché sont remplacées dans v' par des copies distinctes, à partir de la seconde.

En considérant s la suite des enfants d'un nœud à l'issue de sa construction (resp. de son affectation) avec la valeur v, et p la suite de nœuds apparaissant lors du parcours en profondeur de v', on observe que $s = p$. En effet, lors du parcours, les nœuds non attachés sont systématiquement copiés pour être ajoutés, et les nœuds non attachés apparaissant plusieurs fois le sont à partir de la deuxième occurrence. On a donc trouvé une valeur v', bien typée selon la définition du type de nœud, dont le parcours produit les enfants du nœud, qui est donc bien typé. □

13.4 Back-end OCaml

Dans cette section, nous montrons d'abord comment encoder une forme (un peu simplifiée) des constructions de nœuds de FidoML en OCaml. Puis nous donnons la projection de la grammaire vers cet encodage.

Projection des types de nœuds Afin d'encoder le typage des nœuds de FidoML dans celui d'OCaml, il faut pouvoir : (1) définir un type OCaml pour chaque type de nœud étiqueté de la grammaire (incompatible avec les autres), (2) un type générique, et la possibilité de transtyper un nœud de type étiqueté vers ce type, et (3) la possibilité de discriminer le type générique vers un type étiqueté. En OCaml, il y a trois encodages (raisonnables) possibles, chacun ayant ses avantages et inconvénients.

1. La première méthode est basée sur l'utilisation de types complètement abstraits. Il s'agit de définir un type abstrait node générique, ainsi qu'un type abstrait a_node et une fonction de coercition explicite de type a_node -> node pour chaque étiquette a.

   Le principal avantage de cette méthode est qu'elle utilise le système de types du cœur du langage, et est donc portable à d'autres variantes d'ML.

2. L'autre possibilité pour encoder la généricité des nœuds est naturellement d'encoder les nœuds étiquetés à l'aide des variants polymorphes d'OCaml.

   La technique est d'utiliser un type 'a node, contenant un variant polymorphe représentant l'étiquette et dont le type coïncide avec le paramètre 'a. Un nœud d'étiquette t sera alors de type [`$t$] node, et le type générique sera du type [`$t_0|\cdots|$`t_n] node, où les t_i sont toutes les étiquettes définies par la grammaire.

   Ainsi, c'est directement la coercition d'OCaml qui est utilisée pour passer d'un type spécifique à un type générique. De plus, on gagne un encodage un peu plus expressif, permettant d'encoder directement les alternatives simples, de façon similaire à l'extension proposée pour le système de types de FidoML.

   La troisième opération requise, spécialiser un type générique vers un type étiqueté, est possible grâce au filtrage par motifs sur les variants polymorphes d'OCaml, qui peut effectuer une spécialisation similaire à celle de FidoML.

3. La troisième possibilité est la même que la deuxième, mais dans laquelle les types sont abstraits pour le programmeur. Cette solution présente l'avantage de masquer la représentation, et permet même de limiter l'utilisation des variants au niveau des types (on parle de paramètres fantômes).

   En utilisant les annotations de variance d'OCaml (en spécifiant type +'a node), on peut faire en sorte que le passage d'un type spécifique à un type plus générique soit fait par la coercition d'OCaml.

   Cependant, le passage d'un type générique à un type étiqueté spécifique ne peut plus être fait par le filtrage d'OCaml. Si on veut conserver cette fonctionnalité, il convient de fournir une fonction de spécialisation par étiquette, effectuant une vérification dynamique en fonction de la représentation interne des étiquettes.

En utilisant la troisième possibilité, on peut commencer à projeter notre exemple vers la signature qui suit. On génère un type +'a node dont le paramètre doit être un sous-type du type variant polymorphe any_tag qui encode l'ensemble d'étiquettes. La fonction match_node effectue la spécialisation, elle prend un nœud générique, et une fonction par étiquette, applique la fonction correspondant à l'étiquette effective du nœud, et renvoie son résultat.

```
(* ensemble des étiquettes *)
type any_tag = [ `dimen | `point
               | `rotation | `scale | `translation | `transformgroup
               | `rect | `circle | `shape | `group
               | `title | `meta | `head | `body | `canvas ]

(* type abstrait *)
type +'a node constraint 'a = [< any_tag ]

(* spécialisation *)
val match_node :
   any_tag node
   -> ([`dimen] node -> 'a) -> ··· ([`canvas] node -> 'a)
   -> 'a
```

Projection des propriétés De façon générale, les propriétés peuvent être vues de deux façons différentes : (1) soit comme des valeurs de première classe associant une clef à une valeur, (2) soit, comme en FidoML, comme des entités intrinsèquement dépendantes d'un nœud et n'étant pas représentées par des valeurs.

La première vision présente un intérêt pour la création de documents de façon fonctionnelle, permettant de définir plusieurs nœuds ayant des propriétés définies par ailleurs. C'est par exemple le cas dans XHTML.M. Dans le cadre du document impératif, cette vision n'a pas beaucoup de sens. Il existe d'ailleurs des implantations de DOM utilisant une telle notion de valeur représentant un attribut, et pour assurer la cohérence du système, elles introduisent un mécanisme d'interdiction de partage dynamique similaire à celui des nœuds.

Nous choisissons donc la seconde, et pour l'implanter, chaque nom/type de propriété est donc projeté vers deux fonctions d'accès et d'affectation de cette propriété pour un nœud donné. Afin de ne permettre d'utiliser ces fonctions que sur les nœuds définissant ces propriétés, les étiquettes des nœuds pouvant être passés à ces fonctions sont celles des nœuds les définissant dans la grammaire. On définit éventuellement une fonction renvoyant un type option, pour les types de nœuds ayant une propriété optionnelle. On peut donc étendre la projection de notre exemple de grammaire aux propriétés comme suit.

```
val get_value : [ `dimen ] node -> int
val set_value : [ `dimen ] node -> int -> unit

val get_name : [ `meta ] node -> string
val set_name : [ `meta ] node -> string -> unit

val get_u_opt : [ `meta ] node -> [ `px | `pt | `in ] option
val set_u : [ `meta ] node -> [ `px | `pt | `in ] -> unit
```

Projection des enfants La projection vers OCaml est très proche de celle vers FidoML, à ceci près que les alternatives peuvent être implantées sans définition annexe de type somme grâce aux variants polymorphes. Pour simuler la structure de construction, on peut par exemple implanter une fonction de construction pour chaque type de nœud, qui devra prendre en paramètres les enfants et les valeurs initiales des propriétés. Il est alors possible d'utiliser le nommage de la grammaire pour spécifier plusieurs fonctions de construction, dans le cas d'une alternative, ou utiliser les arguments nommés (resp. optionnels) d'OCaml, dans le cas d'une séquence (resp. d'un opérateur ?). Nous pouvons alors donner la projection (d'un extrait) de notre exemple de grammaire.

```
val make_dimen : ?u:[ `px | `pt | `in ] -> value:int -> [> `dimen ] node
val make_point : x:[ `dimen ] node -> y:[ `dimen ] node -> [> `point ] node
val make_rotation : center:[ `point ] node -> angle:[ `dimen ] node -> [> `rotation ] node
(* ... *)
val make_tranform_from_rotation : [ `rotation ] node -> [> `transform ] node
val make_tranform_from_scale : [ `scale ] node -> [> `transform ] node
val make_tranform_from_translation : [ `translation ] node -> [> `transform ] node
val make_tranform_from_transformgroup : [ `transformgroup ] node -> [> `transform ] node
(* ... *)
val make_rect_absolute :
  topleft:[ `point ] node -> bottomright:[ `point ] node -> [> `rect ] node
val make_rect_relative :
  center:[ `point ] node -> height:[ `dimen ] node -> width:[ `dimen ] node -> [> `rect ] node
(* ... *)
val make_head : [ `title ] node -> [ `meta ] node list -> [> `head ] node
val make_body : [ `shape ] node list -> [> `body ] node
val make_canvas : [ `head ] node -> [ `body ] node -> [> `canvas ] node
```

Si on veut implanter le mécanisme de portée en OCaml, la meilleure solution est d'implanter une extension de syntaxe. Cependant, on peut ruser en demandant au programmeur de fournir, au lieu des valeurs des propriétés et des enfants, une fonction de calcul de ces derniers. Cette astuce force l'η-expansion par le programmeur, et la fonction peut alors être appelée explicitement par la bibliothèque entre l'ouverture et la fermeture de la portée du nœud.

Implantation Nous avons pu implanter et tester en pratique un prototype (pour lequel seule la projection vers OCaml est pour l'instant implantée) sur la grammaire de XHTML, et le procédé de nommage automatique, ainsi que quelques élagages et nommages d'alternatives imbriquées suffit à rendre la projection intelligible et utilisable. Cette expérience a exhibé aussi que la séparation de la grammaire extraite automatiquement et des annotations d'adaptation, si elle peut paraître un peu lourde, est clairement un bon choix. En effet, pour XHTML par exemple, de nombreuses versions sont publiées, et les annotations à faire peuvent souvent être réutilisées directement entre deux versions mineures.

13.5 Back-ends possibles

Au travers d'une DTD minimale, mais présentant les problématiques principales (alternatives imbriquées dans des séquences, nommage de sous-expressions, etc.), nous avons montré que cette technique de projection automatique vers le système de types d'un langage existant est viable pour FidoML et pour OCaml. Dans cette section, nous présentons d'autres cibles possibles et leurs intérêts.

Un back-end CDuce À première vue, on peut se demander quel pourrait être l'intérêt d'utiliser notre système de projection de grammaire vers un langage comme CDuce, dont le système de types est conçu spécifiquement pour gérer des grammaires rationnelles.

Nous avons aussi pu voir, au travers d'une DTD exemple que le nommage par le programmeur, s'il peut être contraignant et entrainer un coût de verbosité, peut aussi avoir de bons côtés, en clarifiant, voire documentant, les constructions des nœuds.

D'autre part, les outils existants pour extraire des types CDuce depuis les DTD utilisent la forme complètement expansée, dont nous avons vu qu'elle élimine tous les nommages intermédiaires.

Pour ces raisons, il pourrait être intéressant de dériver un outil d'extraction de grammaire DTD vers des types CDuce, qui bénéficierait de la reconstruction des nommages de la DTD, et qui permettrait de surcroît au programmeur d'ajouter des nommages intermédiaires supplémentaires, et d'élaguer les parties inutiles.

Un back-end Java ? Si nous avons vu que le mécanisme fonctionne pour ML, et qu'il peut même être utile dans un langage XML, il serait aussi intéressant de voir si il est possible de projeter une telle grammaire annotée vers un système de types radicalement différent, comme celui de Java.

Une approche évidente pour implanter des types de nœuds en Java est de considérer une classe de base Node, et d'en dériver une sous-classe par type de nœud. En effet, dans ce cas, la coercition entre un nœud générique et un nœud étiqueté est possible et utilise le système de types de Java.

Il reste alors à typer les enfants et les propriétés. Dans les deux cas, si on souhaite pouvoir utiliser le système de types de Java pour prendre en charge ces concepts, il ne reste alors que les interfaces.

Une possibilité pour uniformiser l'accès aux propriétés, et en même temps vérifier que le type donné à chaque propriété est partout le même, est simplement de définir une interface par propriété. En outre, l'équivalent de l'opération générique prop ? de FidoML, permettant de programmer certains traitements et parcours génériques, peut simplement être simulée avec un instanceof.

La spécification des expressions rationnelles spécifiant les séquences d'enfants est plus complexe. En effet, si, comme lors de la projection vers les fonctione en OCaml, on peut déplier les séquences à la racine des expressions comme arguments de méthodes, il n'est pas possible d'encoder les alternatives aussi facilement qu'avec les types somme ou variants polymorphes d'OCaml. Éventuellement, pour les alternatives à la racine, il serait possible d'encoder chaque cas dans un constructeur différent. Mais pour les alternatives imbriquées, la tâche est un peu plus complexe. Une astuce possible serait d'utiliser une interface par cas de l'alternative, implantée par chaque classe de cette dernière, permettant de les regrouper sous un seul type au sein des expressions. Pour cette dernière tâche, il est clair qu'une détection automatique des alternatives imbriquées, la génération automatique et le nommage sont utiles, sinon indispensables.

Nous ne poussons pas plus loin la présentation des possibilités concrètes pour encoder une grammaire DTD en Java qui est un peu hors de propos ; nous avons présenté suffisamment de points pour conclure que cette méthodologie de projection automatisée de grammaire pourrait s'étendre à un langage de ce type.

13.6 Conclusion sur la méthode et travaux futurs

La première conclusion est que cette approche pragmatique de projection d'une grammaire de document vers un système de types existant, même si elle engendre un peu de bruit et est moins expressive qu'un système de types spécialement conçu, est viable et correcte, et peut être effectuée efficacement grâce à l'automatisation. En particulier, la séparation de la grammaire, des annotations et de la projection maximise les tâches automatiques, et minimise le coût de maintenance et d'adaptation aux nouvelles versions de la grammaire.

D'autre part, si dans les langages typés à la ML, l'automatisation est un plus par rapport à l'existant, l'approche a aussi un intérêt dans les langages de recherche plus complexes, comme dans des langages plus répandus. Dans les langages à la CDuce, le fait de pouvoir partir d'une DTD existante et d'en obtenir des types intelligibles et personnalisés facilement semble un point intéressant pour la démocratisation de ces langages. Réciproquement, une telle approche pourrait amener un peu plus de typage dans le monde des langages à la Java, et amener un peu de sûreté de typage aux solutions existantes pour le Web.

Déconstruction de nœuds Nous avons principalement traité le cas de la construction de document dans les back-ends présentés dans ce chapitre, que ce soit pour FidoML ou OCaml. Cette restriction est correcte, car au final seule la construction doit être typée pour que le système soit bien typé. Il serait cependant intéressant d'étudier s'il est possible, de façon automatique, et sans modifier le système de types (1) d'ajouter à FidoML un filtrage en profondeur des nœuds, effectuant l'inverse de la construction,

à savoir reconstruire une valeur du langage à partir de la s'équence des enfants du nœud, (2) pour OCaml de générer des fonctions de déconstructions bien typées de la même façon.

Conversion de grammaires La plupart des outils disponibles pour convertir les grammaires entre les formats courants DTD, XMLSchema et Relax NG sont limités et peu sûrs. Le résultat est que beaucoup de grammaires trouvables sur le Web pour les formats de documents courants sont fausses. Par exemple, car converties sans vérification depuis un autre format de grammaire. Par exemple la grammaire Relax NG fournie par Microsoft à l'ECMA (*European Computer Manufacturers Association*) pour son format OpenXML (téléchargeable sur la page de la sépcification du format) est le résultat d'une conversion depuis la grammaire XMLSchema originale, et est fausse au point de ne même pas être syntaxiquement correcte. Clairement, il serait intéressant d'adopter une approche similaire à celle développée dans ce chapitre, en passant par une représentation intermédiaire, et permettant si besoin d'assister la conversion par des renommages ou simplifications, le tout de façon vérifiée automatiquement.

III Troisième partie
Vers un langage multi-tiers

14 Tour d'horizon des solutions existantes

Dans ce chapitre, nous donnons un aperçu étendu des problématiques de la programmation Web et d'une sélection de quatre solutions existantes issues du monde de la recherche : HOP, OPA, Links et Ocsigen.

Nous avons choisi de restreindre le nombre de solutions afin de clarifier le discours et éviter au mieux les répétitions. Nous avons alors fait en sorte au maximum que le choix soit suffisant pour couvrir l'éventail des solutions existantes aux différentes problématiques qui nous intéressent dans cette thèse. Et bien sûr, ces solutions ayant été pionnières dans le domaine, nous avons préféré les présenter plutôt que des solutions plus récentes en ré-utilisant les concepts [1].

Cette section est découpée en deux parties, donnant deux angles de vue différents.

Dans une première partie, nous donnons une vision à gros grain, articulée autour de ces solutions multi-tiers complètes, et des buts visés par celles-ci. Nous cherchons à mettre en valeur la diversité des approches et leurs points forts respectifs.

Dans une seconde partie, nous dressons une liste des principales problématiques de la programmation Web. Pour chacune, nous décrivons plus en détail les solutions techniques choisies par les langages multi-tiers. Pour les traits où ces solutions ne couvrent pas toutes les possibilités, nous donnerons aussi une sélection supplémentaire de solutions ne ciblant que le sous-problème concerné.

14.1 Présentation des solutions

Dans cette section, nous présentons l'architecture générale des quatre langages multi-tiers sélectionnés, en respectant au maximum la vision de leurs auteurs. En particulier, nous faisons volontairement ressortir les points forts de chaque approche. Pour chacune, nous donnons un schéma d'une installation typique, permettant de s'imprégner d'un coup d'œil de son architecture.

Deux des solutions sont actives : HOP [📖28], le langage de Manuel Serrano de l'INRIA basé sur Scheme et Ocsigen [📖2], la solution Web issue du laboratoire PPS (*Laboratoire Preuves Programmes et Systèmes*) basée sur OCaml, dans laquelle s'inscrit en partie cette thèse. Les deux autres sont récentes, mais leur état de maintenance est moins clair : OPA [⚓89], le langage développé par la start-up parisienne MLState, et Links [📖9], le langage développé par l'équipe de Philip Wadler à l'université d'Edimbourg.

Ces solutions ont toutes des visions du Web sensiblement différentes. Elles permettent de programmer des types d'applications variés, tant au niveau du matériel mis en jeu que des capacités logicielles intégrées ou interfacées. Techniquement, elles se basent sur des modèles de programmation et visent des publics de programmeurs différents. Cela implique des disparités en termes de taille des applications, de facilité et de souplesse du déploiement et de la maintenance.

14.1.1 HOP : *Programmation du Web diffus*

Le langage HOP est une solution pour programmer le *Web diffus*, terme désignant l'extension du Web à un (très) grand nombre de plates-formes de plus en plus variées et mobiles. De ce fait, HOP est la plus souple des solutions du point de vue du déploiement, de la portabilité et de l'adaptabilité.

1. Par exemple, bien que ces solutions soient proches des problématiques qui nous intéressent, nous ne présentons pas le langage Ur/Web [⚓84] ou la plate-forme Web basée sur Haskell Snap [⚓85], car leurs concepts de base se trouvaient déjà dans les solutions présentées.

Le courtier (*broker*) Pour arriver à exploiter au mieux les différentes cibles, HOP ajoute un processus tiers, le courtier, entre le serveur et le client. Celui-ci est une sorte de proxy, et dispose de plus de droits qu'un navigateur Web, permettant ainsi au client et au serveur d'accéder à des fonctionnalités supplémentaires de la machine. Dans le cadre classique du Web, le courtier permet d'effectuer des tâches comme l'agrégation depuis plusieurs sources Web, ou des communications de type socket. Dans le cadre du déploiement sur plates-formes mobiles, celui-ci permet par exemple aux applications HOP d'utiliser les informations sur l'état du matériel ou de géo-localisation.

Le courtier n'est pas strictement obligatoire pour déployer une application HOP dans le cas où elle peut se contenter des fonctionnalités du navigateur. On se retrouve alors avec une architecture client/serveur. S'il est présent, le courtier est en général situé sur la même machine que le client Web, mais pourrait l'être sur une machine intermédiaire. Pour certains cas d'applications, par exemple s'il s'agit juste d'exploiter les ressources de la machine locale via une interface Web, il est aussi possible de former une application avec uniquement un courtier et un client. La figure 14.1 donne plusieurs exemples de déploiement. Dans cette figure, les processus intervenant dans le déploiement sont représentés par des engrenages, et marqué d'un S pour un serveur, D pour une base de données et B pour un courtier.

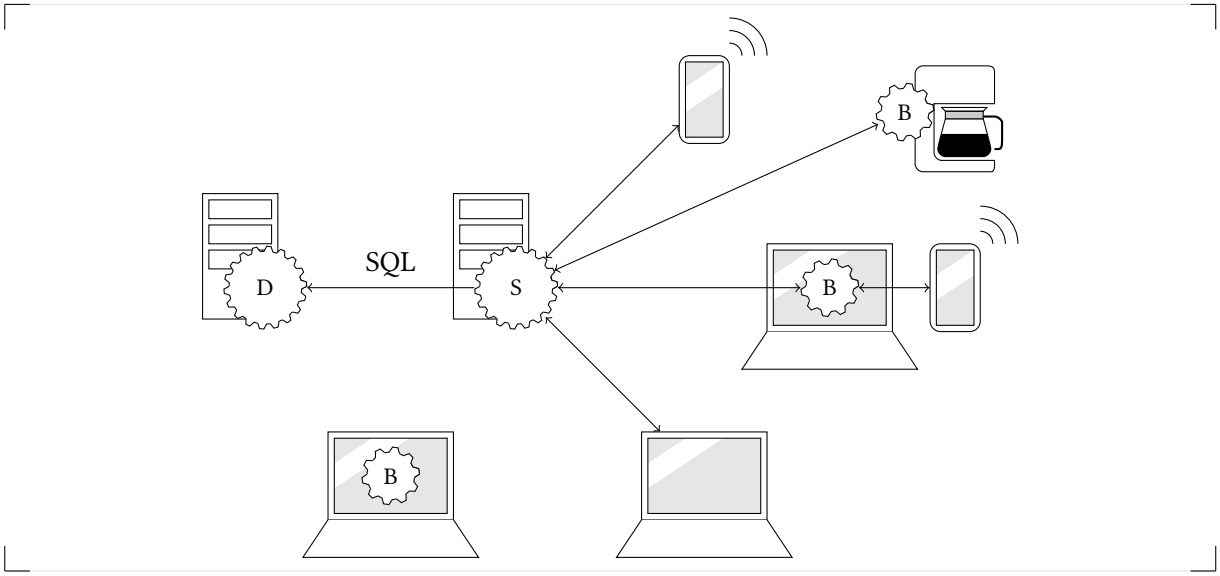


FIGURE 14.1: Architecture d'une installation HOP.

Applications visées Comme toutes les solutions présentées ici, HOP pourrait être utilisé pour implanter à peu près n'importe quel type d'application Web, mais il est spécifiquement conçu pour certains types d'applications.

Une première famille d'applications est la programmation d'interfaces graphiques via le navigateur. En effet, grâce au courtier, il est possible d'accéder aux ressources locales de la machine ou d'accéder à des serveurs distants, rendant possible la création d'applications comme des lecteurs de musique ou des clients de courrier électronique. Pour ces applications, HOP fournit une bibliothèque exhaustive de composants d'interface graphique, et reste compatible avec les technologies classiques HTML/CSS (*Cascading Style Sheet*).

Un autre but principal visé est la domotique. Le courtier de HOP est porté sur divers périphériques mobiles (Android, disques réseau, plates-formes robotiques, etc.), en permettant de gérer leurs capteurs et matériels de communication spécifiques. Chaque périphérique mobile peut alors informer le serveur de la maison de leur état local, et l'utilisateur peut voir ses informations sous forme agrégée via un navigateur, chez lui ou à distance.

14.1.2 OPA : *One Pot Application* [2] [3]

Le point de départ d'OPA est la difficulté à programmer le Web actuellement, de part le nombre de langages et logiciels à connaître, et les problèmes de bogues, de sécurité et de distribution que cela implique. La solution proposée est un langage unique dédié au Web, avec de bonnes propriétés en termes de sécurité, de déverminage, et de déploiement.

Un langage unique et dédié au Web En OPA, un seul langage sert à décrire les données persistantes de l'application, le code de la partie serveur et celui la partie client, et les l'inter-actions entre ces parties. Le tout est de plus intégré dans un système de types, permettant la vérification par le compilateur de la cohérence entre les parties de l'application.

Sécurité Le fait d'avoir conçu le langage pour les besoins spécifiques des applications Web intégrées a en particulier permis d'ajouter au compilateur des vérifications de sécurité. Par exemple, le compilateur exhibe les points d'entrée du code serveur généré, amenant à un accès aux données persistantes, et pas assez protégé par le programmeur. De telles vérifications auraient été beaucoup plus difficiles à implanter en utilisant un langage généraliste existant (et, à plus forte raison, plusieurs).

Déploiement facile L'intérêt principal pragmatique d'OPA par rapport aux autres solutions présentées est sa focalisation sur la facilité de déploiement, voire de distribution des applications Web produites. OPA est conçu pour être une plate-forme de développement d'applications Web *tout-en-un*, intégrant en un seul exécutable le programme serveur, la génération des programmes client, les éventuelles bibliothèques et la base de données, comme résumé par la figure 14.2.

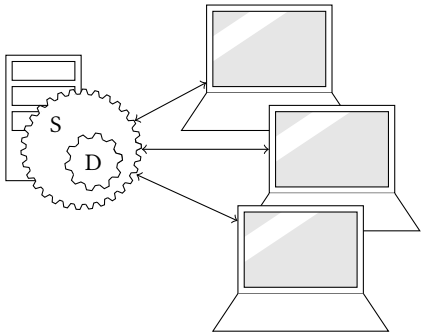


Figure 14.2: Architecture d'une installation OPA.

14.1.3 Links : *Web Programming Without Tier*

Le projet Links ne cherche à révolutionner le monde ni par sa forme ni par son architecture, ou en tout cas beaucoup moins que les autres projets présentés dans ce chapitre. Son but est plutôt d'assainir et d'homogénéiser le développement des applications Web, en réunissant au sein d'un même langage les briques habituelles : code serveur, code client et inter-action avec un moteur de bases de données relationnelle. La cible visée est donc le programmeur Web classique plutôt que le programmeur fonctionnel de haute formation universitaire. L'architecture d'un déploiement Links se rapproche d'un environnement Web classique, comme le montre la figure 14.3. Tristement, la page Web dont les activités récentes

2. La version que nous présentons ici est une pré-version temporairement distribuée au public courant 2010. Les traits présentés ici pourraient donc avoir évolué depuis. 3. Le mode de distribution d'OPA a été changé depuis la rédaction de la thèse, et une version Open-Source est maintenant disponible sous licence AGPL. Après un rapide regard de cette nouvelle version, il apparaît que les points techniques présentés dans ce chapitre restent valides (mêmes s'ils pourraient être précisés).

sont des projets *undergraduate* et le peu de publications liées semblent indiquer que Links n'a pas eu le succès escompté.

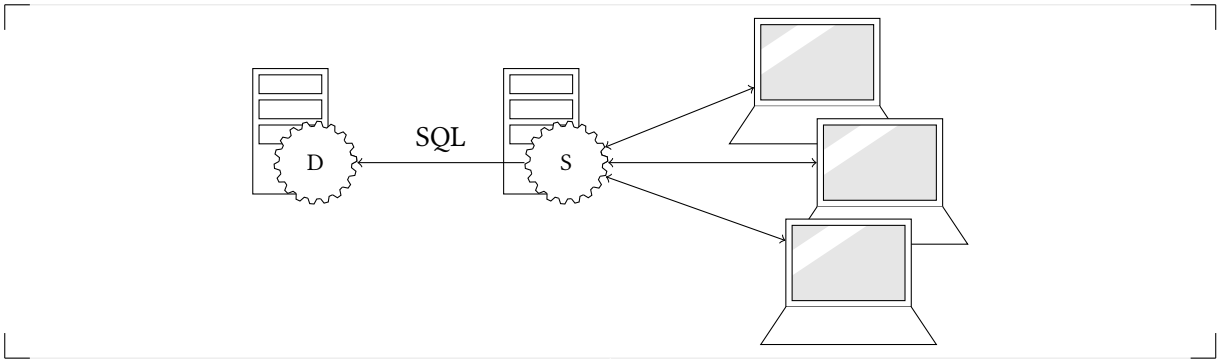


FIGURE 14.3: Architecture d'une installation Links.

Un langage pour le développeur Web Comme OPA, le projet Links a choisi de définir un nouveau langage, au lieu de se baser sur un environnement existant. Cependant, dans le but de viser le programmeur Web, le choix a été fait d'utiliser une syntaxe proche de JavaScript pour le code (client et serveur), et un DSL proche de SQL pour les accès aux données.

Avec de bonnes propriétés Derrière la syntaxe familière au développeur Web se trouve des fondations scientifiques fortes. Le modèle événementiel impératif de JavaScript est remplacé par un modèle de concurrence par passage de messages à la Erlang [📖66], et le cœur du langage est statiquement et fortement typé.

De même, si le système ne va pas jusqu'à vérifier statiquement l'intégrité des requêtes par rapport au schéma de base de données, le fait que celles-ci soient construites à partir d'un DSL interdit par construction les injections de code par ce biais et permet une optimisation automatique à la compilation, ce qui est un grand pas par rapport aux langages de script Web habituels.

Le programmeur fonctionnel typé pourrait regretter que le XML produit ne soit pas typé par rapport à une grammaire, et que le DOM du client et le XML produit par le serveur ne soient pas unifiés, mais les deux imposant un système de type et une sémantique avancés, c'est probablement un choix pragmatique raisonnable lorsque le but est de cibler le programmeur Web.

14.1.4 Ocsigen/Eliom : *Applications Web en OCaml*

Le projet Ocsigen est probablement le plus ambitieux de ceux présentés ici, étant donné (1) qu'il cherche à offrir un langage unique pour programmer le serveur, le client et l'accès aux données dans un même système de types statique, (2) qu'il vise la versatilité quant aux applications visées, et (3) le tout en se basant sur le vénérable langage OCaml, en permettant d'utiliser les bibliothèques existantes. Le projet est découpé en modules indépendants : Server, le serveur HTTP généraliste, Eliom, le module de développement de haut niveau, js_of_ocaml ou OBrowser, pour l'exécution de code client, et PG-OCaml ou Macaque, pour les requêtes de base de données.

De belles URL Dans les langages actuels du Web, l'association des documents aux URL est souvent gérée automatiquement, et généralement liée à des détails techniques d'implantation (système de fichiers, identifiants de base de données, etc.). Une innovation majeure d'Eliom est la définition d'une notion claire et incontournable de *service* pour associer les requêtes aux documents au sein d'une application Web, et surtout la classification simple et claire de ces services. Chaque classe de service correspond à un type d'action possible du client, et l'ensemble des besoins est couvert par les différentes classes.

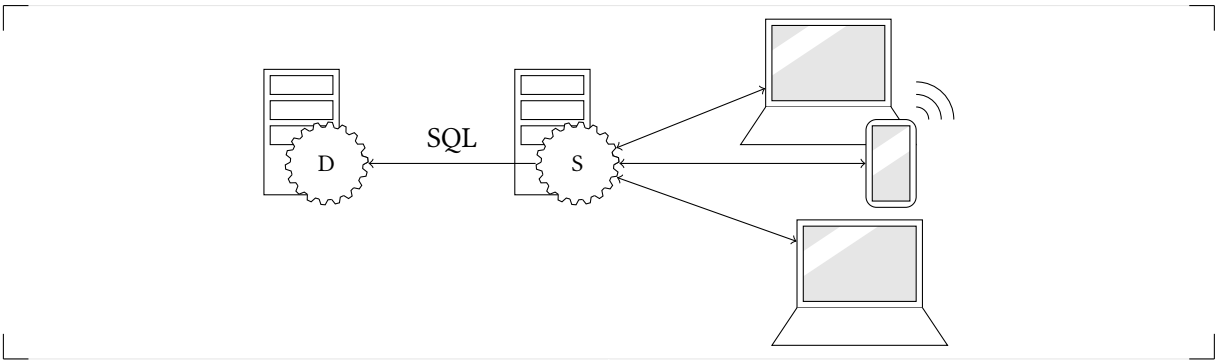


FIGURE 14.4: Architecture d'une installation Ocsigen.

Vers un Web 2.0 bien typé Ces dernières années ont vu une série d'expériences et de réflexions sur la programmation client au sein d'Ocsigen, de façon bien typée et cohérente avec la programmation serveur. Le résultat est la version 2.0 d'Ocsigen, incluant une version d'Eliom dédiée à la programmation du serveur et du client, avec le même modèle de concurrence et le même système de types. Le langage utilisé est toujours OCaml mais il est cependant nécessaire d'utiliser des extensions de syntaxe, en particulier car le code du serveur et du client peuvent se trouver dans un même fichier.

Concrètement, si la version 2.0 est compatible avec les versions précédentes et permet toujours de réaliser des sites Web *classiques*, elle introduit un nouveau cadre spécifiquement conçu pour la conception d'*applications Web* intégrées avec client riche, concept très en vogue actuellement bien que très difficile à programmer avec les outils existants. Dans le même esprit, Eliom 2.0 introduit les différentes technologies pour le *client riche* à la mode introduites par HTML 5.

Typage statique intégral Si la figure ressemble fortement à celle que nous avons donnée pour Links, la différence est qu'avec Ocsigen, toutes les flèches sont bien typées : (1) le langage de base, OCaml est typé statiquement et fortement, (2) le document XML transmis au client est bien typé vis-à-vis de la grammaire du document (dans les limites présentées dans la partie 2, en particulier au niveau des *id*s), (3) l'inter-action utilisateur au travers est bien typée grâce au mécanisme de services, (4) les communications dynamiques entre le serveur et le client sont bien typées statiquement, et des vérifications dynamiques sont effectuées pour assurer la sécurité du serveur, et (5) les accès à une base de données sont bien typés, en utilisant des extensions de syntaxe dédiées aux requêtes SQL.

14.2 Problématiques d'un langage pour Web

Nous avons présenté en première partie l'architecture générale des applications visées par les solutions, sans détailler concrètement leurs implantations et les langages de programmation associés. Dans cette section, nous allons donc détailler les solutions techniques pour chacune des problématiques importantes spécifiques à la programmation Web. Nous commençons par décrire les aspects programmatoires : modèles d'exécution, de concurrence, de communication, et de compilation multi-cibles. Puis, nous présentons les différents aspects fonctionnels : manipulations de documents, accès aux données, sécurité et passage à l'échelle.

14.2.1 Langages et modèles de navigation

Nous commençons donc, dans cette section, par présenter chacun de ces langages et leurs modèles d'exécution, en particulier au niveau des inter-action client/serveur.

HOP HOP est dérivé du langage Scheme, auquel il ajoute des constructions spécifiques. Le code serveur et le code client peuvent être partagés dans un même fichier.

Le code de plus haut niveau d'un fichier de programme HOP est destiné au serveur. Ce code peut déclarer de nouveaux points d'entrées à l'aide de la construction define-service. Chaque service a un nom, définissant l'URL par laquelle le client peut accéder, des paramètres, et un corps. Le corps est une expression Scheme, évaluée à chaque appel du service par un client pour générer une nouvelle page, ce que l'on appelle la phase d'initialisation.

L'expression de génération de page doit renvoyer un arbre de document, qui sera transmis au client sous forme d'XML. Pour ceci, HOP introduit des fonctions spécifiques à la création d'éléments de page, dont la composition ressemble volontairement à la notation XML. Au sein de cette expression, le programmeur peut insérer du code à destination du client, avec un opérateur d'échappement ~. Réciproquement, le programmeur peut utiliser au sein du code client le résultat d'expressions serveur avec l'opérateur d'échappement $.

Concrètement, les expressions serveur échappées dans le code client dont nous venons de parler sont calculées lors de la phase d'initialisation. Si le programmeur veut en changer la sémantique pour que l'évaluation soit faite à la demande, par exemple lors d'un évènement dans le navigateur, il doit utiliser l'appel explicite de service dans le code client. Pour ceci, il appelle la fonction with-hop, en lui passant en paramètre le service à appeler et la fonction de rappel à exécuter une fois que le serveur a répondu.

Une fois la page envoyée au client, plus rien ne s'exécute sur le serveur. Cependant, le programmeur peut, au cours de la génération de la page, définir d'autres services dynamiquement. Ces services sauvegardent le contexte de leur création, en particulier les données spécifiques à ce client. Ils permettront de reprendre l'exécution de code sur le serveur, en restaurant le contexte d'évaluation existant lors de leur création. C'est le modèle de programmation du Web par continuations [🕮46]. Des appels vers ces services définis dynamiquement peuvent bien sûr être insérés dans le code client (ou comme URL de liens de la page), et l'exécution sur le serveur peut alors être reprise à l'initiative du client suite à un évènement (ou à un clic sur un lien).

Concrètement, les services définis sont des continuations du langage. Le serveur HOP maintient pour cela une table de continuations appelables, qui est nettoyée régulièrement, en supprimant les services arrivés à expiration.

Dans ce modèle, seul le client a la main sur la reprise de l'exécution. Pour permettre aussi au serveur de relancer l'exécution pour un client donné, HOP définit un mécanisme d'évènements serveurs, auxquels le client peut réagir, de façon similaire à la façon de réagir aux évènements du navigateur.

Le comportement de l'évaluation de HOP, en particulier des différentes combinaisons de ces échappements et mécanismes de communication, n'est pas trivial. Manuel Serrano et al. donnent la spécification du modèle d'évaluation côté client comme côté serveur et des communications au sein d'une même sémantique dénotationnelle dans [🕮45].

Ocsigen/Eliom Ocsigen est basé sur le langage OCaml, auquel il ajoute un certain nombre de constructions spécifiques implantées sous forme de bibliothèques et d'extensions de syntaxe. Le code serveur et le code client peuvent être partagés dans un même fichier. Le modèle d'exécution d'Ocsigen est clairement très similaire à celui de HOP que nous venons de présenter, nous nous concentrons donc sur les différences notables de conception avec ce dernier.

La différence principale avec HOP est bien sûr que tous les concepts introduits sont typés statiquement, le document XML produit, les échappements client/serveur, les appels de services et les évènements serveur. Nous précisons ici les points liés à la navigation, et reviendrons sur les autres dans les sections spécifiques qui suivent.

Si le mécanisme de base de la navigation par services est le même, l'implantation dans Eliom est différente, reflétant le caractère typé de la solution.

- en HOP, le programmeur peut faire renvoyer différents types de données au client, et c'est la forme de l'appel depuis le client qui donne une sémantique à cet appel de service. Concrètement, un appel de service peut résulter d'un appel d'URL par l'utilisateur, ou d'une requête depuis le code client, avec dans ce dernier cas différentes utilisations possibles du résultat par le code client.
- Avec Eliom, la notion de service est scindée en différents types de services, correspondant aux différentes actions de navigation possibles. Cette vision est utile sur plusieurs points. (1) Tout d'abord, c'est une aide à la conception, puisque pour une action de navigation donnée, le programmeur n'a qu'à choisir le type de service à implanter. (2) Ce mécanisme permet de conserver de belles URL, en particulier, une notion de *co-service* est introduite, permettant d'effectuer une action sur le serveur depuis n'importe-quelle URL, et ce sans changer cette dernière. (3) Le principe de faire des services des valeurs de première classe, valable en Eliom comme en HOP, permet déjà d'éliminer la plupart des liens morts, en ne permettant de générer des liens qu'à partir de services existants. Cependant, il reste possible de générer des liens morts, à cause d'arguments mal-formés. Pour régler ces problèmes, d'une part, Eliom introduit le typage des paramètres empêchant la création de requêtes aux paramètres mal formés. (4) Une autre source de liens morts est l'appel direct d'URL par l'utilisateur, alors que le programme serveur n'est censé répondre à cette URL qu'en présence de paramètres *post*. C'est par exemple le cas si l'utilisateur a ajouté la réponse à un formulaire dans ses marque-pages. Pour ceci, Eliom impose au programmeur de spécifier un service de secours à appeler en cas de paramètres *post* manquants. (5) Grâce à tous ces avantages techniques, Eliom permet de programmer facilement et façon sûre une navigation complexe, et ce sans l'obligation d'exécuter du code côté client.

L'inter-action avec un client spécifique se fait, comme nous l'avons déjà expliqué, par le mécanisme de continuations.

- En HOP, c'est uniquement ce mécanisme qui est disponible. Éventuellement, le programmeur peut construire lui-même un mécanisme supplémentaire de personnalisation pour chaque client, par exemple en utilisant une base de données externes.
- Eliom, au contraire, introduit un mécanisme de sessions centralisé. Chaque client possède un identifiant de session, et le programmeur peut enregistrer dans la session les services et les données temporaires. D'une part, ce mécanisme permet de faciliter et uniformiser la programmation, et d'autre part, il permet une gestion automatique plus fine de la durée de vie des données et services temporaires.

Pour sa version 2.0, Eliom offre un modèle se concentrant sur le développement d'une *application Web* dans son ensemble, c'est à dire d'un ensemble de services cohérents entre eux. Cela se concrétise dans le langage par (1) la possibilité d'ajouter des directives destinées au client au plus haut niveau du fichier Eliom, qui seront partagées par toutes les pages de l'application côté client, et (2) la possibilité de changer de service depuis le programme client, sans recharger la page complètement. Ces mécanismes permettent de piloter la navigation Web côté client et, outre permettre de fluidifier le chargement des pages, offrent la possibilité (1) de conserver un état d'une page sur l'autre, utile pour implanter des tâches de fond à l'application, comme par exemple la lecture de musique, et ce (2) tout en conservant la navigation par services, permettant de conserver la gestion des URL et donc des marques pages, y compris si la navigation se fait côté client.

OPA Le langage OPA est un langage ne se basant pas sur un langage existant. Sa syntaxe s'inspire des langages fonctionnels typés à inférence, en imposant de plus un style monadique. C'est un langage statiquement typé, le système de types est principalement basé sur une notion d'enregistrements extensibles.

Le schéma d'évaluation est un peu moins évident que celui d'HOP/Ocsigen. Il n'y a pas de notion de service, c'est au programmeur d'associer les documents aux URL. Pour cela, il définit une fonction principale pour son serveur, prenant une URL non restreinte en paramètre, et devant renvoyer une

page. Ce choix peut être défendu par la relative souplesse qu'il donne au programmeur, et par le fait que l'appel de code serveur depuis le code client est complètement automatique et ne fait pas intervenir les URL.

Les effets proposés par le langage sont limités à la mutation de valeurs globales dans la base de données (cf. la section sur les données), ainsi qu'aux effets sur le document côté client. De fait, concrètement, il *semble* que le code calculatoire soit dupliqué entre le client et le serveur, et que seules les parties faisant intervenir la *base de données* sont remplacées dans le code client par des appels distants. On peut supposer que le schéma de compilation est aidé dans cette tâche par la structure monadique.

Le langage propose aussi un mécanisme de canaux d'évènements entre le serveur et le client pour permettre les communications bilatérales.

Links Le langage Links, comme nous l'avons présenté en introduction, a une syntaxe inspirée de JavaScript. Il un système de types statiques à la ML augmenté d'une forme de variants polymorphes.

Le mécanisme de programmation Web par continuations n'est pas implanté, comme dans HOP et Ocsigen, par la définition de services locaux côté serveur, mais par le paradigme send/suspend (introduit à l'origine par Krishnamurthi, en langage Scheme, dans le serveur Continue [🕮21]). Concrètement, la fonction prédéfinie sendSuspend prend une page en paramètre, et fige le calcul dans une continuation, qui est envoyée au client pour relancer le calcul sur le serveur à la demande.

Links ne dispose d'aucun mécanisme de description d'URL, le langage utilise les URL uniquement pour reconstruire la continuation mentionnée ci-dessus. Chaque fichier Links possède une partie servant à décrire la page principale de l'application, ensuite, les URL générées au sein d'une application Links sont toutes de la forme …/programme.links ?_cont=….

En plus du mécanisme send/suspend, le code client peut aussi faire directement des appels de fonctions distants, et réciproquement. Ici la granularité des communications est directement l'appel de fonction (en opposition à HOP/Ocsigen, où le langage dispose d'une granularité plus fine via des échappements, et à OPA ou le compilateur décide de la répartition du code). Au niveau du langage, cette fonctionnalité est implantée en marquant les fonctions avec les mots-clefs client et server. Si les fonctions contiennent des opérations uniquement disponibles d'un côté, elles doivent être marquées par le programmeur en accord. Par contre les fonctions n'utilisant que le cœur du langage peuvent ne pas porter de marque de localisation, auquel cas elles sont compilées vers les deux parties, et la version locale est utilisée lors d'un appel de fonction du langage. Concrètement, un appel de fonction distante ne peut être initié que depuis le client, puisque le serveur ne s'exécute plus une fois la page envoyée. Un tel appel est fait par une requête HTTP. Cependant, durant cet appel, le serveur peut faire appel à une fonction client. Dans ce cas, le serveur ne renvoie pas directement le résultat, mais renvoie à la place au client une demande d'exécution d'appel de fonction, ainsi que la continuation du calcul sur le serveur. Le client va alors exécuter cet appel, et transmettre le résultat par HTTP, avec la continuation. La réponse à cet appel sera alors celle de l'appel initial (ou éventuellement une autre demande, si le serveur a besoin d'effectuer d'autres rappels au client pour donner son résultat).

14.2.2 Compilation et déploiement

Maintenant que nous avons parlé du modèle d'exécution du langage, en particulier de la répartition du code entre le client et le serveur, intéressons-nous aux problématiques pratiques pour compiler et déployer ce code sur les différentes parties.

En HOP Le code du serveur/courtier est compilé à l'aide d'une version spécifique du compilateur Scheme (écrit en Scheme) Bigloo [🕮29] vers du code natif efficace (ou alternativement pour la portabilité, vers du code-octet pour la JVM). Pour la partie cliente, le système utilise judicieusement les facilités de meta-programmation du langage Scheme. Le code du client est compilé à la volée vers JavaScript, par

le code du serveur. Comme nous l'avons expliqué, les échappements serveur (ceux avec l'opérateur ~, et non le mot clef with-hop) sont calculés lors de la phase d'initialisation, et le résultat est inséré dans l'expression à compiler. Le code client d'une application HOP est donc spécialisé pour une exécution donnée.

Côté déploiement, HOP fournit un serveur générique. Le programmeur peut simplement lancer le serveur HOP avec le fichier programme de son application en paramètre. Bien sûr, le serveur est configurable via un fichier de configuration, et il est possible de paramétrer un serveur pour charger un ensemble de scripts spécifiques, ou pour charger automatiquement les scripts dans un répertoire, auquel cas le programmeur n'aura qu'à y déposer ses fichiers de script pour qu'ils soient pris en charge par le serveur.

En Ocsigen Le langage OCaml ne disposant pas de facilités de manipulation de code à l'exécution, le modèle adopté est la compilation statique du code client, en même temps que celle du serveur. Une alternative possible serait d'utiliser un outil comme MetaOCaml, mais cela nécessiterait pour le programmeur d'installer et utiliser un compilateur modifié, ce qui serait un frein au succès du projet, et nécessiterait des coûts de maintenance supplémentaires.

Concrètement, un préprocesseur effectue la phase de séparation, et génère deux fichiers OCaml. Une fois séparés, le code du client et du serveur sont compilés avec le compilateur ocamlc en utilisant la (même) bibliothèque standard OCaml, enrichie de bibliothèques spécifiques à chaque partie. Le schéma est en fait un peu plus compliqué : (1) Tout d'abord, le préprocesseur génère aussi un fichier intermédiaire interne, servant à vérifier la cohérence des types entre les deux codes, en utilisant le typeur standard OCaml. De plus, il ne fait pas que séparer le code, mais insère automatiquement du code nécessaire au fonctionnement, par exemple pour gérer l'encodage et le décodage des valeurs transmises. (2) Après le découpage du code, le code serveur doit être passé au travers des diverses extensions de syntaxe (pour le XML, les bases de données, etc. dont nous parlerons dans la suite). (3) Le code client, une fois compilé vers du code octet OCaml, est recompilé vers JavaScript avec l'outil js_of_ocaml. D'autre part, le code doit aussi passer par une extension de syntaxe spécifique pour l'inter-opérabilité avec JavaScript introduite par js_of_ocaml. Bien entendu, la distribution Ocsigen fournit des scripts de compilation pour effectuer toutes ces passes.

Une fois l'application compilée, son lancement se fait de façon similaire à celle présentée pour HOP, en plaçant les fichiers compilés à un endroit spécifié dans le fichier de configuration du serveur.

En OPA Le code client est pré-compilé en même temps que le code serveur, et, comme pour Ocsigen, la spécification se fait par passage de paramètres. Le compilateur est implanté en OCaml, il produit du code intermédiaire en langage OCaml. Le binaire produit est donc le produit de la compilation de ce code intermédiaire par le compilateur ocamlopt.

Pour implanter l'aspect tout-en-un, le code du client et le schéma de la base de données sont stockés dans le binaire du serveur sous forme de (chaînes de caractères) constantes.

Nous ne pouvons donner que peu d'informations sur le schéma de compilation d'OPA, le peu que nous décrivons l'étant à titre indicatif, puisque reposant sur la simple observation du code compilé, et non sur la spécification du schéma de compilation qui n'est pas publique.

En Links Dans links, comme en Ocsigen et OPA, le code serveur et le code client sont compilés à l'avance plutôt qu'à la volée. Cette compilation est faite par défaut au lancement du serveur, mais peut être optionnellement être faite à l'avance. Le compilateur Links est écrit en OCaml.

Dans links, le développement n'a clairement pas été centré sur les possibilités de déploiement. Links ne dispose pas de fichier de configuration aussi poussé que HOP ou Ocsigen, et une instance du serveur est dédiée à une seule application dont le programmeur donne le fichier Links principal sur la ligne de

commande. Un aspect très intéressant qui découle de ce fonctionnement simple est que Links dispose d'un interprète inter-actif (*toplevel*), utile pour l'expérimentation et la mise au point de programme.

HaXe Les solutions présentées requièrent obligatoirement la permission de lancer un logiciel serveur spécifique. En pratique, cela donne bien évidemment plus de liberté dans la conception du serveur, mais rend tout simplement impossible leur utilisation par le grand public à large échelle. En effet, les solutions d'hébergement à bas coût utilisent des serveurs mutualisés entre plusieurs clients, et pour assurer la sécurité, seul le serveur Web installé par l'hébergeur est utilisable, et ce soit avec les langages de script fournis, soit au mieux avec des programmes personnels devant répondre au standard CGI.

De ce point de vue, HaXe, langage riche et statiquement typé développé dans la petite entreprise Motion Twin et que nous avons déjà évoqué en introduction, est probablement la solution la plus souple. En effet, le langage peut être compilé vers les langages de scripts fortement déployés comme PHP, ainsi que des langages généralistes comme C++ ou Java permettant de créer des CGI. La solution n'est cependant pas parfaite, puisque la bibliothèque standard est dépendante de la cible (hormis la bibliothèque minimale commune), et qu'il faut par conséquent adopter les mécanismes de la plate-forme sous-jacente.

14.2.3 Modèle de concurrence, gestion d'évènements

Le fait que le langage soit le même sur toutes les parties est nécessaire, mais pas suffisant pour rendre homogène la programmation entre les parties. En particulier, le modèle de concurrence d'une application en code natif sur le serveur, et celui d'un script dans le navigateur sont très différents. Cette section présente comment les différentes solutions traitent cette problématique de la concurrence. Nous ne parlons pas d'OPA par manque d'informations.

En HOP Le modèle de concurrence reste proche des plates-formes sous-jacentes. D'une part car cela permet de conserver un schéma de compilation raisonnablement simple et efficace, et d'autre part simplement car ce n'est pas un point sur lequel les efforts de recherche ont été concentrés.

Concrètement, côté serveur, il est possible d'utiliser les threads POSIX (*Portable Operating System Interface*) si le programmeur veut exécuter du code en concurrence avec la boucle de réponse aux requêtes. Côté client, c'est le modèle événementiel de JavaScript qui est repris. Les gestionnaires d'évènements sont exécutés de façon séquentielle par la boucle de l'interprète JavaScript, et les appels au serveur via la syntaxe with-hop se font de façon asynchrone, en passant la fonction à rappeler une fois la réponse obtenue, de façon similaire à JavaScript. Si le mécanisme est le même, on peut toutefois argumenter que ce style de programmation à la CPS est clairement plus lisible en Scheme qu'en JavaScript.

En Ocsigen La concurrence est entièrement gérée par la bibliothèque de threads coopératifs Lwt [📖34]. Le choix a été fait historiquement pour le serveur original, car le modèle se prête bien à la programmation de ce type d'applications, en permettant de traiter les requêtes de nombreux clients sans avoir à supporter le coût des threads systèmes.

Lwt étant implanté non pas directement dans le langage, mais sous forme de bibliothèque de combinateurs de type monade, son utilisation dans une partie de l'application rend obligatoire son utilisation dans tout le reste du programme. En particulier, les bibliothèques effectuant des entrées/sorties, de longs calculs, ou utilisant les exceptions doivent être adaptées. De même, Eliom, le module de développement Web de haut niveau est écrit en style monadique, et le code écrit par le programmeur doit aussi l'être.

Si ce style CPS imposé n'est pas forcément au goût de tous, il peut être facilité par une extension de syntaxe ajoutant une structure syntaxique do de monades à la Haskell. De plus, puisque le langage ne dispose pas d'opérateurs de contrôle de type call/cc, les continuations implantant les services doivent de toutes façons être écrites en style CPS.

Côté serveur, c'est un choix bien adapté étant donné le déterminisme qu'il apporte, et que l'importance des entrées/sorties entraîne automatiquement des points de coopération en quantité suffisante pour que le programmeur ne se préoccupe pas de la granularité ou d'éventuels blocages.

Côté client, le modèle coopératif de Lwt s'implante naturellement au dessus de la boucle d'évènements de JavaScript, et l'écriture en CPS des points de coopération par le programmeur fait que le code est proche du JavaScript équivalent, et donc que le schéma de compilation de js_of_ocaml peut rester simple et efficace, sans nécessiter de transformation CPS automatique potentiellement coûteuse des parties non concurrentes.

En Links Links implante un mécanisme de concurrence homogène sur le client et le serveur, par passage de message, basé sur un calcul de processus typé. Le programmeur dispose des trois mots-clefs spawn,! et receive pour respectivement lancer un nouveau processus et récupérer sa valeur de retour (la structure prend une expression Links en paramètre qui est le corps de ce processus), envoyer et recevoir des messages avec un autre processus (avec possibilité de choix externe entre plusieurs processus pour la réception). Les auteurs de Links revendiquent leur inspiration par Erlang [📖66] et Mozart [📖26] dans la conception du mécanisme de concurrence.

Côté serveur, les threads système sont utilisés par l'interprète. Côté client, le mécanisme est implanté en utilisant la boucle d'évènements de JavaScript. Pour cela, une transformation automatique de programme en forme CPS est effectuée. Cette transformation utilise comme points de contrôle les primitives de concurrence spawn et receive, mais aussi les appels distants et des fonctions bloquantes prédéfinies comme sleep. Les auteurs indiquent cependant une limitation de la méthode de compilation, faisant que le mécanisme de concurrence ne fonctionne pas lors de l'exécution de rattrapeurs d'évènements, qui doivent alors soit être soit courts et n'utilisant pas la concurrence, soit se contenter de lancer un processus concurrent avec spawn, puis terminer [📖9] [4].

Programmation Réactive Fonctionnelle Les paradigmes de programmation réactive et programmation réactive fonctionnelle semblent une bonne abstraction pour la programmation événementielle du navigateur. Si nous n'avons pas connaissance d'un projet de langage réactif pour le Web, plusieurs expériences on été menées du côté de la programmation fonctionnelle réactive.

Au sein du projet Ocsigen, des tentatives ont été faites pour intégrer la programmation réactive fonctionnelle, via les bibliothèques Froc [⚓77] et React [⚓78]. Concrètement, l'approche n'a pas été retenue, en partie car les gains par rapport à Lwt ne sont pas clairs, étant donné le prix de l'introduction d'une nouvelle bibliothèque imposée au programmeur. De plus, l'implantation de ce modèle de programmation nécessite des références faibles, afin de libérer automatiquement les nœuds devenus inaccessibles par ailleurs dans le graphe de propagation d'événements. Or ce mécanisme n'est pas disponible en JavaScript, entraînant potentiellement des fuites mémoire. Pour les évènements spécifiques au DOM, Ocsigen propose une bibliothèque expérimentale de combinateurs de gestion d'évènements, utilisant les flèches (*arrows* [📖44]) de Hughes.

Le langage FlapJax [📖24] reprend le cœur du langage et la syntaxe de JavaScript, et remplace le mécanisme de boucle d'évènements par les notions de *flux d'évènements* (et de *comportements*) de la programmation réactive fonctionnelle [📖35]. Une partie très convaincante est la modification automatique du DOM et des CSS à partir de *comportements*. Le langage permet d'insérer dans une page HTML et dans la feuille de style CSS associée des échappements vers des *comportements*, et ces parties du document seront automatiquement modifiée à chaque occurrence du comportement. Les exemples donnés sur le site [⚓75] montrent que le langage est très pratique et convaincant pour décrire des comportements locaux, et interfacer les évènements prédéfinis du navigateur, et pour écrire de petits programmes à l'aspect réactif évident. Cependant, la lecture devient plutôt cryptique dès que les exemples deviennent

4. Ce problème est similaire à celui que nous avons décrit, qui empêche la concurrence préemptive lors de l'exécution de fonctions de rappel dans OBrowser au chapitre 4.

moins triviaux, comme par exemple le glisser/déposer. Il paraît alors difficile d'imaginer un programme dont toute la logique est écrite dans ce paradigme, interface graphique et communications avec le serveur comprises. On peut donc conclure sur le sujet en disant que, si le modèle est intéressant, il reste du travail avant de pouvoir l'utiliser dans une solution de programmation Web généraliste.

14.2.4 Création et manipulation de documents

Cette section présente les modèles de documents utilisés dans les différentes solutions et leurs limitations. Nous avons déjà rapidement présenté les solutions au chapitre 7 pour introduire la problématique du document. Dans cette section, nous reprenons cette présentation en ajoutant l'aspect client/serveur.

En HOP La possibilité d'utiliser des caractères non alphanumériques est utilisée dans HOP pour utiliser une syntaxe proche du XML. Par exemple, la fonction de création de liens hypertexte s'appelle tout simplement `<A>`. Le langage fournit ainsi des fonctions pour les différents éléments des grammaires HTML et SVG. Un intérêt majeur est que le programmeur peut virtuellement étendre l'ensemble des éléments, simplement en définissant ses propres fonctions de composition d'éléments existants, et en les nommant de la même façon.

La même syntaxe est utilisée côté client et côté serveur pour la création de nœuds. De plus HOP donne une implantation du DOM côté serveur, afin de fournir une API homogène. La bibliothèque standard de HOP comporte toutes les fonctions de manipulation de DOM classiques. Cependant, le DOM côté serveur n'a pas la même sémantique, et autorise le partage et les cycles. Pour corriger cela, une fonction normalize est utilisée en interne par le serveur afin de corriger la forme d'arbre. La documentation ne donne pas la spécification exacte, mais il est facile de voir dans le XML envoyé au client qu'un arbre contenant du partage est expansé par copie, ce qui peut par exemple amener à des duplications implicites d'attributs *id*.

En plus de proposer la même API des deux côtés, HOP permet, grâce au mécanisme d'échappements, d'utiliser de façon transparente les parties de document crées côté serveur, dans le code client, en particulier le corps des rattrappeurs d'évènements. Cette fonctionnalité a déjà été illustrée figure 7.5. Une restriction est que les nœuds utilisables côté client doivent faire partie du document envoyé. Le programmeur ne peut par exemple pas créer un nœud côté serveur sans l'attacher dans l'arbre, passer la référence vers ce nœud par un échappement client, et l'attacher plus tard côté client avec une opération DOM. Cette évaluation des primitives de création et manipulation a été spécifiée dans la sémantique opérationnelle [📖5] de HOP.

En Ocsigen Le module de typage de la grammaire XHTML.M est utilisable côté client. L'interface est la même, mais l'implantation construit directement des nœuds DOM plutôt qu'une représentation intermédiaire. Ainsi, on conserve la même syntaxe pour créer les nœuds du document des deux côtés. Cependant, l'implantation au dessus du DOM n'est pas correcte en cas d'introduction de partage, comme nous avons vu au chapitre 7, c'est donc une bonne aide au typage, mais le programmeur doit rester vigilant pour ne pas subir de déplacements implicites.

Du point de vue des modifications, l'API du DOM est rendue disponible au programmeur sur le client uniquement. Le programmeur peut transtyper explicitement une valeur de XHTML.M en un nœud non typé, sur lequel il peut utiliser les primitives de modification bas niveau. Contrairement à HOP, les primitives du DOM sur la représentation intermédiaire du serveur ne sont pas implantées.

Au niveau client/serveur, l'extension de syntaxe d'Eliom permet un mécanisme similaire à celui de HOP, permettant d'utiliser côté client les références vers les nœuds créés côté serveur, grâce au mécanisme d'échappements.

En OPA/Links Les solutions utilisées dans OPA et Links sont très proches. Pour la création de parties de document, OPA et Links permettent d'utiliser la même syntaxe côté client et côté serveur, en reprenant une syntaxe très proche du XML. La création côté serveur utilise une représentation intermédiaire permettant le partage, il est donc possible, comme avec HOP et Ocsigen, d'obtenir des *id*s dupliqués dans le XML envoyé au client.

Pour OPA, nous avons vu que l'unique source d'effets dans le code serveur est la modification de la base de données. Côté client, il est toujours possible d'altérer la base de données par appel distant, et il est aussi possible de modifier le DOM [5]. Le langage offre deux constructions de modification : l'ajout d'un enfant à un nœud, et le remplacement de tous les enfants d'un nœud. Links offre les mêmes manipulations, simplement elles sont données comme fonctions de la bibliothèque et non comme constructions du langage.

La différence majeure avec HOP et Ocsigen est que l'implantation sur le client de la création de nœuds n'utilise pas directement le DOM, mais une représentation intermédiaire comme sur le serveur. Ainsi, la création d'un sous-arbre DOM à partir d'une représentation intermédiaire est explicite et utilise une expansion comme sur le serveur, permettant de régler le problème du partage lors de la création de nœuds. D'autre part, dans les opérations de modification du DOM, le contenu ajouté est obligatoirement la représentation intermédiaire, et non un nœud DOM existant. Concrètement, en OPA, ceci est assuré car les nœuds DOM générés ne sont tout simplement pas utilisables comme valeurs du langage, et en Links, les nœuds DOM ont le type node, et les nœuds de la représentation intermédiaire le type xml. Avec cette restriction des modifications, les déplacements implicites ne peuvent tout simplement pas se produire, puisque les nœuds ajoutés dans l'arbre sont toujours issus d'une expansion fraîche.

Si le contenu ajouté à l'arbre est donné dans la représentation intermédiaire, la cible de la modification doit par contre être un nœud du DOM. Cependant, puisqu'une partie de document dans la représentation intermédiaire peut être expansée zéro ou plusieurs fois, il n'est pas possible de lui associer un nœud dans le DOM. Il faut donc que les langages permettent de sélectionner un nœud du DOM cible de la modification, sans pour autant pouvoir utiliser les références prises lors de la création.

Pour choisir ces nœuds du DOM, en OPA, la syntaxe de modification des nœuds utilise directement les ids comme partie gauche de l'affectation. Links propose dans la bibliothèque standard les principales primitives du DOM, permettant de parcourir l'arbre à bas niveau. Concrètement, cela fait peu de différence, par exemple, les exemples sur le site de Links utilisent tous la fonction getNodeById.

La solution n'est clairement pas satisfaisante en l'état pour plusieurs points. (1) Elle introduit la possibilité de duplications de *id*s côté client. (2) Cette duplication est d'autant plus problématique puisque la seule façon pour le programmeur de choisir le nœud à modifier est d'utiliser son *id*. (3) Le fait de ne sélectionner les nœuds que par leur ids rend quasiment impossible le typage des modifications en fonction du type de nœud. (4) Si le programmeur veut construire des composants réutilisables, il doit faire attention à générer manuellement des *id*s uniques pour chaque nœud DOM de chaque instance du composant, et faire attention à ne pas rentrer en collision avec le reste du programme.

Il apparaît donc que la solution d'utiliser une représentation intermédiaire n'est pas convaincante, puisque la seule façon d'y ajouter les modifications du document est de se reposer sur le mécanisme d'ids, qui introduit d'autres problèmes difficiles à résoudre. Pour Links, les auteurs assument n'avoir pas fait du traitement du document une priorité, étant donné que d'autres recherches existaient par ailleurs. Pour OPA, on peut espérer que la version finale corrigera ces problèmes.

14.2.5 Accès aux données

L'accès aux stockage de masse de données est un point primordial pour beaucoup de types d'applications Web. Historiquement, les solutions Web utilisent un SGBDR SQL, puisque ces systèmes sont conçus spécialement pour traiter de grandes masses de données, en offrant un point de vue transaction-

5. Il est aussi possible d'écrire des modifications de DOM dans le code serveur, acceptées par le compilateur mais qui génèrent une erreur à l'exécution.

nel pratique pour gérer automatiquement les mises-à-jour concurrentes. Cette section présente cette interface des langages Web avec les moteurs de bases de données.

En HOP Comme nous avons vu à la section précédente, HOP ne se veut clairement pas destiné à écrire des applications manipulant un grand nombre d'utilisateurs et une grande quantité de données. L'API de HOP côté serveur fournit seulement une interface simpliste au SGBDR SQLite [⚓100], utilisant des requêtes sous forme de chaînes. Cependant, HOP étant basé sur le langage Scheme, un programmeur souhaitant utiliser un SGBDR plus sérieusement pourrait utiliser une solution tierce. En particulier, on peut citer SchemeQL [🕮36], dans lequel les requêtes sont exprimées sous forme de S-expressions, introduisant la composabilité, et empêchant par construction les failles liées à la manipulation de requêtes sous forme de chaîne.

En Ocsigen Le projet Ocsigen utilisant le langage OCaml, il est possible d'utiliser les solutions existantes pour OCaml (sous réserve qu'elles se composent bien avec Lwt). Il existe principalement deux solutions, toutes les deux basées sur un modèle de requêtes explicites à un SGBDR SQL : PG OCaml [⚓99] et Macaque [🕮27].

PG OCaml est une interface spécifique à la base de données PostgreSQL. Elle utilise un mécanisme spécifique avancé de ce SGBDR, qui permet de vérifier qu'une requête correspond au schéma de la base de données. PG OCaml est une extension de syntaxe, qui ajout une couche d'inter-opérabilité entre les types d'OCaml et ceux de PostgreSQL, et effectue la vérification statique de toutes les requêtes. Avec cette technique, le typage est complètement sûr, et il est possible d'utiliser toutes l'expressivité du dialecte de PostgreSQL. Si du point de vue typage, on peut voir cela comme un avantage, du point de vue langage, c'est plutôt un inconvénient : le programmeur OCaml n'a pas forcément envie de programmer en langage SQL, et pourrait par exemple préférer manipuler des structures de données de haut niveau. De même, le programmeur fonctionnel préférerait, comme pour tout le reste du langage, se reposer sur une gestion automatique (et non basée sur des requêtes explicites) des accès et mises-à-jour.

Macaque, développé dans le cadre du projet Ocsigen, est une interface de haut niveau à un SGBDR compatible SQL sous forme de DSL. En réalité, Macaque est limité à PostgreSQL, mais le mécanisme général pourrait être porté à d'autres systèmes. L'idée générale, reprise sur Haskell/DB [🕮22], est d'ajouter un sous-langage de requêtes sous forme de compréhensions. Chaque forme de requête SQL supporté a été implantée manuellement et façon réfléchie, et ne ressemble pas forcément à la syntaxe SQL originale. L'interface est donc de plus haut niveau et largement plus plaisante au programmeur fonctionnel. Outre la syntaxe de plus haut niveau, le gros avantage par rapport à PG OCaml est que les requêtes sont paramétrables et composables. L'inconvénient qui en résulte cependant est que les requêtes ne peuvent pas être vérifiées statiquement par le mécanisme de PostgreSQL.

En OPA En tant que solution *tout-en-un*, OPA fournit son propre système de données persistantes. Il ne s'agit pas d'un SGBDR, mais plutôt un stockage persistant des données globales du programme. Le système prend en charge les types de base et les tables d'associations indexées par des entiers ou des chaînes. Comme nous l'avons expliqué dans la description du langage, c'est l'unique source d'effets globaux sur le serveur, et ceux-ci sont encapsulés dans une structure monadique.

En Links Links fournit une interface aux SGBDR, via un DSL très proche du dialecte SQL standard. Les requêtes peuvent être partiellement forgées à l'exécution, mais ne sont pas des valeurs de première classe. Le compilateur effectue des simplifications et optimisations automatiques de requêtes. Côté sûreté d'exécution, Links ne propose pas de typage statique des requêtes par rapport au schéma de base de données. Cependant, le fait d'utiliser un DSL reste une grande avancée par rapport à l'utilisation de requêtes dans des chaînes, du point de vue sécurité.

14.2.6 Passage à l'échelle, élasticité

Un aspect de plus en plus important à prendre en compte dans les solutions Web est le déploiement à large, voire très large échelle où les utilisateurs connectés simultanément se comptent en centaines de milliers. Les solutions présentées ici n'ont clairement pas mis l'accent principal sur cette problématique. Le projet HOP, par exemple, affiche explicitement sur son site que le serveur n'est pas conçu pour traiter un grand nombre d'utilisateurs et n'a pas été testé dans ce cadre. Cependant, de la conception même de chacune de ces solutions, on peut dégager des points intéressants, facilitant ce passage à l'échelle.

En Ocsigen Un point de conception intéressant est que le modèle de concurrence coopérative d'Ocsigen est intrinsèquement bon pour gérer un nombre important de connexions, sans exploser en ressources comme si chaque client déclenchait la création d'un thread ou processus système. Il permet aussi de maintenir une répartition de la charge, et donc un temps de réponse, équitable entre les clients, contrairement à une simple boucle de réponse aux requêtes. Bien sûr, ces arguments sont valables dans la mesure où la coopération est correctement effectuée par le programmeur et les bibliothèques.

Le gros point négatif pour le passage à large échelle du modèle à services de HOP et Eliom est le stockage de continuations. Le modèle peut cependant être utilisé à moyenne échelle, moyennant la limitation de la durée de vie de ces continuations. Pour ceci, Eliom définit dans son mécanisme centralisé de sessions l'enregistrement de services temporaires de session. Il permet aussi la définition de données de session, permettant de découpler les données des services, et donc de limiter le nombre de services créés. Eliom définit de plus plusieurs niveaux de persistance des données et services, et permet au programmeur de gérer finement les temps d'expiration des divers objets de sessions, et se charge de limiter leur durée de vie en conséquence.

D'autre part, lors de l'OCaml Meeting 2010 [⚓74], deux utilisateurs d'Ocsigen présentaient leur adaptation du modèle d'Ocsigen sur architectures distribuées. Dario Texeira présentait une implantation de distribution en Ocsigen, sous la forme d'un serveur Ocsigen primaire jouant le rôle de proxy, et redirigeant les clients vers plusieurs serveurs Ocsigen exécutant le code principal de l'application Web. Ce modèle est bien sûr à utiliser dans le cas d'un très grand nombre de clients, et est valide si les clients ne partagent que peu de données, où si l'accès aux données est centré sur un SGBDR déportée sur une machine tierce. À l'inverse, William Le Ferrand a présenté un modèle où le nombre de clients n'est pas le goulet d'étranglement, mais plutôt le temps de calcul nécessaire à chaque client. Sa plate-forme CoreFarm [⚓73] est un service Web de rendu d'images par lancer de rayons, et couple Ocsigen pour la partie Web et la gestion des travaux, et JoCaml pour la distribution élastique des calculs sur une architecture nuageuse (*cloud*).

Ces expériences de terrain montrent que le modèle est assez souple pour être adapté à certain types d'applications Web massives. Mais elles montrent surtout que l'utilisation d'un langage existant peut permettre, à moindre coût grâce à la ré-utilisation de composants existants, de s'adapter à différents modèles de déploiement et permettre au projet de toucher un plus grand nombre d'utilisateurs.

En OPA Les bibliothèques récentes de stockage de masse à haute élasticité sont basées sur le principe de dé-localisation du stockage, et sur une structure simpliste de type table de hachage. Il semble donc paradoxalement que les choix faits initialement de façon adaptée aux petites applications Web *tout-en-un* (1) d'une structure de la base de données simpliste, et (2) d'isolation des effets dans le langage grâce au style monadique, pourraient permettre au langage d'utiliser un stockage de masse à grande échelle, sans avoir à modifier le modèle du langage.

En Links Dans HOP et Ocsigen, le stockage des continuations implantant les services temporaires nécessaire prend de la place en mémoire sur le serveur. Même si des efforts sont faits pour optimiser la durée de vie de ces services, le principe même du mécanisme fait qu'il ne peut fonctionner à très large échelle sans adaptation.

Dans Links, nous avons vu que la navigation personnalisée pour le client est implantée avec la notion sensiblement différente du send/suspend. Ce mécanisme, s'il est un peu moins expressif, est plus facile à implanter de façon à ne pas prendre de place sur le serveur. Dans Links, l'implantation du send/suspend enregistre tout l'état du serveur nécessaire à la reprise, et le transmet au client. Cet état est alors retransmis au serveur lorsque le client demande la reprise de l'exécution.

D'autre part, le modèle de concurrence par passage de messages, s'il est implanté actuellement dans l'interprète du serveur avec des threads, pourrait se prêter à la distribution automatique.

14.2.7 Sécurité

La sécurité est un thème impossible à contourner pour une solution Web. Concrètement, il s'agit d'empêcher les individus malveillants d'utiliser les points d'entrée du serveur d'une façon non prévue, qui pourrait amener à la fuite ou la perte d'informations, ou à la surcharge ou au plantage du serveur. Nous avons déjà présenté dans l'introduction de la thèse les grands types de problèmes de sécurité : usurpation d'identité, injection de code serveur, injection de requêtes SQL, insertion de code client malveillant amené à être affiché sur les autres clients de l'application, etc. Suivant le type d'applications visées, les problèmes et solutions pratiques ne sont pas forcément les mêmes. Dans cette section, nous décrivons les principaux mécanismes de sécurité mis en place dans chacune des solutions.

En HOP La solution n'étant volontairement pas orientée vers la création d'applications massivement multi-utilisateurs et ouvertes au public, le traitement des injections de code client ou SQL n'est pas une priorité de conception. Par exemple, les fonctions SQL de la bibliothèque standard utilisent des chaînes, c'est donc au programmeur de faire d'éventuelles vérifications.

Par contre, étant donné qu'un des buts principaux visés est la domotique, la priorité est clairement mise sur la gestion d'utilisateurs et les contrôles d'accès. En effet, si nous prenons l'exemple d'une application permet de piloter sa maison à distance depuis le Web, il est peu probable que de nombreuses personnes s'y connectent, par contre, il est important que seules les personnes autorisées puissent y avoir accès, et éventuellement que chaque personne puisse avoir des droits différents.

Cette gestion des contrôles d'accès se fait via l'intermédiaire du fichier de configuration du serveur, qui demande à déclarer les utilisateurs autorisés. Elle est accessible ensuite par le programmeur via un module de la bibliothèque standard. Des efforts ont été faits pour valider le modèle, en particulier la sémantique opérationnelle [📖5] donne une formalisation des contrôles d'accès dans HOP.

En Ocsigen En Ocsigen, la sécurité est en majeure partie assurée par la compilation et le typage statique de l'ensemble de l'application. Pour le reste, c'est la bonne conception du modèle qui assure la sécurité.

- L'injection de code serveur est simplement impossible puisqu'il est compilé à l'avance.
- L'injection de code SQL est impossible, puisque les chaînes de requêtes sont cachées par une représentation abstraite, qui est de plus vérifiée par typage statique.
- Si le programmeur utilise le module Eliom pour les applications Web de la version 2.0, le code client envoyé par le serveur ne peut être qu'un appel de code client existant, pré-compilé, et les types des paramètres passés entre le serveur et le client sont bien typés statiquement, l'injection de code client n'est donc pas possible. Avec les versions précédentes, ou si le programmeur utilise les primitives bas-niveau de création ou manipulation du document, la tâche revient par contre au programmeur.
- Ocsigen est de par sa conception peu enclin à tomber face aux tentatives de surcharge. Le modèle de threads coopératifs du serveur permet de supporter un grand nombre de connexions, et donc de se protéger contre une bête attaque bas-niveau. Les mécanismes de limitations en nombre et en temps d'expiration des objets de session permettent de résister à une tentative de surcharge plus

intelligente, cherchant à saturer la mémoire ou à faire exploser le temps de réponse en saturant le nombre d'utilisateurs connectés à une application.
– Pour empêcher les tentatives de corruption du serveur, les types des paramètres des services, utiles pour la mise au point de l'application, sont systématiquement conservés à l'exécution pour vérifier toutes les requêtes, il est donc impossible de passer des paramètres mal formés, ou d'en passer trop ou pas assez. Il en va de même pour tous les types de communications entre les parties.

En OPA Pour les mêmes raisons de typage statique qu'Ocsigen, OPA n'est pas sensible aux problématiques d'injection de code serveur et client. Quant à l'injection de code de base de données, il n'a pas de sens puisque les accès à la base de données ne passent pas par un dialecte texte.

Du point de vue de la corruption de l'intégrité du serveur par forge de requêtes, étant donné que le découpage est fait de façon automatique, il n'est pas forcément évident de se rendre compte par lecture du programme des points d'entrée à protéger. Pour permettre au programmeur de protéger son application, le compilateur donne alors une liste des points d'entrée générés qui mènent à des accès directs à la base de données. C'est alors au programmeur d'insérer au bons endroits les éventuelles vérifications d'accès nécessaire.

En Links Comme nous l'avons dit, le serveur Links ne stocke pas en mémoire les informations de chaque client, puisqu'il enregistre son état dans les continuations passées au client. Ajouté au modèle de concurrence par passage de messages, Links peut résister aux attaques de surcharge par conception.

Il en va tout autrement de la corruption du serveur, ou de l'usurpation d'identité, étant donné que dans la version de Links actuelle, les continuations en provenance du client ne sont vérifiées que de manière minimale. Il est possible de modifier leurs arguments, en particulier en modifiant leur type, de récupérer les informations d'un autre utilisateur, simplement en modifiant son identifiant de base de données dans une continuation, étant donné que ce numéro (qui devrait rester privé sur le serveur) fait partie de la continuation et est donc transmis au client, etc.

Étant donne ce constat, Baltopoulos et Gordon ont proposé TinyLinks [📖3], une version sûre de Links basée sur la cryptographie, où les continuations sont signées par le serveur. Pour montrer la correction de leur approche, tout d'abord, ils sélectionnent un ensemble d'attaques et identifient les points faibles au niveau source. Ils étendent alors le langage avec des assertions que le programmeur peut insérer, par exemple, vérifier qu'un identifiant de base de données transmis par le client est le même que celui effectivement dans la base de données, définissent des exemples d'implantations des attaques à empêcher, et insèrent les assertions nécessaires. Ils proposent alors un système de types et d'effets permettant de vérifier la correction de ces assertions automatiquement au niveau source. Finalement, ils définissent un schéma de compilation utilisant la cryptographie pour signer les continuations, et montre que si les assertions sont vérifiées par leur système de types et d'effets, alors elles sont inviolables si le programme est utilisé avec leur schéma de compilation.

Il existe aussi une seconde déclinaison de Links, SELinks de Hicks et al., qui cherche à ajouter des notions de contrôle d'accès et de niveaux d'accréditation directement dans le langage. Les résultats sont intéressants, mais sortent du cadre des problèmes de sécurité intrinsèques au Web, le thème principal de ces travaux étant les politiques de contrôle d'accès, le Web n'en étant que le domaine d'application.

14.3 Conclusion

En première partie de ce chapitre, nous avons présenté une vue d'ensemble des solutions Web de recherche, et expliqué les buts visés par leurs auteurs et les spécificités principales. Dans la section précédente, nous avons dressé une liste des points importants dans la conception et l'implantation d'une solution Web. Pour chacun de ces points, nous avons pu décrire les solutions mises en place par chacune des solutions Web, et ainsi les comparer. Le tableau des pages 208 et 209 récapitule ces comparaisons.

	HOP
1. Généralités	
- Cible	mobilité, domotique
- Mode de distribution	open-source
- Développement	actif (INRIA)
2. Langage	
- Langage	Scheme
- Typage	dynamique
- Langage d'implantation	Scheme
2. Compilation et déploiement	
- Compilation serveur	code natif ou JVM
- Compilation client	JavaScript
- Séparation/génération du code	dynamique
- Granularité	échappements
3. Modèle de navigation et communication	
- Gestion d'URL	services
- Appels client → serveur	via service
- Appels serveur → client	✗
- Événements distants	✓
4. Concurrence et gestion d'événements	
- Modèle	boucles d'événements + préemptif
- Implantation	systèmes sous-jacents
6. Création et manipulations de documents	
- Typage	✗
- API de création	homogène
- API de manipulation	homogène (sémantique différente)
- Re-liaison de nœuds serveur → client	✓ (si nœud de page)
7. Accès aux données	
- Forme des requêtes	SQL textuelle
- Typage du schéma	✗
- Typage des requêtes	✗
- Optimisation	✗
8. Passage à l'échelle	
- Limitation de l'empreinte mémoire	✗
- Distribution	multi-serveur HOP possible
- Stockage élastique	bibliothèques
9. Sécurité	
- Prévention d'injection	✓ (sauf SQL)
- Requêtes non forgeables	✓ (typage dynamique)

OPA	Links	Ocsigen
applications tout-en-un bi-licence propriétaire et AGPL actif (MLState)	polyvalent open-source inactif (U. of Edimburgh)	polyvalent open-source actif (Paris 7/IRILL)
OPA statique OCaml	Links statique OCaml	OCaml + extensions statique OCaml
code OCaml JavaScript statique automatique	interprète JavaScript/CPS statique fonction	Code OCaml JavaScript statique échappements
manuelle automatique ✗ ✓	✗ appel de fonction appel de fonction ✗	services via un service ✗ ✓
- -	passage de messages transformation CPS	coopératif style CPS (Lwt)
✗ homogène client uniquement ✗	✗ homogène client uniquement ✗	création (sauf *ids*) homogène client uniquement ✓
intégrées au langage ✓ ✓ -	DSL à la SQL ✗ ✓ ✓	SQL/compréhensions ✓ (PG OCaml) ✓ Déléguée à PostgreSQL
- ✗ ✗ (design compatible)	par conception ✗ ✗	programmable possible (ex. JoCaml) bibliothèques
✓ ✓ (avertissements à la compil)	✓ ✗ (✓ avec TinyLinks)	✓ ✓ (vérifications à l'appel)

De ce tour d'horizon, il est impossible de conclure à un *classement* des langages. En effet, toutes ces solutions mettent en place des combinaisons différentes de solutions aux problématiques du Web, certains projets partageant les mêmes solutions pour certaines problématiques et d'autres très différentes sur les autres. Nous avons pu voir que les solutions à chaque problème, mais aussi la composition de ces solutions impliquent toutes des avantages et des inconvénients.

Mais ce tour d'horizon n'est pas pour autant exempt d'enseignement. La conclusion la plus évidente que nous pouvons tirer est qu'il reste du travail de recherche dans le domaine, de nouvelles solutions à chacune des thématiques à trouver, et des combinaisons à essayer. De plus, après avoir passé en revue chacune des problématiques, nous pouvons conclure en donnant les solutions les plus convaincantes, et compatibles avec toutes les problématiques de la programmation Web à une solution Web complète, à prendre comme modèle pour les futures solutions de programmation Web.

De HOP On retient la notion de courtier, et les possibilités qu'elle ajoute d'une part aux applications Web pour profiter des ressources matérielles, et d'autre part aux applications classiques pour utiliser le navigateur comme interface. Des travaux d'expérimentation seraient cependant à mener, pour voir si le courtier est vraiment préférable et pas trop limitatif par rapport à un greffon de navigateur.

D'autre part, il apparaît clairement que l'approche de l'utilisation d'un langage existant est possible. Bien sûr, la conclusion est à pondérer par la grande expressivité du langage Scheme, en particulier ses possibilités de méta-programmation sont très utiles ici, et ne sont pas une fonctionnalité courante des langages.

D'Ocsigen On retient clairement le mécanisme de services, en particulier nous avons vu qu'il fonctionne avec la navigation par liens comme avec la navigation dynamique par code client.

On peut aussi conclure qu'il est possible de trouver des solutions à la plupart des problématiques du Web avec un système de types de langage généraliste pour les rendre plus fiables, faciles à déboguer et sûres. Il faut bien sûr relativiser car celui d'OCaml est tout de même l'un des plus expressifs. Cependant, la nécessité d'utiliser des extensions de syntaxe, ainsi que les encodages complexes dans le système de types, permettent moins de conclure sur la validité de l'approche d'extension d'un langage que pour Scheme avec HOP. De façon plus précise, l'approche fonctionne bien en pratique, mais il semble difficile de viser un public qui n'est pas déjà familier avec des notions avancées de programmation et de typage.

D'OPA Il faut retenir le côté tout-en-un. La complexité de d'installation, configuration et déploiement est clairement un point faible des solutions comme HOP et Ocsigen, limitant le succès au niveau du grand public. Pour aller plus loin dans ce sens, il faut aussi retenir les possibilités de compilation vers les langages de scripts déjà largement déployés à la HaXe.

De Links Le point qui ressort par rapport aux autres solutions est clairement le modèle d'exécution du langage. Sur le plan théorique, le modèle de concurrence choisi est sain, expressif et adaptable. En pratique, il est bien intégré au langage, se comporte indifféremment sur le client et le serveur, et permet des appels distants de façon complètement transparente dans les deux sens. Le modèle est de plus bien adapté au passage à l'échelle, d'une part car il est implanté sans avoir à maintenir d'état sur le serveur, et d'autre part car il est intrinsèquement distribuable.

Le point négatif de Links est bien sûr l'exploitation inappropriée des URL. Il serait intéressant de travailler sur la compatibilité du modèle à services avec celui de Links.

15 Conclusion et perspectives

Dans l'introduction de cette thèse, nous regrettions que la notion de document inter-actif, que nous considérons comme centrale dans le Web, soit mal-menée dans les solutions de programmation Web répandues. De même, nous regrettions que les solutions de recherche ne fournissent pas de vision du document sûre et homogène entre le client et le serveur. En particulier, le point noir évoqué se situait au niveau des modifications du document, où toutes ne fournissent qu'une solution très proche de l'API bas niveau fournie par le navigateur.

Dans la partie I, nous avons décrit OBrowser, le résultat de nos travaux menés pour programmer le navigateur. La technique présentée permet de programmer de façon statiquement typée, avec l'ensemble du langage OCaml, y compris le modèle de concurrence préemptif. Nous avons aussi présenté un mécanisme d'inter-opérabilité permettant d'utiliser l'environnement du navigateur, en particulier le document, de façon intégrée avec la couche objet d'OCaml, du point de vue des types comme du modèle objet.

En pratique, OBrowser a servi de plate-forme expérimentale de développement client, dans le projet Ocsigen ainsi que dans d'autres expériences réalisées au cours de cette thèse. Mais, scientifiquement parlant, le résultat important est que la programmation du client peut tout à fait se faire avec des modèles de langage, de typage et de concurrence différents de ceux de JavaScript, et que la conception d'une solution de programmation Web de recherche peut, et même devrait, se faire sans obligatoirement se plier à ces choix. À la fin de cette expérience, nous avons cependant constaté que si OBrowser permettait de s'abstraire du langage et du système de types, il n'était pas suffisant pour s'abstraire du modèle de document, qui restait différent de celui utilisé sur le serveur et difficile à typer.

Dans la partie II, nous avons alors cherché à proposer un modèle de document utilisable pour la création comme la manipulation, et ce de façon bien typée. Nous avons commencé par formaliser une API minimale de création et manipulation de document impératif, similaire à celle des navigateurs, en particulier avec les mêmes déplacements implicites pour maintenir la structure d'arbre que nous souhaitions éliminer. Nous avons alors proposé une sémantique alternative, où les déplacements sont remplacés par des copies en profondeur. Mais le document contient des données annexes attachées à la structure d'arbre, comme les valeurs fonctionnelles servant à réagir aux événements, qu'il faut prendre en compte lors de la copie pour que celle-ci ait du sens. Afin de délimiter l'ensemble des valeurs à copier, nous avons alors introduit une notion de portée de nœud, la copie d'un nœuds entraînant, en plus de celle de ses enfants dans la structure d'arbre, celle de tous les objets sous sa portée.

Pour valider ce modèle de document, nous l'avons utilisé pour décrire l'évaluation d'un petit (mais non trivial) langage généraliste, muni de structures spécifiques de création et manipulation de document. Nous avons de plus défini un système de types pour le langage, afin de montrer que le modèle de document permet la création et les manipulations de document bien typées. Le langage présenté est basé sur ML, avec une structure syntaxique délimitée spécifique aux nœuds. La sémantique du langage fait correspondre la portée du nœud à la portée lexicale de la construction, de façon à ce que le programmeur puisse contrôler facilement la localité d'un objet à un nœud, simplement en plaçant sa définition à l'intérieur ou à l'extérieur de la structure syntaxique.

Tout au long de cette partie, nous avons cherché à faire en sorte que les résultats développés soient adaptables à d'autres langages et environnements. Nous avons défini le modèle du document comme une une API classique, nous avons cherché à définir un système de types pour les nœuds permettant

d'exprimer la plupart des traitements courants mais suffisamment simple pour être exprimable dans des langages existants et nous avons proposé une approche générique au typage de la grammaire spécifiquement conçue pour être adaptée à des langages existants.

Dans cette partie III, nous avons donné un tour d'horizon des langages pour le Web issus de la recherche. Pour cela, nous avons présenté chacune des solutions de façon générale, puis nous avons dressé une liste des problématiques principales des langages pour le Web, et, pour chacune, présenté la solution implantée par chacun des langages. Nous avons en particulier mis l'accent sur le traitement du document, et les implications sur les autres traits du langage.

Dans ce dernier chapitre, nous concluons l'approche développée dans cette thèse, en proposant des pistes pour la conception d'une solution de programmation Web typée, muni d'une notion de document sûre et homogène entre client et serveur, basée sur celle de FidoML. Le but n'est bien entendu pas de donner une spécification précise, mais de montrer en quoi les travaux présentés dans cette thèse ouvrent des perspectives de recherche en programmation Web. Pour ceci, nous présentons concrètement comment les modèles uniformes et polyvalents de langage et de document peuvent profiter aux différentes problématiques du Web évoquées au chapitre 14.

15.1 Vers un langage Web centré sur le document

Nous avons vu au chapitre 14 que parmi tous les langages, aucun n'offre un modèle d'exécution et de communications client/serveur parfaitement cohérent. Différentes méthodes doivent être employées (échappements, appels de services, canaux d'événements, etc.), suivant la phase de l'exécution et la partie initiant la communication. De plus, certaines solutions ne permettent pas les communications au cours de certaines phases, ou seulement de façon unidirectionnelle.

La raison principale à ces limitations est la même que celle qui rend le modèle de document non uniforme dans ces solutions. Leurs auteurs ont cherché à construire des plates-formes complètes en partant des technologies disponibles (et de plus, pour certains projets, en les intégrant les unes après les autres), et non à partir d'un modèle abstrait uniforme et complet. Évidemment, il y a de bonnes raisons derrière ce choix. En particulier, c'est en grande partie le fait de se baser sur des techniques existantes éprouvées, quitte à ne pas avoir un modèle parfaitement homogène, qui fait qu'HOP et Ocsigen sont des plates-formes aujourd'hui utilisables en production.

Dans cette section, nous présentons comment, en partant de notre modèle de document impératif, il est au contraire possible d'obtenir un modèle uniforme de programmation des différentes parties et des communications entre celles-ci. Pour ceci, nous commençons par étendre le modèle de document impératif ^{c}DOM pour prendre en compte des opérations de document à distance. Puis nous étendons FidoML pour l'adapter à une architecture distribuée, en y incorporant ces opérations distantes. Enfin, nous appliquons ce modèle général de langage généraliste distribué avec manipulations de document au cas particulier du Web.

Dans l'optique d'une implantation utilisant le langage OCaml et OBrowser, nous indiquons ponctuellement des possibilités ou limitations techniques dans ce cadre.

15.1.1 Extension distante de ^{c}DOM

La première étape nécessaire pour aller vers un langage Web centré autour de ^{c}DOM est d'étendre ce dernier pour l'adapter à une utilisation distante, en y ajoutant des fonctionnalités de manipulations à distance et de transmission de documents.

Manipulations distantes de document Notre spécification du document impératif a volontairement été pensée de façon à pouvoir être implantée de façon distante, via des technologies classiques de type

RPC (*Remote Procedure Call*). Concrètement, ^{c}DOM est défini comme un jeu de primitives, chacune pouvant être associée à une procédure distante. De plus, les différents types des entrées et sorties des primitives sont spécifiés précisément, facilitant leur implantation correcte à l'aide des types disponibles pris en charge par le mécanisme de RPC utilisé.

Transmission de documents Dans un modèle distribué où les parties savent toutes manipuler des documents, on voudrait, en plus du mécanisme de manipulations distantes, permettre de transmettre des documents, ou parties de documents. ^{c}DOM ne définit pas de primitive pour cela, mais le modèle est facile à étendre, puisqu'une telle opération se rapporte à une sérialisation, en utilisant les informations dynamiques de portée pour délimiter l'ensemble des objets à prendre en compte, comme pour la copie locale. Il suffit alors d'utiliser le mécanisme de RPC pour transmettre la valeur sérialisée, et à l'arrivée d'utiliser les primitives locales de ^{c}DOM pour reconstruire la structure de document à partir de la valeur sérialisée.

Il y a cependant une différence importante avec la copie locale. Lorsqu'un objet à copier contient un pointeur vers un objet qui n'est pas à copier, ce pointeur est conservé tel-quel lors d'une copie locale. Lors d'une copie distante, il convient que la copie contienne, à la place du pointeur original, une référence distante vers la cible de ce pointeur. L'implantation de ^{c}DOM doit alors être adaptée pour utiliser le mécanisme de RPC sous-jacent à la demande, afin que les références distantes présentes dans le document soient utilisées de façon identique aux pointeurs locaux.

Migration de documents Les deux fonctionnalités précédentes peuvent ne pas être satisfaisantes pour exprimer certains traitements. L'utilisation d'une succession d'appels distants peut s'avérer trop coûteuse, et la duplication liée à la copie non souhaitable. Une troisième alternative est alors la migration de document, permettant la transmission d'un document sans duplication.

En pratique, le mécanisme est très proche de la copie distante décrite précédemment, en ajoutant le fait que les pointeurs présents dans les objets non migrés, dont la cible est un des objets migrés doivent être transformés dynamiquement en références distantes. Par exemple, le pointeur vers la racine ayant servi à demander la migration doit devenir une référence distante vers la racine migrée après l'opération.

Gestion mémoire Un point impossible à ignorer est que ces mécanismes impliquent des problématiques non triviales de ramassage mémoire distribué. Nous ne donnons malheureusement pas de solution magique, mais pouvons tout de même citer plusieurs hypothèses simplificatrices dont nous disposons dans le contexte d'un FidoML client/serveur.

- Tout d'abord, si la gestion mémoire peut s'avérer critique sur le serveur, elle l'est clairement moins dans le navigateur. La conception de l'algorithme de ramassage peut donc être facilitée en se concentrant sur la correction et l'efficacité sur le serveur et la minimisation des communications, en se permettant pour cela des concessions, par exemple sur le délai avant ramassage ou le temps d'un cycle de collecte, dans le navigateur.
- Dans le même ordre d'idée, dans la plupart des applications Web, les données de chaque utilisateur doivent être locales à sa session, on peut donc concevoir un ramassage séparé pour les données inter-utilisateurs et les données locales à chaque utilisateur.
- Plus techniquement, la migration transparente de documents, et de façon similaire la généralisation aux objets du langage proposée à la section suivante, pourrait intervenir directement dans la conception du ramassage, en migrant automatiquement sur l'autre partie les objets devenus inaccessibles localement, participant à la réduction des cycles inter-parties et réduisant les communications lors des phases de ramassage suivantes.

15.1.2 Extension distribuée de FidoML

Avant de présenter comment étendre FidoML pour aller vers un langage spécifique au Web, commençons par étudier comment le langage peut être étendu à une utilisation sur architecture distribuée, en profitant des extensions distantes du modèle de document.

Migration de code Pour permettre l'utilisation d'une extension distribuée de ^{c}DOM dans FidoML, comme expliqué à la section précédente, il faut définir la migration des valeurs des types primitifs. Parmi celles-ci, les seules valeurs non triviales à migrer sont les expressions du langage, utilisées dans l'évaluation de FidoML pour représenter le corps des fonctions dans les fermetures. Une solution possible est de transmettre le code. Mais d'une part cette approche n'est pas systématiquement souhaitable pour des raisons de sécurité, et d'autre part, cela requiert des mécanismes non triviaux comme la métaprogrammation (à la HOP), ou la transmission dynamique de modules (à la Java RMI (*Remote Method Invocation*)). L'autre solution est la transmission de pointeurs de code uniquement (dans le cas d'une implantation de FidoML utilisant OCaml, c'est d'autre part la seule possibilité offerte par le langage).

Re-liaison Dans le cas de la transmission de pointeurs de code, il faut un mécanisme de re-liaison (*relink*), permettant de faire correspondre le pointeur de code de chaque fonction à celui de la même fonction sur la machine de destination. Pour qu'un tel mécanisme fonctionne sans erreur dynamique, il faut que tous les pointeurs envoyés correspondent à des fonctions présentes sur la machine de destination. Concrètement, cela signifie que les programmes doivent être identiques sur toutes les machines. Le mécanisme peut être rendu un peu plus souple via le chargement dynamique de modules, mais les éventuels modules à charger doivent tout de même être disponibles sur toutes les parties.

Une autre possibilité est de marquer les pointeurs de code par la machine sur laquelle ils ont été définis. Le langage se charge alors d'appeler directement le code si celui-ci provient de la machine locale, et d'effectuer de façon transparente un appel distant dans le cas contraire. Ce second mécanisme est plus intéressant dans notre cas, car il ajoute au langage une possibilité de communication entre les parties plus légère que les opérations distantes sur les documents, simplement par la transmission de fonctions de rappel au sein des valeurs transmises. Concrètement, pour implanter un mécanisme d'appels de fonctions FidoML distantes au dessus d'un mécanisme RPC simple, on peut ajouter sur chaque machine une fonction d'application générique sur chaque machine, et on sérialise l'environnement de fermeture et les arguments lors de l'appel.

Cependant, l'insertion systématique d'appels distants n'est pas adaptée à tous les problèmes, et introduit un sur-coût inutile dans le cas où la fonction migrée est aussi disponible sur la machine de destination, en particulier pour les fonctions de la bibliothèque standard. On voudrait alors pouvoir utiliser une approche hybride, utilisant automatiquement la re-liaison locale pour les fonctions communes, et les appels distants pour les autres. Éventuellement, dans le cas des fonctions communes, on voudrait aussi permettre au programmeur de choisir par un marquage explicite si la fonction doit s'exécuter sur la machine ou la fermeture a été crée, ou sur la machine ou la fermeture se trouve après d'éventuelles migrations.

Mécanisme d'informations dynamiques de portée généralisé Les informations dynamiques de portée ont été ajoutées pour permettre les copies implicites de nœuds en profondeur, et comme nous l'avons vu, elles sont utiles à la transmission de parties de documents, par migration ou par copie distante. Pour rappel, en FidoML, nous avons cherché à faire apparaître clairement au programmeur la portée d'un nœud, grâce à une structure syntaxique délimitée. L'idée était de rendre plus tangible et légitime la différence de comportement entre l'intérieur et l'extérieur de la structure au niveau du code source.

Il paraît alors naturel d'étendre ce mécanisme aux autres entités du langage décrites par des structures délimitées. Dans FidoML, on pourrait par exemple ajouter des informations dynamiques de portée

aux enregistrements, et ainsi pouvoir les copier en profondeur ou les transmettre. En OCaml, on voudrait de façon similaire ajouter des informations de portée aux objets et aux modules, afin d'obtenir automatiquement les opérations de clonage et de transmission.

D'autre part, les informations de portée pourraient être utilisées pour d'autres mécanismes du langage que la délimitation des objets à prendre en compte lors de la copie ou de la migration. Concrètement, le principe est d'adapter dynamiquement le comportement d'une opération du langage sur un objet, en fonction de la portée dans laquelle celui-ci se trouve. Pour donner un exemple, dans le mécanisme de re-liaison hybride proposé plus haut, plutôt que de demander au programmeur d'annoter chaque fonction, on pourrait imaginer que les fonctions définies dans le module dans lequel a été construit l'objet migré utilisent un appel distant, tandis que les fonctions externes au module (provenant par exemple de la bibliothèque standard) soient exécutées localement.

15.1.3 FidoML à la conquête du multi-tiers

Nous avons vu comment étendre FidoML/^{c}*DOM* pour ajouter des possibilités de communication par transmission de documents et d'objets du langage, en se basant sur un mécanisme de communication simple de type RPC. Voyons maintenant comment ce modèle peut être utilisé dans le cadre d'une architecture Web, et comment il pourrait être étendu pour traiter des traits avancés, tels que l'interface avec les SGBDR et le passage à l'échelle, sans compromettre l'homogénéité du langage.

Mécanisme RPC Depuis le début de cette section, nous parlons d'un mécanisme simple de type RPC. Mais dans le cadre d'un navigateur et d'un serveur Web communicant par le protocole HTTP, la disponibilité d'un tel mécanisme n'est pas en réalité acquise. Cependant, le projet Links en montre la faisabilité, en fournissant dans le langage les appels de fonctions distantes, dans les deux sens et potentiellement imbriqués.

Dans le but d'étudier la faisabilité d'implanter FidoML en utilisant OCaml, nous avons de notre côté réalisé une expérience utilisant OCaml sur le serveur et OBrowser sur le navigateur, et des deux côtés le mécanisme de threads pour permettre les appels distants bilatéraux, potentiellement imbriqués, et sans interrompre le fonctionnement du programme principal en concurrence. Concrètement, nous avons utilisé le mécanisme de sérialisation d'OCaml pour transmettre les environnements des fermetures et les paramètres, en ayant pris soin de l'instrumenter pour réécrire les fermetures locales afin de les transformer en appels distants, et réciproquement.

La couche bas niveau d'un mécanisme d'appels de fonctions distantes est donc possible à implanter sur une architecture Web classique, y compris en utilisant un langage existant.

Cycle de vie d'une page Les concepts généraux de communication que nous avons présentés n'imposent pas de restrictions précises sur le modèle d'exécution. Afin d'en évaluer au mieux les possibilités, nous nous plaçons dans le cadre d'une architecture donnant un maximum de latitude, à savoir le couple *client riche* (navigateur moderne avec JavaScript activé)/*serveur stateful* (conservant en mémoire des informations, continuations ou processus pour chaque session), similaire à celle des solutions de recherche HOP et Ocsigen. Le cycle de vie d'une page correspond alors au schéma suivant.

1. Lors de l'initialisation, le serveur construit le document correspondant. Dans ce document, il met en place des fonctions de rappel pour les événements. Celles-ci peuvent être des fonctions communes aux deux parties (bibliothèque standard, manipulations de document, etc), des fonctions spécifiques au serveur (accès à la base de données, etc.), ou éventuellement des fonctions spécifiques au client obtenues lors d'un échange précédent.
2. Le document est alors migré vers le client, et la re-liaison des fonctions effectuées. Côté client, les fonctions communes ou spécifiques au client s'exécuteront sur le client et manipuleront le document fraîchement construit. Les fonctions spécifiques au serveur sont transformées en appels

distants. Côté serveur, les références au document et aux objets migrés avec sont transformés en références distantes.

3. Les communications avec le serveur se font alors automatiquement lorsque le client utilise un appel distant généré lors de la migration. Les communications du serveur au client peuvent se faire de la même façon, ou par manipulations distantes du document.

Sessions En programmation Web, une session représente à la fois la séquence d'inter-actions avec un utilisateur et les données utilisées durant celle-ci. La plupart des solutions de développement Web implantent ces données de session sous la forme d'une table associative manipulée explicitement par le programmeur. Il semble alors naturel d'introduire au langage une construction syntaxique représentant une session, munie d'informations de portée. Du point de vue technique, cela pourrait permettre éventuellement la migration ou la duplication de session, servir au mécanisme de re-liaison ou aider la gestion la durée de vie des données de session. De plus, du point de vue du programmeur, notre argument est que la structuration syntaxique rend plus facile à appréhender la session, en délimitant concrètement ce qui s'y trouve de ce qui est global.

Services Nous avons parlé plus haut du modèle d'exécution pour une requête spécifique, mais pas du modèle de navigation global. Comme nous avons vu au chapitre précédent, le modèle à service à la Ocsigen est de notre point de vue le plus convaincant, sur le plan de l'absence de liens morts, du typage, et de la lisibilité des URL. Un mécanisme de services est a priori compatible avec le modèle présenté jusqu'ici, et il semble là encore naturel de l'intégrer au langage sous forme d'une structure syntaxique, munie d'informations de portée à l'exécution, et sensible à la portée dans laquelle elle est définie. Une intégration de la sorte permettrait probablement de rendre plus naturel le fait que la durée de vie et les possibilités de migration changent si le service est défini dans la portée d'une session ou non.

Répartition de charge (*load balancing*) Les techniques de manipulations et copies distantes et de migration d'objets du langage pourraient être utilisées pour permettre un passage à l'échelle, en particulier sur architectures nuageuses. L'idée est d'avoir un nombre de serveurs ajusté au nombre de connexions, et de migrer les sessions en cas d'ajout ou de suppression de serveur, en utilisant la structure syntaxique de session évoquée précédemment.

Dans ce cadre, la migration des objets permet de répartir dynamiquement la mémoire utilisée. Mais pour que le mécanisme soit utile en pratique, il faut aussi répartir la charge processeur de la même façon. Une approche possible serait la migration des threads lancés dans la portée de l'objet migré. Un tel mécanisme implique des difficultés de conception et d'implantation non triviales, mais des travaux existent sur le sujet. En particulier, la bibliothèque HirondML [🕮42] montre la faisabilité technique de la migration de threads en OCaml entre deux instances du même programme.

Accès aux données Les solutions Web que nous avons présentées au chapitre 14 permettent soit d'utiliser un SGBDR externe, en intégrant plus ou moins les requêtes, soit d'utiliser une forme intégrée mais très basique de base de données. Avec ce constat, on pourrait croire que l'intégration complète d'un SGBDR dans un langage est un problème insolvable. Pourtant, en 1996, Buneman et Ohori ont proposé un mini ML muni de primitives d'algèbre relationnel [🕮41]. Leur variante du langage permet de considérer les rangées des tables de la base de données sous une forme classique d'enregistrements extensibles, valeurs de première classe du langage. En particulier les références indirectes explicites sous forme d'indices dans les tables, classiques en bases de données SQL, sont masquées et vues comme des références classiques du langage. Ainsi, le programmeur n'a pas à manipuler ces références manuellement, et la durée de vie des rangées peut être prise en charge automatiquement comme pour le reste de la mémoire, éliminant toute une classe de bogues et de fuites.

Mais en pratique, il n'est pas judicieux de laisser gérer toutes les données du langage par la base de données, en particulier en cas d'utilisation d'un SGBDR externe. On aimerait par exemple que le programmeur puisse choisir quels enregistrement devront rester en mémoire et quels autres devront être stockés dans la base de données, pour ensuite les utiliser de façon indifférenciée. Mais si le système gère automatiquement les références, un tel mécanisme n'est pas évident à mettre en place, car il pourrait aboutir à l'existence de pointeurs de la base de données vers les données en mémoire. À titre d'exemple, un tel pointeur peut être introduit lors d'une affectation d'une donnée en mémoire à un champ d'une donnée en base de données. Ce phénomène est problématique principalement en cas d'arrêt du programme en présence de tels pointeurs, la base de données résultante contenant alors des pointeurs fantômes. D'autre part, ils introduisent des cycles rendant potentiellement plus complexe la gestion mémoire.

Nous pensons alors que l'utilisation d'un mécanisme d'informations dynamiques de portée peut constituer le point de départ d'une solution aux problèmes que nous venons d'évoquer. En particulier, elle permettrait de choisir automatiquement si les enregistrements doivent se trouver en mémoire ou en base de données suivant la portée dans laquelle ils sont construits, ainsi que de faciliter la copie ou la migration en profondeur des enregistrement en mémoire vers la base de données à la demande (et inversement) pour empêcher l'introduction de pointeurs arrières. Bien entendu, il s'agit d'une piste de recherche, et la mise au point d'une telle technique constituerait un travail important. Mais le résultat offrirait un mécanisme d'accès aux données du SGBDR souple et transparent, et reposant sur les mêmes concepts de base que les inter-actions entre les autres parties de l'application.

Modèles d'exécution alternatifs Si le modèle *client riche/serveur stateful* est intéressant du point de vue recherche, et permet au programmeur d'exprimer des comportements complexes, nous avons évoqué le fait au chapitre 14 qu'il est impossible à déployer chez les hébergeurs mutualisés, rendant les solutions qui se basent dessus inutilisables par le grand public. De façon symétrique, les projets prenant comme point de départ la facilité de déploiement ou le passage à l'échelle, comme HaXe ou Links, imposent un modèle d'exécution plus restrictif. Dans les deux cas, ces choix sont reflétés directement dans le langage, et le programmeur doit écrire son code en se pliant au modèle d'exécution choisi.

Nous pensons que le modèle uniforme de documents, couplé aux possibilités de migration, pourrait permettre de lever ce couplage entre le modèle d'exécution et l'architecture de déploiement. Dans l'idéal, le programmeur écrirait son programme avec le modèle décrit jusqu'ici, et celui-ci pourrait être déployé alternativement sur d'autres architectures Web. Voyons par exemple comment ce principe pourrait être appliqué aux deux architectures restreintes les plus répandues.

1. L'architecture la plus répandue actuellement est le couple client riche (les statistiques indiquent que les gens utilisent de plus en plus des navigateurs à jour, disposant des dernières technologies)/serveur stateless (la plupart des hébergeurs permettent uniquement l'usage de scripts du type PHP, ou de programmes CGI). De plus, il est fort probable que cette architecture continue à être majoritaire dans les années à venir, pour des raisons techniques de sécurité et de facilité d'administration.

   Dans ce cadre, une solution se rapportant en partie à celle de la solution de recherche Links, pourrait permettre de programmer de la même façon que pour un serveur stateful, en délégant une partie du travail au client riche. L'idée est de migrer la session de l'utilisateur sur le client à la fin de la phase d'initialisation, et de faire en sorte que les appels distants depuis le client migrent temporairement cette session sur le serveur durant l'appel. En plus de la migration de la session, le modèle de document uniforme pourrait permettre de plus de migrer les services associés à la session côté client, limitant ainsi les appels distants aux accès aux données, puisque la génération de page peut être faite côté client.
2. Une autre architecture qu'il serait intéressante de prendre en charge est le couple client léger/serveur stateful. Si pour la majorité des ordinateurs personnels, on peut compter sur la présence d'un na-

vigateur récent, il existe des situations où ce n'est pas le cas, et qu'on aimerait ne pas négliger. En particulier, on peut citer certains navigateurs spécifiquement conçus pour l'accessibilité, les restrictions d'entreprise limitant la prise en charge de JavaScript, les machine très légères, etc.

Dans ce cas, le modèle uniforme de document pourrait permettre de partager une large partie du code entre un programme pour client riche ou pour client léger. Le principe repose sur une solution symétrique au cadre précédent, c'est-à-dire migrer sur le serveur les traitements normalement effectués par un client riche. Concrètement, le serveur conserverait en mémoire l'état normalement stocké sur le client, en particulier le document. Ainsi, chaque action du client, au lieu d'effectuer un appel de fonction local, entraîne un appel distant effectuant le même traitement sur l'état conservé sur le serveur. L'idée est alors de renvoyer au client une sérialisation XML du document complet, compatible avec des navigateurs moins riches, lorsque celui-ci est modifié à la suite d'une action.

De façon plus générale, on peut entrevoir que l'uniformité du modèle de langage et de document, associés à un mécanisme de communications automatiques, pourraient permettre de programmer sans se préoccuper du déploiement, et de choisir a posteriori la répartition entre le client et le serveur.

Références

Bibliographie

[🕮1] Lennart Augustsson. **Compiling pattern matching**. Actes de *Functional Programming Languages and Computer Architecture (FPCA'85)*, vol. 201 de *Lecture Notes in Computer Science (LNCS)*, pages 368 à 381. Springer, 1985.

[🕮2] Vincent Balat. **Ocsigen : Typing Web interaction with Objective Caml**. Actes de *Workshop on ML (ML'06)*, pages 84 à 94. ACM, 2006.

[🕮3] Ioannis G. Baltopoulos et Andrew D. Gordon. **Secure compilation of a multi-tier web language**. Actes de *4th International Workshop on Types in Language Design and Implementation (TLDI'09)*, pages 27 à 38. ACM, 2009.

[🕮4] Véronique Benzaken, Giuseppe Castagna, et Alain Frisch. **CDuce : An XML-Centric General-Purpose Language**. Actes de *8th International Conference on Functional Programming (ICFP'03)*, pages 51 à 63. ACM, 2003.

[🕮5] Gérard Boudol, Zhengqin Luo, Tamara Rezk, et Manuel Serrano. **Towards reasoning for web applications : an operational semantics for Hop**. Actes de *Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA'10)*, pages 3 à 14. ACM, 2010.

[🕮6] Benjamin Canou, Vincent Balat, et Emmanuel Chailloux. **O'Browser : Objective Caml on browsers**. Actes de *Workshop on ML (ML'08)*, pages 69 à 78. ACM, 2008.

[🕮7] Emmanuel Chailloux et Grégoire Henry. **O'Jacaré, une interface objet entre Objective Caml et Java**. Actes de *Dixième conférence Langages et Modèles à Objets (LMO'04).*, vol. 10 de *L'Objet*, pages 75 à 88. Hermès, 2004.

[🕮8] Emmanuel Chailloux, Grégoire Henry, et Raphael Montelatici. **Mixing the Objective Caml and C# Programming Models in the .NET Framework**. Actes de *International Workshop on Multiparadigm Programming with Object Oriented Languages (MPOOL'04)*. 2004.

[🕮9] Ezra Cooper, Sam Lindley, Philip Wadler, et Jeremy Yallop. **Links : Web Programming Without Tiers**. Actes de *5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, pages 266 à 296. Springer-Verlag, 2006.

[🕮10] Stéphane Ducasse, Adrian Lienhard, et Lukas Renggli. **Seaside a Multiple Control Flow Web Application Framework**. Actes de *International Smalltalk Conference (ISC'04)*, Lecture Notes in Computer Science (LNCS), pages 231 à 257. Springer, 2004.

[🕮11] Martin Elsman et Niels Hallenberg. **Web Programming with SMLserver**. Actes de *5th International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, pages 74 à 91. Springer-Verlag, 2003.

[🕮12] Martin Elsman et Ken Friis Larsen. **Typing XHTML Web Applications in ML**. Actes de *6th International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 224 à 238. Springer-Verlag, 2004.

[🕮13] Fabrice Le Fessant et Luc Maranget. **Optimizing pattern matching**. Actes de *6th International Conference on Functional Programming (ICFP '01)*, pages 26 à 37. ACM, 2001.

[🕮14] Alain Frisch. **OCaml + XDuce**. Actes de *11th International Conference on Functional Programming (ICFP'06)*, pages 192 à 200. ACM, 2006.

[🕮15] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, et Michael Franz. **Trace-based just-in-time type specialization for dynamic languages**. Actes de *Conference on Programming Language Design and Implementation (PLDI'09)*, pages 465 à 478. ACM, 2009.

[🕮16] Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, et Uri D. Zarfaty. **Local Hoare reasoning about DOM**. Actes de *27th Symposium on Principles of Database Systems (PODS'08)*, pages 261 à 270. ACM, 2008.

[🕮17] Arjun Guha, Claudiu Saftoiu, et Shriram Krishnamurthi. **The Essence of JavaScript**. Theo D'Hondt, editeur, actes de *24th European Conference on Object-Oriented Programming (ECOOP'10)*, vol. 6183 de *Lecture Notes in Computer Science (LNCS)*, pages 126 à 50. Springer, 2010.

[🕮18] Phillip Heidegger, Annette Bieniusa, et Peter Thiemann. **DOM Transactions for Testing JavaScript**. Leonardo Bottaci et Gordon Fraser, editeurs, actes de *5th International Academic and Industrial Conference on Testing – Practice and Research Techniques (TAIC PART'10)*, vol. 6303 de *Lecture Notes in Computer Science (LNCS)*, pages 211 à 214. Springer, 2010.

[🕮19] David Herman et Cormac Flanagan. **Status report : specifying javascript with ML**. Actes de *Workshop on ML (ML'07)*, pages 47 à 52. ACM, 2007.

[🕮20] Sam T. Hochstadt et Matthias Felleisen. **The design and implementation of typed scheme**. Actes de *35th Symposium on the Principles of Programming Languages (POPL'08)*, pages 395 à 406. ACM, 2008.

[🕮21] Shriram Krishnamurthi. **The Continue Server**. Veronica Dahl et Philip Wadler, editeurs, actes de *5th Symposium on the Practical Aspects of Declarative Languages (PADL'03)*, vol. 2562 de *Lecture Notes in Computer Science (LNCS)*, pages 2 à 16. Springer, 2003.

[🕮22] Daan Leijen et Erik Meijer. **Domain specific embedded compilers**. Actes de *2nd Conference on Domain-Specific Languages (DSL '99)*, pages 109 à 122. ACM, 1999.

[🕮23] Sergio Maffeis, John C. Mitchell, et Ankur Taly. **An Operational Semantics for JavaScript**. G. Ramalingam, editeur, actes de *6th Asian Symposium Programming Languages And Systems (APLAS'08)*, vol. 5356 de *Lecture Notes in Computer Science (LNCS)*, pages 307 à 325. Springer, 2008.

[🕮24] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, et Shriram Krishnamurthi. **Flapjax : A Programming Language for AJAX Applications**. Actes de *24th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*, pages 1 à 20. ACM, 2009.

[🕮25] Manish Pokharel et Jong Sou Park. **Cloud computing : future solution for e-governance**. Actes de *3rd International Conference on Theory and Practice of Electronic Governance (ICEGOV '09)*, pages 409 à 410. ACM, 2009.

[🕮26] Peter Van Roy. **Convergence in Language Design : A Case of Lightning Striking Four Times in the Same Place**. Masami Hagiya et Philip Wadler, editeurs, actes de *9th Functional and Logic Programming Symposium (FLP'06)*, vol. 3945 de *Lecture Notes in Computer Science (LNCS)*, pages 2 à 12. Springer, 2006.

[🕮27] Gabriel Scherer et Jérôme Vouillon. **Macaque : interrogation sûre et flexible de base de données depuis Ocaml**. Actes de *Vingt et unièmes Journées Francophones des Langages Applicatifs (JFLA'10)*, pages 165 à 194. Studia Informatica, 2010.

[🕮28] Manuel Serrano, Erick Gallesio, et Florian Loitsch. **Hop, a language for programming the web 2.0**. Actes de *21st Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 975 à 985. ACM, 2006.

[🕮29] Manuel Serrano et Pierre Weis. **Bigloo : a portable and optimizing compiler for strict functional languages**. Alan Mycroft, editeur, actes de *2nd Static Analysis Symposium (SAS'95)*, vol. 983 de *Lecture Notes in Computer Science (LNCS)*, pages 366 à 381. Springer, 1995.

[🕮30] Basile Starynkevitch. **OCamlJIT : a faster Just-In-Time OCaml Implementation**. Actes de *1st MetaOCaml Workshop*. ACM, 2004.

[🕮31] Peter Thiemann. **A Type Safe DOM API**. Gavin Bierman et Christoph Koch, editeurs, actes de *10th International Symposium on Database Programming Languages (DBPL'05)*, vol. 3774 de *Lecture Notes in Computer Science (LNCS)*, pages 169 à 183. Springer, 2005.

[🕮32] David Ungar et Randall B. Smith. **Self : The Power of Simplicity**. Actes de *21st Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 227 à 242. ACM, 1987.

[🕮33] Benoît Vaugon, Philippe Wang, et Emmanuel Chailloux. **Les microcontrôleurs PIC programmés en Objective Caml**. Actes de *Vingt deuxièmes Journées Francophones des Langages Applicatifs (JFLA'11)*, pages 177 à 208. Studia Informatica, 2011.

[🕮34] Jérôme Vouillon. **Lwt : a cooperative thread library**. Actes de *Workshop on ML (ML'08)*, pages 3 à 12. ACM, 2008.

[🕮35] Zhanyong Wan et Paul Hudak. **Functional reactive programming from first principles**. Actes de *Conference on Programming Language Design and Implementation (PLDI'00)*, pages 242 à 252. ACM, 2000.

[🕮36] Noel Welsh, Francisco Solsona, et Ian Glover. **SchemeUnit and SchemeQL : Two Little Languages**. Actes de *3rd Workshop on Scheme and Functional Programming (SFP'02)*. Cambridge University Press, 2002.

[🕮37] Phillip Heidegger et Peter Thiemann. **Recency Types for Analyzing Scripting Languages**. Theo D'Hondt, editeur, *24th European Conference on Object-Oriented Programming (ECOOP'10)*, vol. 6183 de *Lecture Notes in Computer Science (LNCS)*, pages 200 à 224. Springer, 2010.

[🕮38] Xavier Leroy et François Rouaix. **Security properties of typed applets**. J. Vitek et C. Jensen, editeurs, *Secure Internet Programming - Security issues for Mobile and Distributed Objects*, vol. 1603 de *Lecture Notes in Computer Science (LNCS)*, pages 147 à 182. Springer-Verlag, 1999.

[🕮39] Peter Thiemann. **Towards a Type System for Analyzing JavaScript Programs**. Mooly Sagiv, editeur, *Programming Languages and Systems*, vol. 3444 de *Lecture Notes in Computer Science (LNCS)*, pages 140 à 140. Springer, 2005.

[🕮40] Frédéric Boussinot. **FairThreads : mixing cooperative and preemptive threads in C : Research Articles**. *Concurrency and Computation : Practice and Experience*, vol. 18, pages 445 à 469, 2006.

[🕮41] Peter Buneman et Atsushi Ohori. **Polymorphism and type inference in database programming**. *ACM Transactions on Database Systems (TODS)*, vol. 21, pages 30 à 76, 1996.

[🕮42] Emmanuel Chailloux, Vivien Ravet, et Julien Verlaguet. **HirondML : Fair Threads Migrations for Objective Caml**. *Parallel Processing Letters (PPL)*, vol. 18(1), pages 55 à 69, 2008.

[🕮43] Grégoire Henry, Michel Mauny, et Emmanuel Chailloux. **Typer la dé-sérialisation sans sérialiser les types**. *Technique et Science Informatiques*, vol. 26(9), pages 1067 à 1090, 2007.

[🕮44] John Hughes. **Generalising monads to arrows**. *Science of Computer Programming (SCP)*, vol. 37(1-3), pages 67 à 111, 2000.

[🕮45] Manuel Serrano et Christian Queinnec. **A simplified multi-tiers semantics for Hop**. *Higher Order Symbolic Computation (HOSC)*, pages 1 à 23, 2010.

[🕮46] Christian Queinnec. **Continuations and Web Servers**. *Higher-Order and Symbolic Computation (HOSC)*, vol. 17(4), pages 277 à 295, 2004.

[🕮47] Milner Robin. **A Theory of Type Polymorphism in Programming**. *Journal of Computer and System Sciences*, vol. 17(3), pages 348 à 375, 1978.

[🕮48] François Rouaix. **A Web navigator with applets in Caml**. *Computer Networks and ISDN Systems*, vol. 28(7-11), pages 1365 à 1371, 1996.

[🕮49] Didier Rémy et Jérôme Vouillon. **Objective ML : An effective object-oriented extension to ML**. *Theory And Practice of Object Systems*, vol. 4(1), pages 27 à 50, 1998.

[🕮50] Peter Thiemann. **A typed representation for HTML and XML documents in Haskell**. *Journal of Functional Programming*, vol. 12, pages 435 à 468, 2002.

[🕮51] David Ungar, Craig Chambers, Bay-Wei Chang, et Urs Hölzle. **Organizing programs without classes**. *LISP and Symbolic Computation*, vol. 4, pages 223 à 242, 1991.

[🕮52] Andrew K. Wright. **Simple imperative polymorphism**. *LISP and Symbolic Computation*, vol. 8, pages 343 à 355, 1995.

[🕮53] Jérôme Vouillon et Vincent Balat. **From Bytecode to Javascript : the Js of ocaml Compiler**, 2011. Disponible à l'adresse `http://www.pps.jussieu.fr/~balat/publications/vouillon_balat-js_of_ocaml.pdf`.

[🕮54] Mark Bergsma. **Wikimedia architecture**. Technical report, Wikimedia Foundation Inc., 2007.

[🕮55] Mathias Bourgoin, Benjamin Canou, Emmanuel Chailloux, Adrien Jonquet, et Philippe Wang. **OC4MC : Objective Caml for Multicore Architectures**. Technical report, International Symposium on Implementation and Application of Functional Languages (IFL'09), 2009.

[🕮56] Xavier Leroy. **The ZINC experiment : an economical implementation of the ML language**. Technical report 117, INRIA, 1990.

[🕮57] Xavier Leroy, Didier Rémy with Damien Doligez, Jacques Garrigue, et Jérôme Vouillon. **The Objective Caml system release 3.12.0 Documentation and user's manual**. Technical report, INRIA, 2010.

[🕮58] Benedikt Meurer. **OCamlJIT 2.0 - Faster Objective Caml**. Technical report, Universität-GH Siegen, 2010.

[🕮59] Emmanuel Chailloux, Pascal Manoury, et Bruno Pagano. **Développement d'applications avec Objective CAML**. O'Reilly France, 2000. version librement consultable à l'adresse `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/`.

[🕮60] David Flanagan. **JavaScript : The Definitive Guide**. O'Reilly & Associates, Inc., 1998.

[🕮61] ECMA (European Association for Standardizing Information et Communication Systems). **ECMA-357 : ECMAScript for XML (E4X) Specification**. ECMA, second edition, 2005.

[🕮62] ECMA (European Association for Standardizing Information et Communication Systems). **ECMA-262 : ECMAScript Language Specification**. ECMA, fifth edition, 2009.

[🕮63] Adele Goldberg et David Robson. **Smalltalk-80 : the language and its implementation**. Addison-Wesley Longman Publishing Co., Inc., 1983.

[🕮64] Benjamin C. Pierce. **Types and programming languages**. MIT Press, 2002.

[🕮65] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, et Jacob Matthews. **Revised [6] Report on the Algorithmic Language Scheme**. Cambridge University Press, 2010.

[🕮66] Robert Virding, Claes Wikström, et Mike Williams. **Concurrent programming in ERLANG (2nd ed.)**. Prentice Hall International (UK) Ltd., 1996.

[🕮67] Pierre Weis et Xavien Leroy. **Le langage Caml**. Dunod, 1999. version plus récente et librement téléchargeable à l'adresse `http://caml.inria.fr/pub/distrib/books/llc.pdf`.

Références Web

[⚓68] Recommandation DOM du W3C
http ://www.w3.org/DOM/

[⚓69] Validateur du W3C
http ://validator.w3.org/

[⚓70] Page d'OBrowser
http ://www.ocsigen.org/obrowser/

[⚓71] Site du projet Ocsigen
http ://www.ocsigen.org/

[⚓72] Google Closure, bibliothèque de composants JavaScript
http ://code.google.com/closure/library/

[⚓73] CoreFarm.org, architecture de calcul distribué en Ocsigen + JoCaml
http ://www.corefarm.org/

[⚓74] OCaml Meeting 2010, programme et liens vers les présentations
https ://forge.ocamlcore.org/forum/forum.php ?forum_id=584

[⚓75] FlapJax, Programmation fonctionnelle réactive en JavaScript
http ://www.flapjax-lang.org

[⚓76] Prototype, simulation du modèle à classes en JavaScript
http ://www.prototypejs.org/

[⚓77] FROC, bibliothèque de programmation réactive fonctionnelle en OCaml
http ://code.google.com/p/froc/

[⚓78] React, bibliothèque de programmation réactive fonctionnelle en OCaml
https ://github.com/camlunity/ocaml-react

[⚓79] Typage XHTML dans Ocsigen avec XHTML.M
http ://ocsigen.org/tyxml/dev/manual/

[⚓80] GWT, suite de développement Web en Java de Google
http ://code.google.com/webtoolkit/

[⚓81] Page personnelle de Benjamin Canou
http ://www.pps.jussieu.fr/~canou/

[⚓82] Moteur de navigateur WebKit
http ://www.webkit.org/

[⚓83] Django, environnement de développement Web en python
http ://www.djangoproject.com/

[⚓84] Ur/Web, un langage fonctionnel pur pour le Web
http ://www.impredicative.com/ur/

[⚓85] Snap, une plate-forme Web basée sur Haskell
http ://snapframework.com/

[⚓86] Symphony, environnement complet pour PHP
http ://www.symfony-project.org/

[⚓87] Cool URIs don't change
http ://www.w3.org/Provider/Style/URI

[⚓88] Lambdoc, librairie de manipulation de documents en OCaml
https ://forge.ocamlcore.org/projects/lambdoc/

[⚓89] OPA (One Pot Application), langage Web propriétaire
http ://www.mlstate.fr/

[⚓90] Article du blog de Brendan Eich sur JavaScript, 3 Avril 2008
http ://weblogs.mozillazine.org/roadmap/archives/2008/04/popularity.html

[⚓91] Interview de Brendan Eich par ComputerWorld, 31 Juillet 2008
http ://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/ ?fp=4194304

[⚓92] Comparatif des navigateurs par Ars Technica, Juin 2010
http ://arst.ch/l6o

[⚓93] Comparatif en temps réel des moteurs JavaScript par l'équipe Mozilla
http ://arewefastyet.com/

[⚓94] Comparatif des navigateurs par Tom's Hardware, Juillet 2010
http ://www.presence-pc.com/tests/comparatif-navigateurs-23306/

[⚓95] Site récapitulatif des compatibilités HTML 5 entre navigateurs
http ://caniuse.com/

[⚓96] Une histoire de Caml
http ://caml.inria.fr/about/history.fr.html

[⚓97] Ocamljs, le compilateur d'OCaml vers JavaScript
http ://jaked.github.com/ocamljs

[⚓98] HaXe, un langage compilant vers les langages du Web
http ://haxe.org

[⚓99] PG'OCaml, de Richard Jones, interrogation de PostgreSQL
http ://pgocaml.berlios.de/

[⚓100] SQLite, moteur de base de données SQL simple
http ://www.sqlite.org/

[⚓101] Node.JS, programmation serveur en JavaScript
http ://www.nodejs.org/

[⚓102] Ruby on Rails, programmation Web en Ruby
http ://rubyonrails.org/

Acronymes

A
ADT ..... Algebraic Data Type (Type de Données Algébrique)
AJAX ..... Asynchronous JavaScript And XML
AOT ..... Ahead Of Time
API ..... Application Programming Interface
ASP ..... Active Server Pages
AST ..... Abstract Syntax Tree (Arbre de Syntaxe Abstraite)

B
BDR ..... Base de Données Relationnelle
BNF ..... Bachus Naur Form

C
CGI ..... Common Gateway Interface
CLR ..... Common Language Runtime (machine virtuelle .Net)
CMS ..... Content Management System (gestionnaire de contenu)
CPS ..... Continuation Passing Style
CSS ..... Cascading Style Sheet

D
DOM ..... Document Object Model
DSL ..... Domain Specific Language
DTD ..... Document Type Definition

E
E4X ..... ECMAScript for XML
ECMA ..... European Computer Manufacturers Association

F
FFI ..... Foreign Function Interface (interface de fonctions externes)
FTP ..... File Transfer Protocol (Protocole de transfert de fichiers)

G
GWT ..... Google Web Toolkit

H
HTML ..... Hyper Text Markup Language
HTTP ..... Hyper Text Transfer Protocol

I
IDE ..... Integrated Development Environment (Environnement de développement)
IDL ..... Interface Description Language
INRIA ..... Institut National de Recherche en Informatique et Automatique

J
JIT ..... Just In Time
JSP ..... Java Server Pages
JVM ..... Java Virtual Machine

L
LAMP ..... Linux Apache MySQL PHP
LIP6 ..... Laboratoire d'informatique de Paris 6
LIP6 ..... Laboratoire d'Informatique de Paris 6

M
MVC ..... Modèle Vue Contrôleur
MVC ..... Model View Controler

N
NAS ..... Network Attached Storage (stockage de masse en réseau)
NPAPI ..... Netscape Plug-in API

O
OPA ..... One Pot Application

P
PHP ..... Personal Home Page
POSIX ..... Portable Operating System Interface
PPS ..... Laboratoire Preuves Programmes et Systèmes
PPS ..... Preuves Programmes et Systèmes

R
RMI ..... Remote Method Invocation
RPC ..... Remote Procedure Call
RPC ..... Remote Procedure Call

S
SGBDR ..... Système de Gestion de Base de Donnée Relationnelle
SML-NJ ..... Standard ML of New Jersey
SML ..... Standard ML
SQL ..... Search and Query Language (Langage de requêtes de bases de données)
SVG ..... Scalable Vector Graphics

U
URL ..... Uniform Resource Locator

W
W3C ..... World Wide Web Consortium
WHATWG ..... Web Hypertext Application Technology Working Group
WYSIWYG ..... What You See Is What You Get

X
XHTML ..... eXtensible Hyper Text Markup Language
XML ..... eXtensible Markup Language

Annexes

A Introduction à JavaScript

Cette annexe est destinée au lecteur peu familier avec le langage JavaScript. Nous présentons rapidement son histoire, et donnons une introduction pratique aux traits principaux du langage ainsi qu'à l'interface avec l'environnement du navigateur.

Historique JavaScript [🕮60] est né durant la seconde guerre des navigateurs entre Netscape Navigator et Internet Explorer. Brendan Eich l'a conçu en 1995 pour Netscape, avec une syntaxe proche de Java, mais en s'inspirant plutôt des langages Scheme [🕮65] et Self [🕮32] pour la sémantique. Dans plusieurs interviews [⚓91] [⚓90], Brendan Eich explique que l'idée de départ était d'ajouter un langage accessible aux programmeurs débutants, pour relier des composants plus lourds comme les greffons ou les applets Java. Il fût ensuite inclus dans Netscape Navigator 2.0 en septembre 1995. Microsoft répliqua en incluant une variante nommée JScript dans Internet Explorer 3.0 la même année.

Le langage fût normalisé par l'ECMA en 1997 [🕮62], et renommé ECMA-262 afin de ne pas favoriser une des parties. Depuis, plusieurs versions sont sorties, et les navigateurs s'efforcent de répondre au standard. Cependant, une fois JavaScript au cœur de la guerre des navigateurs, à chaque cycle les fabricants ajoutaient leurs extensions personnelles incompatibles avec les autres navigateurs. La version 4 du standard a même été abandonnée en cours de rédaction faute d'entente entre les vendeurs, et la version 5 suivante a misé sur l'intégration des diverses extensions et incompatibilités au standard.

Au final, pour comprendre l'état de JavaScript il y a quelques années, il faut garder à l'esprit cette construction chaotique démarrée sur un cœur de langage conçu par une seule personne, à la base sans l'ambition de devenir ce qu'il est aujourd'hui, et faite d'extensions propriétaires, d'implantations non conformes, dont les spécifications arrivent après les implantations pour palier aux incompatibilités.

JavaScript aujourd'hui Heureusement, le nerf de la guerre d'aujourd'hui n'est plus à l'introduction d'extensions personnelles, mais à la compatibilité [⚓95] et surtout à la performance [⚓92] [⚓94]. À vrai dire, deux sur les quatre implantations majeures étant Open Source, l'ajout d'extensions propriétaires non documentées est devenu caduque. Le langage, tout comme le reste de l'environnement HTML 5, reste le plus souvent amélioré par des extensions propriétaires, mais ces extensions sont ensuite rapidement discutées par le consortium WHATWG (*Web Hypertext Application Technology Working Group*), et implantées dans les autres navigateurs. Cette vision optimiste reste à modérer par le délai entre les versions majeures des navigateurs très variable selon les vendeurs, et les implantations parfois imparfaites lors de leur première sortie. Il reste donc toujours un délai de quelques années avant de pouvoir être sûr qu'une nouvelle partie du langage fonctionne sur la majorité du parc des navigateurs installés.

L'autre face du Web Aujourd'hui, l'autre plate-forme pour les applications Web est Adobe Flash. Or cette plate-forme, si elle se distingue par son unique implantation propriétaire souvent décriée pour son instabilité, est étrangement similaire. Le modèle d'exécution est très événementiel, l'IDE permettant visuellement d'attacher des comportements à des événements. D'autre part, le langage pour écrire ces comportement est ActionScript, une implantation conforme et performante d'ECMA-262. Les techniques proposées dans OBrowser pourraient donc être utilisées pour implanter une machine virtuelle au sein de l'environnement Flash.

Au final, JavaScript est un langage complet et muni de concepts avancés, mais avec des particularités et un environnement d'exécution particuliers voire déroutants et différent des langages applicatifs

classiques. Cette section présente le langage en lui même, ses traits les plus inhabituels et surtout ceux nécessaires pour comprendre la description de l'implantation d'OBrowser.

A.1 Cœur du langage

Cette section présente les spécificité du cœur du langage JavaScript. Les traits impératifs classiques (structures de contrôle, affectations, etc.) sont inspirés des langage du type de Java et supposés connus du lecteur.

Valeurs Les valeurs immédiates sont les flottants double précision, les booléens, les chaînes non mutables et les valeurs spéciales null et undefined. Il n'y a pas d'entiers. Les valeurs construites sont les objets, avec un certain nombre d'objets prédéfinis par le langage comme les chaînes et les fonctions.

Objets Les objets sont des tables associatives dont les éléments sont appelés propriétés. Les propriétés des objets sont indexées par des chaînes. La syntaxe permet d'utiliser des variables ou constantes numériques, auquel cas les interprètes ne font pas en général de conversion vers des chaînes, mais utilisent des représentations optimisées ad hoc. Il est possible d'ajouter, de modifier et de supprimer dynamiquement les propriétés des objets, ainsi que de les parcourir. L'accès à une propriété non définie ne provoque pas d'erreur mais renvoie la valeur spéciale undefined. Le code ci-dessous donne un aperçu de la syntaxe pour créer, modifier et accéder à ces tables.

```
1 var p = {};        /* création d'un objet vide */
2 p["x"] = 12;       /* ajouts de la propriété */
3 p["y"] = 22;
4 p["x"] = 15;       /* affectation de propriété */
5 var u = p["z"];    /* accès à une propriété non définie,
6                       u vaut alors undefined */
7 var p2 = {
8   /* définition de propriétés à la création */
9   x: 13,
10  y : 33
11 };
12 /* parcours d'objet */
13 for (i in p2) {
14   alert (i + " :" + point[i]) ;
15   /* affiche "x : 13" puis "y : 33 "*/
16 }
```

Certains objets prédéfinis, ou fabriqués à partir d'objets prédéfinis ont des restrictions quant à l'ajout ou la modification de propriétés. De plus, les accesseurs de propriétés des objets prédéfinis étant définis de manière procédurale, la lecture ou l'écriture d'une propriété peut avoir des effets de bord. Par exemple, le navigateur corrige la plupart du temps les valeurs affectées aux objets prédéfinis si celles-ci ne sont pas conforme à l'API plutôt que de lever une erreur. Une affectation peut aussi parfois aboutir à des comportements complexes que l'on attendrait plutôt d'un appel de méthode. Par exemple, dans le navigateur, affecter la propriété window.location avec une chaîne contenant une URL revient à demander un changement de page.

Types objets Le type de base des objets est Object. De nombreux types sont prédéfinis comme les tableaux Array, les expressions rationnelles RegExp, les chaînes String, etc. On peut créer une valeur d'un certain type objet T avec la syntaxe new T(), et tester l'appartenance d'un objet à un type avec instanceof. Le langage inclus des construction facilitant la création des objets prédéfinis. Par exemple, la syntaxe {} vue plus haut est une abréviation de new Object(), et [1,2,3,...] est une abréviation de new Array(1,2,3,...).

Le langage est *tout-objet* : les immédiats sont automatiquement encapsulés dans des objets prédéfinis (de type Number pour les valeurs numériques, String pour les chaînes), et la définition de fonction crée un objet instance du type Function.

Modularité Il n'y a pas de mécanisme intégré de modules. Tous les programmes JavaScript appelés dans une page sont lus les uns après les autres dans le même environnement global. Il n'y a pas non plus de mécanisme de chargement dynamique. Certaines bibliothèques JavaScript simulent un mécanisme de modules en utilisant les objets, et le chargement dynamique en utilisant la fonction eval.

A.2 Caractère fonctionnel

La syntaxe function f(args)code permet de définir une fonction. Les fonctions sont des valeurs comme les autres, et le nom f de la fonction désigne cette valeur fonctionnelle. Une syntaxe alternative sans le nom permet même de définir des fonctions anonymes.

Fonctions et portée lexicale La fonction est la seule construction permettant de restreindre la portée des variables en JavaScript. Une variable déclarée avec le mot-clef var à l'intérieur d'une fonction y sera restreinte.

Il est possible de définir des fonctions à l'intérieur de fonctions. Dans ce cas, le corps de cette fonction peut accéder aux variables locales des blocs supérieurs. Si une variable n'est pas trouvée dans la fonction courante, alors on cherche dans la fonction au dessus.

Comme en Java, le programmeur peut structurer son code avec des paires d'accolades, mais elles ne définissent en aucun cas des blocs lexicaux, ce qui est souvent source de bogues chez les débutants ou lorsqu'on porte du code d'un langage à syntaxe proche. En d'autres termes, si le programmeur déclare une variable avec var dans un bloc d'accolades, celle-ci existe toujours après l'accolade fermante. C'est également le cas pour les blocs des structures de contrôle.

Fermetures et environnements lexicaux Puisque les fonctions sont des valeurs, il faut embarquer les environnements lexicaux dans les valeurs fonctionnelles. Un point important, est que ces environnements lexicaux sont utilisés par référence. Concrètement si des variables locales sont modifiées ou ajoutées après la définition de la fonction locale dans une des fonctions parentes, alors cet effet sera visible lors de l'appel de la fermeture. De façon similaire, les modifications effectuées sur les variables d'une fonction parente seront visibles lors de l'appel suivant de la fermeture. Le code suivant donne un exemple de ce comportement, utilisé pour générer des fonctions de comptage.

```
1 function make_counter (){
2   var cpt = 0; /* compteur local */
3   return (function () {
4     return (++cpt);
5   })
6 }
7
8 /* deux appels créent deux instances de cpt */
9 f = make_counter ();
10 g = make_counter ();
11
12 /* affiche 1 2 1 2 */
13 alert( f () + "" + f () + "" + g () + "" + g ());
```

Mais si dans le code précédent ce fonctionnement est utile, il est souvent source d'erreurs, et déconcertant pour un langage à syntaxe à la Java. Le code suivant montre comme il est facile d'écrire du code bogué pour le programmeur débutant. Ici, Le programmeur a voulu écrire une fonction prenant en paramètres un objet o et un tableau t de noms, et initialisant les propriétés de l'objet o dont le nom

apparaît dans t, avec une fonction affichant son nom. Concrètement, le programmeur voulait observer les événements se produisant sur un élément de la page.

```
function witness_events (o,t) {
  for (var i = 0;i < t.length;i++) {
    /* on lie une fonction anonyme à chaque événement... */
    o[t[i]] = function () {
      /* ...qui affiche le nom de l'événement */
      debug(t[i] + " called !");
    }
  }
}

witness_events (document, ["onclick", "onkeypress", "ondrag"]);
```

Le résultat est que toutes ces fonctions vont imprimer "ondrag called", puisque la variable i dans la fonction générée est résolue comme variable du bloc de la fonction bogus, qui est accédé par référence, et que "ondrag" est sa valeur à la fin de la boucle.

S'il réussit à déceler l'erreur, le programmeur débutant tentera sûrement de corriger l'erreur en ajoutant une variable locale au bloc de la boucle, ce qui ne changera rien, puisque comme nous l'avons dit, seuls les blocs des fonctions comptent. Le code suivant est une possibilité de correction.

```
function witness_events (o,t) {
  function make_witness (n) {
      /* n est locale et différente
         pour chaque appel de make_witness */
      return function () {
        debug(n + " called !");
      }
  }
  for (var i = 0;i < t.length;i++) {
    /* on copie la valeur instantanée de t[i]
       lors de l'appel à make_witness */
    o[t[i]] = make_witness (t[i]);
  }
}

witness_events (document, ["onclick", "onkeypress", "ondrag"]);
```

Éléments de portée dynamique En plus des portées locales, JavaScript définit un environnement lexical global. Dans les navigateurs, cet environnement correspond à l'objet window, et il est possible dynamiquement de l'enrichir, l'appauvrir ou le modifier. Si un nom x utilisé dans le corps d'une fonction ne correspond pas à une définition locale, alors l'accès, la modification ou l'ajout est fait dans cet environnement global.

D'autre part, une syntaxe with(o){/* code */} existe, qui permet d'empiler dynamiquement un objet sur la pile d'environnements lexicaux dans un bloc délimité par des accolades. Les propriétés de cet objet participent alors à la résolution des noms de variable en lecture et en écriture, mais étrangement pas en ajout via le mot clef var, les nouvelles variables sont crées dans le bloc function au dessus et non dans l'objet.

Fonctions variadiques La définition des fonctions est encore un point où le choix de la syntaxe reprise de Java est discutable. En Java, le nombre de paramètres d'une fonction est fixe, et chaque argument est nommé. En JavaScript, les fonctions sont variadiques, de façon similaire à Scheme. Il est possible de passer un nombre arbitraire d'arguments à une fonction, quels que soient ceux précisés lors de sa définition. Les arguments manquants valent alors undefined, et les arguments supplémentaires sont accessibles via la variable spéciale arguments. De fait, il n'y a alors pas de mécanisme d'application partielle.

A.3 Modèle objet

Nous avons présenté la structure de tables associatives des objets de JavaScript. Au dessus de cette représentation, le langage implante des primitives objets, selon un modèle à prototypes.

Tout d'abord le langage propose des constructions syntaxiques pour l'accès aux champs des objets et l'appel de méthode.

```
1 var t = document.title;                          /* accès à un champ */
2 document.location = "http ://www.ocsigen.org/"; /* modification de champ*/
3 window.alert("bouh");                            /* appel de méthode */
```

Concrètement, les champs sont simplement les propriétés et les méthodes les propriétés dont les valeurs sont des fonctions. Il est possible de redéfinir simplement une méthode en affectant la propriété associée avec une fonction personnalisée. Dans le corps de la fonction, le mot-clef this représente l'objet cible de l'appel de méthode (contrairement à Java, le this n'est jamais implicite en JavaScript). Une touche supplémentaire de portée dynamique dynamique est apportée par la possibilité de choisir la cible d'un appel de méthode. Cela se fait en appelant sur l'objet fonction la méthode prédéfinie call, avec l'objet qui devra prendre le rôle de this.

```
1 var o = {
2   /* champ t */
3   t : 3,
4   /* methode m */
5   m : function(s) {
6         debug(s + "--" + this.t);
7       }
8 };
9 o.m ("X");                /* affiche "X--3" */
10 o.m ();                  /* affiche "undefined--3" */
11
12 var f = o.m;             /* récupère la valeur fonctionnelle */
13 f.call ({});             /* affiche "undefined--undefined" */
14 f.call ({},"X");         /* affiche "X--undefined" */
15 f.call ({t:"Y"},"X");    /* affiche "X--Y" */
16
17 f ();                    /* affiche "undefined--undefined"
18 f.call (null);             si this est null ou undefined, il est
19                            redirigé vers l'environnement global */
```

Prototypes Si un champ (resp. une méthode) o.n est définie dans un objet o, alors c'est lui (resp. elle) qui sera le résultat de la résolution de nom par l'opération o.n. Si ce n'est pas le cas, alors avant de renvoyer undefined, le langage effectue la même recherche dans un objet associé à un champ spécial qu'on appelle le *prototype* de l'objet. Cette recherche se propage de la même façon sur le prototype du prototype, jusqu'à arriver à l'objet prédéfini Object.prototype, maillon terminal de toutes les chaînes de prototypes. On dit que l'objet *hérite* des champs et méthodes de son prototype.

Lors d'une affectation o.n = expr, c'est toujours l'objet o lui-même qui est modifié, et non son prototype. Si, avant l'affectation, le prototype de o définissait une propriété n mais que o n'en définissait pas, alors la propriété n est ajoutée à o, et masque celle de son prototype.

Attention à ne pas se tromper, le prototype d'un objet n'est pas stocké dans son champ prototype. S'il est accessible au programmeur, ce qui n'est pas forcément le cas, c'est en général sous un nom comme __proto__ est utilisé.

Types objet personnalisés En JavaScript, un type d'objet personnalisé est constitué d'un constructeur et d'un prototype. Concrètement, le constructeur d'objets de type T est la valeur fonctionnelle (en d'autres termes l'objet de type Function) résultant de l'évaluation d'une définition de fonction nommée T. Lorsque l'utilisateur utilise alors la syntaxe new T(), un nouvel objet de type T est créé, et la fonction

T est appelée, dans le corps de laquelle this représente l'objet en cours de création. Le code ci-dessous montre par exemple la création d'une classe point, dont le constructeur initialise les composantes.

```
function Point(x,y) {
  this.x = (x != undefined) ? x : 0;
  this.y = (y != undefined) ? y : 0;
}

var p = new Point(3);
/* affiche 3 */
debug (p.x);
/* affiche 0 */
debug (p.y);
```

L'objet T (le constructeur) possède un champ prototype qui contient au départ un objet vide. C'est cet objet qui est affecté comme prototype des objets de type T créés par ce constructeur. Ainsi, le programmeur peut affecter cet objet, et les méthodes et champs seront accessibles à tous les objets de type T, comme dans l'exemple étendu ci-dessous.

```
function Point(x,y) {
  if (x != undefined)
    this.x = x;
  if (y != undefined)
    this.y = y;
}
Point.prototype.x = 0;
Point.prototype.y = 0;

Point.prototype.toString = function() {
  return (this.x + "x" +  this.y);
}

/* affiche (3,4) */
debug ((new Point(3,4)).toString());
/* affiche (0,0) */
debug ((new Point()).toString());
```

Pour obtenir un chaînage de prototypes, et que T hérite d'un autre type U, la seule possibilité en JavaScript est que le prototype de l'objet stocké dans le champ prototype de T soit une instance de U. C'est assez problématique car cela implique d'appeler le constructeur de U lors de l'héritage, ce qui n'est pas toujours une bonne idée si par exemple celui-ci effectue des effets de bord. Pour cela, beaucoup de bibliothèques effectuent une copie de prototypes plutôt qu'un chaînage (c'est en particulier ce que nous faisons dans notre implantation de l'interface entre les modèles objets). Le code ci-dessous montre ces deux possibilités d'héritage dans le modèle objet de JavaScript.

```
/* par chainage */
function U() { }
U.prototype.x = 1;
function T() { }
T.prototype = new U();
T.prototype.y = 2;

/* par copie */
function U() { }
U.prototype.x = 1;
function T() { }
for (i in U.prototype)
   T.prototype[i] = U.prototype[i];
T.prototype.y = 2;

/* test */
var t = new T() ;

/* affiche "1 2" dans les deux cas */
debug (t.x + " " + t.y);
```

A.4 L'environnement d'un navigateur

JavaScript en lui-même permet de manipuler des objets et des valeurs primitives comme nous venons de le voir. Pour fonctionner, il utilise des types objet primitifs tels que Function, Array ou Object : le type de base de tous les objets, la fin de toutes les chaînes de prototypes.

De même, l'inter-action avec l'environnement du navigateur se fait via un ensemble d'objets et fonctions prédéfinis. Nous avons aperçu plus haut l'utilisation de l'objet window, qui renferme toutes les variables globales définies par les scripts s'exécutant dans la même fenêtre du navigateur, ainsi que d'autres informations prédéfinies, et un pointeur vers l'objet document.

Le DOM Un document Web est décrit sous forme d'arbre, et transféré sur le client dans une forme linéarisée au format XML. L'objet document est la racine de l'arbre représentant le document à l'exécution. On accède aux fils dans l'arbre via des propriétés prédéfinies firstChild, nextSibling, etc. Il est aussi possible de modifier dynamiquement le document (ajouter, supprimer, déplacer des éléments gra-

phiques de la page Web) en agissant sur les propriétés et en appelant des méthodes prédéfinies de cet objet et de ses fils. Cette API de manipulation du document est appelée DOM.

Les Événements et le DOM Il est possible de rendre un document inter-actif en affectant des fonctions de rappel (*callbacks*) à des propriétés prédéfinies des objets du DOM. Par exemple, lorsque l'utilisateur clique sur un élément graphique de la page correspondant dans le DOM à l'objet o, la méthode o.onClick() est appelée si elle a été définie par le programmeur. Si au contraire, le programmeur n'a défini de gestionnaire d'événement, celui-ci sera propagé dans l'arbre. La propagation peut s'effectuer vers le parent ou les enfants (mécanisme appelé *bubbling*, contrôlable via des fonction prédéfinies).

Modèle d'exécution Le modèle d'exécution de JavaScript est entièrement basé sur ces événements. Ainsi, on dispose d'un événement window.onLoad permettant d'exécuter du code lorsque la page est totalement chargée, afin d'initialiser les scripts. Il est aussi possible de déclencher une temporisation, et d'attacher une fonction qui s'exécutera lorsque l'événement de fin de temporisation se produira via la fonction prédéfinie setTimeout. De même, l'attente sur ressource se fait de manière asynchrone via les événements, en attachant une fonction de rappel à exécuter lorsque la ressource change d'état (réception d'une donnée depuis le serveur par exemple).

Les gestionnaires d'événements sont classiquement placés dans une file consommée indéfiniment par la boucle événementielle, et la prise en compte des événements, ainsi que la mise à jour visuelle de la page sont faites à chaque tour de boucle. Il est alors indispensable que les scripts eux-mêmes, s'ils sont non triviaux, soient programmés de façon événementielle : pas d'attente active et découpage du code pour permettre de rendre la main quand il faut à la boucle d'événements. Étrangement, JavaScript ne dispose pas pour autant de mécanisme de continuation ou de co-routines [1], et il est nécessaire d'encoder ces mécanismes à la main. La seule primitive approchante est la fonction setTimeout citée plus haut, qui permet de demander explicitement qu'une fonction de rappel soit exécutée au prochain tour de boucle. L'exemple ci-dessous montre un exemple de script changeant la couleur du fond de page chaque seconde. La boucle est décomposée en deux fonctions, se rappelant mutuellement via setTimeout pour rendre la main au navigateur entre chaque appel.

```
1  function black() {
2    document.body.style.backgroundColor = 'black';
3    window.setTimeout (white, 1000);
4  }
5  function white() {
6    document.body.style.backgroundColor = 'white';
7    window.setTimeout (black, 1000);
8  }
9  document.onload = function () {
10   black ();
11 }
```

A.5 Vision formelle

Depuis que l'utilisation de JavaScript s'est clairement imposée pour la programmation Web client, le monde de la recherche a cherché à donner des fondations au langage.

Évaluation Clairement, de part sa conception et la diversité des implantations, le modèle d'évaluation de JavaScript n'est pas trivial, en témoignent les idées fausses de beaucoup de programmeurs. Le premier effort pour clarifier ce modèle a été fait à l'époque de la conception de JavaScript 2.0 (ECMA-262 4), qui devait être spécifié par un interprète de référence en SML [📖19]. Comme nous l'avons expliqué en

1. Des extensions du type co-routines sont proposées par Mozilla pour les prochaines versions, mais la portabilité interdit de les utiliser pour l'instant.

introduction de cette annexe, cette version a été abandonnée pour le standard de facto que les concepteurs de navigateurs ont imposé avec HTML 5. Les travaux suivants ont donc cherché à spécifier JavaScript tel-quel. Maffeis et Al. [📖23] et Krishnamurthi et Al. [📖17] ont aussi proposé indépendamment deux sémantiques opérationnelles. Les premiers cherchent à formaliser en une trentaine de page les 200 pages de la spécification ECMA [📖62] informelle en langue anglaise, le résultat est presque complet, il manque cependant quelques constructions syntaxiques complexes. Les seconds ont utilisé une approche différente, en formalisant un langage minimal λ_{JS}, et en projetant les constructions de haut niveau de JavaScript.

Sémantique statique De la même façon que des efforts ont été menés pour spécifier l'évaluation de JavaScript, plusieurs travaux récents cherchent à apporter un certain degré de sécurité statique au langage, à la fois pour la sûreté d'exécution et pour la sécurité Web. La majorité des travaux se concentrent néanmoins sur l'analyse statique de problèmes de sécurité précis. L'exception notable est constituée par les travaux de Peter Thiemann [📖39, 37], qui cherche à typer le langage JavaScript dans son ensemble, avec un système de types dédié, en utilisant des techniques modernes de synthèse de types par résolution de contraintes ou analyse statique.

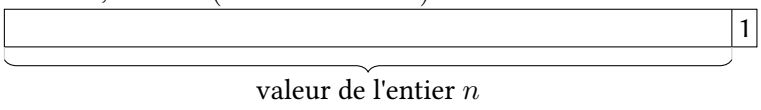
La machine virtuelle et les rouages d'OCaml

La machine virtuelle de d'OCaml ou ZAM2, qui a succédé à la ZAM [1] de Caml Light, introduite dans la nouvelle implantation du langage Caml par Xavier Leroy en 1990 [📖56] [⚓96]. Elle interprète un code-octet spécifique, assez restreint mais couvrant un large spectre permettant de bonnes performances tant en fonctionnel qu'en impératif ou en programmation par objets.

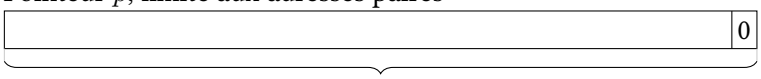
B.1 Les valeurs

Les valeurs du langage sont représentées dans la machine soit par des entiers, soit par des blocs pointés. Les données n'embarquant pas d'information de types à l'exécution, la distinction entre entier et pointeur est faite en utilisant un bit comme discriminant (les entiers manipulés ont alors un bit de moins que le mot machine). Ce format est schématisé par le graphique ci-dessous.

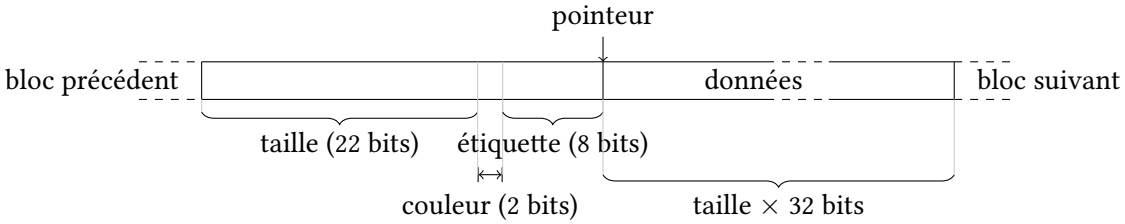
– Entier n, limité à $(\textit{mot machine} - 1)$ bits :



– Pointeur p, limité aux adresses paires :



Chaque bloc pointé est précédé d'un en-tête, stockant des méta-données sur celui-ci. Il contient la taille du bloc en mots machine, quelques bits dits de *couleur* servant à la gestion mémoire automatique, et une étiquette (un entier sur 8 bits) permettant de distinguer plusieurs formats de blocs, sur laquelle nous reviendrons. L'exemple ci-dessous est la représentation sur une machine 32 bits du bloc dans le tas OCaml, la valeur étant le pointeur vers le début des données du bloc, après l'en-tête.



Les données des blocs issus des types algébriques d'OCaml sont simplement des suites de valeurs. Dans le cas d'un type somme, l'étiquette est utilisée pour encoder le constructeur. Dans le cas d'un type enregistrement, tableau ou n-uplet, l'étiquette vaut 0. Les 246 premières étiquettes sont disponibles pour cette utilisation, les étiquettes suivantes sont réservées pour des formats de blocs spécifiques. Les formats spéciaux entre 246 et 249 sont des suites de valeurs comme les blocs de données normaux, en particulier, ils peuvent être parcourus par le ramasse-miettes. Les blocs d'étiquette 250 (`No_scan_tag`) ont un format spécifique et sont opaques pour la gestion mémoire.

1. L'acronyme ZAM signifie *ZINC Abstract Machine*. ZINC est le nom de code du projet Caml Light, c'est l'acronyme récursif de *ZINC Is Not Caml*.

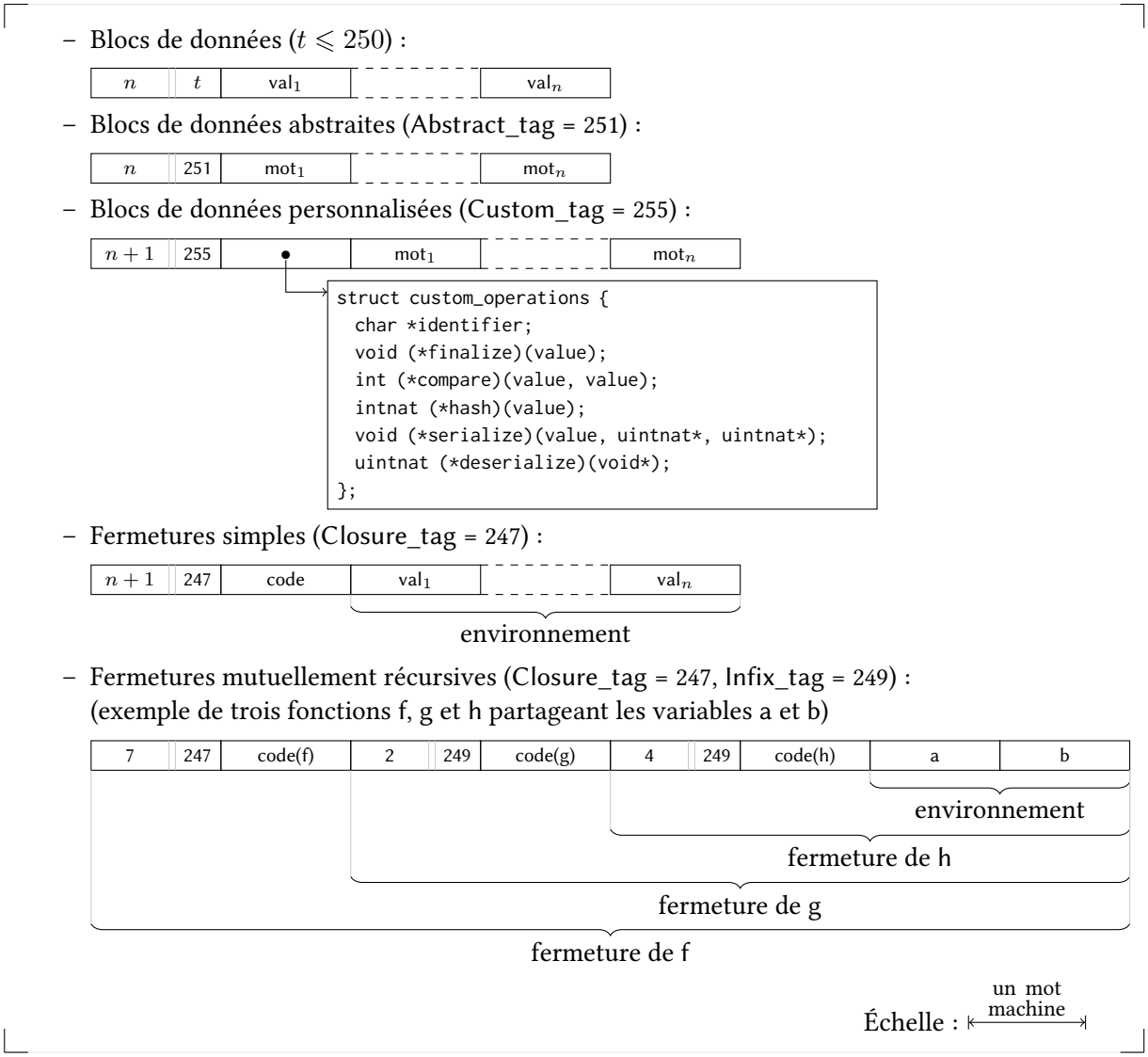


FIGURE B.1: Représentation des blocs en OCaml.

- Abstract_tag et Custom_tag sont les étiquettes pour les blocs contenant des valeurs externes venant du monde C. Dans le premier cas, les données sont complètement abstraites (opaques), dans le second, le premier mot est un pointeur vers une structure C contenant des primitives personnalisées (sérialisation, etc.), permettant de rendre la valeur un peu plus intégrée à la bibliothèque d'exécution.
- Les fermetures, avec l'étiquette Closure_tag, sont constituées d'un pointeur de code et d'un environnement : suite des valeurs constituant les variables libres et paramètres déjà passés de la fonction.
- Les fermetures correspondant aux définitions de fonctions mutuellement récursives sont un cas particulier de bloc Closure_tag. D'autres en-têtes portant l'étiquette Infix_tag sont inclus dans la même zone allouée, ainsi la première fonction est le bloc lui-même, et les autres sont des pointeurs au milieu du bloc. Les bits de taille des en-têtes de blocs infixes sont détournés pour indiquer le décalage par rapport au début du bloc fermeture. L'environnement est à la fin de ce bloc, et est donc partagé entre des différentes fonctions (le code de chacune des fonctions accédant aux mêmes variables de l'environnement avec des indices différents).
- Les flottants (systématiquement en double précision) sont encapsulés dans un bloc d'étiquette

Double_tag, dont les données font les 64 bits nécessaires.
- Les valeurs construites ne contenant que des flottants sont optimisés en blocs Double_array_tag dont les données sont formées par une suite contiguë de flottants 64 bits.
- les chaînes de caractères, String_tag, sont compatibles avec le format C, une suite d'octets terminée par un octet nul. La fin du segment de données est remplie de zéros pour obtenir un multiple de la taille du mot machine, puisque la taille des données des blocs est donnée en mots machine.

Les différents formats de blocs sont récapitulés et représentés graphiquement dans la figure B.1.

B.2 Le jeu d'instructions

La ZAM2 est une machine à pile et à environnement (tableau de valeurs indexées par des entiers), avec un registre supplémentaire accu pour les résultats temporaires (qui peut être vu comme le sommet de la pile). Le code est une séquence d'instructions et de constantes utilisées comme paramètres de ces instructions. Les adresses de saut sont directement les indices dans cette séquence.

Pile La machine définit bien sûr les instructions de base de manipulation de pile (PUSH, POP, ACC, ASSIGN, etc.) ainsi que quelques variantes et combinaisons de celles-ci pour améliorer un peu les performances et la taille du code.

Arithmétique Elle dispose aussi d'instructions spécifiques pour les constantes, l'arithmétique et les opérations logiques entières (CONSTINT, XORINT, ADDINT, etc.), Les opérations sur les flottants sont faites via des appels externes.

Blocs La machine dispose aussi d'un jeu d'instructions pour allouer, modifier et accéder au contenu des blocs (MAKEBLOCK, MAKEFLOATBLOCK, SETFIELD, GETFIELD, etc.). Au niveau de la machine, les accès aux blocs ne sont pas typés, et le code-octet est supposé correct. Du code incorrect pourrait par exemple dé-référencer des entiers ou accéder à des indices en dehors des blocs. Les vérifications sont faites au niveau du langage et de la compilation.

Globales Les données globales sont indexées par des entiers et accédées par des instructions spécifiques (GETGLOBAL, SETGLOBAL, etc.).

Branchements La machine dispose d'instructions classiques de branchement conditionnel déplaçant le pointeur de code courant en fonction de l'accumulateur (BRANCHIF, SWITCH). Le filtrage par motif est entièrement compilé en branchements simples et n'apparaît pas dans le code-octet.

Traits fonctionnels La machine implante le mécanisme Push/Enter en appel par valeur via un jeu d'instructions dédiées à la création et le rappel de valeurs fonctionnelles :
- CLOSURE construit une fermeture en capturant l'environnement situé au sommet de pile,
- APPLY modifie l'environnement pour celui de la fermeture passée en paramètre, et appelle son code, l'argument de la fonction étant sur la pile
- Des instructions APPLY2, APPLY3 permettent d'appliquer plusieurs paramètres à la fois. Elle empilent les arguments et mettent à jour le registre spécial extra_args utilisé par le mécanisme d'application général avec le nombre de paramètres passés en plus du premier.
- PUSH_RETADDR permet de sauvegarder l'état actuel sur la pile avant une application.
- Les instructions de la famille APPTERM sont utilisées pour les appels terminaux, en réutilisant l'espace en pile utilisé par l'appel en cours.

- C'est GRAB, qui est toujours la première instruction du corps d'une fonction, qui implante le mécanisme d'application générale. Elle vérifie si elle a assez d'arguments passés sur la pile pour s'appliquer, en regardant le registre extra_args. Dans le cas contraire, elle renvoie une nouvelle fermeture, dont l'environnement est étendu de ces arguments. Le pointeur de code de cette fermeture ne pointe pas vers cette instruction GRAB, mais vers une instruction RESTART, toujours compilée juste avant, qui extrait l'environnement de fermeture et le replace sur la pile et restaure extra_args, afin que GRAB puisse faire à nouveau son travail.
- RETURN est toujours la dernière instruction d'une fonction (non récursive terminale), aussi utilisée pour la compilation de l'application générale. Si la valeur renvoyée par la fonction est une fermeture, et qu'il reste des arguments à appliquer sur la pile, c'est cette instruction qui effectue cette nécessaire application. Sinon, elle saute à l'adresse déposée par PUSH_RETADDR.

L'accès à l'environnement de fermeture est réalisé par l'instruction ENVACC (et ses déclinaisons).

Les fermetures mutuellement récursives sont crées grâce à l'instruction CLOSUREREC, et on peut accéder à une fonction du même bloc récursif via les instructions de la famille OFFSETCLOSURE (qui effectuent des décalages de pointeurs au sein de la fermeture, comme expliqué dans la section sur les blocs).

Exceptions OCaml utilise un mécanisme d'exceptions où la pose de rattrapeur et le déclenchement sont équivalents et faibles en coût. Concrètement, OCaml utilise un chaînage de rattrapeurs d'exceptions posés sur la pile. Le registre spécial trap_sp contient un pointeur vers le dernier rattrapeur, afin de de remonter la pile jusqu'à celui-ci en temps constant lors d'une levée d'exception. Ces opérations sont faites via les instructions PUSHTRAP, POPTRAP et RAISE.

Les objets L'introduction des objets dans OCaml a seulement introduit les instructions de résolution de méthodes GETPUBMETH et GETDYNMETH, utilisant une recherche dichotomique dans un dictionnaire de noms de méthodes hachés. Le reste de la gestion des objets est entièrement écrite en OCaml, via l'interface interne bas niveau non typée de manipulation des valeurs en mémoire. Ainsi, les objets sont des valeurs OCaml normales, incluant la table des méthodes, les méthodes elles-mêmes sous forme de fermetures, etc. Le calcul des tables de méthodes n'est pas fait à la compilation mais est retardé à l'initialisation du programme.

B.3 Interface avec C

OCaml définit une API simple d'appel de fonction C depuis OCaml, ainsi qu'une FFI de manipulation des données depuis C. La machine dispose de l'instruction CCALL (et de déclinaisons à plusieurs paramètres) permettant d'appeler un pointeur de code selon la convention d'appel du C, avec un certain nombre d'arguments.

Les fonctions définies ainsi en C prennent en paramètres des valeurs OCaml via le type C value et doivent renvoyer une valeur OCaml. Il est possible de construire des valeurs depuis C avec une API de manipulation des valeurs, mais il faut alors bien faire attention à construire des valeurs correctes et les enregistrer comme vivantes, sous peine de casser le ramasse-miettes.

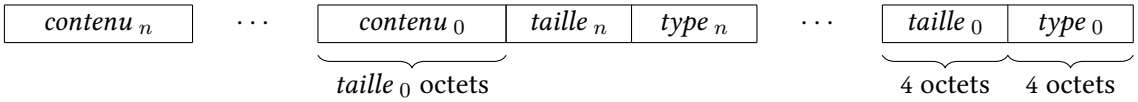
De façon symétrique, il est possible, à partir d'une valeur fonctionnelle OCaml et d'une valeur jouant le jeu d'argument, de retourner un moment dans le monde OCaml depuis C avec les fonctions de la famille caml_callback. Il est aussi possible de déclencher des exceptions avec les fonctions caml_raise.

B.4 Format du fichier de code-octet

Le fichier exécutable n'est pas une simple suite d'instructions. Il contient, en plus du programme lui-même, un environnement permettant de paramétrer la machine virtuelle pour qu'elle interprète

correctement de code : données globales, primitives utilisées, etc.

Concrètement, le fichier est décomposé en sections, de la façon suivante :



On le lit donc en partant de la fin, section après section. Les sections principales sont les suivantes, identifiées par leur type sur quatre octets :

- **CODE** : contient les instructions, les unes à la suite des autres. Chaque instruction est codée sur quatre octets, pour une lecture et une indexation plus rapide. Dans la même veine, les arguments éventuels d'une instruction sont stockés dans des mots de quatre octets à la suite de cette dernière. Ainsi, les pointeurs de code présents (par exemple les arguments des branchements) sont directement des indices de mots.
- **DATA** contient une structure sérialisée contenant toutes les données globales pré-calculées à la compilation, et des emplacements vides initialisés au lancement du programme.
- **PRIM** contient les associations entre les noms des implantations en C des primitives et les entiers utilisés dans le code pour les désigner. Il s'agit concrètement d'une série de chaînes séparées par des zéros, le numéro associé à chaque primitive étant simplement son ordre dans cette séquence.
- **DLLS** contient les noms des bibliothèques dynamiques utilisées par le programme.
- **DBUG** contient des informations de déverminage pour l'outil ocamldebug.

B.5 Sérialisation

Le langage propose un module Marshal permettant de linéariser en mémoire ou dans un flux une valeur construite. Les valeurs sérialisées sont des réflexions systématiques et bas niveau de la structure mémoire, et donc dépendantes de l'implantation de la machine virtuelle, de la taille du mot machine, etc. Le caractère non typé de la machine fait que cette sérialisation ne peut être assurée correcte du point de vue des types, et une valeur mal formée peut casser la machine. Ce mécanisme est nécessaire dans l'implantation d'une ZAM puisque les données globales d'un programme OCaml sont sérialisées par le compilateur dans le fichier code-octet. Puisque nous aimerions qu'un programme OBrowser puisse communiquer facilement avec un serveur en OCaml, il faut que le format de sérialisation soit identique. Des travaux sont en cours [🕮43] pour donner à ce mécanisme une certaine sécurité, pour éviter de casser la machine et la gestion mémoire, ce qui est un minimum dans un environnement de programmation Web.

Algorithme L'algorithme de sérialisation est le suivant : la valeur à sérialiser est parcourue en profondeur, et les blocs parcourus sont recopiés dans l'ordre dans un segment mémoire. Lors de la copie, un pointeur vers un bloc est transformé en l'indice de ce bloc dans la valeur sérialisée.

Concrètement, l'algorithme principal parcourt un bloc. Lorsque qu'il rencontre un pointeur vers un bloc non encore sérialisé, ce dernier se voit associé comme indice (dans une table d'indices) le nombre de blocs déjà rencontrés, et il est mis dans une liste d'attente. Si il rencontre un pointeur vers un bloc déjà présent dans la table d'indice, il utilise cet indice. Lorsque le parcours s'arrête, on relance l'algorithme sur le premier élément de la liste d'attente. On arrête lorsque la liste d'attente est vide. Cet algorithme prend en charge naturellement les structures comportant des cycles ou du partage.

Pour désérialiser une valeur, la possibilité la plus simple est d'allouer tous les blocs nécessaires, puis de les initialiser tous dans un second temps en remplaçant les indices par les pointeurs déjà créés.

Format de sérialisation En réalité, les blocs ne sont pas recopiés exactement selon la représentation des données en mémoire. Dans la valeur sérialisée, chaque segment utile est précédé d'un en-tête sur

un octet, qui décrit sa taille et comment l'interpréter. Les entiers, par exemple, sont encodés dans un segment de 1, 2, 4 ou 8 octets, suivant s'ils sont précédés de l'en-tête INT8, INT16, INT32ou INT64 (c'est une optimisation de la taille de la valeur sérialisée, ils doivent bien sûr, être désérialisés à la taille du mot machine).

La figure B.2 donne les différents en-têtes, comme les interpréter, ainsi que deux exemples de sérialisation.

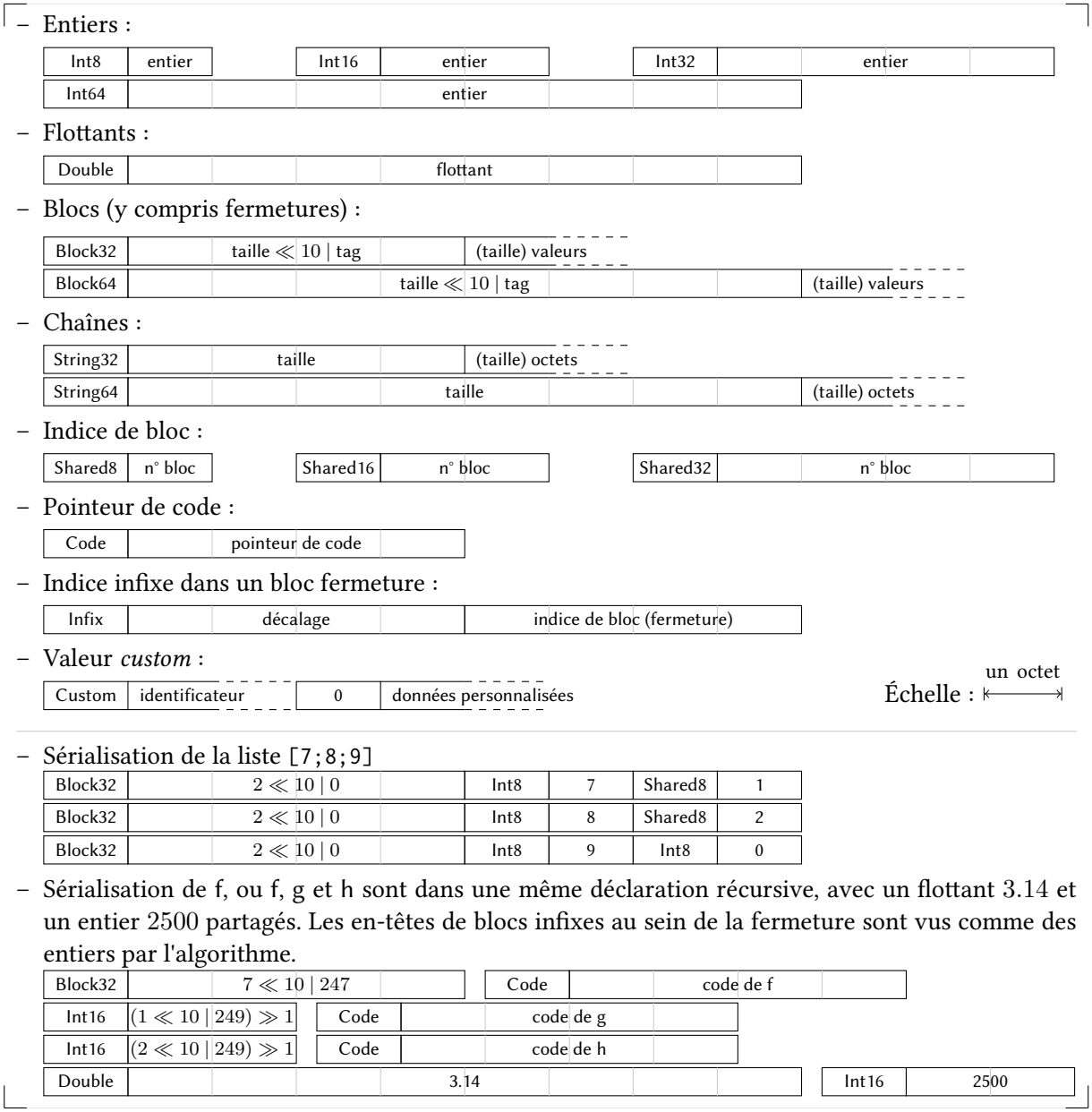


FIGURE B.2: Segments du format de sérialisation.

C Implantations de *fDOM*

Nous donnons deux implantations : la première est en OCaml dans un environnement classique, où OCaml est à la fois le langage d'implantation et le langage hôte. La seconde est en JavaScript pour le langage d'implantation, et en OCaml, via l'utilisation d'OBrowser, pour le langage hôte. Les deux sont spécialisées pour gérer des documents de type page Web, avec des nœuds spécialement réservés pour le contenu textuel.

C.1 Implantation en OCaml

Dans cette implantation, on cherche à implanter *fDOM* en OCaml, pour être utilisé depuis OCaml. Une utilisation possible serait la manipulation de document côté serveur dans un environnement de programmation Web. Commençons par quelques explications permettant de décrypter, dans l'ordre, le code donné à partir de la page 246.

Paramètres L'instanciation $\mathcal{I}$ des paramètres de *fDOM* est définie comme suit :

- $\mathcal{I}(Tag)$ = `Element of string | Text`
  On interprétera les nœuds texte en utilisant leur propriété *"textContent"* pour contenu textuel.
- $\mathcal{I}(Imm)$ = `String of string | Int of int`
- $\mathcal{I}(Key) = \mathcal{I}(Imm)$
- $\mathcal{I}(E \cup \{nil\})$ = $\mathcal{I}(E)$ `option`
- $\mathcal{I}(\{nil\})$ = `unit`
- $\mathcal{I}(Int)$ = `int`
- $\mathcal{I}(Enum(E))$ = $\mathcal{I}(E)$ `list`

État Un point important à noter est le type document. Afin de s'intégrer au modèle mémoire du langage il y a plusieurs possibilités pour représenter le document :

1. On peut considérer que les racines du document inutilisées par ailleurs sont définitivement perdues. Dans ce cas, on ne maintient pas de liste des racines, et on n'implante pas la primitive `roots`. Cette solution implique qu'un nœud peut être détaché de son parent si celui-ci n'est plus utilisé. Cette solution est facile à mettre en place dans une implantation proche de la spécification comme ici, mais serait plus difficile dans une implantation utilisant des arcs bidirectionnels pour optimiser les parcours.
2. On peut aussi considérer que c'est au programmeur de libérer les nœuds, implanter la primitive `roots` et en rajouter une de libération. C'est un peu ce qui se passe lorsqu'on utilise l'API C des moteurs de DOM courant, qui utilisent un modèle mémoire indépendant à base de comptage de références.
3. Ici, nous proposons une version intermédiaire, où la primitive `roots` renvoie une sur-approximation de l'ensemble des racines, grâce à l'utilisation de références faibles.

Fonctions auxiliaires L'implantation a besoin de quelques fonctions auxiliaires pour

- supprimer de temps en temps les racines effectivement supprimées par le ramasse-miettes de l'ensemble de racines faibles,

- tracer l'exécution, en imprimant à chaque fois la primitive et la règle de sémantique choisie,
- et effectuer des recherches dans l'arbre.

Primitives Ici, on a choisi une implantation réentrante, en passant en argument supplémentaire à chaque primitive le contexte du document dans lequel on se trouve. On a ainsi ajouté une primitive `create` construisant un nouveau document. Alternativement, on aurait pu encapsuler un document dans un module ou un objet. Une impression de trace des primitives et règles exécutées a été ajoutée au code des primitives afin d'observer le déroulement du programme. La figure C.1 présente un exemple proche de celui de la figure 7.2, codé avec cette implantation, ainsi que la trace d'exécution ainsi produite.

```
open Fdom ;;
let d = create_dom () ;;
let p = create_black d (Element "P") ;;
let n1 = create_black d (Element "N") ;;
let n2 = create_black d (Element "N") ;;
set d p (String "test") (Imm (Int 12)) ;;
bind d p n1 ; bind d p n2 ;;
bind d n1 n2 ;;
```

(a) Code

```
Prim: create dom          Rule: create dom
Prim: create black        Rule: create black
Prim: create black        Rule: create black
Prim: create black        Rule: create black
Prim: set                 Rule: set
Prim: bind                Rule: attach
Prim: bind                Rule: attach
Prim: bind                Rule: move
```

(b) Trace d'exécution

FIGURE C.1: Exemple d'utilisation de *fDOM* en OCaml

Code OCaml :

```
(* --------------- PARAMÈTRES -------------- *)

(* paramètres *)
type imm =
  | Int of int
  | String of string
type tag =
  | Element of string
  | Text
type key = imm

(* implantation *)
type obj =
  (* objets (noirs et blancs) *)
  | Black of tag * properties * links
  | White of properties
and links =
    (* liens vers les enfants *)
    obj list ref
and properties =
    (* ensemble de propriétés des objets *)
    (imm * value) list ref
and value =
   (* valeurs des propriétés *)
  | Imm of imm
  | Obj of obj

(* type principal *)
type document = obj Weak.t list ref


(* --------- FONCTIONS AUXILIAIRES --------- *)

(* exception levée sur appel incorrect *)
exception Bad_args

(* réduit la liste des racines faibles *)
let update dom =
  dom := List.filter
    (fun w -> Weak.get w 0 <> None)
    !dom

(* trace un appel de primitive *)
let trace p r =
  Printf.fprintf stdout
    "Prim : %-20s Rule : %-20s\n%!" p r

(* ajoute une racine faible *)
let add_root dom r =
  let wr = Weak.create 1 in
  Weak.set wr 0 (Some r) ;
  dom := wr :: !dom

(* récupère une liste de racines non faibles *)
let unweaken dom =
  List.fold_left
    (fun r w ->
      match Weak.get w 0 with
        | None -> r
        | Some v -> v :: r)
    [] !dom

(* trouve le parent d'un nœud *)
let find_parent dom n =
  let rec find l =
    List.fold_left
      (fun r p ->
        match p with
          | Black (_, _, links) ->
            if List.exists
              ((==) n) !links then
              Some p
            else
              (match find !links, r with
                | (Some _ as r'), None
                | None, (Some _ as r') -> r'
                | None, None -> None
                | _ -> assert false)
          | White _ -> assert false)
      None l
```

```
  in find (unweaken dom)

(* teste si a est un ancêtre de n *)
let ancestor dom a n =
  let rec check n =
    let p = find_parent dom n in
    match p with
      | None -> false
      | Some p when p = a -> true
      | Some p -> check p
  in
  a != n && check n


(* ------------ RÈGLES D'ACCÈS ------------ *)

let children dom n =
  match n with
    | White _ -> raise Bad_args
    | Black (_,_,links) ->
      trace "children" "children" ;
      List.length !links

let child dom p n =
  match p with
    | White _ -> raise Bad_args
    | Black (_,_,links) ->
      try
        let r = List.nth !links n in
        trace "child" "child" ;
        Some r
      with
        | Failure "nth"
        | Invalid_argument "List.nth" ->
          trace "child" "child-unbound" ;
          None

let roots dom =
  trace "roots" "roots" ;
  unweaken dom

let get dom n k =
  match n with
    | White pp | Black (_,pp,_) ->
      try
        let v = List.assoc pp k in
        trace "get" "get" ;
        Some v
      with
        | Not_found ->
          trace "get" "get-unbound" ;
          None

let tag dom n k =
  match n with
    | White _ -> raise Bad_args
    | Black (t,_,_) ->
      trace "tag" "tag" ;
      t



(* ------------ RÈGLES À EFFETS ------------ *)

let create_dom () =
  trace "create dom" "create dom" ;
  ref []

let create_black dom tag =
  trace "create black" "create black" ;
  let n = Black (tag, ref [], ref []) in
  add_root dom n ;
  n

let create_white dom =
  trace "create white" "create white" ;
  White (ref [])

let detach dom n =
  match n with
    | Black (_,_,_) ->
      begin match find_parent dom n with
        | Some (Black (_, _, links)) ->
          trace "detach" "detach-1" ;
          links :=
            List.filter ((!=) n) !links
        | None -> trace "detach" "detach-2" ;
        | Some (White _) -> assert false
      end
    | White _ -> raise Bad_args

let bind dom p n =
  if ancestor dom n p then
    (* interdiction des cycles*)
    raise Bad_args ;
  match p with
    | Black (_,_, links) ->
      begin match find_parent dom n with
        | None ->
          trace "bind" "attach";
          links := n :: !links
        | Some (Black (_, _, links') as p') ->
          trace "bind" "move";
          (* suppression du partage *)
          links' :=
            List.filter ((!=) n) !links' ;
          links := n :: !links
        | Some (White _) -> assert false
      end
    | White _ -> raise Bad_args

let set dom n k v =
  match n with
    | White (props) | Black (_,props,_) ->
      if List.exists
        (fun (k',_) -> k' <> k) !props
      then (
        trace "set" "modify" ;
        props :=
          (k, v) :: (List.filter
                       (fun (k',_) -> k' <> k)
                       !props)
      ) else  (
        trace "set" "set" ;
        props := (k, v) :: !props
      )

let unset dom n k v =
  match n with
    | White (props) | Black (_,props,_) ->
      if List.exists
        (fun (k',_) -> k' <> k) !props
      then (
        trace "unset" "unset-1" ;
        props := (List.filter
                    (fun (k',_) -> k' <> k)
                    !props)
      ) else  (
        trace "unset" "unset-2" ;
        props := (k, v) :: !props
      )
```

C.2 Implantation en JavaScript/OCaml pour OBrowser

Nous nous intéressons maintenant à une implantation utilisable en OCaml via OBrowser, et dont les primitives sont écrites en JavaScript, en utilisant le DOM du navigateur.

Paramètres L'instanciation $\mathcal{I}$ des paramètres de *fDOM* est définie comme suit :

- $\mathcal{I}(Tag)$ = `Element of string | Text`
  On interprétera les nœuds texte en utilisant leur propriété *"textContent"* pour contenu textuel.
- $\mathcal{I}(Imm)$ = `string`
- $\mathcal{I}(Key) = \mathcal{I}(Imm)$
- $\mathcal{I}(E \cup \{nil\}) = \mathcal{I}(E)$ `option`
- $\mathcal{I}(\{nil\})$ = `unit`
- $\mathcal{I}(Int)$ = `int`
- $\mathcal{I}(Enum(E)) = \mathcal{I}(E)$ `array`

Ils diffèrent un peu de l'implantation précédente pour des raisons de simplicité et de lisibilité, mais il serait raisonnablement simple d'avoir exactement la même interface, dans le but de partager du code entre client et serveur.

État L'état du document est celui de l'implantation du DOM du navigateur. Le type obj est un type abstrait pour OCaml, pouvant être un objet noir ou blanc.

Il n'est pas possible de créer plusieurs documents différents, le document sur lequel les primitives travaillent est celui de la page en cours. Contrairement à l'implantation précédente, les primitives n'ont donc pas de paramètre de type document.

Primitives Pour l'interface, on utilise le mot clef **`external`** d'OCaml, référençant des fonctions JavaScript effectuant la traduction entre les primitives de *fDOM* et les méthodes de l'implantation concrète du DOM du navigateur. La description de l'interface entre OCaml et JavaScript dans OBrowser a été donnée au chapitre 2.

Les paramètres transmis aux primitives sont encapsulés dans des types ocaml, qui sont alors traduits vers les valeurs adéquates pour les méthodes du DOM du navigateur. Pour cela, on utilise la représentation des données uniforme d'OCaml/OBrowser. De façon symétrique, les résultats des méthodes du DOM sont encapsulés dans des valeurs OCaml du bon type avant d'être renvoyés.

Pour la primitives `create`$^\bullet$, on utilisera les nœuds texte prédéfinis dans le cas où l'étiquette d'un nœud est `Text`, puisque le DOM du navigateur les considère différemment.

Validité Contrairement à la version précédente entièrement en OCaml, c'est le DOM du navigateur qui vérifie que les paramètres donnés aux primitives sont corrects et cohérents avec l'état du document. C'est aussi lui qui empêche la création de partage et de cycles. Si une primitive est mal appelée, le programme recevra une exception JavaScript du DOM, qui sera relayée en exception OCaml.

En particulier, en cas d'appel mal défini de primitive, il risque de recevoir les exceptions suivantes :

- `INDEX_SIZE_ERR` : mauvais index d'enfant
- `HIERARCHY_REQUEST_ERR` : liaison impossible (cycle, partage)
- `NOT_FOUND_ERR` : erreur de paramètre, nœud demandé non trouvé

Mais aussi que d'autres liées à des restrictions supplémentaires des pages Web par rapport à celles du document impératif, ou à des restrictions du navigateur.

Code JavaScript :

```
/* ------- DÉFINITIONS AUXILIAIRES ------- */

// représentation des valeurs ML
var value_Obj_tag = 0;
var value_Prim_tag = 1;

/* -------------- PRIMITIVES ------------- */

function fdom_create_white (unit) {
    /* objet JavaScript vide */
    return {};
}

function fdom_create_black (tag) {
    try {
        var r;
        if (is_int (tag)) {
            /* tag = Text */
            r = document.createTextNode ("");
        } else {
            /* tag = Element */
            r = document.createElement (
                string_val (tag.get (0))
            );
        }
        return r;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_children (node) {
    try {
        return node.childNodes.length;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_detach (node) {
    try {
        node.parent.removeChild (node);
        return UNIT;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_bind (node, child) {
    try {
        node.appendChild (child)
        return UNIT;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_child (node, idx) {
    try {
        if (idx < 0
            || idx >= node.childNodes.length)
            return 0 /* None */ ;
        else {
            var r = mk_block (n, 0) /* Some */;
            r.set (0, node.childNodes[idx]);
            return r;
        }
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_roots (unit) {
    try {
        var r = mk_block (1, 0);
        r.set (0, document);
        return r;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_properties (obj) {
    try {
        var n = 0;
        var vals = [];
        for (var k in obj) {
            vals[n] = val_string (k);
            n++;
        }
        var r = mk_block (n, 0);
        for (i = 0;i < n; i++)
            r.set (i, vals[i]);
        return r;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_get (obj, key) {
    try {
        var v = obj[string_val (key.get (0))];
        if (v == undefined)
            return 0 /* None */ ;
        else {
            var r = mk_block (n, 0) /* Some */;
            r.set (0, v);
            return r;
        }
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_set (obj, key, val) {
    try {
        var rval;
        if (val.tag == value_Obj_tag) {
            rval = val.get (0);
        } else { /* value_Prim_tag */
            rval = string_val (val.get (0));
        }
        obj[string_val (key.get (0))] = rval;
        return UNIT;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}

function fdom_unset (obj, key) {
    try {
        obj.removeAttribute (
            string_val (key.get (0))
        );
        return UNIT;
    } catch (e) {
        this.failwith ("DOM:" + e.message);
    }
}
```

Code OCaml :

```
(* -------------- PARAMÈTRES ------------- *)

type prim = string
 and key = prim
 and tag =
   | Element of string
   | Text


(* ---------------- ÉTAT ----------------- *)

type obj

(* valeurs transitant entre les deux mondes *)
type value =
  | Obj of obj
  | Prim of prim


(* --------- PRIMITIVES D'ACCÈS ---------- *)

external children
  : obj -> int = "fdom_children"
external child
  : obj -> int -> obj option = "fdom_child"
external roots
  : unit -> obj array = "fdom_roots"
external properties
  : obj -> prim array = "fdom_properties"
external get
  : obj -> prim -> value option = "fdom_get"


(* --------- PRIMITIVES À EFFETS --------- *)

external create_black
  : tag -> obj = "fdom_create_black"
external create_white
  : unit -> obj = "fdom_create_white"
external detach
  : obj -> unit = "fdom_detach"
external bind
  : obj -> obj -> unit = "fdom_bind"
external set
  : obj -> prim -> value -> unit = "fdom_set"
external unset
  : obj -> prim -> unit = "fdom_unset"
```

Résumé

Le but de cet thèse est de contribuer à rendre la programmation Web plus flexible et plus sûre qu'elle ne l'est avec les solutions répandues actuellement. Pour ceci, nous proposons une solution dans la lignée de la famille de langages ML, qui laisse un maximum de liberté au programmeur de part son côté multi-paradigmes, tout en offrant un degré de sûreté important grâce au typage statique.

Dans une première partie, nous montrons qu'il est possible de programmer le navigateur sans se plier au style de JavaScript. Notre solution est OBrowser, une implantation en JavaScript de la machine virtuelle OCaml. L'implantation prend en charge l'ensemble du langage OCaml et de sa bibliothèque, y compris le modèle de concurrence préemptif. Nous présentons de plus un mécanisme d'inter-opérabilité entre les couches objet de JavaScript et d'OCaml, permettant d'utiliser de façon bien typée l'environnement du navigateur avec les objets d'OCaml.

Dans une seconde partie, nous fournissons une API de manipulation du document plus sûre et de plus haut niveau que le DOM des navigateurs. En particulier, nous cherchons à éliminer les déplacements implicites effectués par le DOM pour maintenir la forme d'arbre, qui limitent les possibilités de typage statique. Nous donnons d'abord *fDOM*, un modèle formel minimal similaire au DOM. Puis nous proposons *cDOM*, un modèle alternatif ou les déplacements sont remplacés par des copies. Nous décrivons ensuite FidoML, un langage basé sur ML, permettant les manipulations bien typées du document grâce à l'utilisation de *cDOM*. Dans toute cette partie, nous faisons attention à ce que les solutions données soient aussi adaptables que possible.

Dans une troisième partie, nous montrons comment les travaux, jusqu'ici principalement présentés dans le cadre du navigateur, s'appliquent à un contexte multi-tiers. Nous donnons d'abord un tour d'horizon des plates-formes multi-tiers proches issues de la recherche. Nous décrivons en particulier les solutions qu'elles apportent à un ensemble de problématiques spécifiques à la programmation Web. Puis nous concluons en présentant les grandes lignes d'un langage multi-tiers mettant à profit les travaux des deux premières parties dans les solutions à ces différentes problématiques.

Abstract

The goal of this thesis is to contribute to make Web programming safer and more flexible than it is in the solutions prevalent today. To achieve this goal, we propose a solution based on the ML language family, which brings freedom to the programmer by its multi-paradigm aspect, while providing an important level of safety thanks to static typing.

In the first part, we show that it is possible to program the browser without sticking to the style of JavaScript. Our solution is OBrowser, an OCaml virtual machine in JavaScript. The implementation supports the whole OCaml language and its library, including the preemptive concurrency model. We additionaly present a mechanism for inter-operability between the object layers of JavaScript and OCaml, that allows to use the browser's API in a type-safe way, using OCaml objects.

In the second part, we give an API for document manipulations, designed to be safer and more high-level than the browser's DOM. In particular, we aim at eliminating the implicit moves performed by the DOM to maintain the tree structure, and which limit the possibilities of static typing. First, we give *fDOM*, a minimal formal model similar to the DOM. Then, we propose *cDOM*, an alternative model in which moves are replaced by copy operations. We then describe FidoML, a language based on ML equiped with document manipulation features, that are well typed thanks to the use of *cDOM*. Throughout this part, we make a special effort to design solutions flexible enough to be used in languages other than ML.

In the third and final part, we show how the work, so far presented in the context of the browser, can be applied to a multi-tier model. First, we give an overview of related multi-tier research platforms. In particular we describe the solutions they provide to a selected set of language aspects specific to Web programming. Then, we conclude by giving the outline of a multi-tier language, that uses the work the first two parts to built solutions to these language aspects.