Programmation Web Réactive dans un cadre typé statiquement pour l'orchestration de contenus multimédia riches

Rémy El Sibaïe Besognet

THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Rémy EL SIBAÏE BESOGNET

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Programmation Web Réactive dans un cadre typé statiquement pour l'orchestration de contenus multimédia riches

(Version préliminaire)

Directeur de thèse	M. Emmanuel Chailloux	Sorbonne Université
Rapporteurs	M. Manuel Serrano	INRIA Sophia Antipolis
	M. Peter Van Roy	Université catholique de Louvain
Examinatrice et Examinateurs	Mme. Emmanuelle Encrenaz	Sorbonne Université
	M. Jean-Ferdinant Susini	CNAM
	M. Benjamin Canou	Tezos

Remerciements

Table des matières

Chapitre 1

Introduction et état de l'art

Le Web est aujourd'hui la plateforme d'application la plus répandue, partagée par la majorité des terminaux de la planète de par sa facilité d'accès et les vitesses de transmission grandissantes des réseaux de communication. Tout type de machine a aujourd'hui une interface Web, de la montre connectée, à la télévision en passant par les téléphones mobiles et la domotique. Depuis longtemps, le Web n'est plus limité à la navigation au travers de documents statiques, mais expose des applications d'interface graphique (GUI) complexes proposant des interactions, des animations et du contenu. Les briques qui composent ces applications ne sont pas toutes présentes au début de leur exécution et sont agrégées par le biais du téléchargement depuis d'autres sources que le serveur d'origine puis liées dynamiquement aux programmes. Les communications deviennent de plus en plus riches et nombreuses de ces derniers. En plus du contenu, vidéo, musical ou textuel, un grand nombre de communications est généré par des éléments tiers : les publicités, les instances de réseaux sociaux traçant les utilisateurs, ou les accumulateurs de données statistiques permettant d'analyser la navigation de l'utilisateur. De part son aspect normatif, l'application Web remplace progressivement l'application de bureau en « client lourd », pour simplifier la portabilité et donc la diffusion du logiciel. Il est alors naturel que ce type de programme ai un accès de plus en plus profond à la machine qui l'exécute. Les interfaces de programmation (API, pour *application programming interface*) des navigateurs gèrent donc de plus en plus de capteurs et de matériaux spécifiques : le GPS, l'accéléromètre, les écrans tactiles, le stockage des données ou même les cartes graphiques. Il est cependant difficile de prévoir dans quelle condition de performance, d'affichage ou de compatibilité le programme sera exécuté. Les bibliothèques de code les plus utilisées se voient même forcées de gérer durant leur exécution les problèmes de version de navigateur, de taille d'écran ou de qualité du réseau. Certaines bibliothèques sont mêmes dédiées à cet aspect.

Une application Web échange effectivement des informations avec un environnement d'exécution inconnu avant son initialisation, c'est-à-dire le matériel de la machine (écran tactile, souris, GPS), et communique aussi avec des serveurs distants ou d'autres applications. Ces entrées-sorties sont calculées en continu par le moteur d'exécution du navigateur. Les entrées sont abstraites sous forme d'évènements dans le programme client qui peut y réagir de façon indépendante. Les sorties du programme résultant de ces réactions peuvent être la modification de l'affichage graphique et les communications échangées entre l'application et le serveur. La complexité vient de la nécessité de traiter simultanément les entrées selon une règle définie et en fonction d'un état. Pour cela, le programmeur doit souvent décrire la relation de dépendance complexe entre les entrées et les sorties.

JavaScript [25] est le langage de programmation *de facto* pour le client Web. Il est accompagné de l'interface de programmation DOM [38] (*Document Object Model*), permettant la manipulation de la page sous la forme d'une structure de données arborescente. Les spécifications de JavaScript et du DOM,

qui représentent le standard, sont écrites et mises à jour par le consortium W3C. L'implémentation du moteur d'exécution ne dépend, lui, que du navigateur. L'interface de programmation DOM propose un modèle concurrent dirigé par les évènements pour gérer les interactions communes à tous les navigateurs. Ce modèle à caractère impératif encourage les effets de bord, c'est-à-dire la mutation d'un état global. Cette situation peut être problématique en présence de concurrence et décourager l'écriture d'abstractions expressives en forçant le programmeur à mélanger le code destiné à l'affichage avec celui gérant les interactions et celui destiné à calculer. En temps normal, on souhaiterait découper l'application pour que chaque partie ait une tâche logique à accomplir mais la disposition du modèle évènementiel peut avoir des conséquences négatives sur le réusinage de code ou l'ajout de nouvelles fonctionnalités. En outre, si les erreurs de programmations sont fréquentes, les outils de support répandus ne proposent aucune vérification, en termes d'analyse statique et de typage. On ne peut donc pas se reposer sur une application que l'on a pas lourdement testé. En d'autres termes JavaScript et les outils standards ne sont pas une bonne approche pour résoudre les problématiques liées aux interactions dans une application Web.

Les besoins de technologies adaptées se font donc naturellement sentir par le programmeur Web, au travers des constructions du langage et de sa sémantique, mais aussi au travers du modèle de concurrence qu'il propose. Le choix du modèle de concurrence est fondamental pour le bon comportement du programme et aura une forte incidence sur la complexité d'écriture et de maintenance de ceux-ci. Beaucoup de technologies récentes tentent de résoudre ces problématiques. Certaines plateformes de programmation, par exemple, cherchent à séparer les responsabilités dans le programme entre les calculs, la gestion de l'affichage et la réaction aux évènements. On regroupe cette approche sous la dénomination MVC pour Modèle-Vue-Contrôleur ou MV* selon les variations possibles. On peut faire le rapprochement entre MVC et les patrons de conception en programmation objets qui, plus que changer le modèle de programmation, tenter d'imposer un consensus sur les bonnes pratiques. On peut citer les plus répandues de ces technologies, telles qu'Angular.js [62], Ember.js [68] ou React.js [82]. Du côté des langages de programmation, la tendance est de s'éloigner de JavaScript, de choisir un langage avec une syntaxe, une sémantique ou un modèle de concurrence plus sûr et de proposer un compilateur de ce langage vers JavaScript. Certains langages sont nouveaux et dédiés à la programmation Web, comme Elm [24], d'autres s'inspirent directement de JavaScript, comme TypeScript [88] et apportent une dimension de sûreté grâce au typage. D'autres encore partent d'un langage de programmation préexistant mais proposent une intégration dans JavaScript. C'est le cas de Scala [85] ou OCaml [61]. Dans plusieurs de ces techniques, la concurrence s'inscrit dans un modèle flot de données. Les éléments la constituant sont considérés comme des valeurs construites par applications successives de fonctions. Cette approche fonctionnelle de la programmation d'interfaces graphiques ne satisfait pas systématiquement les besoins, en particulier quand des effets de bord sont nécessaires, avec des interactions discrètes (e.g. un clic sur un bouton) plutôt que continues (e.g. les coordonnées GPS). Ces techniques de programmation seront présentées plus loin.

Une application pour le client Web est un système concurrent calculant un ensemble de valeurs de sortie à partir d'un ensemble de valeurs d'entrée à un rythme régulier. Du point de vue du programme, ce rythme dépend des itérations de la boucle d'interaction, qui s'exécute en continu, prend en compte les évènements entrants et traite leurs réactions séquentiellement. On peut donc dire qu'un programme client évolue sur le temps logique plutôt que sur le temps réel. Même la lecture d'une vidéo est interfacée par une suite d'évènements contenant un horodatage, auquel le programme peut réagir. Une itération de la boucle prend un temps dépendant du code exécuté en réaction aux évènements. On pourrait donc utiliser une machine à états dont l'exécution est calquée sur un temps logique. À chaque nouvel instant, la machine extrait, à partir de l'ensemble des entrées et de son état précédent, des valeurs de sortie. Cette représentation des programmes est très proche du modèle de programmation réactive-synchrone dont les qualités ne sont plus à démontrer.

Dans le cadre de cette thèse, nous proposons de subvenir aux besoins du programmeur en appliquant les pratiques du modèle synchrone-réactif au développement d'applications du client Web. La programmation synchrone-réactive apporte des constructions de haut niveau pour décrire la concurrence dans un programme, ainsi qu'une notion de communication au travers des signaux qui permet de représenter les entrées-sorties. En outre, l'ordonnancement automatique des tâches et les vérifications qui l'accompagnent font gagner au programmeur un temps précieux, qu'il pourrait allouer à d'autres problématiques. De plus, ce type de programmation rend la concurrence explicite en la calquant sur une abstraction du temps réel. Elle simplifie la composition de programme et facilite le réusinage de code.

Notre solution est un langage de programmation, nommé pendulum, qui se base sur le langage Esterel. Il est dédié à la programmation synchrone et s'intègre en tant qu'extension de syntaxe dans le langage OCaml. Il diffère des langages synchrones par une interopérabilité avec son contexte d'exécution, le client Web, et son modèle de concurrence, les évènements. Ce travail montre comment plonger un langage synchrone dans un langage généraliste, puis dans un navigateur Web, et comment profiter du modèle d'exécution de ce dernier calqué sur le modèle synchrone pour capturer les interactions efficacement. Avant de présenter notre solution, nous décrivons plus précisément le fonctionnement des applications Web, les systèmes sur lesquels elles reposent et l'état de l'art en matière de nouvelles technologies de programmation.

1.1 Programmation d'applications pour le client Web

Actuellement, la programmation Web client se définit principalement par l'interface que les navigateurs proposent au programmeur. Ces technologies, qui regroupent entre autres les langages HTML, CSS et JavaScript, sont spécifiées par le W3C. Avant de trouver une alternative à ce modèle, il convient de le présenter en détail et d'en montrer les failles. Un large panel de domaines est brassé, c'est pourquoi on démarre par l'introduction des termes concernant les modèles et les technologies de programmation en jeu.

1.1.1 Modèles et techniques de programmation

Les modèles de programmation

Un modèle de programmation, ou un paradigme [54], est une façon d'aborder l'expression d'algorithmes. On n'écrira pas un programme de la même façon dans le modèle fonctionnel ou dans le modèle impératif par exemple. Les langages sont généralement multi-paradigmes mais mettent souvent l'accent sur un modèle en particulier qui permet de donner une idée du style de programmation qu'un langage propose.

La programmation impérative est un modèle dans lequel le programme est une suite d'instructions exécutées séquentiellement. Chaque instruction produit une modification de la mémoire et donc un nouvel état. On appelle ce type d'opération un *effet de bord*, car l'instruction elle même ne fait pas que produire une valeur de retour, mais a aussi des conséquences sur la mémoire, les variables, etc. Les langages assembleurs, le langage C, Ada, ou Rust, par exemple, entrent dans cette catégorie.

La programmation fonctionnelle à l'inverse du modèle précédent agit par construction de nouvelles valeurs, et considère les valeurs crées comme *immuables*, c'est-à-dire non-modifiables. La brique de base du programme fonctionnel est l'expression, qui correspond à un calcul. Le nom de ce modèle est dû au

fait que l'on considère les fonctions comme des valeurs que l'on peut manipuler, passer en paramètre à d'autres fonctions, et les rendre comme résultat de calculs. Un langage purement fonctionnel, comme Haskell, refuse les effets de bord, les affectations de variables et utilise des méthodes d'encapsulation pour gérer les entrées-sorties. D'autres langages fonctionnels tendent à intégrer un certain nombre de fonctionnalités impératives pour des questions d'efficacité et de simplicité de programmation, comme la modification de la mémoire par effet de bord. Les langages des familles ML et Lisp sont de ceux-là.

La programmation orientée objets est un paradigme de programmation dans lequel les concepts sont représentés par des *objets*. Ces derniers sont des structures de données contenant des valeurs, que l'on appelle attributs et des fonctions que l'on appelle des méthodes. Les méthodes décrivent le comportement de l'objet, et les attributs décrivent son état interne. Les méthodes ont implicitement accès à l'état interne et peuvent le modifier. Les langages objets possèdent un langage de description des types objet, que l'on appelle classe. Ces dernières supportent une forme de sous-typage grâce à l'héritage permettant à une classe d'être sous-type d'une autre. De nombreux langages sont munis d'un modèle objet [87] : Java, C++, C#, JavaScript et Python.

La programmation concurrente permet d'exprimer une partie d'un programme comme un ensemble de tâches à effectuer simultanément. Il y a plusieurs types de concurrence. On peut parler d'une exécution entrelacée lorsqu'une tâche est mise en pause pour laisser une autre s'exécuter. Si une instruction ne peut pas s'interrompre pour laisser place à une autre, on dit qu'elle est atomique. C'est le cas en programmation *multi-thread*. On distingue deux cas : les modèles préemptifs, ou compétitifs, dans lesquels le changement de contexte se fait sur demande de l'ordonnanceur (ou *scheduler* en Anglais) du système qui exécute le programme. Le programmeur n'a pas la main sur ces changements et ils sont en majorité imprévisibles. A l'inverse, dans les modèles collaboratifs comme les FairThreads [11], les tâches passent la main volontairement en fonction d'une instruction du programme (souvent appelée `yield`). Une exécution simultanée peut aussi se faire sans interruption du fil d'exécution, dans le cas de la programmation parallèle, où plusieurs tâches peuvent se voir allouer chacune un noyau de calcul physique ou un fil d'exécution logique indépendant.

Le point commun de tous ces modèles, est que le comportement du programme est difficile à prévoir, et fait disparaître le déterminisme de l'exécution séquentielle. Dans les modèles collaboratifs, même si l'on peut prévoir le changement de contexte, on ne peut pas savoir à quelle tâche l'ordonnanceur va donner la main. En particulier, l'ordre dans lequel les tâches accèdent à la mémoire et aux données partagées est non-déterministe. Une dépendance implicite peut se créer entre deux tâches au travers de ce partage et ainsi causer des problèmes à l'exécution si on ne sait pas dans quel ordre l'accès aura lieu. On parle de situation de compétition, ou *race condition*, lorsqu'une tâche modifie une ressource pendant la lecture de celle-ci par une autre tâche. Ce genre de situation peut mener à de grosses incohérences et des bugs difficiles à reproduire. Pour s'assurer qu'une tâche accède à une ressource de façon atomique, on utilise en général des primitives de verrouillage forçant la synchronisation entre les processus. Les verrous mal utilisés peuvent provoquer des interblocages lorsqu'au moins deux tâches attendent réciproquement que l'autre libère une ressource, ou des famines, signifiant qu'une tâche devant accéder à une ressource n'a jamais la main sur celle-ci lorsqu'elle est disponible.

Ce type de comportement rend la programmation concurrente intrinsèquement difficile à concevoir et à mettre au point. Elle est néanmoins utile, voir indispensable, dans de nombreux cas, pour des raisons de performances, ou des problématiques de gestion d'entrées-sorties bloquantes (écoute dans un canal de communication, affichage à l'écran, etc.).

Typage et analyse statique

Pour vérifier qu'un programme est conforme à sa spécification, il existe plusieurs approches. Un moyen répandu est d'écrire manuellement ou de façon partiellement automatique, des tests et d'exécuter des sous-parties du programme dans le cadre de ces tests, en vérifiant que les sorties correspondent aux résultats prévus. L'écriture de tests est néanmoins très exigeante en temps de programmation et en lignes de code, et ne fournit en aucun cas une vérification complète, puisque c'est le programmeur qui décide de ce qui est testé. L'introduction de tests met aussi le programme dans un contexte différent que le contexte d'exécution et peut alors fausser son résultat. C'est d'autant plus vrai dans les programmes concurrents où il est très difficile de provoquer une exécution des tâches dans un ordre précis.

Dans le domaine des méthode formelles, on préfère prouver la validité du programme avant son exécution. Des outils de preuve assistée comme Coq [23], aux ateliers de raffinement de spécification comme la Méthode B [1], l'approche est de prouver et programmer à la fois, et de ne construire un programme exécutable que s'il est validé. Ces méthodes demandent une implication très importante du programmeur, ainsi que des connaissances avancées en informatique théorique et en logique. D'autres approches, permettent d'automatiser partiellement l'analyse statique :

- La vérification de modèle, ou *model checking*, cherche à infirmer l'existence d'un ensemble d'états d'erreur en parcourant l'ensemble des exécutions possibles du programme, souvent dans une représentation différente que le code source d'origine.
- L'interprétation abstraite se fait par l'exécution partielle, ou sur-approximée du programme, pour en extraire les propriétés.

Dans ces cas là on confronte deux entités séparées qui sont la spécification formelle et le programme. Si elles sont plus facilement automatisables, elles peuvent aussi ne pas réussir à trouver la réponse en un temps raisonnable ou détecter des faux positifs.

Enfin, la technique d'analyse statique la plus automatique, et donc la plus répandue dans les langages de programmation, est le *typage* qui est souvent embarqué avec les distributions de langages. Un type caractérise les valeurs manipulées par le langage. Le système de types est un ensemble de règles qui s'applique sur cette caractérisation en décrivant quelles sont les constructions du langage autorisées, dont le calcul ne mènera pas à une erreur. Si une expression du langage est bien typée, alors elle pourra être exécutée sans erreur du point de vue du système de types. Ces expressions peuvent être par exemple de type entier, nombre flottant, liste ou chaîne de caractères. Si on considère que *successeur* est une fonction qui prend un entier et qui rend un entier, on veut pouvoir encoder cette information dans le système de types pour restreindre cette fonction et afficher une erreur de compilation dans le cas d'une mauvaise utilisation, comme *successeur*("*hello*"). Les langages dérivés de ML (SML [49], OCaml [44], etc) ou Haskell implémentent des systèmes de types riches, où toutes les vérifications de type se font statiquement, et au travers desquels il est possible de représenter des structures de données complexes et réutilisables. Le polymorphisme, par exemple, permet d'utiliser des types paramétrés par des types, ou des fonctions polymorphes, qui prennent en paramètre des valeurs de n'importe quel type à condition que la fonction n'implique pas une construction ou une utilisation du paramètre comme un type monomorphe. Il est alors possible de décrire des structures de données telles que les listes chaînées ou les tables de hachage, et de réutiliser la même structure quel que soit le type contenu dans la liste. De plus, ces systèmes de types stricts, comme OCaml ou Haskell, utilisent l'inférence de type, permettant au typeur de rechercher et générer des types adaptés aux expressions, et donc automatiser une partie de la spécification du code écrit.

L'expressivité d'un langage typé peut sembler réduite, car une partie des programmes qui s'exécutent sans erreur se verront tout de même refusés par le compilateur. De plus, l'expressivité du typage tel qu'il est implémenté dans les compilateurs de langages généralistes est réduite, pour que l'analyse statique

soit la plus automatique et la plus pragmatique possible. Néanmoins, le typage statique embarqué avec le compilateur a des avantages indéniables dans la programmation, à la fois dans la recherche et dans l'industrie. Il permet de détecter des erreurs simples mais très nocives et difficiles à détecter lorsque l'on débogue. Il permet donc d'éviter une partie des tests. C'est aussi un avantage pour le réusinage de code, sur des projets larges où la moindre modification d'une représentation de valeur peut avoir des conséquences à de nombreux endroits. Cela permet aussi d'avoir des informations de type avant l'exécution, pratiques pour l'environnement de programmation et la documentation. Pour finir, un langage avec un système de type riche pousse souvent le programmeur à réfléchir au code qu'il écrit puisqu'il doit être conforme à certaines exigences.

Cela concerne le typage dit *statique* mais il est aussi possible d'appliquer des règles de typage dynamiquement, donc pendant l'exécution du programme. Cette approche est à double tranchant, car si elle donne plus de souplesse au programmeur, elle dissimule aussi les erreurs de typage qui ne seront découvert qu'à l'exécution, et donc nécessite d'autant plus de tests. La programmation du client Web se repose sur ce genre de technologies, mettant en avant le prototypage logiciel rapide et l'accessibilité. On notera l'existence du typage graduel [58], qui permet à un système de faire coexister à la fois le typage statique et dynamique. Le programmeur est alors libre d'annoter les parties du programme pour lesquelles il requiert un typage statique.

Extensions de syntaxe et langages dédiés

La métaprogrammation est une faculté d'un langage permettant aux programmes de manipuler des programmes comme des valeurs et les exécuter. La réflexivité est la faculté d'un programme à faire de l'introspection, c'est-à-dire analyser et modifier son propre code dynamiquement. Les langages de la famille Lisp par exemple proposent des transformations de code via les macros, et des évaluations et créations dynamiques de programmes au travers des S-expressions. Certains langages statiquement typés proposent tout de même une forme d'introspection pendant l'exécution comme Java et d'autres une forme statiquement typée de la métaprogrammation, comme MetaOCaml [43].

Il existe aussi des macros dont l'application se fait à la compilation, pendant une étape que l'on appelle macro-expansion. Ces macros prennent en entrée une expression du langage, peuvent appliquer des transformations arbitraire sur cette dernière et rendent une nouvelle expression. C'est celle-ci que le compilateur du langage transformera. Ce modèle de macros apparaît dans la famille des langages Lisp, mais beaucoup d'autres langages de programmation implémentent leur propre approche. On repose dans cette thèse sur le moteur de macros PPX [44, Chapter 8 Language extensions] du langage OCaml, qui permet de lier au compilateur des programmes de transformation d'expressions, décrites en OCaml exécutés avant l'étape de typage. La contrainte de PPX est que le code généré est représenté comme un arbre de syntaxe abstraite d'OCaml. On ne peut donc pas construire de nouvelles constructions syntaxiques dans ce moteur de macros.

Ces différentes techniques ont plusieurs intérêts. Premièrement, l'adaptabilité automatique du programme à certains changements de contexte est plus aisée lorsque le programme peut modifier son propre comportement. Ensuite, grâce à l'utilisation des macros, on a la possibilité d'automatiser certaines tâches répétitives de programmation, comme par exemple générer automatiquement une fonction de lecture pour un type précis, construit à partir de types connus. C'est le but de l'extension Deriving [91] basée sur PPX par exemple, qui permet de générer automatiquement et statiquement le code dépendant d'un type créé par l'utilisateur : l'égalité entre deux valeurs de ce type, une fonction d'affichage, la conversion dans un format binaire, etc.

Il est aussi possible de séparer complètement le contexte d'évaluation de certaines parties du pro-

gramme par annotation [55]. Par exemple, dans Eliom [4], une plateforme pour le développement Web en OCaml, l'extension de syntaxe permet de séparer le code client du code serveur. Le premier est compilé vers JavaScript via Js_of_ocaml et le second est compilé par le compilateur code-octet ou code-machine et intégré au serveur Web de l'application. Malgré tout, ces parties sont écrites dans un même fichier et interagissent lors du typage. Cette méthode d'extension du langage permet aussi de relier automatiquement les parties connectées entre les deux contextes en générant les identifiants des éléments de la page du client depuis le serveur ou même les communications pour accéder à une variable du serveur depuis le client. En plus de gagner en productivité, automatiser permet donc de générer du code cohérent qui sera le même partout et limiter les erreurs. Ces exemple ne sont évidemment pas exhaustifs et on peut citer de nombreuses autres utilisations similaires des macros statiques.

Le dernier avantage que nous citerons permet de résoudre un problème différent : un langage de programmation est dit généraliste quand il est utilisé pour résoudre toute sorte de problèmes, du logiciel système aux applications mobiles en passant par les programmes distribués. Ces langages sont généralement multi-paradigmes : impératif, fonctionnel, objet, concurrent, etc. Néanmoins, si l'on prend ces langages sans ajout, il leur manque en général les fonctionnalités pour gérer au mieux des problématiques spécifiques avec tout le support nécessaire. Par exemple, un langage généraliste peut donner l'accès des bibliothèques permettant de se connecter à des bases de données variées et le programmeur peut écrire sa requête SQL au format chaîne de caractères. Malheureusement, le compilateur est incapable de détecter si la syntaxe de la requête est correcte ou non. Celle-ci n'utilise pas de constructions du langage, empêchant compilateur de raisonner sur cette dernière. On utilise alors un langage dédié (ou DSL, pour *Domain Specific Language*) qui peut parfois être écrit dans le même fichier de code avec un marquage syntaxique explicite. Lorsque que le compilateur du langage hôte rencontre ce marqueur, il passe la main au moteur d'extension syntaxique qui vérifie la bonne formation des requêtes et pourrait éventuellement les optimiser. Par exemple, la bibliothèque PgOCaml [81] propose ce genre de support en OCaml. On peut aussi citer l'extension de syntaxe de Js_of_ocaml [79], qui permet de gérer les constructions propres à JavaScript en simplifiant l'intégration avec le langage cible. Les extensions de syntaxes sont couramment utilisées en JavaScript aussi, où elles permettent à l'utilisateur de construire les éléments graphiques personnalisés avec une syntaxe balisée comme HTML, comme dans React.js.

1.1.2 Programmation du client Web

Le navigateur est un programme qui récupère des ressources présentes sur le Web, les interprète, et les affiche dans une fenêtre graphique. La localisation d'une ressource est précisée au moyen d'une adresse URI, [1], décrivant le chemin pour y accéder. Le transfert est effectué via le protocole HTTP [2]. On parlera ici de ressources sous forme d'applications. Elles sont généralement composées d'un ensemble de documents qui décrivent la structure de la page en langage HTML [3] accompagnée de ses feuilles de style en langage CSS [4] et d'un ensemble de scripts interactifs en langage JavaScript. Ces différents composants d'une application sont fournis par le serveur en fonction du contenu de la requête HTTP d'origine. Une application Web peut alors être exécutée sur la majorité des terminaux du parc informatique mondial : ordinateurs personnels, téléphones intelligents, télévisions, objets connectés, etc. Elle est le moyen le plus facile pour présenter à la fois un logiciel et du contenu à n'importe quel utilisateur.

Il existe plusieurs méthodes pour décrire la partie client d'une application Web. La solution standard

1. Uniform Resource Identifier
2. HyperText Transfert Protocol
3. Hypertext Markup Language
4. Cascaded StyleSheets

est la combinaison des langages cités ci-dessus. La spécification de ces langages est décrite précisément par les normes du W3C [33] et EcmaScript [25]. L'implémentation du moteur d'exécution et de rendu peut néanmoins différer selon les navigateurs et leur version, avec des conséquences sur le comportement du programme.

JavaScript est un langage typé dynamiquement, décrit par le standard EcmaScript. Il est impératif, fonctionnel et objet. Sa syntaxe rappelle le C et le Java. Néanmoins, il tire certaines de ses propriétés des langages fonctionnels comme les fonctions d'ordre supérieur. Le typage des valeurs est très permissif et le langage est muni d'un modèle objet à base de *prototypes*, c'est-à-dire dans lequel il n'y pas de classe de référence et seulement des objets héritant des propriétés de leurs parents. Ce langage est plutôt utilisé dans le contexte du Web, la programmation client de la majorité des navigateurs modernes, mais aussi dans la programmation des serveurs Web avec Node.js [75]. Son utilisation se généralise cependant, grâce à l'efficacité croissante de ses implémentations. Les moteurs d'exécution actuels utilisent la compilation *juste-à-temps* (JIT pour *Just in time* en Anglais), compilant le code source vers du code-octet, interprété par une machine virtuelle, comme dans les navigateurs Firefox [77], Safari [76] et Edge [74] ou vers du code machine comme dans le navigateur Google Chrome [89].

Le DOM ou Document Object Model [38], est une interface de programmation mise à disposition par les navigateurs, indépendamment du langage de programmation, pour manipuler l'interface graphique, et les entrées-sorties du programme client. Le *document* représente la fenêtre du navigateur sous la forme d'un arbre d'éléments graphiques et de conteneurs hiérarchisés. Les modifications appliquées à cette structure de données sont directement propagées dans la vue du navigateur.

1.1.3 Modèle d'exécution du client Web

Les programmes exécutés par le client Web présentent par nature un grand nombre d'interactions et sont majoritairement dirigés par ces dernières. La majorité du code s'exécute en réponse aux entrées du programme. L'interface de programmation DOM met à disposition des outils pour répondre à ces entrées du programme sous la forme de programmation évènementielle, ainsi que le nécessaire pour manipuler les sorties du programme, à savoir l'interface graphique, et les communications.

Programmation évènementielle et boucle d'interaction

Les entrées du programmes capturées par le navigateur, les évènements, sont toujours déclenchés par une cible. Cette cible peut être un élément graphique, comme un bouton ou la fenêtre principale. Elle peut aussi être un objet en mémoire qui représente une entrée globale que le navigateur reçoit : la position géographique, l'accéléromètre, une connexion réseau. Un composant peut être la cible de plusieurs types d'évènements, comme la *fenêtre du programme* qui peut être la cible du *déplacement de la souris*, ou du *clic de souris*. Pour désigner un couple formé par un composant cible et un type d'évènement, on parlera de comportement évènementiel. Un évènement est donc une occurrence particulière d'un comportement, et il est représenté par une valeur composée de champs multiples. Un même comportement peut générer plusieurs occurrences qui sont des valeurs distinctes du programme.

Le navigateur implémente une boucle infinie, que l'on appelle une *boucle d'interaction*, qui lit ces valeurs et exécute la réaction prévue. La réaction est différée par rapport à l'apparition de l'occurrence du comportement qui en est à l'origine. Il est donc impossible de prévoir dans quel ordre seront traitées

ces différentes entrées si elles arrivent au même moment. C'est pourquoi on parle de comportement asynchrone concurrent.

Les navigateurs Web [39] implémentent ce genre de programmation concurrente. Pour faciliter l'accès au programmeur, toute l'exécution est basée sur les évènements et il n'est pas possible d'avoir accès au code de la boucle d'interaction, ni à la file où sont enregistrés les évènements. L'interface de programmation DOM donne toutefois, pour simplifier, la possibilité d'enregistrer une fonction *de rappel*, ou *callback* pour un comportement évènementiel. En JavaScript, on peut déclarer une réaction à un comportement évènementiel de la façon suivante.

```
function f () { alert ("Hello world"); }
document.getElementById("btn").addEventListener("click", f);
```

On définit la fonction `f` qui affiche une alerte à l'écran. On accède à l'élément ayant `"btn"` pour `id` dans le DOM, et on y affecte la fonction `f` comme gestionnaire de l'évènement nommé `"click"`. Lorsque le navigateur, qui a la charge de l'interface bas niveau, reçoit l'information du matériel, un message est ajouté dans la file avec l'évènement correspondant à l'entrée-sortie. A chaque itération de la boucle d'interaction, la file des évènements est vidée. Pour chaque évènement qu'elle contient, on parcourt l'arbre DOM à la recherche de l'élément qu'il cible depuis la racine. Si la cible est atteinte, la recherche s'arrête, on appelle le gestionnaire (ici `f`), puis on remonte dans l'arbre, jusqu'à la racine en déclenchant tous les gestionnaires correspondant à cet évènement que l'on rencontre. On rappelle que l'exécution des gestionnaires d'évènement est atomique, c'est-à-dire que la fonction `f` est exécutée jusqu'à ce qu'elle termine. Les préemptions sont évitées pour minimiser les risques de situations de compétition.

Mise-à-jour de la vue graphique

La mission principale du navigateur est d'afficher le contenu choisi par l'utilisateur dans la fenêtre graphique. Cette tâche est effectuée par le moteur de rendu, déclenché directement après une modification du DOM par les scripts en exécution. On peut observer dans le schéma 1.1, que le moteur de rendu est appliqué en plusieurs passes. Une première, le *reflow*, construit un arbre intermédiaire (arbre de rendu) qui décrit sous forme de boîtes l'imbrication des composants et leurs tailles. Une deuxième, le *redraw*, transpose cet arbre dans une représentation sous forme de pixels.

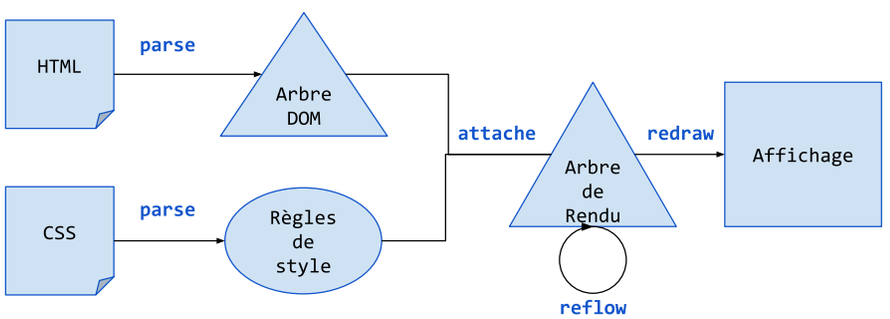


FIGURE 1.1 – Flux de construction de la page Web

Quand un élément de la page est modifié, les deux tâches n'ont pas lieu systématiquement. Certaines actions peuvent être appliquées directement sur les pixels parce qu'elles n'ont qu'un impact faible sur la

vue graphique (comme l'opacité). Dans ce cas aucune des deux fonctions n'est appelée. De même, une modification visible pourrait impacter l'affichage sans modifier les tailles des boîtes du *render tree*. On appellera la tâche *redraw*. L'exécution de ces deux fonctions peut être coûteux, en particulier *reflow*, et le travail du programmeur est, entre autre, de minimiser les appels à ces dernières.

Il est possible de volontairement repousser la modification du DOM par la fonction de rappel. Si plusieurs modifications de la page sont appliquées à la suite, le navigateur déclenchera plusieurs mises-à-jour et appels à la fonction *redraw* de façon forcée. Au lieu de ça il est possible d'utiliser la fonction d'ordre supérieur `requestAnimationFrame` permettant de repousser l'exécution de la fermeture passée en paramètre au prochain *redraw*. L'appel à *redraw* est de toute façon nécessaire, sachant que le navigateur rafraîchit l'affichage à une fréquence de 60hz. Le programme laisse ainsi au soin du navigateur le choix du moment de ce rafraîchissement, pour éviter d'être obstrué par les calculs graphiques et perdre en temps d'exécution.

1.1.4 L'orchestration des interactions

Comme expliqué en introduction, la partie client d'une application est sujette aux interactions avec le monde extérieur. Une des difficulté dans l'écriture d'un programme client est la gestion de ces interactions, d'origine variées.

Le multimédia riche est un terme qui désigne des applications Web modernes, centrées sur le client, où l'expérience utilisateur est devenue un problème clef. Ces applications peuvent mêler plusieurs canaux d'interactions, comme de la vidéo, de la discussion instantanée, des documents PDF, du texte brut, des animations, qui communiquent parfois entre eux.

Les difficultés d'orchestration Rien que pour s'initialiser, une application a déjà exécuté un certain nombre de requêtes sur le réseau. Certaines de ces données d'initialisation peuvent dépendre les unes des autres : synchronisation du stockage local en fonction de l'authentification, récupération d'un contenu dépendant de la géolocalisation ne sont que quelques exemples où de multiples évènements doivent être pris en compte.

L'application peut se trouver dans différents états avant de compléter une tâche, donc si l'un de ces comportements asynchrones génère une erreur, comme la perte de connexion ou du signal GPS, le cas doit être traité selon les différents états de l'application. Ces erreurs sont monnaie courante quand il s'agit d'un terminal mobile par exemple. Or, la concurrence basée sur les évènements est un frein à la composabilité de ces comportements. Le programmeur est forcé de passer par des variables globales pour partager l'information entre les différentes fonctions de rappel, alors que la consistance de cet environnement global n'est pas du tout assuré. Décrire le système comme des réactions à des évènements indépendants ne pousse pas l'utilisateur à se poser les questions importantes : quelle est la relation de dépendance entre mes données, dans quel ordre doivent se passer les choses, et comment doit réagir mon programme si cet ordre n'est pas respecté ?

1.2 De nouvelles technologies de programmation pour le client Web

L'attrait de la programmation pour le client Web repose sur une apparence de simplicité, où le programmeur peut écrire le prototype d'une petite application en quelques lignes de code et la mettre en production. Cette situation apparaît pour plusieurs raisons, mais en particulier parce que ces technologies sont assemblées non pas pour des considérations techniques ou théoriques à l'origine, mais grâce à

leur assimilation par le plus grand nombre. HTML semble accessible pour un non-initié et JavaScript est très flexible. Cette simplicité n'existe qu'en apparence, car en vérité, plus l'application grossit et plus les complications rencontrées sont nombreuses. Cette flexibilité qui semble au départ être une qualité devient un défaut car il devient plus difficile de détecter les erreurs dans son programme.

Le réusinage de code devient aussi une opération risquée, à moins d'écrire de nombreux tests, délicats à mettre en place dans le cas d'une application graphique, avec de nombreuses entrées-sorties, ou pouvant être exécuté sur de nombreux supports. Pour palier à cela, il y a eu énormément d'efforts de la part des communautés Web, de l'industrie et de la recherche dans le domaine pour proposer de nouvelles approches et une programmation plus haut niveau que le standard du W3C, introduit dans la sous-section 1.1.2. Ces approches portent à la fois sur l'expressivité, la sûreté et le modèle de concurrence.

1.2.1 Pour la sûreté et l'expressivité

Les langages de programmation typés statiquement

L'idéologie de JavaScript est d'accepter le maximum de programmes syntaxiquement corrects et d'y trouver une sémantique pour laquelle l'exécution se passe sans erreurs, autant que faire se peut. Cela peut mener à deux problèmes évidents. Premièrement, il est possible de faire une erreur de typage, malgré les efforts du système, et celle ci ne peut être détectée qu'à l'exécution (comme par exemple l'accès à membre d'un objet indéfini). Deuxièmement, si l'interprète est capable de trouver un sens à un programme, il ne sera pas toujours celui souhaité par le programmeur. Celui-ci n'a pas forcément pris le temps de lire la sémantique exacte et la définition du langage, voir des différentes versions de ce dernier : EcmaScript 4, 5, 6, les différences entre les navigateurs, etc. On ajoute donc un facteur d'imprévu dans la logique de programmation.

Remarque. *Par exemple, tous les programmeurs JavaScript ne seront pas capables de répondre à la question : « Que vaut* `(new Array(3) == ",,")` *? ». L'opérateur* `==`*, l'égalité faible, converti automatiquement une des deux opérandes pour qu'elles soient toutes les deux du même type. Ici le tableau est converti en chaîne de caractères. Cette opération convertie chaque élément du tableau en chaîne et les sépare par des virgules. Hors, le tableau est vide mais de taille trois, donc deux séparateurs apparaissent dans la chaîne :* `",,"`*. L'évaluation de cette expression vaut donc* `true`*. On ne détecte pas d'erreur mais le comportement ne sera pas celui attendu, à moins de bien connaître la documentation. En résumé, l'absence de typage statique rend la sémantique difficile à comprendre en plus de laisser passer les erreurs.*

C'est pour éviter ce genre de problèmes que les nouvelles approches en faveur d'un typage statique du langage de programmation client apparaissent dans l'industrie et la recherche. On peut citer TypeScript [88], un JavaScript avec des éléments de typage, ou Flow [70], un outil de vérification de types pour ce même langage. D'autres initiatives proposent de passer au delà de JavaScript en le considérant comme un "assembleur du client Web", et de concevoir un compilateur d'un langage nouveau ou préexistant vers JavaScript. Il y a de nombreux exemples, mais on peut parler en particulier de ScalaJS [85] et surtout Js_of_ocaml [61], un compilateur de code-octet OCaml vers JavaScript sur lequel repose le travail présenté dans cette thèse. Js_of_ocaml apporte, en plus du compilateur, une interface pour utiliser la bibliothèque DOM de façon typé. Grâce à cet outil il est possible d'utiliser tout le code OCaml existant ou bibliothèque compilée en code octet vers un programme client Web, à condition qu'il ne fasse pas appel du code C externe ni ne fasse d'hypothèse sur la machine d'exécution et son matériel. Il existe d'autres exemples de compilation d'OCaml vers JavaScript, comme BuckleScript[64], en particulier utilisé dans le cadre du langage ReasonML [83], une variante syntaxique d'OCaml orientée pour les programmeurs Web. BuckleScript propose une compilation de source à source, dans le but de faciliter la lecture et le

débogage du code généré. Une autre approche est d'embarquer dans le client, écrit en JavaScript, la machine d'exécution du langage, comme le fait O'Browser[17] avec l'interpréteur de code-octet d'OCaml. Dans ces travaux, il est souvent question d'utiliser des bibliothèques JavaScript depuis le code OCaml. Ce problème est généralement résolu par l'utilisation d'un langage de FFI (Foreign Function Interface) [6] permettant de décrire dans le langage des types d'OCaml l'interface de la fonction externe.

D'autres projets visent à améliorer le support du langage JavaScript, notamment sa spécification. La documentation EcmaScript [25] est exhaustive, mais loin d'être formelle. C'est pourquoi ces projets proposent de nouvelles sémantiques pour le langage, comme des sémantique opérationnelles [45], parfois au travers de versions minimales du langages [40]. Certaines spécifications formelles sont plutôt appliquées sur le DOM [35], pendant que d'autres approches s'intéressent à la sémantique de mise-à-jour du document [16].

Les modèles Web multitier

On divise généralement l'architecture d'une application Web en plusieurs parties qui résolvent différentes problématiques : l'affichage à l'utilisateur, la logique et les règles qui régissent les données, comme leurs droits d'accès par exemple ou encore le stockage de ces données. On dit d'une application qu'elle est *3-tier*, ou trois niveaux, si elle est composée de

- La présentation : ici le client, généralement décrit en JavaScript et HTML
- Le traitement métier : la logique encodée dans le programme serveur (Node.js, PHP, etc), et qui répond aux requêtes
- L'accès aux données : le logiciel de gestion de base de données

Le nombre de couches peut augmenter arbitrairement selon les choix d'architecture. Cette division a plutôt pour origine une considération technique, de langage et de localisation du code. Elle avait aussi du sens alors que le travail de logique et de présentation, c'est-à-dire générer le HTML, était sur le serveur, et que le client avait plutôt un but de présenter les données sans interactions. Aujourd'hui, l'intelligence se situe de plus en plus dans le client, et le serveur a une mission d'extraction des données depuis la base pour les fournir aux clients logiciels dans le format attendu.

Ce découpage tend d'ailleurs à séparer des parties du code sémantiquement connectées et à les regrouper avec d'autres n'ayant aucune relation. Les communications doivent être redéfinies explicitement de par et d'autre du client et du serveur. Écrire le code deux fois peut entraîner des différences entre les deux descriptions d'interface et donc des erreurs à l'exécution difficile à détecter. Ce découpage propose généralement l'utilisation de plusieurs langages, cassant la cohérence de l'application, demandant un changement de contexte pour le programmeur et de multiple compétences, tout en dupliquant obligatoirement le code commun entre client et serveur.

C'est pour répondre à ce problème que la programmation que l'on appelle ici *multitier*, propose de réunir l'application en un seul programme, utilisant le même langage en partageant l'environnement. Décrire une seule fois une communication permet d'assurer une cohérence entre le client et le serveur en empêchant les erreurs de formatage et de typage des données. La multiplication des traductions de données d'un format à l'autre pour traverser les différents systèmes et représentations demande des ressources de développement supplémentaires. De plus, certaines partie du code, similaires entre client et serveur, peuvent être automatiquement factorisées. La programmation multitier donne aussi la possibilité de partager un environnement d'exécution et d'accéder à des valeurs du serveur côté client et inversement, voir même entre les clients, le tout sans faire l'effort ni prendre le risque de décrire la communication explicitement. Pour finir, utiliser un unique langage permet au programmeur de n'utiliser qu'un ensemble

restreint d'outils de programmation, une seule base de connaissance et une seule logique de programmation à la fois pour le client et le serveur. Fort de tous ces avantages, plusieurs technologies ont fait le choix de la programmation multitier, en particulier dans les initiatives de recherche.

Le projet OCsigen [4] est un ensemble d'outils pour le développement d'applications Web. Il contient d'abord Js_of_ocaml, qui rend possible l'exécution de programmes OCaml dans le navigateur, grâce à une compilation du code-octet vers JavaScript. On trouve aussi Tyxml, bibliothèque contenant des fonctions permettant de décrire et générer du HTML valide, vérifié par le typage du langage. Lwt [60] est la bibliothèque de concurrence par défaut et propose des fils d'exécution coopératif avec une interface de programmation monadique. On citera pour finir Ocsigenserver qui prend le rôle du serveur Web. Toutes ces technologies sont regroupées dans l'outil principal Eliom [5] et son langage de programmation spécifique [52][53], qui est un atelier de programmation d'applications client-serveur multitier. En plus de ces outils, Eliom, propose une extension de langage localisant explicitement le code : client, serveur ou partagé. La plateforme le compile ensuite automatiquement vers la bonne cible, et génère les identifiants pour accéder aux valeurs du client depuis le serveur et inversement. Les références aux variables partagées sont compilées automatiquement vers des communications HTTP, quand celles-ci sont nécessaires. Le typage des valeurs est appliqué globalement sur le code client et le code serveur avant la compilation, on est donc assuré de la validité des formats d'échange. Il est aussi possible d'appeler des fonctions distantes du serveur depuis le client (RPC, *Remote Procedure Call*) en précisant la structure des transportées.

Du côté client, Js_of_ocaml propose une interface de programmation pour le DOM dans le langage des types d'OCaml et du sucre syntaxique [5] qui permet d'accéder à des opérations propre au langage cible comme

- L'accès à un champ d'un objet : `b##.value`
- La mise-à-jour d'un champ : `b##.value := Js.string "myval"`
- L'appel de méthode : `video##play`

Le modèle objet de JavaScript, sous forme de prototypes, est donc simulé dans celui d'OCaml qui est structurel ave. Les objets JavaScript sont représentées par le type `Js.t` qui est abstrait et certaines valeurs nécessitent un changement de type explicite, comme les chaînes de caractères, mutables en OCaml et non-mutables en JavaScript, ou les booléens. L'interface de programmation reproduit au maximum la définition du DOM où les changements sont des conséquences du système de types. Par exemple :

1. `document.createElement("button")`

2. `Dom_html.(createButton document)`

La version Js_of_ocaml (2) force à typer statiquement l'élément créé. S'il est ensuite utilisé dans le code, il ne pourra fournir que les méthodes et champs d'un élément de type `buttonElement`, et le typeur pourra relever les utilisations incohérentes. A l'inverse, le code JavaScript, ne permet pas de typer statiquement parce que le choix de l'élément se fait par une valeur du programme calculée dynamiquement (ici une chaîne de caractères). La seule information de type est donc que c'est un élément, jusqu'à ce que le code soit exécuté. L'interface de programmation de Js_of_ocaml exhibe une fonction pour chaque type d'élément existant dans la spécification du DOM.

5. L'opérateur ## est utilisé par proximité avec #, l'appel de méthode en OCaml

Links [22] est un langage de programmation créé pour servir les besoins de la programmation multitier. En plus de proposer, comme Eliom, du typage et une cohérence plus forte entre client et serveur, une partie du langage est aussi dédiée à l'accès à la base de données. Une partie du code est compilée vers SQL, une autre vers le client et une troisième vers le serveur. Contrairement à Eliom, seule les fonctions sont localisées comme étant côté client ou côté serveur. Le langage inclut aussi la syntaxe XML pour décrire les interfaces. Links propose aussi un véritable modèle de concurrence dans le client Web, au travers d'un modèle par passage de message, inspiré des langages Erlang [3] et Mozart [2].

HOP [56] est un langage dérivé de Scheme pour la programmation d'applications Web multitier. De même que Eliom, il propose une syntaxe particulière pour localiser le code entre client et serveur, et des références mutuelles entre les deux entités. Les services Web sont des valeurs de première classe du programme au même titre que les fonctions. Néanmoins, là où Eliom tend à vouloir gommer complètement les tiers, HOP les conserve plus explicitement en considérant le programme client comme une valeur calculée par le programme serveur, à la manière d'une S-expression en Scheme. Une fonctionnalité spécifique à HOP est de pouvoir envoyer un programme sur le client depuis le serveur construit dynamiquement. La plateforme HOP se décline aussi en HOP.js et est complètement compatible avec JavaScript [57] avec une implémentation exhaustive de la bibliothèque standard de Node.js pour le côté serveur. Pour donner une idée des possibilités de HOP, voici un programme, `hello`, qui crée un service Web accessible depuis l'URL `/hello?who=<...>`.

```
service hello( { who: "foo" } ) {
  return <div onclick=~{ alert( "Hi " + ${who} + "!") }>
           hello
         </div>;
}
```

Ce programme est appelé côté serveur et la valeur `~{ alert( "Hi " + ${who} + "!")}` est un meta-programme qui doit s'exécuter dans le client lorsque l'utilisateur clique sur le div défini en syntaxe HTML. Ce meta-programme contient un meta-programme de niveau deux, `${who}`, exécuté sur le serveur et contenant la valeur `who` reçue dans les paramètres de l'URL.

UR/Web [20] est un langage de programmation, dérivé de Ur, fonctionnel et statiquement typé. Comme les technologies précédentes, celui-ci se base sur une unification du client et du serveur sous la forme d'un seul programme, permettant l'appel à des RPC. La mise-à-jour des éléments de la page se fait par propagation de l'état du serveur vers le client à travers un modèle de signaux réactifs.

1.2.2 Pour l'orchestration des interactions

Comme dit dans la sous-section 1.1.4, le modèle de concurrence proposé par l'interface DOM est trop simple pour gérer les interactions de façon expressive. Comme dans la partie précédente, de nouvelles approches issues de l'industrie, de la recherche et du logiciel libre proposent des alternatives. Toutes ces approches reposent sur l'exécution purement séquentiel du moteur JavaScript et sur la nécessité de communiquer, au moins implicitement, avec la couche d'évènements de l'API DOM.

MVC et au delà

Avec les outils standards de programmation du client Web, le script JavaScript modifie la page Web décrite sous forme d'arbre en mémoire par effets de bord.

```
var button = document.getElementById("button");
button.style.visibility = "hidden";
```

Le code ci-dessus, permet par exemple d'accéder à un élément par son identifiant, puis de modifier une propriété, ici sa visibilité. On connaissait depuis longtemps les bibliothèques JavaScript comme JQuery [71], qui augmente la bibliothèque standard et ajoute du sucre syntaxique pour manipuler le DOM de manière concise, en écrivant plutôt

```
$("button").css.("visibility", "hidden");
```

Les outils plus riches proposent une méthode au programmeur pour organiser son application : le MVC (Modèle-Vue-Contrôleur). Le modèle est la couche qui gère la représentation des données en mémoire et les calculs. La vue est une représentation de l'affichage graphique, et le contrôleur administre les entrées du programme. Les outils MVC que l'on rencontre couramment sont Backbone [63], Angular [62] ou Ember.js [68]. Ils aident à lier les composants entre la page HTML et le script qui s'exécute de façon transparente. La page devient une description à *trous*, où ceux-ci représentent des variables accessibles directement depuis le script. Par exemple le code Angular suivant lie une valeur du modèle appelé `name` à un champ de texte. Cette valeur va évoluer lorsque l'utilisateur écrit dans ce champ de texte. Ensuite la syntaxe `{{ name }}` va automatiquement afficher la valeur `name` dans le paragraphe lorsqu'elle évolue.

```
<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p>You wrote: {{ name }}</p>
</div>
```

On appelle cette technique du *data binding*. Elle permet au programmeur de s'affranchir de l'écriture de la fonction de mise-à-jour graphique. D'autres constructions existent permettant d'itérer sur une structure de données et de l'afficher en formattant son contenu.

La bibliothèque React.js propose de changer à nouveau les « bonnes pratiques » en supprimant les modifications explicites de la page par effet de bord. A la place, on décrit des composants spécifiques ainsi que la façon dont ils se comportent face à un évènement. Quand le modèle est modifié, les composants de la page se modifient alors automatiquement. On peut voir cela comme une forme de programmation flots de données. React.js gère plutôt l'équivalent de la vue, et plusieurs possibilités s'offrent au programmeur pour décrire l'état de l'application qu'est le modèle, comme Redux [84].

La programmation réactive

Un programme réactif interagit en continu avec son environnement. Il décrit un calcul qui génère des données en sortie à partir des données en entrée. La programmation réactive peut avoir plusieurs approches, fonctionnelle en flot de données, ou impérative. En programmation fonctionnelle réactive [31] (FRP), la progression du calcul s'effectue par applications successives de transformations sur les données d'entrée et non par mutation. Les constructions sont alors uniquement des expressions et valeurs,

comme les éléments de la page, sont des flots qui évoluent au cours du temps. Quand la valeur d'un flot change, la mise-à-jour se propage vers les calculs qui en dépendent. Un programme fonctionnel réactif peut être vu comme une structure de graphe de flots où les nœuds sources sont les valeurs en entrée, les nœuds intermédiaires sont les calculs, et les nœuds terminaux sont les sorties du programme. Cette forme de modèle réactif peut être implémenté en tant que bibliothèque de n'importe quel langage généraliste comme OCaml avec React [65] ou Scala avec Scala-react [46].

Ces modèles de programmation en flots de données sont hérités des réseaux de Kahn [42] et de la programmation de circuits électroniques. Certains langages synchrones comme Lustre [19] implémentent ce type de programmation réactive en flots de données. En Lustre, l'accent est mit sur la validité du programme vérifié statiquement. On souhaite s'assurer que la relation de causalité entre les entrées et les sorties est cohérente mais aussi pouvoir utiliser des outils d'analyse statique [41] pour vérifier que le programme est conforme à sa spécification. Le travail de ces outils est facilité par la représentation sous forme de circuit séquentiel des programmes Lustre. Une vision similaire dans un style impératif trouve ses racines dans la programmation synchrone dérivée d'Esterel [9]. Les langages synchrones étant un sujet majeur de ce travail, la sous-section 1.3.1 est destinée à leur présentation.

Les idées de la programmation réactive se sont répandue dans les communautés de développement Web, grâce notamment à FlapJax [48], Elm [24] et React.js. Ce dernier en proposent une version affaiblie centrée sur la mise-à-jour de la page. Ces langages montrent qu'il est possible de se passer de la programmation évènementielle pour capturer les interactions propres à la programmation client Web. Elm propose par exemple un langage à part entière inspiré d'Haskell adaptant entièrement l'interface DOM avec une représentation FRP, rendant le tout cohérent et élégant.

Représenter la page Web par des données continues, mises à jour par propagation, est adapté à certains comportements. Par exemple un champ de texte possède continuellement une valeur donc sa définition et les éléments qui en dépendent sont faciles à décrire dans un modèle fonctionnel réactif. A l'inverse certains comportements ne correspondent pas à une valeur, comme un bouton sur lequel on clique, mais plutôt à un déclenchement d'action. On peut évidemment le représenter avec une valeur booléenne, mais si on veut capturer séparément l'appui et le relâchement, on est obligé de donner deux informations pour chacun de ces évènements et on doit affiner en spécifiant quelle valeur produira un déclenchement. Il manque donc une notion simple de déclenchement dans ce type de programmation, notion que l'on trouve plutôt dans les représentations du type machine à états. C'est d'ailleurs la conclusion à laquelle en sont venus les auteurs de Elm d'après leurs récentes communications concernant la mise-à-jour `0.17` de leur langage. Cette dernière modifie les constructions du langage pour ne plus parler de signaux, mais de *subscriptions*, c'est-à-dire des comportements auxquels auquel le programme réagit. Ils se placent donc à mi-chemin entre le flot de données et la machine à états, à la manière du langage Lucid Synchrone [18].

1.3 L'approche proposée par cette thèse

Dans la section 1.1, nous avons introduit les standards de programmation pour le client Web, avec en particulier le langage JavaScript et l'interface de programmation DOM. Nous avons montré que, malgré l'apparente simplicité d'utilisation de ces techniques, le problème à résoudre est complexe. Si JavaScript et le DOM sont efficaces pour décrire un prototype d'application client avec très peu d'interactions, il devient rapidement difficile d'ajouter de nouveaux comportements sans complexifier l'architecture du code et y ajouter des erreurs. L'absence de vérification statique des types dans le langage est une des origines de cette situation, mais c'est surtout le modèle de programmation concurrente qui est à mettre en cause : trop faible pour exprimer un programme avec des dizaines d'évènements dont les réactions sont

interdépendantes, il est inadapté à la description d'une machine à états complexe. Dans la sous-section 1.1.4 sur l'orchestration, on explique que le seul moyen de composer les réactions issues de plusieurs évènements est d'utiliser un environnement de variables globales décrit et mis à jour manuellement. On considère cette pratique de programmation mauvaise car source d'erreurs. On ne peut pas s'assurer que les données de l'état courant sont cohérentes et que l'environnement est correctement mis-à-jour entre chaque réaction aux évènements.

Section 1.2, nous avons présenté plusieurs approches récentes pour améliorer la situation proposée par le W3C et les navigateurs. Il y a d'abord celles qui proposent d'ajouter à JavaScript de nouvelles constructions ou vérifications en s'inspirant d'autres langages, comme du typage statique, ajoutant à la fois de l'expressivité et de la sûreté. Certaines approches partent directement de JavaScript, et d'autres préfèrent utiliser un langage complètement différent compilable vers le client Web. Nous avons ensuite présenté différentes techniques de programmation concurrente qui tentent d'améliorer les conditions actuelles. La part belle est faite aux modèles qui intègrent ensemble mise-à-jour et description graphique, comme le modèle fonctionnel-réactif, permettant d'écrire de façon concise le procédé de construction et de mise-à-jour des éléments, avec une forme de propagation des données depuis les évènements en entrée, vers l'affichage de la page. On explique aussi pourquoi ce modèle ne nous convient pas complètement : parce qu'il lui manque la possibilité de décrire une machine à états, plus proche du modèle d'une application Web graphique.

On souhaite donc proposer une nouvelle approche, qui pourra concilier la représentation sous forme de machine à états avec la manipulation des valeurs. Ce langage doit pouvoir traiter un grand nombre d'entrées de natures différentes et avoir des constructions de suffisamment haut niveau pour exprimer l'exécution concurrente, l'interruption et les communications entre différentes tâches pour permettre de composer les réactions aux évènements. Il existe un modèle de programmation adapté à ce types de programmes : *le modèle synchrone-réactif avec un style impératif à la Esterel*.

1.3.1 Le modèle synchrone-réactif

La première introduction de ce modèle remonte aux langages Esterel [9], Lustre [19] et Signal [37]. Esterel, dont les versions les plus récentes sont v5 [8] et v7 [7], propose un langage de programmation basé sur des expressions de processus, pouvant être exécutés en parallèle ou en séquence dont les instructions émettent de l'information qui se propage dans l'environnement d'exécution. Esterel a donc une approche plus impérative, tandis que Lustre et Signal offrent une approche flots de données. Les signaux en Esterel sont les valeurs de première classe. Ils sont munis d'une information de présence (absent ou présent) et peuvent aussi transporter une valeur à la manière d'un évènement. La singularité du modèle synchrone est que l'exécution se déroule par étapes successives que l'on appelle *instants*. Tous les processus en parallèle s'exécutent durant ce même instant, et leur cadencement sur le temps réel est décidé par une horloge globale extérieure au programme. Les constructions d'Esterel permettent une programmation modulaire via l'opérateur de composition parallèle de deux instructions : `(i || j)`. Deux tâches parallèles peuvent partager des signaux, qui sont définis par leur portée lexicale. Émettre un signal propage l'information de présence dans le programme et ce signal conserve son état jusqu'à la fin de l'instant. La modularité de la composition parallèle est possible parce que les communications entre deux programmes au travers des signaux peuvent être vérifiées statiquement. La sémantique s'appuie sur deux principes :

- L'hypothèse synchrone
- Le déterminisme

Le premier principe énonce que l'émission des signaux est instantanée, et la deuxième que le résultat

attendu en sortie ne dépend que du calcul des entrées et non pas d'une disposition d'un état interne de la mémoire. En conséquence de l'hypothèse synchrone, il est possible de décrire des programmes incohérents, où il est impossible de savoir si un signal est absent ou présent. On parle alors de problèmes de causalité. Un exemple le plus courant est quand l'émission d'un signal dépend de son absence. Effectivement, une fois que l'état d'un signal a été déterminé à un endroit du programme il doit rester le même jusqu'à la fin de l'instant. Cette sûreté ne vient donc pas gratuitement et impose certaines contraintes sur les programmes synchrones. D'abord, il est nécessaire de vérifier le programme par analyse statique pour supprimer ces problèmes de cohérence, mais la vérification du programme peut rejeter une classe de programmes qui semble correct au programmeur. Ensuite, il n'est pas possible d'exécuter une récursion potentiellement infinie via les constructions du langage. La seule façon de répéter un calcul avec une boucle est d'accumuler les données au travers des instants. Malgré tout, ces contraintes sont compensées par le gain en expressivité que les abstractions du langage apportent à la programmation concurrente.

Le programme `ABRO` suivant est un bon représentant de programme synchrone. Il a trois signaux d'entrée en argument : `A`, `B` et `C`, et un signal de sortie `O`. Le programme s'exécute en une boucle infinie. Cette boucle attend la présence disjointe des évènements `A` et `B`. Quand les deux ont été présents, le programme émet `O` et recommence la boucle. Si le signal `R` est présent lors de l'exécution de la boucle, on recommence la boucle sans émettre `O`.

```
module ABRO:
  input A, B, R;
  output O;
  loop
    [ await A || await B ];
    emit O
  each R
end module
```

On peut avoir envie de modifier ce programme, et d'ajouter un signal d'entrée supplémentaire, une nouvelle condition. Pour montrer à quel point les programmes sont composables, on peut, à partir d'`ABRO` écrire le programme `ABCRO` en le mettant en parallèle avec lui même. Le signal local `AB` représente le signal sortant `O` du premier appel, et on le passe en tant que `A` du second.

```
module ABCRO:
  input A, B, C, R;
  output O;
  signal AB in
    run ABRO [ signal AB / O ]
  ||
    run ABRO [ signal AB / A, C / B ]
end module
```

La composition parallèle de programmes est une grande qualité du modèle à la Esterel. Si on souhaite ajouter un nouveau comportement à une application et qu'il interagit et partage des données avec les comportements déjà écrits, on peut simplement l'ajouter en parallèle des autres. On ne se pose pas de question quant à l'ordre d'exécution. C'est le compilateur qui va déterminer cet ordre, et s'il en existe un. Le travail du programmeur se limite donc à savoir si il a bien exprimé les émissions et lecture de signaux, puis il peut s'attarder sur d'autres questions plus algorithmiques.

L'approche flots de données

Lustre [19] est un langage synchrone réactif à flot de données dont les applications principales sont les systèmes de traitement du signal et l'embarqué critique. Un programme Lustre est un ensemble d'opérateurs, ou de nœuds, décrits par des équations où, à gauche du symbole égal se trouve un ensemble de signaux, et à sa droite l'expression de leurs valeurs à chaque instant. Les expressions produisent toutes un flot de données et les constructions sont purement fonctionnelles. A chaque instant, les valeurs des signaux sont calculées à nouveau, et de façon concurrente. L'ordre dans lequel les expressions seront évaluées ne dépend pas de l'ordre dans lequel elles sont définies mais des dépendances entre les signaux.

On illustre l'utilisation du langage dans un exemple très simple qui implémente un nœud construisant la suite de Fibonacci. Étant dans un langage synchrone, il n'est pas possible d'écrire de récursion instantanée. On va donc donc construire une nouvelle valeur à chaque instant. `f` est le résultat de sorti du calcul et il est défini comme une suite de valeurs commençant par `1`, puis ayant la valeur `pre x` ensuite, c'est à dire la valeur de `x` à l'instant précédent. Le signal `x` vaut `1` au premier instant puis la somme de `f` et de la valeur `f` à l'instant précédent. On observe qu'il y a une dépendance mutuelle entre les deux variables, sauf que l'opérateur `pre` précise qu'on accède à la valeur de l'instant précédent, déjà calculée et qui n'impose alors pas de dépendance dans l'instant courant. La seule dépendance est donc celle de `x` vers `f`, calculé en premier.

```
node fibonacci() returns (f: int);
  var x: int;
let
  x = 1 -> f + pre f;
  f = 1 -> pre x;
tel
```

Les flots prennent successivement les valeurs suivantes :

x	1	2	3	5	8	...
f	1	1	2	3	5	...

D'autres langages explorent l'approche flots de données synchrone. Lucid Synchrone [18] par exemple incarne une version à la ML de ce modèle, qui a été étendu avec des machines à états par la suite et mélange les deux approches. Les constructions à la ML comme le filtrage par motif et le typage statique strict en font un langage à la fois sûr et expressif. Le principe de mélanger les machines à état avec les flots de données a été aussi intégrée à Scade [32], une version moderne et industrielle de Lustre. OCaLustre [59] propose une intégration d'un Lustre sous forme d'extension de syntaxe OCaml pour la programmation de microcontrôleurs dans le but de minimiser la consommation de mémoire et profiter d'une programmation synchrone flots de données typée.

La non-réaction à l'absence

Bien qu'il soit possible de vérifier les incohérences des programmes Esterel par analyse statique, certaines solutions proposent de les supprimer complètement par construction en interdisant la réaction instantanée à l'absence d'un signal. Cette idée naît de l'approche réactive [14] dans les termes de Frédéric Boussinot qui propose de restreindre le modèle à la Esterel en supprimant la réaction instantanée à l'absence des signaux, sous la forme du langage SL [14]. La réaction à une absence est forcément décalée

à l'instant suivant, et plus aucune incohérence n'est donc possible. On remarque finalement que l'expressivité est équivalente puisqu'on pourra écrire dans ce modèle des programmes qu'Esterel refuse mais qui sont correctes du point de vu de la causalité. En outre, comprendre les erreurs de l'analyse statique d'Esterel peut être ardu pour un non-initié, et les contourner peut être un avantage. Les SugarCubes [13] sont une implémentation de ce modèle en Java.

1.3.2 Apport de la programmation synchrone au client Web

L'exécution d'un programme dans le navigateur alterne entre prise en compte des évènements dans la boucle d'interaction, exécution des fonctions de rappel et mise-à-jour de la vue graphique. Le navigateur simule une horloge qui déclenche l'exécution de code à chaque itération de la boucle. Un programme réactif-synchrone calcule un ensemble de valeurs de sortie à partir des entrées à intervalle régulier et de façon concurrente. On peut donc facilement calquer l'exécution d'un programme synchrone-réactif sur l'horloge globale du client Web. On peut alors considérer les évènements du client Web comme les entrées du programme, et les éléments graphiques du DOM comme les sorties du programme. Proposer une telle méthode permettrait de simplifier le travail du programmeur en liant automatiquement, par la compilation, les entités du programme synchrone aux entités du programme client. On peut ainsi remplacer la programmation évènementielle, par le modèle synchrone-réactif, plus expressif, plus prévisible, et dont les erreurs de programmation seront plus facilement détectables. On peut aussi profiter de la fonction d'animation du navigateur décrite partie 1.1.3 pour accumuler les évènements entre deux rafraîchissements de l'affichage et déclencher l'horloge au moment du rafraîchissement.

1.3.3 Contributions

Pour défendre nos affirmations, nous proposons dans cette thèse l'introduction d'un langage synchrone-réactif nommé pendulum, que l'on décrit par un plongement du modèle à la Esterel dans un contexte de programmation dans le client Web. On étend par ailleurs ce dernier pour faciliter le branchement entre les entrées-sorties du client Web, les évènements, avec les entrées-sorties du programme synchrone, les signaux. Le dessein visé est également de faciliter la définition de l'horloge synchrone par le programmeur, ou plus simplement, de le libérer de la question : "Quand le programme doit-il réagir ?". On montrera aussi comment adapter automatiquement l'horloge synchrone au moteur d'exécution du navigateur pour profiter des meilleures performances, comme discuté partie 1.3.2. En outre, Le compilateur du langage pendulum fait partie de ce travail et l'implantation en est faite en OCaml. Il compile le code source pendulum vers OCaml, en se présentant comme une extension syntaxique basée sur l'outil PPX. On utilise une technique existante de compilation pour Esterel appelée GRaphCode, qui représente le programme sous forme d'arbre de flot de contrôle. Cet arbre de flot de contrôle est transformé pour respecter un ordonnancement statique du programme synchrone. Les erreurs de programmation synchrones sont détectées si aucun entrelacement des chemins parallèles n'est possible. Le résultat de la compilation génère un objet du langage hôte, OCaml, qui contient l'environnement du programme synchrone, ainsi qu'une méthode `react`, traduction du graphe de flot de contrôle vers du code exécutable, permettant l'exécution d'un instant, et connectée aux fonctions de rappel des évènements du client Web. Pour montrer les possibilités de ce langage, la distribution est accompagnée d'applications réalisées pour le client Web et mobile mettant en pratique différentes interactions, et que nous présentons dans la dernière partie de ce manuscrit.

1.3.4 Travaux connexes

L'environnement d'exécution des programmes synchrones-réactifs est divers. Ces langages ont été conçus pour être compilés vers des circuits de systèmes embarqués à faibles ressources, avec des contraintes de certifications exigeantes, comme dans l'avionique. Mais de par leur expressivité et leurs qualités de sûreté de nouvelles applications se sont ajoutées. Un programme Esterel peut par exemple être compilé vers un programme C sous forme d'automate à états, puis en code machine. Des expérimentations pour écrire des programmes synchrones en Web ont déjà été menées dans le cas de la plateforme HOP avec le langage HipHop [10], qui propose d'ajouter une couche de programmation synchrone au langage Hop. La partie synchrone est intégrée comme un langage dédié : un indicateur syntaxique indique que l'on entre dans le contexte synchrone. Le code est ensuite compilé grâce aux macros Scheme vers un arbre de syntaxe abstraite. Le moteur d'exécution évalue ensuite cet arbre conformément à la sémantique d'Esterel. Contrairement à notre approche, HipHop considère une compilation très dynamique, et donc ne vérifie pas que le programme est causalement correct avant son exécution. Néanmoins, cela apporte un intérêt important en termes de dynamicité, telle que la capacité d'échanger des programmes par le réseau, ou d'augmenter dynamiquement le nombre de programmes exécutés en parallèle.

Le mariage du typage fort à la ML et de la programmation synchrone-réactive est une problématique de recherche qui a été plusieurs fois étudiée. On a parlé précédemment de Lucid Synchrone qui choisir le mariage de Lustre et ML. Il existe aussi ReactiveML [47]. ReactiveML diffère du modèle à la Esterel, puisque qu'il est inspiré du modèle réactif de ReactiveC [12] et du travail mené sur les SugarCubes [15]. Ce langage se compile vers OCaml et permet d'utiliser un sous ensemble de ses constructions. La compilation d'un programme ReactiveML génère un ensemble de fermetures imbriquées dont l'exécution correspond à un instant du programme synchrone. Comme dans les SugarCubes, l'ordonnancement des tâches est dynamique, c'est-à-dire que le choix de l'ordre d'exécution ne se fait pas à la compilation mais pendant l'exécution du code.

Un autre axe de recherche propose une vision fonctionnelle de la programmation Web réactive. C'est le cas de langages dont il est question dans la partie 1.2.2, comme FlapJax [48] et Elm [24]. Cette approche s'est démocratisée et on la retrouve dans langages comme Scala.React qui peut être utilisé dans le client Web grâce à Scala.js. C'est aussi le cas d'OCaml et sa bibliothèque React (à ne pas confondre avec son homonyme en JavaScript) qui prend de plus en plus d'importance dans la plateforme Eliom pour la programmation d'applications Web et mobile. L'approche d'Eliom permet d'utiliser la programmation réactive fonctionnelle pour décrire des relations entre des signaux du client et des signaux du serveur, mais aussi de générer les signaux du programme client depuis le serveur en initialisant leur fonction de mise-à-jour et leur valeur initiale. Le langage Ur/Web [20] choisit une approche similaire pour faire évoluer la page Web en fonction de la modification des données.

1.3.5 Plan de thèse

Dans le chapitre suivant, nous détaillons les constructions du langage pendulum en les illustrant par des exemples, ainsi que les extensions proposées au langage pour l'inclure une démarche de programmation Web. Ensuite nous indiquerons comment les différentes couches, client Web, OCaml et synchrone s'articulent durant l'exécution et comment le programmeur peut instrumenter un programme synchrone depuis le langage hôte. Dans le troisième chapitre, nous décrivons à notre manière la méthode de compilation GRaphCode pour poursuivre dans le chapitre quatre avec la compilation de ce graphe de flot de contrôle vers un programme OCaml lié à la bibliothèque Js_of_ocaml. Nous donnerons en particulier la génération du code et des structures de données qui gèrent l'environnement d'exécution. Dans le cinquième chapitre, nous présentons nos résultats liés aux expérimentations avec pendulum et en particulier

la corrélation entre le modèle d'exécution du navigateur et une horloge synchrone. Nous exposerons aussi la mise en pratique de cette constatation pour gagner en temps d'exécution sur la réaction aux évènements. Dans le sixième chapitre, nous détaillerons plusieurs exemples avec code à l'appui, qui montrent chacun différents aspects de la programmation Web et comment ils s'écrivent en pendulum, ainsi qu'une extension faite à notre langage pour l'interopérabilité avec la plateforme Eliom. Nous conclurons finalement sur les apports de ce travail et les possibilités futures de pendulum.

Chapitre 2

Pendulum, un langage synchrone pour le Web

Pendulum [28][29] est un sous-ensemble du langage de programmation Esterel [9]. Il est dédié, et conçu pour être intégré dans un langage généraliste hôte. Pendulum permet d'écrire des programmes, équivalents des modules en Esterel, avec une sémantique réactive-synchrone, au travers d'une interface de programmation pour les déclencher depuis le programme hôte. On commence par présenter le noyau synchrone puis on établit les connexions entre ce langage et son langage hôte. Le reste du chapitre se concentrera sur des nouvelles constructions ajoutées au noyau synchrone, facilitant l'interopérabilité avec un contexte de programmation Web.

Remarque. *Bien que le cœur de pendulum soit conçu pour s'exécuter avec n'importe quel langage généraliste comme hôte, le langage hôte doit avoir la contrainte de pouvoir s'exécuter dans un client Web pour profiter des nouveautés apportées par pendulum. A cet égard, le choix pourrait porter arbitrairement sur JavaScript ou tout autre langage ayant pour ce dernier pour cible de compilation. Nous avons choisi OCaml en tant que langage hôte pour les expérimentations, avec Js_of_ocaml pour la partie client Web. Néanmoins, le cœur de l'implémentation est agnostique de la cible de compilation, de même que la section 2.1 d'introduction du langage. Dans les exemples de code des sections qui suivent, on présente des éléments de programmation de différents niveaux, selon si on se situe dans du code synchrone, dans le langage hôte, ou dans les fonctionnalités du client Web. Pour des besoins de clarté, un code couleur est appliqué aux différentes constructions (**pendulum**, **OCaml**, **Js_of_ocaml**)*

2.1 Le noyau synchrone

Un programme pendulum est une composition parallèle (`||`) ou séquentielle (`;`) d'instructions, ou tâches, agissant par effet de bord sur l'environnement d'exécution. L'exécution se déroule pas à pas en une suite d'*instants*. Au cours d'un instant, il n'est pas possible d'allouer de la mémoire indéfiniment ou de boucler à l'infini en utilisant uniquement les constructions du langage. On considère donc les calculs et la transmission des informations comme étant instantanés. Ces instructions sont listées dans la figure 2.1, où les *header* sont les en-têtes pour la définition des arguments du programme, et *stmt* (pour *statement*) le corps des instructions du programme.

prog : := *header* stmt*	*programme*
header : := (**input** | **output**) *ident* ;	*arguments*
stmt : := **emit** *ident ocaml-expr*	*émission d'un signal valué*
| **nothing**	*ne rien faire*
| **pause**	*attendre l'instant suivant*
| **present** *test stmt stmt*	*branchement conditionnel*
| **loop** *stmt*	*boucle infinie*
| **exit** *label*	*échappement de bloc*
| *stmt* **||** *stmt*	*parallèle*
| *stmt* **;** *stmt*	*séquence*
| **let** *ident* **=** *ocaml-expr* **in** *stmt*	*signal local*
| **trap** *ident stmt*	*bloc interruptible*
| **!** *ocaml-expr*	*expression hôte atomique et instantanée*
| **suspend** *test stmt*	*bloc suspendu*
test : := *ident*	*expression de test de présence d'un signal*

FIGURE 2.1 – Grammaire de pendulum

On présente d'abord les signaux, qui constituent les valeurs de première classe du langage. On explique ensuite comment composer les programmes et comment définir la fin et la mise en pause d'un programme synchrone, pour finir sur l'expression du flot de contrôle du programme et l'exécution atomique d'expression du langage hôte.

Les signaux

Un signal est une structure de données avec un état interne qui peut être *Présent* ou *Absent*, ainsi qu'une valeur transportée. On peut le représenter de cette façon par un type de données OCaml. Dans ce langage, le type `signal` est paramétré par le type de la valeur transportée.

```
type state = Present | Absent
type 'a signal = {
  mutable value : 'a;
  mutable state : state;
}
```

Au début d'un nouvel instant, l'état du signal est inconnu et une valeur par défaut lui a été attribuée à sa construction. L'instruction `emit` permet de changer son état et sa valeur. L'instruction

```
emit a 1
```

L'instruction ci-dessus change l'état du signal `a` à `Present` et met sa valeur à `1` en supposant que `a` transporte des valeurs de type entier. La valeur transportée persiste entre les instants, contrairement à l'état du signal qui passe automatiquement à absent à la fin de l'instant. Les tests de présence ne sont exécutés au cours de l'instant que quand l'état de présence du signal testé a été fixé : soit il a été émis, soit plus aucune émission n'est atteignable et on sait qu'il sera absent pendant cet instant. L'instruction

suivante permet de tester la présence de a et de choisir une branche `emit b` ou `emit c` en fonction du résultat :

```
present a (emit b) (emit c)
```

On observe qu'il y a une asymétrie entre les informations de présence et d'absence, parce qu'il est possible de déclarer la présence par l'émission mais rien ne permet de déclarer l'absence d'un signal. La syntaxe autorise aussi l'omission de la branche *else* :

```
present a (emit b)
```

automatiquement traduite vers l'instruction neutre `nothing` :

```
present a (emit b) nothing
```

La définition d'un signal local est un bloc d'instructions qui correspond à la portée du signal, à la façon du mot clef `let` à la ML. Cette portée est lexicale donc il est visible de toutes les tâches parallèles que le bloc englobe.

```
let a = () in
emit a || present a (emit b) (emit c)
```

Composition de programmes

Il est possible de composer les tâches séquentiellement ou en parallèle. La concurrence s'exprime par la composition de deux tâches, soit deux instructions. Le code suivant signifie que les instructions p et q s'exécutent simultanément, sur la même horloge. Leurs instants démarrent et terminent au même moment.

```
p || q
```

On peut aussi exécuter deux instructions, l'une à la suite de l'autre avec l'opérateur de séquence (;). Dans le programme suivant, l'exécution de q commence instantanément lorsque celle de p termine.

```
p ; q
```

L'arrêt du programme

Un programme réactif-synchrone peut avoir plusieurs états à la fin d'un instant. Si il ne reste plus rien à exécuter, on dit qu'il est terminé, et exécuter un instant n'aura aucun effet. C'est par exemple le cas du programme suivant après le premier instant d'exécution :

```
nothing
```

Le programme peut aussi être dans un état de pause, qu'il atteint lorsqu'au moins un processus en parallèle atteint l'instruction `pause`. Elle signifie l'interruption du flot de contrôle du programme pour l'instant courant. Le programme suivant exécute d'abord `emit s1` puis s'arrête.

```
emit s1; pause; emit s2
```

Il reprend ensuite son exécution à partir du `pause` et exécute `emit s2` puis termine.

```
emit s1; pause; emit s2
```

Flot de contrôle

La répétition se fait au moyen de l'instruction `loop`, qui est une boucle infinie. Par exemple, le programme ci-dessus émet le signal `a` à chaque instant.

```
loop (
  emit a 0;
  pause
)
```

Le langage interdit cependant l'exécution de plus d'un tour de boucle par instant, bloquant par ce moyen la récursion instantanée et l'accumulation de mémoire. Si aucune instruction `pause` n'est présente syntaxiquement, elle est ajoutée automatiquement à la fin de la boucle. Comme cette construction boucle indéfiniment, elle va de pair avec un comportement d'interruption, qui apparaît sous la forme d'échappement.

Pour interrompre un comportement, le langage est muni d'une structure d'échappement à la manière des exceptions dans les langages généralistes, c'est-à-dire une interruption définitive de l'instruction en cours d'exécution, et un saut à la fin de cette instruction. Ce dernier peut être par exemple utile pour définir une condition de fin de boucle.

```
trap t (
  loop (
    present a (exit t);
    pause
  ))
```

Dans le code ci-dessus, on interrompt l'exécution de la boucle et on saute à la fin du bloc `trap` quand l'instruction `exit T` est exécutée, donc si `a` est présent. Si plusieurs labels ont le même nom, le bloc interrompu sera le label plus proche du `exit`. Dans le cas où plusieurs `exit` ont lieu en même temps et que les blocs `trap` sont imbriqués, c'est le bloc le plus à l'extérieur qui est interrompu.

```
trap t1 (
  trap t2 (
    loop (
      exit t1 || exit t2;
      pause
   )))
```

Dans le programme si dessus, c'est le label `t1` qui est pris en compte alors que les deux `exit` sont exécutés en même temps.

Exécution atomique d'instructions du langage hôte

On permet au programmeur d'exécuter une expression du langage hôte, pour faire appel à des instructions qui ne sont pas prises en charge par pendulum. Dans le cas où le langage hôte est OCaml, l'instruction

```
loop (
  !(print_string "Hello");
  pause
)
```

exécute à chaque instant l'expression OCaml qui suit l'opérateur `(!)`.

2.2 Exécution d'un programme synchrone dans un programme hôte

Un programme embarqué dans un programme hôte, quelque soit la méthode de compilation, peut être considéré comme un sous programme de ce dernier. On peut alors le voir comme une fonction. Le modèle d'exécution d'un programme synchrone est très différent du modèle d'exécution d'un langage algorithmique *généraliste*, comme C, Ada ou OCaml, parce qu'il s'arrête explicitement sur certaines instructions jusqu'au prochain tic d'horloge. S'il est possible qu'un programme traditionnel s'interrompt et reprenne son exécution, un programme synchrone atteint forcément un état de pause. Sa phase d'exécution est bornée et d'après l'hypothèse synchrone, elle est instantanée.

La fonction du programme hôte qui représente le programme synchrone embarqué ne l'exécute pas entièrement mais s'arrête quand un instant est terminé, c'est-à-dire quand le programme atteint l'instruction `pause`. Pour démarrer l'instant suivant, on doit rappeler la *fonction d'instant* qui reprend depuis l'exécution précédente en fonction de l'état de l'environnement. Dans le vocabulaire des langages synchrones, on dit que la suite des appels à la fonction d'instant constitue une horloge du programme synchrone. Si on appelle $\mathcal{E}_p^i$ l'environnement d'exécution du programme p après l'instant i et $react_p$ la fonction d'instant du programme, on peut décrire l'évaluation ($\rightarrow$) symboliquement de la façon suivante :

$$react_p(\mathcal{E}_p^i) \rightarrow \mathcal{E}_p^{i+1}$$

2.2.1 Le langage en pratique

On applique l'idée à notre situation : on embarque le langage pendulum comme extension dans le langage OCaml via l'outil PPX décrit dans la sous-section 1.1.1. Ce dernier permet de marquer une expression dans le programme source, pour qu'elle soit ciblée par le préprocesseur et transformée à la compilation. Dans notre cas, il est possible d'appliquer la marque `%sync` à une expression `let` ou `let in`, pour que la variable ainsi créée devienne l'identifiant du programme synchrone :

```
let%sync nom = code
```

Ici, le couple (*nom*, *code*) est envoyé au préprocesseur `ppx_pendulum`, c'est-à-dire notre compilateur, lui même écrit en OCaml, qui le transforme en un nouveau couple ou *code* est le programme synchrone compilé. La compilation d'un programme OCaml étendu avec pendulum doit aussi faire la liaison avec la bibliothèque d'exécution `pendulum` qui donne les types et les primitives nécessaires à l'exécution. La ligne de compilation de ce type de programme a la forme suivante :

```
# ocamlfind ocamlc -package pendulum -ppx ppx_pendulum < nomfichier >
```

Cette ligne de commande signifie que l'on fait appel au compilateur de code-octet ocamlc en liant le paquet pendulum et en utilisant son moteur PPX comme préprocesseur. Le code généré contient l'ouverture automatique du module `Pendulum.Signal` de cette bibliothèque, permettant la manipulation des signaux dans les expressions atomiques OCaml par exemple.

2.2.2 Générateur et instance

On décrit notre premier programme synchrone, `afficheA`, dont le code est le suivant :

```
let%sync afficheA =
  input a;
  loop (
    present a
      !(print_string "A");
    pause
  )
```

À droite du symbole =, la syntaxe est celle de pendulum et le code est traité et transformé par l'extension de syntaxe. Le type de `afficheA` n'est pas directement celui d'un programme synchrone avec une fonction de réaction. C'est un *générateur* du programme *afficheA*. Un générateur est une valeur de type objet possédant une méthode `create` qui prend en paramètre les valeurs initiales des signaux et construit une instance du programme *afficheA* effectivement exécutable. Dans la syntaxe des types objets d'OCaml, on écrit :

```
val afficheA : < create : 'a → 'a pt >
```

Remarque. *Les symboles < > désignent les objets dans le langage des types d'OCaml et ils encadrent la liste des noms de méthodes et de leurs types.*

Dans ce code, *pt* est le type du programme synchrone exécutable. Le type de `a` est déterminé par l'inférence de type, mais comme il n'est pas utilisé dans `afficheA`, c'est un type paramétré et on lui attribut la variable de type `'a`. On peut donc l'appeler avec une valeur arbitraire à la création.

Remarque. `#` *est l'opérateur d'appel de méthode en OCaml*

On choisit par exemple le type *unit* dont l'unique valeur est `()` et qui correspond moralement à une valeur sans information.

```
let afficheA_p = afficheA#create ()
```

Un programme synchrone est un objet du programme hôte qui contient une mémoire et un environnement d'exécution local. On peut avoir besoin d'exécuter plusieurs instances de ce programme en parallèle qui ne partagent pas de mémoire avec les autres instances. C'est pour cette raison que l'on passe par un générateur au lieu de construire directement un programme et sa mémoire. On peut ensuite appeler la méthode `react`, qui ne prend pas de paramètre.

```
afficheA_p#react
```

Le programme *afficheA*, à chaque instant, vérifie si le signal `a` est présent, et si oui, affiche la chaîne de caractères `"A"`, si non, se met en pause. Puisque `a` est absent, n'ayant pas été déclaré présent explicitement, il est donc normal que l'appel à `react` ci-dessus n'ait aucune conséquence visible. Les instances générées possèdent par-ailleurs une fonction pour chaque signal d'entrée, qui met à jour la valeur et fixe l'état à présent pour l'instant suivant. On fait appel à la méthode `afficheA_p#a`.

```
afficheA_p#a ();
afficheA_p#react
```

avec la sortie standard suivante :

```
stdout > A
```

On passe `()` comme nouvelle valeur du signal `a`, et après un deuxième appel à la fonction de réaction, la chaîne de caractères s'affiche sur la sortie standard. On a le type de l'instance, et le type du générateur :

```
(* générateur *)
val afficheA : < create : 'a → < a : 'a → unit; react : unit >>

(* instance *)
val afficheA_p : <  a : unit → unit; react : unit >
```

2.2.3 Les signaux de sortie

Il est aussi possible d'utiliser des signaux de sortie. Le cas échéant, on doit passer une fonction de rappel pour chaque signal de sortie pour exprimer le fait que l'émission du signal déclenchant la fonction accède à l'environnement du programme hôte et peut exécuter des effets de bord. On prend maintenant l'exemple du programme suivant :

```
let%sync afficheA2 =
  input a;
  output b;
  loop (
    present a (emit b "A");
    pause
  )
```

La méthode `create` prend dorénavant deux paramètres : la valeur initiale pour `a` et un couple composé de la valeur initiale et d'une fonction de rappel pour `b`.

```
let afficheA2_p = afficheA2#create () ("", print_string) in
afficheA2_p#a ();
afficheA2_p#react
```

Ce programme affiche affiche :

```
stdout > A
```

Il possède le type :

```
val afficheA2 : < create : 'a → (string, (string → unit)) →
                  <  a : 'a → unit ; react : unit >>
```

2.2.4 Passage du programme synchrone au programme hôte

On est en présence de deux *niveaux* de programmation, et il est important de noter les points de passage d'un *niveau* à l'autre. Dans la syntaxe donnée dans la partie précédente, ce passage se fait en présence du symbole non-terminal *ocaml-expr*. On observe que cela apparaît quand le code fait référence aux valeurs concrètes transportées par les signaux, qui sont celles du langage hôte (e.g. **emit**, **let**) ainsi que l'exécution instantanée d'une expression du langage hôte (**!**). Depuis ces expressions, il est possible de faire référence aux signaux du programme en cours, qui sont définis en fonction de leur portée : locale (**let**) ou globale (**input**, **output**). Par exemple, dans le programme suivant *affiche*, on émet `b` avec la valeur du signal `a`, s'il est présent. L'opérateur `!!` extrait la valeur contenue dans un signal.

```
let%sync affiche =
  input a;
  output b;
  loop (
    present a (emit b !!a);
    pause
  )
```

Remarque. *On notera que l'opérateur* (`!!`) *n'existe que dans le langage hôte et non dans pendulum. Il est fourni par le module* `Pendulum.Signal`. *Son type est* (`'a signal → 'a`)

On peut ensuite observer dans le résultat que la valeur de `a` mise dans le signal `b` est ensuite passée à la fonction de rappel donnée `print_string` quand le signal est émis.

```
let affiche_p = affiche#create () ("", print_string) in
affiche_p#a "Hello";
affiche_p#react

stdout > "Hello"
```

2.2.5 Modèle objet et typage du programme synchrone

L'utilisation des objets comme interface d'un programme synchrone embarqué dans un programme OCaml se justifie pour représenter un état mémoire que l'on modifie au cours du temps par différentes fonctions. Un objet est plus commode à manipuler qu'un n-uplet de fonctions, surtout lorsque le nombre de composantes grandit, puisqu'un tuple doit être déconstruit systématiquement pour séparer les composantes. De plus, le type d'un programme synchrone dépend du type de ses signaux d'entrée-sorties. Si on avait utilisé une représentation sous forme d'enregistrement, chaque programme synchrone aurait

été généré en même temps que son propre type. Ce dernier aurait été paramétré par un type pour chaque signal en argument du programme, pour laisser le compilateur l'inférer. Un tel type peut être difficile à lire en cas d'erreur. L'utilisation des objets règle ce problème puisque qu'il n'est pas nécessaire de définir une classe, et donc un type, au préalable pour écrire un objet qui l'instancie, et ce grâce au typage structurel : un type anonyme pour cette classe est construit automatiquement. On peut donc avoir un programme synchrone facile à manipuler, dont les types des entrées-sorties sont inférés par le compilateur du langage hôte sans efforts supplémentaires de la part du programmeur.

Les constructions synchrones sont totalement agnostiques des types transportés par les signaux et qui ont été inférés. Seules les expressions OCaml sont donc sujettes aux erreurs de type provoquées par le programmeur. La localisation de ces expressions a été conservée entre le source et l'arbre de syntaxe abstraite pendulum, ainsi que la localisation des différentes définitions de signaux. Une erreur sur l'utilisation d'un signal va se comporter comme une erreur sur une fonction en OCaml. Par exemple l'erreur pour le code mal typé suivant indique précisément où est l'erreur.

```
let%sync p = present a (emit b "1") (emit b 2)
                                           ^

 Error: This expression has type int
 but an expression was expected of type string
```

On peut avoir d'autres exemples où l'erreur est plus difficile à situer, quand la valeur par défaut d'un signal donnée lors de la création et celle manipulée dans le programme divergent.

```
let%sync p = input a; emit a "1"
let p_m = p#create 2
                   ^

 Error: This expression has type int
 but an expression was expected of type string
```

Si l'erreur est bien dans le paramètre, alors le compilateur est précis. Par contre si l'erreur se situe dans l'utilisation du signal, alors le compilateur ne donne pas la localisation attendue. Le programmeur peut néanmoins forcer le type par une annotation sur le signal en argument.

```
let%sync p =
  input (a : int) ; emit a "2"
                           ^

 Error: This expression has type string
 but an expression was expected of type int
```

2.3 Connexion avec le client Web

Nous avons vu dans la partie précédente l'exécution d'un programme synchrone dans un programme hôte issu d'un langage algorithmique généraliste, en l'occurrence OCaml. On souhaite maintenant faire exécuter une application de ce type dans un navigateur Web sous forme de script JavaScript. En utilisant l'outil Js_of_ocaml, dont les détails sont donnés dans la sous-section 1.2.1, il est possible de compiler le code octet issu du programme OCaml vers un script JavaScript, comme on l'observe dans le schéma 2.2.

Cette solution a le bon goût de s'abstraire des modifications ultérieures du langage et n'est pas sensible aux extensions de syntaxe, ni aux bibliothèques liées au programme qui ne se basent pas sur du code

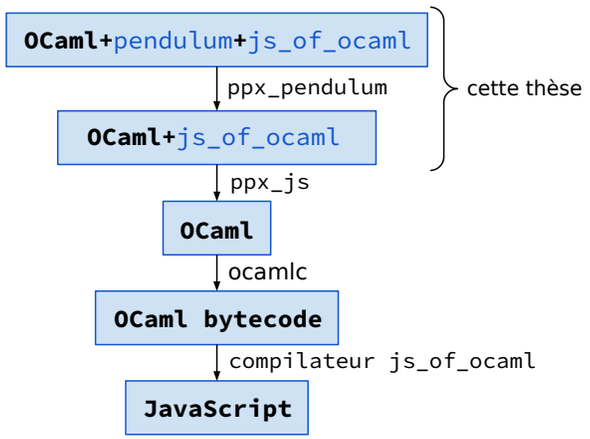


FIGURE 2.2 – Processus de compilation, de pendulum à JavaScript

externe compilé ou dans un autre langage. Cela signifie que l'utilisation de pendulum est complètement transparente vis-à-vis de la compilation du programme vers un exécutable pour le navigateur. On veut maintenant utiliser un programme synchrone pour gérer la partie de l'application qui nécessite de la concurrence, en remplaçant l'utilisation des évènements par un programme synchrone. Dans un premier temps, on veut montrer qu'il est possible de faire réagir le programme synchrone aux évènements en les simulant au travers des signaux, ainsi que de modifier les éléments de la page depuis le programme synchrone.

Pour utiliser la couche des évènements en Js_of_ocaml, on utilise le DOM. On souhaite écrire un programme qui écrit la position du curseur de la souris quand ce dernier se déplace, dans un élément de la page pour l'afficher. On commence par déclarer cet élément, `sp`, que l'on attache au `body` du document.

```
open Dom_html;;
let sp = createSpan document;;
appendChild body sp;;
```

On écrit ensuite le programme synchrone `mouse_loc` paramétré par un signal `coords` contenant les coordonnées du curseur sous forme de chaîne de caractères. Dans le corps du programme, à chaque fois que le signal `coords` est présent, on affecte sa valeur au champ `textContent` de notre élément, ceci ayant pour effet d'afficher la nouvelle valeur dans la page web.

```
let%sync mouse_loc =
  input coords;
  loop (
    present coords !(
      sp##.textContent := Js.some (Js.string coords)
    );
    pause)
;;
```

On remarque que des transformations sont nécessaires pour correspondre à la représentation des type de Js_of_OCaml. Ici, le champ `textContent` peut valoir `null`, il est alors représenté par un type de chaîne

de caractères optionnelle. On construit ensuite notre instance de programme avec la chaîne de caractères vide comme valeur initiale du signal `coords`.

```
let m = mouse_loc#create ("");;

let callback ev =
  m#coords (sprintf "%f, %f" ev##.clientX ev##.clientY);
  m#react
in
window##.onmousemove := handler callback;;
```

Remarque. *`##` est l'opérateur d'appel de méthode de Js_of_ocaml et `##.` l'accès aux champs de l'objet. A ne pas confondre avec `#`, l'appel de méthode en OCaml.*

On doit ensuite définir une fonction qui est appelée quand le navigateur détecte le déplacement de la souris. Notre fonction `callback` prend en paramètre la valeur JavaScript de l'objet évènement contenant les coordonnées du curseur, les transforme en chaîne de caractères et passe la chaîne à la méthode `m#coords`. Cette méthode met le signal du même nom à l'état présent avec la nouvelle valeur pour l'instant suivant. Elle déclenche ensuite un instant du programme avec `m#react`. On affecte ensuite `callback` au champ `onmousemove` de l'objet `window` désignant la fenêtre du navigateur. Dorénavant, `callback` est le gestionnaire de l'évènement *mousemove* sur l'objet `window`.

Remarque. *À chaque irruption de l'évènement* mousemove*, on déclenche un instant d'exécution du programme synchrone et on met à jour l'élément de la page. On observe que dans cette situation, l'horloge du programme synchrone se définit par la suite des exécutions des fonctions de rappel liées à l'évènement. Nous verrons plus tard que cette solution n'est pas toujours la plus adaptée.*

2.4 Les évènements en entrée, le DOM en sortie

Précédemment, on a connecté manuellement le moteur d'évènements de JavaScript à l'exécution du programme synchrone en utilisant les fonctions de rappel. Cela nécessite une implication de la part du programmeur pour une tâche systématique que l'on pourrait automatiser. En forçant l'utilisateur à faire ce travail, on le force aussi à manipuler des évènements alors que l'objectif est d'utiliser un modèle plus sain qui génère lui même les réactions aux évènements. Comme il est dit dans l'introduction dans la sous-section 1.1.3, un comportement évènementiel dans le DOM est représenté par un couple (*element*, *event*), que l'on abrège en (ν, ϵ). Pour faire correspondre directement des signaux à des évènements, on doit pouvoir construire des signaux à partir de ce couple. Pour cela, on étend la syntaxe de pendulum de la façon suivante :

$$\begin{aligned} \mathit{test} &::= \text{ident} \\ &\mid \mathit{ident}\texttt{\#\#}\mathit{event} \end{aligned}$$

Dans ce modèle, on ajoute un nouveau genre de valeurs dans le langage, qui n'interagit pas avec les signaux : les éléments. Ils représentent directement les évènements du DOM et ne peuvent être utilisés qu'en un seul point de la syntaxe : l'expression de test des signaux. Dans une expression de test de présence, on peut maintenant lier un élément ν du DOM passé en paramètre du programme et un nom d'évènement ϵ pouvant apparaître sur cet élément au moyen de l'opérateur de liaison évènementielle : `##`. Si on reprend l'exemple de la partie précédente, on peut écrire `window##onmousemove`, avec `window`. On peut alors définir le comportement du signal ainsi créé.

Définition 1. *Pour tout élément v du DOM et pour tout évènement qui y est associé ϵ, on a le signal v##ϵ tel que :*

- *v##ϵ est défini globalement en en-tête du programme comme* `input`.
- *Le type de v##ϵ est t_ϵ* `option signal`*, où t_ϵ est le type de l'évènement désigné par ϵ et où* `type 'a option = Some of 'a | None` *est un type OCaml et signal est le typé définit section 2.1*
- *v##ϵ est fixé à présent pour l'instant logique i_n si et seulement si la fonction de rappel affectée au champ ϵ de v a été appelée avant le début de l'instant.*
- *A la fin de l'instant i_n, v##ϵ est fixé à absent.*

La présence de cette expression de signal a des conséquences sur la génération de code et dans le programme synchrone. Implicitement, une fonction de rappel est construite et affectée à l'élément v et exécute un code équivalent à celui présenté dans l'exemple de la section précédente : elle donne une valeur au signal pour l'instant suivant et déclenche l'appel à la méthode `react`. La valeur du signal a un type option pour représenter le cas initial, où l'évènement n'est pas encore apparu. Après cela, la valeur du signal de type t_ϵ `option` est conservée pour les instants qui suivent, comme toutes les valeurs transportées. Pour passer les éléments en paramètres d'un programme, on ajoute une nouvelle construction pour les arguments de programme en en-tête.

header ::= ...
| **element** *ident* ;

On peut maintenant écrire l'instruction de test du programme de la partie précédente avec notre nouvelle construction et la définition de notre élément `window` en argument

```
element window;
...
  present window##onmousemove !( ... );
```

La valeur d'un signal défini par liaison évènementielle n'est pas très commode à manipuler en tant que valeur optionnelle car elle ajoute du bruit au programme synchrone et limite la lisibilité. De plus, si certaines données de l'objet évènement ne sont pas nécessaire il serait plus pratique d'avoir une façon de simplifier la valeur en la projetant vers un autre type systématiquement un instant logique, mais que l'on accumule les valeurs jusqu'au prochain instant. On va donc généraliser la déclaration des éléments en argument, en permettant au programmeur de préciser chaque évènement dont ils peuvent être la cible avec une fonction qui peut à la fois exprimer la projection, et l'accumulation des valeurs.

header ::= ...
| **element** *ident gather*? ;

gather ::= **{** (*event* **=** *ocaml-expr*, *ocaml-expr* **;**)+**}** ;

Une accumulation d'évènements se traduit par un couple de valeurs du langage hôte. La première valeur du couple est de type `('a)`, valeur initiale de l'accumulateur. La deuxième attend une valeur fonctionnelle de type `('a' → event → 'a)`.

Remarque. *Les évènements en Js_of_ocaml sont décrits par la classe* `event` *et les classes qui en héritent. La classe de base est définit de la façon suivante :*

```
class type event = object
  inherit [element] Dom.event
end
```

Dans pendulum, une contrainte de type est ajoutée pour forcer la fonction d'accumulation à prendre un paramètre dont le type de classe est hérité d'`event`.

On peut par exemple réécrire la déclaration de l'élément `window` de la façon suivante. On notera qu'il n'y a pas besoin d'accumuler les coordonnées dans ce cas, donc le premier paramètre `acc` est inutile si on souhaite juste faire une projection mais tout de même exigé par le type. C'est pourquoi on utilise _ pour l'ignorer.

```
element window {
  onmousemove = "",
    (fun _ ev → sprintf "%f, %f" ev##clientX ev##clientY)
};
```

Le signal contient les valeurs successives des coordonnées et non plus la dernière occurrence de l'évènement sous forme d'objet. Une utilisation de la fonction d'accumulation pourrait être de construire un historique inversé des coordonnées des clics de la souris entre deux instants :

```
element window {
  onmouseclick = [],
    (fun acc ev → (ev##clientX, ev##clientY) :: acc)
};
```

Les entrées sont maintenant spécifiées, mais il serait intéressant de voir comment améliorer l'interaction avec les sorties du programme synchrone. Jusque là, on se repose sur l'utilisation des expressions atomiques sous forme d'effets de bord. On souhaite une méthode plus intégrée au programme synchrone, en ayant la possibilité d'émettre directement une valeur dans une propriété d'un élément du DOM. De cette façon, la manipulation de ce dernier peut être intégrée de la même manière que les signaux. On ajoute donc une dernière construction syntaxique : la liaison de propriété.

stmt ::= ...
| **emit** *ident*##.*property ocaml-expr*

Remarque. *La construction non-littérale property réfère à un nom de propriété associé à l'élément dans le DOM. Si la propriété n'existe pas dans l'interface de Js_of_OCaml, une erreur de type sera détectée à la compilation. Comme on se place dans un contexte statiquement typé, on considère que la définition du DOM passe par l'interface Js_of_ocaml. D'un point de vue typage, les deux sont considérés équivalents. Il en va de même pour les évènements qui correspondent eux aussi à un champ de l'élément.*

Fort de ces nouveaux ajouts, on peut réécrire entièrement l'exemple de la partie précédente. On remplace le signal d'entrée `coords` par deux éléments, `textbox`, notre composant textuel, et `window` qui représente la fenêtre du programme et qui reçoit l'évènement `onmousemove`.

```
let%sync mouse_loc2 =
  element textbox;
  element window {
    onmousemove = "",
      (fun _ ev → sprintf "%d, %d" ev##.clientX ev##.clientY);
  };

  loop (
    present window##onmousemove
      (emit textbox##.textContent
         Js.(some (string .!!(window##onmousemove))));
    pause)
;;
let m2 = mouse_loc2#create (textbox, window);;
```

Le programme en tant que bloc contient toutes les informations nécessaires à sa compréhension par le programmeur. Néanmoins, le corps du programme synchrone ne comprend plus que la relation de dépendance suivante : « À chaque instant, si le signal indiquant que le curseur de la souris a changé de position est présent, on écrit les coordonnées stockées dans ce signal sous forme de chaîne de caractères dans un champ de texte ».

Avec les extensions proposées à ce langage synchrone, il est possible de manipuler le document qui représente les interactions de l'application dans le client Web. Les entrées physiques et les communications entrantes du programme, capturées sous forme d'évènements, sont traduites en signaux manipulés par le programmeur. Ce dernier peut utiliser la programmation synchrone comme couche de concurrence principale, sans avoir conscience des évènements sur lesquels le système repose. L'encodage de l'arbre de syntaxe abstraite de pendulum se trouve dans le module `Ast` du projet pendulum à l'adresse suivante : https://github.com/remyzorg/pendulum/tree/master/src/preproc.

On souhaite par la suite ajouter des outils pour gérer et comprendre l'horloge du programme dont les déclencheurs sont, jusque-là, les occurrences d'évènements. Ce choix est trop simpliste pour correspondre aux normes de la programmation Web moderne. Dans la section 5.1, « Exécuter un programme synchrone dans le navigateur », on présente des modifications au modèle de cette partie pour rendre plus flexible la manipulation de l'horloge, et comment rendre un programme synchrone efficace en temps d'exécution du point de vue du navigateur. Dans le chapitre suivant, nous ouvrons le capot de pendulum pour décrire le processus de compilation du code source synchrone vers un arbre de flot de contrôle des instants du programme.

Chapitre 3

Compilation : du synchrone vers le flot de contrôle

Il existe différentes approches de compilation permettant d'intégrer les constructions d'un DSL à un langage hôte plus généraliste pour engendrer un programme unique. A un extrême, l'approche purement dynamique, qui est de représenter le programme embarqué en une valeur du programme hôte, à la manière des S-expressions en Scheme. On peut ensuite l'exécuter avec une fonction d'évaluation générique. L'approche la plus statique est de générer du code dans le programme hôte qui simule le programme embarqué avec la sémantique du DSL. On génère alors une fonction d'évaluation pour chaque programme dédié. La première solution donne plus de liberté sur le programme, et notamment la possibilité de le modifier dynamiquement. La deuxième semble être plus efficace en vitesse d'exécution, parce qu'elle économise un niveau d'interprétation : le programme embarqué est directement exécuté par le moteur du programme hôte et non pas par une fonction d'évaluation écrite dans le programme hôte. Dans le cas d'une intégration statique, on peut utiliser le système de types du langage hôte pour avoir des garanties sur le programme dédié, alors que dans le cas d'une exécution dynamique, on doit typer statiquement le code du programme hôte séparément avant le typage du langage hôte. Dans les deux cas, l'environnement est réifié sous la forme d'une structure de données du langage hôte et il est modifié par l'interpréteur ou la fonction qui simule le programme. Il est ensuite possible de trouver des solutions intermédiaires. Il est par exemple possible de compiler le code dédié dans une représentation plus efficace, plus rapide à interpréter comme du code-octet ou encore de générer une fonction d'évaluation ad-hoc dans laquelle une partie de la logique du code dédié est présente.

Pour compiler un programme Esterel [51, Partie 3, Compiling Esterel], il est possible de choisir la méthode dynamique : construire un objet du langage hôte et l'évaluer. Cela correspond à implanter la sémantique comportementale d'Esterel dans la fonction d'évaluation. Les approches statiques consistent à compiler le programme vers un automate à états finis, un circuit, ou un graphe de flots de contrôle. La compilation vers un automate fini, bien que générant le code le plus efficace possible, puisqu'il décrit tous les états possibles du programme, a tendance à exploser en termes de taille d'exécutable. La représentation en circuit n'a pas ce problème de taille de code, mais comme chacune des portes logiques doivent être évaluées à chaque instant, la vitesse d'exécution du circuit est très inférieure à celle de l'automate. Dans la recherche d'une représentation plus efficace, deux compilateurs Esterel, Saxo-RT [21] et CEC [26], proposent une approche qui apparaît ensuite dans le compilateur Esterel V7 : la compilation à travers une structure de graphe de flot de contrôle. La transformation en graphe de flot de contrôle du programme étant guidée par la syntaxe, la taille de la structure est quasi-linéaire en la taille du programme source. La

représentation de l'état courant est minimale et le but de cette approche est de n'activer que les parties du programme qui sont effectivement exécutées. Cette méthode génère donc un code exécutable compact, comme la compilation en circuit, tout en approchant la vitesse d'exécution de l'automate. Le défaut de GRC est qu'il est difficile de compiler séparement des morceaux d'un programme.

C'est cette dernière solution que nous avons retenu pour compiler les programmes pendulum vers du code OCaml. On commence par détailler le processus de compilation d'un programme source vers un graphe de flot de contrôle au travers d'un schéma de compilation. On montre ensuite comment transformer ce graphe pour le rendre séquentiel et adapté à la compilation vers un langage algorithmique.

3.1 Définition du GRC

Cette technique de compilation a été proposée par le compilateur CEC [27], et une version complète est décrite dans l'ouvrage *Compiling Esterel* [51]. Elle a pour nom *GRaphCode* (GRC). Le principe est de construire un graphe de flot de contrôle (CFG) à partir du programme dont l'évaluation depuis la source (i.e. le nœud racine) jusqu'à une extrémité (i.e. le nœud puit) exécute un des instants possibles du programme selon l'environnement d'entrée. Le principe permettant à cette représentation d'être compact est qu'elle se trouve à mi-chemin entre une fonction d'interprétation et une mise à plat du programme comme dans l'automate à états finis. Le programme qui est représenté par ce CFG raisonne sur le programme synchrone d'origine. Les différents états du programme sont sous la forme de valeurs et ne sont pas représentés par la structure GRC comme ils le sont dans l'automate. Exceptés les constructions raisonnant sur le programme d'origine, les autres sont directement des tests et des émissions de signaux et ne nécessitent pas une interprétation du programme réifié, gagnant ainsi en vitesse d'exécution. De plus, le résultat de la compilation de GRC a une forme de graphe orienté sans cycle (*directed acyclic graph* ou DAG) où aucune partie du code n'est dupliquée. L'exécution du programme issu du CFG est gagnante par rapport à celle du circuit, parce que parmi les multiples chemins que le CFG contient, un seul est emprunté à chaque instant alors que l'évaluation du circuit est forcément complète.

On présente dans ce chapitre la méthode GRC à notre façon. On donne premièrement les valeurs que manipule le CFG, pour en comprendre les enjeux, puis on détaillera sa structure et sa construction.

3.1.1 L'environnement d'exécution du graphe de flot de contrôle

Le CFG peut être vu comme un programme exécutable et manipule des valeurs. On les range dans trois environnements.

Définition 2 (Environnement des signaux)**.** $\mathcal{F}$ *associe leur nom à leur description. La représentation des signaux est la même que celle rencontrée précédemment : un état de présence et une valeur transportée. De la même manière que le programme source, le graphe peut émettre un signal et changer sa valeur, ainsi que tester la présence. Cet ensemble contient aussi les signaux construits à partir de l'opérateur de liaison évènementielle.*

Définition 3 (Arbre de Sélection)**.** $\mathcal{S}$ *est l'état du programme synchrone. Il permet de garder la trace des instructions où le programme s'est arrêté l'instant précédent, pour savoir où reprendre l'exécution à l'instant suivant. Il y a donc une bijection entre les instructions non-immédiates du programme et les éléments de* $\mathcal{S}$*, pour arbre de sélection. Cet arbre a la même structure que le programme synchrone, mais ne contient que les instructions sur lesquelles le programme s'arrête, c'est-à-dire* `pause` *et ses parents. Chaque nœud contient une valeur booléenne qui est à vrai si l'instruction correspondante est sélectionnée, et faux sinon. On accède aux nœuds de l'arbre par un entier qui les identifie de façon unique.*

Définition 4 (Environnement des codes d'échappement)**.** *$\mathcal{E}$ définit l'ensemble des codes d'échappement activés. Il y a un code par label d'échappement introduit par* `trap` *dans le programme. $\mathcal{E}$ est un tableau clé-valeur d'un label vers son état booléen, activé ou non.*

3.1.2 La structure du graphe de flot de contrôle

La structure principale du graphe $\mathcal{G}$ repose sur l'exécution d'actions qui modifient l'environnement, le branchement conditionnel et le parcours de chemins simultanément. Ces constructions sont représentées dans la figure 3.1. Le graphe est dirigé, on parle de degré sortant pour le nombre d'arcs qui partent du nœud, et degré entrant pour le nombre d'arcs qui y entrent.

action : := **emit** *signal value*
| **exit** *int*
| **enter** *int*
| **return** *label*
| **local** *signal value*
| **inst** (*id*, *signal*[,*signal*])
| **sync** (*int*, *int*)

pred : := **sel** *int*
| **present** *signal*
| **sync** (*int*, *int*)
| **finished**
| **ispaused** *id*
| **code** (*label*)

node : := **call** (*action*, *node*)
| **branch** (*pred*, *node*, *node*)
| **fork** (*node*, *node*)
| **finish** | **pause**

Figure 3.1 – Langage des nœuds du graphe de flots de contrôle

Les nœuds **call**, de degré sortant 1, représentent l'exécution des *actions*. **emit** permet de mettre un signal dans l'état présent dans $\mathcal{F}$ avec une nouvelle valeur. Les actions **enter** et **exit** permettent respectivement de sélectionner et désélectionner un nœud dans $\mathcal{S}$. La sélection ne modifie que le nœud en question, mais désélectionner agit récursivement sur les fils du nœud cible. En termes de programme pendulum, cela signifie que lorsque l'on quitte une instruction contenant un bloc d'instructions on quitte aussi toutes les intructions contenues dans ce bloc. L'action **return** permet d'activer un code d'échappement utilisé dans les nœuds de synchronisation pour savoir quelle sortie doit être empruntée. Pour finir, l'action **local** permet de définir un signal local et l'affectation de sa valeur et **inst** de démarrer l'exécution d'un programme synchrone externe en passant les signaux en paramètres.

L'expression d'un branchement conditionnel est représentée par le nœud **branch**, de degré sortant 2, qui dirige le flot d'exécution vers son premier fils lorsque que l'évaluation du test rend le booleen *vrai*, et son deuxième fils sinon. Le prédicat **sel** répond *vrai* si l'instruction du programme source identifiée de façon unique par l'entier donné est sélectionnée dans $\mathcal{S}$. **sync** agit de façon similaire avec deux instructions et sert, comme son nom l'indique à synchroniser les deux instructions parallèles quand elles terminent,

en pause ou fin d'exécution. **present** a un sens similaire à l'instruction du langage synchrone et répond *vrai* si le signal est présent. **finished** ne prend aucun paramètre et répond vrai quand le programme est *terminé*. On utilise ce prédicat pour rediriger le flot de contrôle vers une feuille **finish** si un instant du programme est déclenché par le programme hôte alors qu'il est *terminé*, c'est-à-dire qu'il ne reste plus rien à exécuter. Pour finir, l'instruction **ispaused** vérifie si un programme synchrone externe exécuté est en pause ou terminé. La dernière construction de degré 2 est **fork**, qui propage le flot de contrôle dans ses deux fils simultanément pour représenter l'exécution parallèle, et **pause** et **finish**, de degré sortant 0, représentent respectivement la mise en pause et la fin du programme, mais surtout la fin de l'instant.

Remarque. *Le prédicat **code** et l'action **sync** ne sont pour l'instant pas pris en compte et seront utiles dans la sous-section 3.3.3.*

3.2 Exemples de transformation en graphe de flot de contrôle

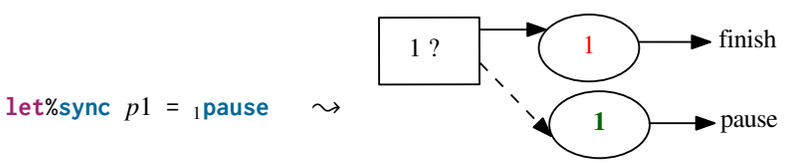
Les instructions d'un programme pendulum peuvent être instantanées ou non. Si une instruction non-instantanée est mise en pause, cela signifie que le flot d'exécution doit revenir en ce point du programme pour en continuer l'exécution. On se sert donc de $\mathcal{S}$ pour savoir si on a déjà exécuté l'instruction et on créé un branchement avec deux chemins possibles : l'instruction est sélectionnée, donc on doit la reprendre ou l'instruction n'est pas sélectionnée, donc on y entre. On va représenter par la suite les transformation des instructions sous forme de graphes de flot de contrôle comme exemples.

Définition 5 (Langage graphique CFG)**.** *Les constructions du CFG sont décrites à partir des symboles suivants. Soient $s \in \mathcal{F}$ et $n \in \mathcal{S}$.*

- *Les nœuds pour les prédicats de présence* $\boxed{s?}$ *et de sélection* $\boxed{n?}$.
- *Les branchements en fonction des valeurs vrai* $\rightarrow$ *et faux* $\dashrightarrow$.
- *Les émissions de signaux* (s)*, les sélections* (n) *et les désélections* (n).
- *L'opérateur de séquence* ; *dans les nœuds est équivalent à un arc vers un nœud **call**. Exemple :* $(n\ ;\ m) = (n) \rightarrow (m)$

Remarque. *On met en indice à gauche des instructions dans le code leur identifiant unique, pour lier plus facilement le code et la représentation du graphe. Exemple :* $_1$`pause` *est l'instruction d'identifiant unique* 1 *dans $\mathcal{S}$.*

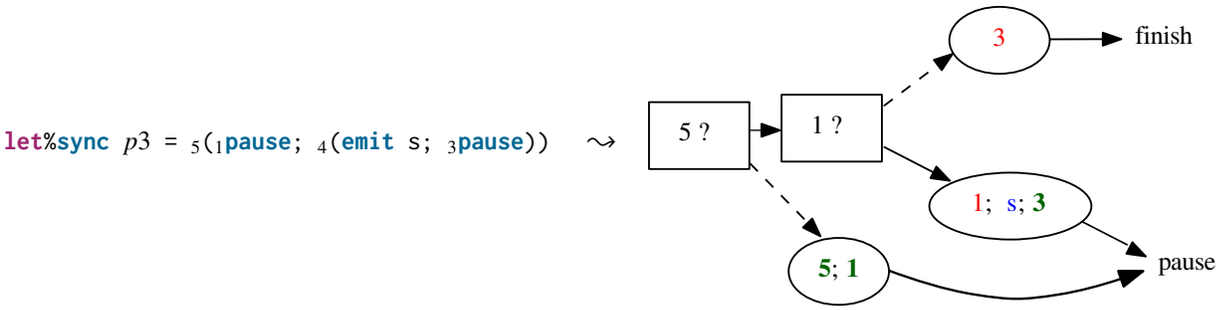
Dans l'exemple qui suit, de traduction de `pause` en flot de contrôle, on observe la divergence entre la première exécution et la seconde. Au premier instant on arrête l'exécution de l'instant en état de pause, et au second instant le programme est terminé.



Ce cas est le programme le plus petit qui puisse se mettre en pause, et donc nécessiter un branchement selon l'état. Effectivement, il n'est pas nécessaire de faire plusieurs cas si l'instruction est instantanée.

Si le programme contient des instructions qui forment des blocs, comme la séquence, ou la boucle, la construction du graphe de flot doit agir récursivement sur les sous-instructions.

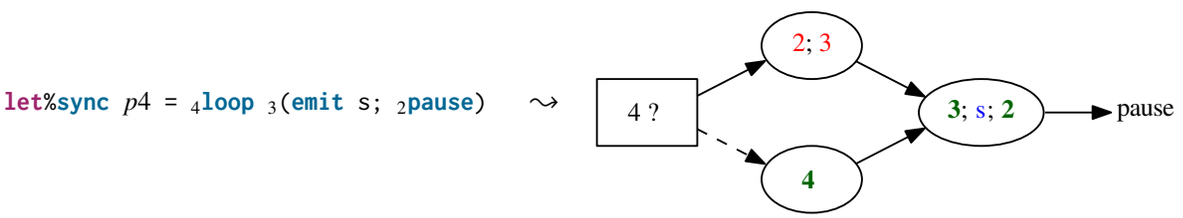
Dans l'exemple suivant, le programme se met en pause à son premier instant, émet s puis se met en pause à son second, puis se termine à son troisième instant.



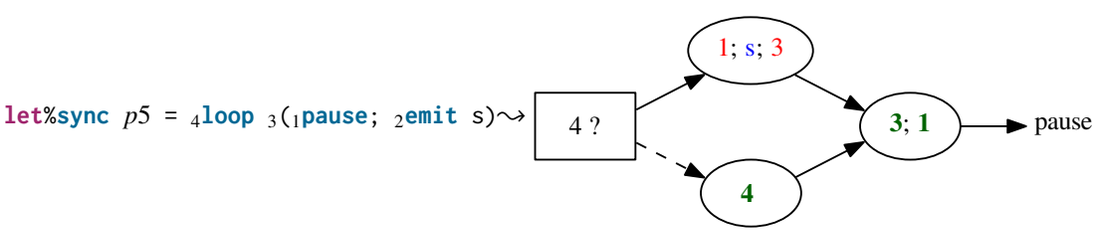
On commence par déterminer si on est déjà dans la séquence principale 5.

- Si non, on la sélectionne, ainsi que sa première instruction $_1$**pause** et on rentre dans l'état pause, fin de l'instant.
- Si oui, on vérifie si $_1$**pause** est sélectionnée.
  - Si oui, on en sort, on émet le signal s, et on entre dans $_3$**pause**, fin de l'instant sur une pause.
  - Si non, on quitte $_3$**pause** et on s'arrête sur la fin de programme

Un autre comportement intéressant est celui de la boucle. Il y a plusieurs caractéristiques de la boucle à représenter. On suppose qu'une boucle compte au moins un pause et qu'elle ne boucle pas dans l'instant. On cherche à représenter la répétition à chaque instant d'un même comportement et la réinitialisation de la boucle quand celui-ci termine On peut présenter deux exemples qui font ressortir ces caractéristiques. Dans ce premier exemple, on place la pause après l'émission du signal.



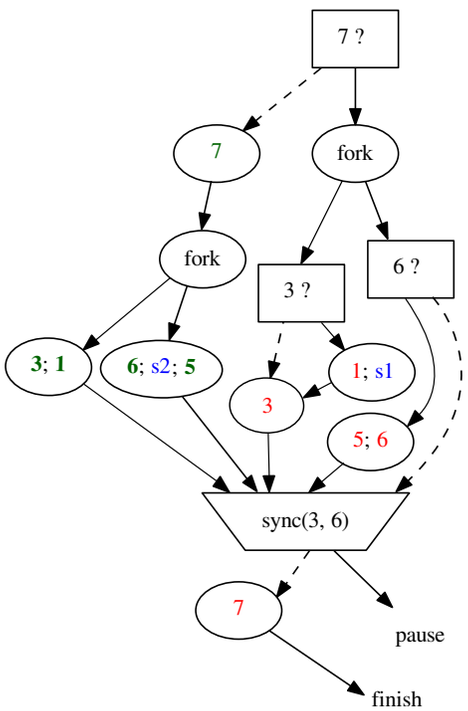
Le premier test (4?) permet de savoir si on se trouve déjà dans la boucle et donc à la première exécution. Si on n'est pas dans la boucle, on y entre (4), et à la suite on entre dans la séquence, on émet *s* et on entre dans $_2$**pause**, fin de l'instant. Si on est déjà dans la boucle, on quitte l'instruction $_2$**pause**, et on exécute la dernière suite d'actions. La seule divergence est donc au premier instant, où l'on est pas encore dans le corps de la boucle et on doit y entrer. Par contre, l'émission du signal a lieu à chaque instant. Voyons maintenant une autre configuration avec la boucle dans un exemple où la pause arrive en premier dans le corps de la boucle.



Les deux cas du premier sont visibles, puisque qu'il n'y a pas d'émission dans ce cas là. Si on est pas dans le corps de la boucle, on sélectionne successivement ce dernier (4) puis la séquence (3) et la pause ((1), fin de l'instant. Si on est déjà dans le corps, on vérifie si on dans l'instruction pause, et si oui, on en sort. On émet ensuite s, on quitte la séquence puis on sélectionne à nouveau la séquence et l'instruction **pause** puis c'est la fin de l'instant. Une observation plus générale est que l'exécution d'une boucle est visiblement sur le graphe infinie puisqu'il n'y a aucun nœud **finish** présent dans le graphe.

Pour terminer cette suite d'exemples, on construit un graphe de flot de contrôle de programme avec du parallélisme, pour comprendre l'utilisation de **fork** et **sync**. L'exemple suivant correspond à une mise en parallèle des corps des deux boucles précédentes. D'un côté, on émet un signal et on se met en pause, et de l'autre, on se met en pause et on émet un signal.

```
let%sync p6 = 3(1pause; emit s1) 7|| 6(emit s2; 5pause)   ⇝
```



On commence par tester si le parallèle est sélectionné : première exécution ou non. S'il ne l'est pas, on le sélectionne (7) et on divise le flot de contrôle en deux avec **fork** :

- Dans la branche de droite, on entre dans l'opérande de droite (6), on émet $s2$ et on entre dans 5**pause**.
- Dans la branche de gauche, on entre dans l'opérande de gauche (3), puis dans l'instruction 1**pause**.

Le flot de contrôle s'arrête au niveau du nœud **sync** et si au moins une tâche est encore active ($\boxed{3\,?} \vee \boxed{6\,?}$). Si une des deux séquences est encore sélectionnée, on redirige le flot vers pause, sinon vers la fin du programme. Si l'instruction parallèle était déjà sélectionnée, on divise aussi le flot de contrôle en deux branches :

- Dans la branche de droite on quitte 6. C'est la fin de l'opérande de droite.
- Dans la branche de gauche, on sort de 1**pause**, on émet $s1$, on sort de 3.

On atteint à nouveau le parallèle qui se comporte comme précédemment.

Maintenant que l'on a exposé l'utilisation du graphe de flot de contrôle pour représenter l'instant d'un

programme synchrone au travers d'exemples, on donne les règles de transformation.

3.3 Le schéma de compilation : Surface et Profondeur

Dans la section précédente, on a observé une séparation des cas entre l'instant où l'on exécute une instruction depuis le début, et les instants où l'instruction est en cours d'exécution. Pour chaque instruction, on va donc construire deux CFGs pour représenter ces deux comportements que l'on appelera Surface et Profondeur.

3.3.1 Notations et définitions

Définition 6 (Les fonctions Surface et Profondeur)**.**
Soit p un programme synchrone représenté par son AST,

- $\mathcal{G}(p)$ *est la fonction qui construit, à partir de p, son graphe de flot de contrôle pour l'exécution d'un instant*
- $\mathcal{S}(p)$ *est la fonction qui construit, à partir de p, son graphe de* Surface*. Le graphe Surface représente le CFG décrivant le premier instant d'une instruction.*
- $\mathcal{D}(p)$ *est la fonction qui construit, à partir de p, son graphe* Profondeur *(*depth *en Anglais). Celui-ci représente le CFG décrivant les instants qui suivent le premier pour une instruction.*

L'opération de construction agit récursivement sur l'arbre de syntaxe abstraite de p, de sorte que le graphe Surface d'une instruction contient aussi le graphe Surface des instructions imbriquées, jusqu'à une pause, ou la fin du programme si son contenu est instantané. De même, le graphe de profondeur d'une instruction peut diriger, à terme, le flot de contrôle vers le graphe Surface de l'instruction suivante. Les fonctions $\mathcal{S}(p)$ et $\mathcal{D}(p)$ sont donc mutuellement récursive. En pratique, si on donne i et j comme instructions avec j qui suit l'exécution de i, $\mathcal{D}(i)$ peut appeler $\mathcal{S}(j)$, mais il n'est pas possible que $\mathcal{S}(i)$ appelle $\mathcal{D}(j)$. Dit autrement, il n'est pas possible d'atteindre au même instant le graphe Surface, et le graphe Profondeur d'une instruction i. Soit une instruction est instantanée et son graphe Profondeur est vide, c'est-à-dire ni pause, ni reprise, soit son graphe Profondeur sera atteint après le premier instant où elle est exécutée.

Définition 7 (La fonction d'évaluation de la transformation)**.**
Soit p un programme synchrone représenté par son AST,

$\rightsquigarrow$ *est la fonction d'évaluation des fonctions de transformation* $\mathcal{G}(p)$, $\mathcal{S}(p)$ *et* $\mathcal{D}(p)$*. La transformation prend en entrée un programme pendulum sous la forme d'un arbre de syntaxe abstraite et génère, par application récursive, un arbre de flot de contrôle GRC.*

On ajoute les arguments de transformation ω, κ *et* Ω *aux fonctions Surface et Profondeur tel que :*

- ω *est le CFG qui encode la continuation où p termine.*
- κ *est le CFG qui encode la continuation où p se met en pause.*
- Ω *est l'ensemble des CFG qui encodent les continuations accessibles par* $p =$ ***exit*** l *où l est un label d'échappement. C'est une association de valeurs ordonnées du label défini le plus à l'extérieur vers le label le plus à l'intérieur (et donc le plus proche de l'instruction courante)*
- $\mathcal{G}(p)$ *ne prend pas d'argument car il initialise leurs valeurs et la récursion.*

Remarque. *Parfois la mise en pause n'est décidée qu'après un test de synchronisation de deux tâches parallèles, comme dans l'exemple de la section 3.2. C'est pour cette raison* κ *correspond à une continuation vers un sous-graphe et pas nécessairement à un arc vers le* ***pause*** *dans le CFG.*

La source est le nœud sur lequel démarre l'exécution d'un instant, et les nœuds puits sont ceux où l'exécuction d'un instant se termine. Pour construire le CFG récursivement, on part des feuilles de l'AST et on remonte à son origine. Le CFG est donc construit des puits à la source. On définit pour cela la règle d'initialisation, qui construit à la fois la source et le puit du graphe et qui est la même pour tous les programmes. la fonction $\mathcal{G}(p)$ ne prend pas les arguments de continuation parce qu'il n'y pas de suite dans l'exécution de p.

$$\mathcal{G}(p) \quad \leadsto \quad branch\ (finished, finish, branch\ (sel\ p, d, s)))$$

$$\text{avec} \begin{cases} d = \mathcal{D}(p, finish, pause, \{\}) \\ s = \mathcal{S}(p, finish, pause, \{\}) \end{cases}$$

Les puits du CFG, **finish** et **pause** sont construits ici et passés en argument à la fonction d'évaluation tel que si p se met en pause, sa continuation soit **pause**, et que si il termine, sa continuation soit **finish**. La source correspond à l'opérande de droite de l'opérateur $\leadsto$. Elle vérifie si le programme est exécutable (i.e. n'est pas **finished**), avec un arc vers **finish** le cas échéant. On peut donc réexécuter un programme terminé, mais le flot de contrôle ira directement sur ce nœud puit. La branche correspondant à $finished = faux$ sert à vérifier si cette exécution est la première, et diverger vers $\mathcal{S}(p)$ ou $\mathcal{D}(p)$, selon l'état de sélection de p. On peut maintenant donner les règles de construction des graphes Surface et Profondeur pour chaque instruction.

Pour des besoins de concision et de clarté, on replace la construction **call** par le symbole de l'opérateur binaire d'appel $\blacktriangleright$, tel que

$$a_1 \blacktriangleright a_2 = call(a_1, a_2)$$

3.3.2 Le schéma de compilation pour chaque instruction

nothing

L'instruction vide se traduit, en Surface et en Profondeur par la même opération : un arc sortant directement vers sa continuation de fin sans nœud. En fait, cette instruction est ignorée et n'apparaît pas dans le CFG.

$$\mathcal{S}(\texttt{nothing}, \omega, \kappa, \Omega) \leadsto \omega \qquad \mathcal{D}(\texttt{nothing}, \omega, \kappa, \Omega) \leadsto \omega$$

pause

L'instruction de mise en pause est elle différente durant les deux instants. Au premier instant, $\mathcal{S}(pause)$, on sélectionne l'instruction dans $\mathcal{S}$ et on dirige le flot de contrôle vers sa continuation de pause, κ. Quand on reprend l'exécution, on désélectionne p et on construit un arc sortant vers sa continuation de fin ω.

$$\mathcal{S}(\texttt{pause}, \omega, \kappa, \Omega) \leadsto enter\ p \blacktriangleright \kappa \qquad \mathcal{D}(\texttt{pause}, \omega, \kappa, \Omega) \leadsto exit\ p \blacktriangleright \omega$$

emit

Pour l'émission de signaux, on manipule une instruction instantanée. Cela est représenté par le fait que l'on doit la faire apparaître dans le graphe Surface mais pas dans le graphe Profondeur. On notera que

l'instruction n'est pas sélectionnée, et qu'elle ne fait que mettre à jour le signal dans $\mathcal{F}$. Dans le graphe Profondeur, on a simplement un arc vers la continuation de fin.

$$\mathcal{S}(\texttt{emit}\ s, \omega, \kappa, \Omega) \rightsquigarrow \mathit{emit}\ s \blacktriangleright \omega \qquad \mathcal{D}(\texttt{emit}\ s, \omega, \kappa, \Omega) \rightsquigarrow \omega$$

atom

L'exécution atomique d'expressions du langage hôte subit exactement le même traîtement, étant aussi une instruction instantanée.

$$\mathcal{S}(\texttt{atom}\ f, \omega, \kappa, \Omega) \rightsquigarrow \mathit{atom}\ f \blacktriangleright \omega \qquad \mathcal{D}(\texttt{atom}\ f, \omega, \kappa, \Omega) \rightsquigarrow \omega$$

La séquence

On commence, avec l'opérateur [;], à traiter les instructions plus complexes qui forment des blocs. Le graphe construit doit propager le flot de contrôle de l'opérande de gauche q à l'opérande de droite r. Le graphe Surface se construit comme la sélection de p, avec $p = q; r$, suivie de la Surface de la première instruction, q.

$$\mathcal{S}(q\ ;\ r, \omega, \kappa, \Omega) \rightsquigarrow \mathit{enter}\ p \blacktriangleright \mathcal{S}(q, \omega\prime, \kappa, \Omega)$$

$$\text{avec } \omega\prime = \mathcal{S}(r,\ \mathit{exit}\ p \blacktriangleright \omega, \kappa, \Omega)$$

On voit que les arguments de la fonction d'évaluation sont modifiés. $\omega\prime$ correspond à la Surface de r, qui doit suivre q s'il est instantané, suivie de $\omega\prime\prime$. Ce dernier graphe encode la fin de p et la continuation qui poursuit l'exécution après p, c'est-à-dire le ω donné au moment de l'appel. Pour construire le graphe Profondeur, on ajoute un test, qui va vérifier si la première instruction est sélectionnée. Si oui, le flot de contrôle doit être propagé sur la reprise de q, donc on construit son graphe Profondeur. Si non, c'est que la seconde exécution s'est mise en pause et on construit le graphe Profondeur de cettte dernière.

$$\mathcal{D}(q\ ;\ r, \omega, \kappa, \Omega) \rightsquigarrow \mathit{branch}(\mathit{sel}\ q, \mathcal{D}(q, \omega\prime, \kappa, \Omega), \mathcal{D}(r, \omega\prime\prime, \kappa, \Omega))$$

$$\text{avec} \begin{cases} \omega\prime = \mathcal{S}(r, \omega\prime\prime, \kappa, \Omega) \\ \omega\prime\prime = \mathit{exit}\ p \blacktriangleright \omega \end{cases}$$

On remarque que les arguments de $\mathcal{D}(q)$ et de $\mathcal{D}(r)$ sont différents. Effectivement, si $\mathcal{D}(q)$ termine q, sa continuation, $\omega\prime$, exécute r. La continuation de fin de r est alors $\omega\prime\prime$, soit la terminaison de p.

Le parallèle

La construction du graphe de deux tâches en parallèle se fait par la composition des nœuds **fork** et **sync**. La Surface correspond à la sélection de p, puis un nœud **fork** qui propage le flot de contrôle aux graphes de Surface de q et r, les deux opérandes.

$$\mathcal{S}(q \parallel r, \omega, \kappa, \Omega) \rightsquigarrow \mathit{enter}\ p \blacktriangleright \mathit{fork}\ (\mathcal{S}(q, \omega\prime, \kappa\prime, \Omega\prime), \mathcal{S}(r, \omega\prime, \kappa\prime, \Omega\prime))$$

$$\mathit{avec} \begin{cases} \kappa\prime = \omega\prime = \mathit{branch}(\mathit{sync}(\Omega, q, r), \kappa, \mathit{exit}\ p \blacktriangleright \omega) \\ \Omega\prime = \mathit{map}\ [l := \kappa\prime]\ \Omega \end{cases}$$

Le point à souligner est que l'on modifie à la fois κ et ω en une même valeur pour que toutes les continuations mènent vers le nœud sync. Celui-ci est connecté à un arc vers la continuation de pause de p si l'une des deux instructions est sélectionnée, sinon, vers la desélection de p et sa continuation de fin ω.

Ensuite, on modifie l'ensemble Ω pour faire pointer toutes les continuations d'échappement vers le nœud sync. On y ajoute un paramètre qui enregistre l'état courant de l'environnement Ω. On notera que $\Omega\prime$ est passé à la fonction de transformation, alors que Ω est conservé dans le nœud. Dans l'état actuel, les échappements, s'ils sont imbriqués dans un parallèle, sautent au nœud sync le plus proche. Dans la sous-section 3.3.3, on ajoute une passe de transformation après la génération du graphe pour propager les échappements après les nœuds de synchronisation.

La construction Profondeur est très similaire à Surface. C'est un nouveau graphe à base de **fork** qui teste pour les deux instructions si elles sont sélectionnées, les exécute si oui et passe directement à la synchronisation sinon.

$$\mathcal{D}(q \parallel r, \omega, \kappa, \Omega) \rightsquigarrow \mathit{fork}\ (\mathit{branch}(\mathit{sel}\ q, q\prime, \kappa\prime), \mathit{branch}(\mathit{sel}\ r, r\prime, \kappa\prime)$$

$$\mathit{avec} \begin{cases} q\prime = \mathcal{D}(q, \omega\prime, \kappa\prime, \Omega\prime) \\ r\prime = \mathcal{D}(r, \omega\prime, \kappa\prime, \Omega\prime) \end{cases} \quad \mathit{et} \begin{cases} \kappa\prime = \omega\prime = \mathit{branch}(\mathit{sync}(\Omega, q, r), \kappa, \mathit{exit}\ p \blacktriangleright \omega) \\ \Omega\prime = \mathit{map}\ [l := \kappa\prime]\ \Omega \end{cases}$$

loop

La boucle apporte la notion de répétition dans la représentation du graphe. Cela ne signifie pas pour autant qu'une boucle puisse être présente dans le graphe puisqu'il est acyclique et représente l'exécution de l'instant. Néanmoins, de multiples exécutions doivent amener à repasser par le même chemin dans le graphe. On représente aussi le fait que si l'exécution du corps de la boucle termine au premier instant, le flot d'exécution ne doit pas poursuivre ni à la continuation de fin de p ni réexécuter q, mais poursuivre vers la continuation de pause.

$$\mathcal{S}(\texttt{loop}\ q, \omega, \kappa, \Omega) \rightsquigarrow \mathit{enter}\ p, \omega, \kappa, \Omega \blacktriangleright\ \mathcal{S}(q, \omega, \kappa, \Omega)$$

Pour cette raison, la continution de fin de $\mathcal{S}(q)$ est κ. La reprise d'une boucle se traduit par la reprise de l'exécution de son corps, $\mathcal{D}(q)$, qui reçoit comme continuation de fin le graphe Surface de q. On retrouve dans la règle de transformation, qui suit la notion de répétition, dans le fait de repasser par le même chemin : $\mathcal{S}(q)$. On peut observer ce phénomène dans l'exemple de transformation du programme $p4$ dans la section 3.2.

$$\mathcal{D}(\texttt{loop}\ q, \omega, \kappa, \Omega) \rightsquigarrow \mathcal{D}(q, \omega\prime, \kappa, \Omega)$$

$$\text{avec } \omega\prime = \mathcal{S}(q, \kappa, \kappa, \Omega)$$

let in

Faire apparaître dans le graphe la définition locale de signal permet de conserver le moment où sa valeur lui est affectée, qui correspond à l'évaluation d'une expression du langage hôte. Le graphe commence par la traditionnelle sélection de p puis l'action **local** qui initialise le signal et dirige un arc vers le graphe Surface de l'instruction dans laquelle le signal est défini.

$$\mathcal{S}(\texttt{let}\ s\ \texttt{in}\ q, \omega, \kappa, \Omega) \rightsquigarrow \mathit{enter}\ p \blacktriangleright \mathit{local}\ s \blacktriangleright \mathcal{S}(q, \omega\prime, \kappa, \Omega)$$

$$\text{avec } \omega\prime = \mathit{exit}\ p \blacktriangleright \omega$$

Le graphe Profondeur se traduit simplement par la reprise de q.

$$\mathcal{D}(\texttt{let}\ s\ \texttt{in}\ q, \omega, \kappa, \Omega) \rightsquigarrow \mathcal{D}(q, \omega\prime, \kappa, \Omega)$$

$$\text{avec } \omega\prime = \mathit{exit}\ p \blacktriangleright \omega$$

present

Il faut maintenir le choix de branche qui a été fait au premier instant de l'instruction et qu'aux instants suivants, on poursuive son exécution quelques soit l'état de présence du signal. Le test de présence est exprimé par le prédicat **signal** qui accède à $\mathcal{F}$.

$$\mathcal{S}(\texttt{present}\ s\ q\ r, \omega, \kappa, \Omega) \rightsquigarrow \mathit{enter}\ p \blacktriangleright \mathit{branch}\ (\mathit{signal}\ s, \mathcal{S}(q, \omega\prime, \kappa, \Omega), \mathcal{S}(r, \omega\prime, \kappa, \Omega))$$

$$\text{avec } \omega\prime = \mathit{exit}\ p \blacktriangleright \omega$$

Selon le résultat du test, on accède soit au graphe Surface de q soit à au graphe Surface de r. Aux instants suivants, on ne teste plus sur le signal, mais sur la sélection de q. Si cette instruction est sélectionnée, c'est qu'elle l'a été précédemment, sinon, c'est r que l'on doit poursuivre.

$$\mathcal{D}(\texttt{present}\ s\ q\ r, \omega, \kappa, \Omega) \rightsquigarrow \mathit{branch}\ (\mathit{sel}\ q, \mathcal{D}(q, \omega\prime, \kappa, \Omega), \mathcal{D}(r, \omega\prime, \kappa, \Omega))$$

$$\text{avec } \omega\prime = \mathit{exit}\ p \blacktriangleright \omega$$

suspend

Cette opération met en pause l'exécution d'un bloc d'instructions si le signal en argument est présent. Il est assez commode de représenter ce comportement dans la représention en CFG, puisqu'à chaque instant, le flot traverse toutes les instructions parentes de celle en cours d'exécution, et il est alors facile de détecter quand la mise en pause doit avoir lieu. Le graphe Surface ne fait qu'exécuter son corps.

$$\mathcal{S}(\texttt{suspend}\ s\ q, \omega, \kappa, \Omega) \overset{\omega\prime,\kappa,\Omega}{\rightsquigarrow} \mathit{enter}\ p \blacktriangleright \mathcal{S}(q)$$

$$\text{avec } \omega\prime = \mathit{exit}\ p \blacktriangleright \omega$$

C'est dans le graphe Profondeur que l'on exécute le prédicat **signal**. Si le signal est présent, on propage le flot de contrôle à la continuation de pause κ, sinon, vers la poursuite de q.

$$\mathcal{D}(\texttt{suspend}\ s\ q, \omega, \kappa, \Omega) \rightsquigarrow \mathit{branch}\ (\mathit{signal}\ s, \kappa, \mathcal{D}(q, \omega\prime, \kappa, \Omega))$$

$$\text{avec } \omega\prime = \mathit{exit}\ p \blacktriangleright \omega$$

run

Pour exécuter un programme synchrone externe, on utilise les constructions **inst**, qui initialise une instance du programme en mémoire, et **ispaused** qui exécute le programme et teste s'il est en état de pause ou de fin après. L'instantiation ne se fait que dans le graphe Surface.

$$\mathcal{S}(\texttt{run}\ id\ signals, \omega, \kappa, \Omega) \rightsquigarrow enter\ p \blacktriangleright inst(id, signals) \blacktriangleright\ branch\ (ispaused\ id, \kappa, \omega\prime)$$

$$\text{avec } \omega\prime = exit\ p \blacktriangleright \omega$$

On dirige le flot de contrôle vers la continuation de pause κ si le programme termine son instant en pause, sinon vers la continuation de fin.

$$\mathcal{D}(\texttt{run}\ id\ signals, \omega, \kappa, \Omega) \overset{\omega\prime,\kappa,\Omega}{\rightsquigarrow} branch\ (ispaused\ id, \kappa, \omega\prime)$$

$$\text{avec } \omega\prime = exit\ p \blacktriangleright \omega$$

trap

Quand on rencontre un bloc d'échappement, on construit un nouveau chemin pour la fin des instructions du bloc, une nouvelle continuation. On doit stocker cette continuation et la lier à un label qui sera utilisé par **exit**. La continuation est stockée dans l'environnement Ω

$$\mathcal{S}(\texttt{trap}\ l\ q, \omega, \kappa, \Omega) \rightsquigarrow enter\ p \blacktriangleright \mathcal{S}(q, \omega\prime, \kappa, \Omega\prime)$$

$$\text{avec} \begin{cases} \omega\prime = exit\ p \blacktriangleright \omega \\ \Omega\prime = \{(l, \omega)\} \cup \Omega \end{cases}$$

On construit un nouveau $\Omega\prime$ qui contient le label d'échappement l lié à la continuation de fin de l'instruction **trap**. Effectivement, quand ce label est activé, le flot de contrôle saute à la fin de l'instruction. Excepté cela, la traduction ne consiste qu'à construire le graphe Surface de q, puis son graphe Profondeur.

$$\mathcal{D}(\texttt{trap}\ l\ q, \omega, \kappa, \Omega) \rightsquigarrow \mathcal{D}(q, \omega\prime, \kappa, \Omega)$$

$$\text{avec } \omega\prime = exit\ p \blacktriangleright \omega$$

exit

C'est ici que l'on voit les conséquences des modifications apportées à l'environnement Ω par le **trap** et le parallèle.

$$\mathcal{S}(\texttt{exit}\ l, \omega, \kappa, \Omega) \rightsquigarrow return\ l \blacktriangleright\ \Omega(l) \qquad \mathcal{D}(\texttt{exit}\ l, \omega, \kappa, \Omega) \rightsquigarrow \omega$$

Cette instruction est instantanée, donc elle n'a pas de graphe Profondeur, mais son graphe Surface correspond à rediriger l'exécution vers la continuation, stockée au label l dans Ω en ayant préalablement activé le code d'échappement correspondant au label l.

3.3.3 Intégration des arcs d'échappement

Dans la section précédente, on a vu que le nœud de synchronisation permet de tester deux indices dans l'arbre de sélection pour connaître l'état des deux tâches et savoir si l'exécution doit attendre l'instant suivant en ce point ou continuer. Néanmoins, pour des raisons d'hypothèses d'ordonnancement, on ne souhaite pas qu'un chemin puisse commencer par un arc qui sorte d'un **fork** sans atteindre à termes le nœud de synchronisation correspondant. C'est pourquoi, lorsque l'on traverse un parallèle, on enregistre le nœud de synchronisation comme une continuation correspondant à l'échappement de tous les labels accessibles depuis l'intérieur du parallèle. Si un échappement a lieu, il doit « dépiler » tous les parallèles dans lequel il est imbriqué en traversant chaque nœud de synchronisation correspondant. Une fois un nœud de synchronisation atteint, on ne doit pas continuer l'exécution normalement, mais déclencher à nouveau un échappement avec le même label. On utilise pour cela la continuation stockée dans l'environnement Ω du nœud **sync**. On pourrait l'écrire en français de la façon suivante :

Pour chaque label de Ω dans l'ordre décroissant de distance (du label le plus englobant au label le plus proche), si ce label renvoie à la valeur vrai *dans $\mathcal{E}$, on saute à la continuation renvoyée par ce label dans Ω.*

Un nœud **sync** n'est plus seulement un test à deux branches mais un test à n branches où la vérification se fait dans un ordre précis. La solution est de transformer le graphe pour faire une décomposition en tests à deux branches. Pour cela on traîte les deux instructions que nous avions laissé de côté en début de chapitre.

$$action \ ::= \ \textbf{sync} \ (int, int)$$

$$pred \ ::= \ \textbf{code} \ (label)$$

Le principe de cette transformation est de remplacer chaque nœud **branch sync** par un nœud **call sync**. Ce dernier va jouer de rôle de nœud de réference et indicateur pour l'ordonnancement. On peut alors déplacer le test de synchronisation et construire les tests d'échappement autour de celui-ci. On cherche les occurrences suivantes.

$$s := branch \ (sync \ (\Omega, p, q), \kappa, \omega)$$

Si deux codes d'échappement sont présents à la fois, on doit sélectionner le bloc d'échappement le plus englobant. On parcourt Ω dans l'ordre de définition inverse et on construit un arbre récursivement pour chaque $\Omega(l_i) \ldots \Omega(l_0)$, ou l_i est plus englobant que l_{i-1}. On décrit l'application de la fonction de transformation τ de la façon suivante :

$$\tau_i(s) \rightarrow \begin{cases} \text{pour } i \neq 0, \ branch \ (code \ l_i, \ \Omega(l_i), \ \tau_{i-1}(s)) \\ \text{pour } i = 0, \ branch \ (code \ l_0, \ \Omega(l_0), \ s) \end{cases}$$

On parcourt Ω et on génère un nœud **code** pour chaque l_i. Un nœud **code** teste la valeur associée à l_i dans $\mathcal{E}$.

On montre l'application de la transformation τ sur un graphe démarré par un nœud de synchronisation dans la figure 3.2. Cette dernière est extraite d'un graphe plus grand généré depuis un véritable programme compilé.

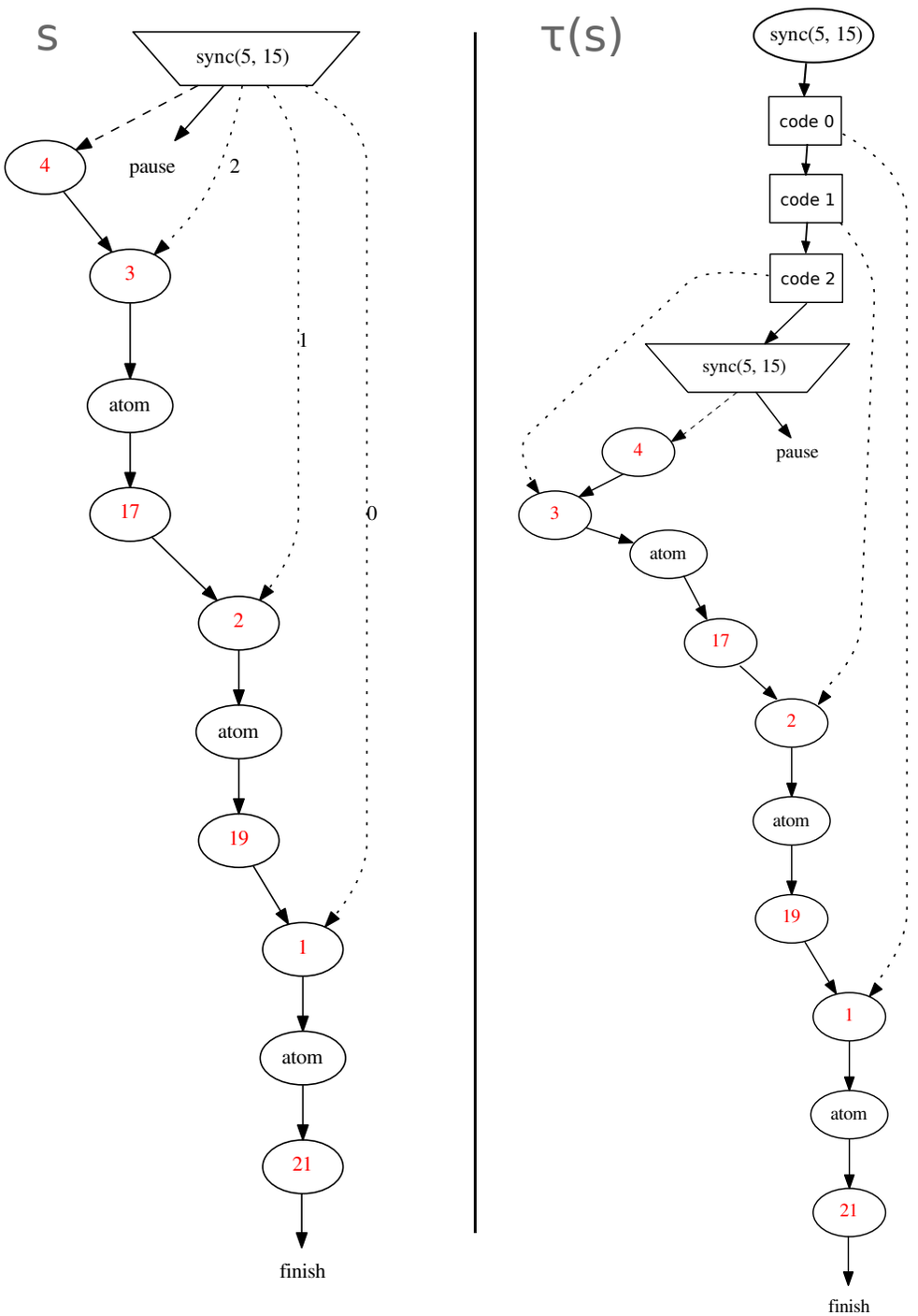


FIGURE 3.2 – Le graphe s et la version avec échappement de parallèles, $\tau(s)$

Sur la figure de gauche on observe le graphe s où les arcs en pointillés sont les arcs stockés dans Ω.

Dans le panneau de droite, on observe la transformation et l'ajout des nœuds de test **code** ainsi que le remplacement du nœud de synchronisation. Des nœuds **code** sortent les chemins en pointillés, signifiant que l_i est associé à vrai dans $\mathcal{E}$.

Une fois cette partie réalisée, le graphe est dans une structure adaptée pour être soumis à l'ordonnancement des chemins parallèles.

3.3.4 Remarque sur l'implémentation : le partage maximal

La représentation du graphe $\mathcal{G}(p)$ peut grandir rapidement si l'on applique naïvement les règles données dans les parties précédentes. De plus, certains parties du graphe seront dupliquées et le résultat sera alors un arbre et non un graphe acyclique orienté. Il devient nécessaire d'adopter une optimisation qui résoud ces deux problèmes.

La mémoïsation est une instanciation du paradigme de la programmation dynamique consistant à conserver les constructions intermédiaires d'un structure de données construite récursivement, pour économiser les calculs ou la mémoire, au cas où un même argument est passé à nouveau à la fonction (comme dans Fibonacci ou la fonction Factorielle). La différence avec la programmation dynamique est que l'on utilise spécifiquement une table de hachage et non un tableau d'entier. Lorsque l'on applique la mémoïsation à des structures de données complexes, comme des arbres [30] ou des graphes, on parle de partage maximal [34], ou *hashconsing*.

Le partage maximal permet ici de sauvegarder les graphes intermédiaires construits pour chaque instruction, à chaque appel à $\mathcal{S}(p)$ et $\mathcal{D}(p)$. Par exemple, quand on construit $\mathcal{D}(\texttt{loop}\ q)$, on doit construire $\mathcal{S}(q)$, qui a déjà été construit dans $\mathcal{S}(\texttt{loop}\ q)$. On peut donc simplement réutiliser le graphe déjà construit. Cela permet d'obtenir un graphe le plus compact possible.

En OCaml, on peut écrire une fonction de mémoïsation générique qui prend en paramètre une fonction et qui, avant chaque appel récursif accède à une table de hachage dans laquelle sont stockés les résultats précédents pour vérifier si on peut le réutiliser. On peut l'écrire de la façon suivante :

```
let memo_rec_build h f =
  let rec g p =
    try Hashtbl.find h p with
    | Not_found →
      let y = f g p in
      Hashtbl.add h p y; y
  in g
```

Dans ce code `f` correspond aux fonctions Surface ou Profondeur. On peut alors les construire non récursives, avec un paramètre supplémentaire, qui sera utilisé pour l'appel récursif à `memo_rec_build`.

```
let surface = memo_rec_build (Hashtbl.create 19) (fun surface → ...)
let depth = memo_rec_build (Hashtbl.create 19) (fun depth → ...)
```

Si la structure de données est proprement partagée, on peut se servir de cette hypothèse pour parcourir le graphe efficacement et avoir un encodage plus précis de la sémantique du graphe. Les graphes présentés dans la section 3.2 sont générés directement à partir des graphes en mémoire et leur forme de DAG n'est possible que grâce au partage maximum. Le partage est forcé en un point plus particulier qui est la construction des nœuds de synchronisation pour l'instruction parallèle. Ils sont construits à plusieurs endroits, mais il n'est nécessaire de n'en construire qu'un par instruction parallèle. Le partage devient

indispensable lorsque l'on effectue des lectures ou des modifications du graphe, comme la génération de code ou l'ordonnancement. Pour considérer un test comme une unité et pouvoir l'ordonnancer ou générer le code correspondant, il faut pouvoir détecter le nœud de jointure des deux branches du test, et donc où il se termine. Détecter la présence d'un même nœud sur deux chemins efficacement n'est possible que si l'égalité physique est strictement équivalente à l'égalité structurelle. Les figures de graphes de flot de contrôle des programmes pendulum présentés dans la partie 3.2 représentent fidèlement le graphe en mémoire et le lien entre les nœuds : un arc est un pointeur du nœud source vers un nœud en mémoire.

3.4 Ordonnancement statique

Un programme pendulum se compile vers un programme séquentiel adapté au modèle d'exécution du navigateur Web. Jusque là, la méthode de compilation produit un graphe de flot de contrôle et son environnement à partir d'un programme représenté par son arbre de syntaxe abstraite. Néanmoins, le CFG construit comporte encore l'expression du parallélisme qu'il n'est pas possible de conserver si l'on veut extraire du code séquentiel de ce graphe. Il est donc nécessaire d'appliquer au graphe une transformation pour supprimer les occurrences de la construction **fork** qui cause une duplication du flot de contrôle. La suppression **fork** n'est pas sans conséquences puisqu'elle implique la linéarisation des chemins sortants de ce nœud. Le GRC étant une représentation à mi-chemin entre une compilation directe et un programme qui raisonne sur le programme synchrone, il est possible de construire statiquement un entrelacement des nœuds, et donc un ordonnancement qui respecte les contraintes de lecture et écriture des signaux. Si on reprend l'exemple *p*6 de la section 3.2, il est facile de produire une linéarisation ce graphe. On rappelle la définition de ce programme :

```
let%sync p6 = 3(1pause; emit s1) 7|| 6(emit s2; 5pause)
```

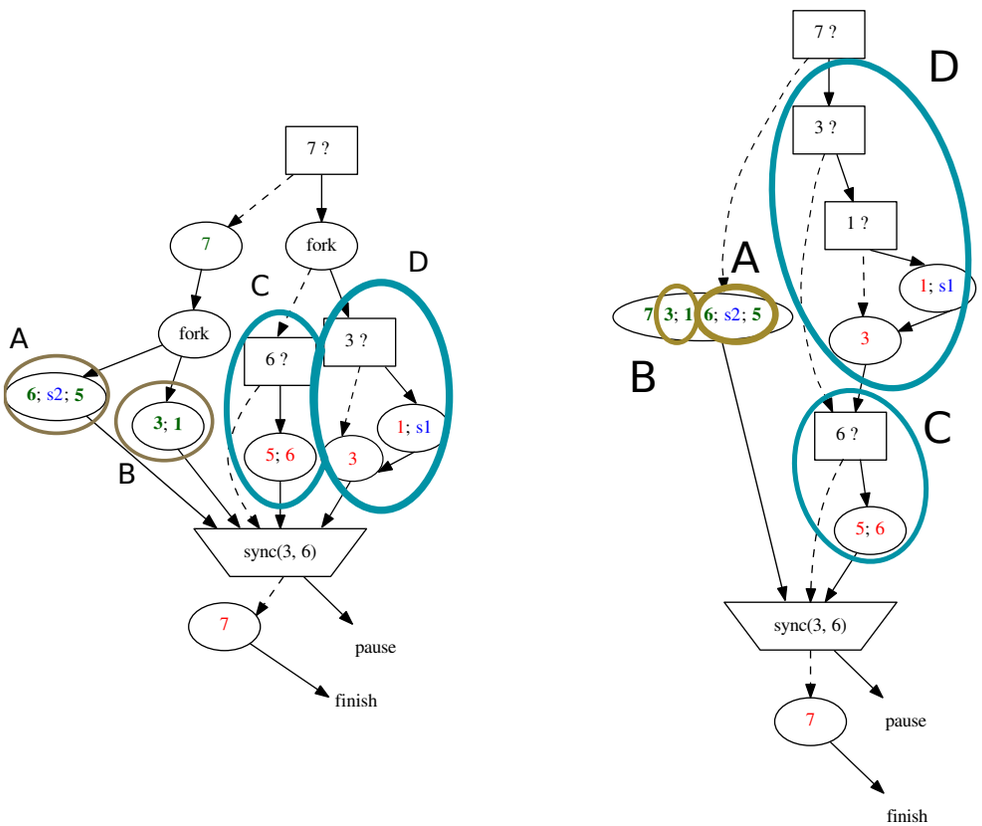


FIGURE 3.3 – Entrelacement des chemins pour le graphe du programme *p*6

L'ordonnancement de ce programme (figure 3.3) nécessite l'entrelacement des graphes *A* et *B* ensemble, et des graphes *C* et *D* ensemble. Cette opération ne pose pas de problème en particulier, puisqu'il n'y a aucun dépendance entre les nœuds de *A* et *B*, et de même pour *C* et *D*. On peut alors choisir une mise

en séquence arbitraire des sous-graphes concernés. En l'occurrence, on choisit d'ordonnancer d'abord le graphe accessible depuis le premier arc rencontré dans la structure de données du nœud **fork**.

On peut observer des programmes moins triviaux où le problème de trouver un ordonnancement nécessite une modification plus profonde de la structure du graphe que le programme précédent. Si par exemple on définit le programme :

```
let%sync p7 =
  present s1 (emit s2;
    present s3
      !(print_endline "hello3"))
  ||
  present s4 (present s2 (
    !(print_endline "hello2");
    emit s3))
```

On présente ensuite le graphe de ce programme réduit à sa forme minimale (figure 3.4) dans lequel on ne conserve que la partie exprimant le parallélisme le moins trivial.

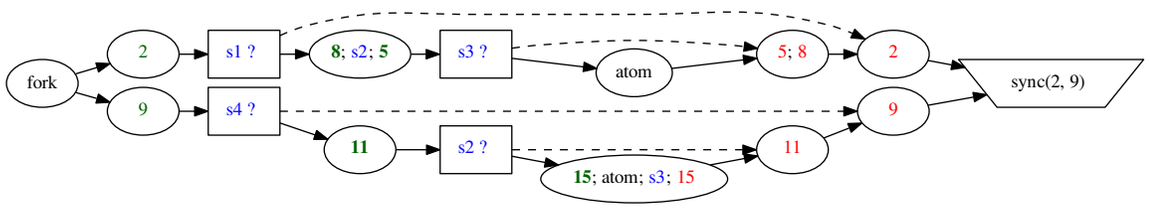


FIGURE 3.4 – Sous-graphe avec **fork** non-trivial de $\mathcal{G}(p7)$

Le nœud source est alors le **fork**. Si on tente de construire manuellement un ordonnancement, observe que l'émission `s2` doit passer avant la lecture de `s2`. De la lecture de ce dernier dépend l'émission de `s3`, qui doit se trouver avant la lecture de `s3`. On peut exprimer l'entrelacement sous la forme du programme synchrone $p7'$ suivant. On observe la nécessité de reconstruire la structure de graphe qui représente le test sur le signal `s1`, et dupliquer l'exécution du test. Une abstraction du problème devient nécessaire. Si on le représente sous forme de programme, le code pourrait être séquentialisé pour devenir le programme ci dessus.

```
let%sync p7' =
  present s1 (emit s2);
  present s4 (present s2 (
    !(print_endline "hello2");
    emit s3));
  present s1
    present s3
      !(print_endline "hello3"))
```

Les différentes étapes pour passer d'un graphe concurrent à un graphe séquetiels :

- Destructuration du graphe (que l'on a appelé Destruction) en listes

- L'entrelacement des listes concurrentes
- La reconstruction du graphe à partir des listes

3.4.1 Destruction du graphe

Le graphe représente un programme. Il décrit deux types de contraintes :

1. Une relation d'accessibilité entre les actions (un ordre partiel)
2. Les conditions d'exécution des actions

La structure des **branch** étant complexe à gérer, avec plusieurs niveaux d'imbrication possibles, on souhaite construire une nouvelle structure dans laquelle ce type de nœud n'apparaît pas mais qui conserverait la même sémantique. Dans un contexte où il n'y a pas de **fork**, et où l'on supprime les **branch**, on ramène toutes les actions au même niveau dans une liste. Néanmoins, cette liste doit conserver les deux règles sémantiques des tests précisées plus haut. On va donc propager dans chaque action de cette liste, l'ensemble des expressions de test donc elle dépend.

Considérons le graphe **branch**(b, f, g), on doit :

- Annoter les nœuds de f qui dépendent de l'expression b avec l'information $True(b)$
- Annoter les nœuds de g qui dépendent de l'expression b avec l'information $False(b)$

Étant dans un **fork**, les graphes f et g ont par construction un sous-graphe en commun, où les deux branches du test se rejoignent. C'est pour cette raison que l'on précise que l'annotation s'applique aux actions *qui dépendent* de b. Pour obtenir la relation entre une action et un ensemble d'expressions de tests, on parcourt les tests récursivement, et on s'arrête aux jointures de ces tests en accumulant les expressions conditionnelles en descendant. Le code de la figure 3.5 est une implémentation de la fonction de transformation. Son type est :

```
type destructed = (action * test_op list) list
val destruct : action list → Fg.t → test_op list → Fg.t → destructed
```

Dans celui-ci, `Fg.t` est le type des arbres de flots de contrôle, `test_op` est une valeur de test d'égalité entre une expression de **branch** et une valeur de vérité, donc `False of test | True of test`. La fonction retourne une valeur de type `destructed`, que l'on définit par une liste d'actions gardées par une liste de conditions

Dans cette fonction, la récursion a deux contextes : `acc`, l'accumulateur de type `action list` que l'on construit en descendant en y ajoutant les actions que l'on rencontre et `tests`, une liste dans laquelle on enregistre le contexte des conditions pour le nœud en cours. On utilise `find_join` pour trouver le nœud où les branches vrai et faux se rejoignent. Cette recherche donne le nœud par lequel passent forcément tous les chemins partant de `fg`. Si des nœuds intermédiaires sont accessibles par plusieurs chemins, ils seront dupliqués dans `acc`. On s'assure que l'on ne perd aucune information et que la liste construite, bien que n'ayant pas la taille optimale, conserve la sémantique du graphe.

On observer que ces deux nœuds dupliqués possèdent au moins une condition incompatible, `True t` et `False t`, et donc ne pourront pas être exécutés au même instant. On filtre ensuite le graphe récursivement en fonction du motif du nœud courant On observe l'application de cette méthode au programme $p7$ de la partie précédente. On rappelle les deux sous-graphes à entrelacer dans la figure 3.6.

- Si le nœud est un **call**, on ajoute l'action à `acc` que l'on passe à l'appel récursif.

```
let rec destruct acc stop tests fg =
  match fg with
  | fg when fg == stop → (* Cas d'arrêt : nœud de synchronisation *)
    acc
  | Call (action, fg') → (* Cas « fg est une action » *)
    destruct ((action, tests) :: acc) stop tests fg'
  | Branch (t, br_true, br_false) → (* Cas « fg est un test » *)
    let stop' = find_join l r in
    let dl = destruct acc stop' (True t :: tests) br_true in
    let dr = destruct dl stop' (False t :: tests) br_false in
    destruct dr stop tests stop'
  | Fork _ | Finish | Pause → assert false (* Impossible par construction *)
```

FIGURE 3.5 – Définition de la fonction `destruct`

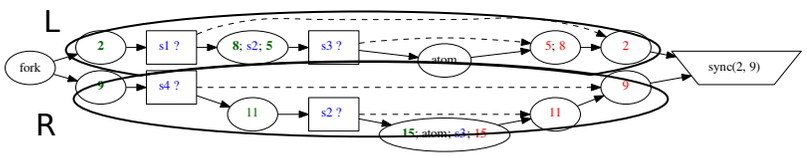


FIGURE 3.6 – Les deux sous graphes L et R

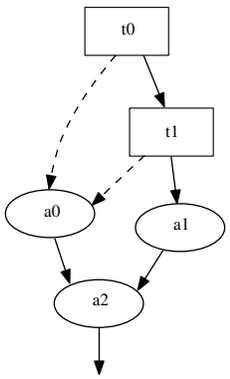
- Si le nœud est un **branch**, on fait appelle à `find_join`, et on considère son résultat `stop'` comme le nouveau nœud de la condition d'arrêt pour les branches vrai et faux sur lesquelles on continue récursivement avec le test précédé de `True` et `False` respectivement. Une fois les deux branches terminées, on relance la récursion sur `stop'` avec le nœud d'arrêt et les tests d'origine et l'accumulation comprenant le parcours des branches `br_true` et `br_false`.
- Sinon, par construction, le nœud d'arrêt est une synchonisation donc on ne peut pas atteindre une feuille (**finish** ou **pause**), et les nœuds **fork** ont été supprimés. Dans cette situation on interrompt le programme.

$$destruct(L) = \left[\begin{array}{lll} \textbf{enter}\ 2 & , & ; \\ \textbf{enter}\ 8 & , & True(s1?) \quad ; \\ \textbf{emit}\ s2 & , & True(s1?) \quad ; \\ \textbf{enter}\ 5 & , & True(s1?) \quad ; \\ \textbf{atom} & , & True(s3?) \wedge True(s1?) \quad ; \\ \textbf{exit}\ 5 & , & True(s1?) \quad ; \\ \textbf{exit}\ 8 & , & True(s1?) \quad ; \\ \textbf{exit}\ 2 & , & ; \end{array}\right]$$

$$destruct(R) = \left[\begin{array}{lll} \textbf{enter}\ 9 & , & ; \\ \textbf{enter}\ 11 & , & True(s4?) \quad ; \\ \textbf{enter}\ 15 & , & True(s2?) \wedge True(s4?) \quad ; \\ \textbf{atom} & , & True(s2?) \wedge True(s4?) \quad ; \\ \textbf{emit}\ s3 & , & True(s2?) \wedge True(s4?) \quad ; \\ \textbf{exit}\ 15 & , & True(s2?) \wedge True(s4?) \quad ; \\ \textbf{exit}\ 11 & , & True(s4?) \quad ; \\ \textbf{exit}\ 9 & , & ; \end{array}\right]$$

FIGURE 3.7 – $destruct(L)$ et $destruct(R)$

Dans la figure 3.7, on observe le résultat de la linéarisation des deux sous-graphes. Les éléments de la liste sont composés d'une action et de la conjonction des expressions booléennes dont elle dépend. Le graphe d'exemple étant assez simple, on observe pas de duplication dans les actions, mais on peut donner la structure minimale qui génère une duplication lors de la transformation en liste.



$$destruct(L) = \begin{bmatrix} a1 & , & True(t0) \wedge True(t1) & ; \\ a0 & , & True(t0) \wedge False(t1) & ; \\ a0 & , & False(t0) & ; \\ a2 & , & & ; \end{bmatrix}$$

FIGURE 3.8 – Duplication dans le résultat de linéarisation

Dans la figure 3.7, l'action $a0$ apparaît deux fois dans la liste construite. Effectivement, il y a deux façons d'atteindre ce nœud dans le graphe, mais il n'est pas considéré comme la jointure du test n'étant pas un nœud commun à tous les chemins. La construction de la liste ne se souvient pas des nœuds déjà rencontrés quand ils ne correspondent pas à une jointure, et ils sont donc dupliqués dans la liste.

3.4.2 Entrelacement

A partir de la structure générée par la linéarisation du graphe donnée dans la partie précédente, on peut entrelacer les deux listes en les lisant simultanément avec des règles de priorité. Ces règles de priorité nécessitent néanmoins une information supplémentaire : pendant le parcours d'une liste, on doit connaître la liste des signaux pouvant être émis dans le futur. On doit donc annoter, pour chaque élément de la liste, l'ensemble des signaux qu'il émet et qui sont émis dans le reste de la liste. En toute logique, plus un élément est placé tôt dans la liste et plus il y aura d'éléments dans l'ensemble des signaux émis. Le code de l'extraction des émissions est donné dans la figure 3.9. Son type est :

```
val extract_em_sets : destructed → SignalSet.t
```

Elle prend la liste créée précédemment et construit un ensemble de signaux.

```
let extract_em_sets =
  let open SignalSet in
  let h = MemoTbl.create 19 in
  let rec extract l =
    try MemoTbl.find h l with | Not_found →
      match l with
      | [] → empty
      | (a, _) :: l' →
        let emts = extract l' in
        let emts = match a with
          | Emit s → add s emts
          | _ → emts
        in MemoTbl.add h l emts; emts
  in extract
```

FIGURE 3.9 – Fonction d'extraction des émissions de signaux de la liste d'action

Au lieu de mettre à jour un environnement passé en paramètre, on utilise une table de hachage globale à tous les appels de la fonction pour mémoïser en fonction de la liste d'actions donnée. On calcule l'ensemble par récursion en remontant dans la liste. On construit ensuite l'entrelacement des deux listes avec cette information. L'implémentation de la fonction d'entrelacement est donnée figure 3.10. Son type est :

```
val interleave_lists : destructed → destructed → destructed
```

Elle prend donc deux listes d'actions gardées, et en renvoie une nouvelle. Chaque cas de récursion agit sur deux éléments de la liste, $d1$ et $d2$ et la récursion est terminale. Les cas où les deux listes sont vides est trivial. Si une des deux listes est vide, on concatène l'autre à l'accumulateur dans l'ordre inverse.

```
let interleave_lists d1 d2 =
  let open SignalSet in
  let rec interleave acc d1 d2 =
    match d1, d2 with
    | [], [] → acc
    | [], d | d, [] → List.rev_append d acc
    | (a1, t1) :: d1', (a2, t2) :: d2' →
      let d1_writes, d1_reads = extract_em_sets d1, to_set t1 in
      let d2_writes, d2_reads = extract_em_sets d2, to_set t2 in
      if inter d1_reads d2_writes = empty then (* 1 *)
        interleave ((a1, List.rev t1) :: acc) d1' d2
      else if inter d2_reads d1_writes = empty then (* 2 *)
        interleave ((a2, List.rev t2) :: acc) d1 d2'
      else error Cyclic_causality (* 3 *)
  in inter [] d1 d2
```

FIGURE 3.10 – Définition de la fonction d'entrelacement

Le dernier cas est celui dans lequel aucune des deux listes n'est vide. La question est de savoir dans quelle liste on sélectionne l'élément. Si aucun des signaux nécessaires pour avancer dans $d1$ n'est émis par $d2$ (**1**) on prend le premier élément de $d1$ et on continue récursivement. Sinon si aucun des signaux nécessaires pour avancer dans $d2$ n'est émis par $d1$ (**2**), on choisit le premier élément de $d2$ et on continue récursivement. Dans le dernier cas (**3**), on a une dépendance circulaire et on ne peut poursuivre ni dans $d1$ ni dans $d2$.

3.4.3 Reconstruction du graphe

On peut trivialement reconstruire un graphe à partir d'une liste en parcourant la liste renversée, en reconstruisant les tests pour chaque action. Il est possible de réunir les actions ayant des conditions communes au moment de reconstruire pour éviter la redondance des tests.

Vers la compilation du graphe en code séquentiel

Dans ce chapitre, on a présenté avec les détails d'implémentation, la technique de compilation d'Esterel connue sous le nom de GRC, des règles de constructions aux passes de transformation du graphe telle que l'ordonnancement des chemins parallèles. L'implémentation de cette partie se trouve dans le module `Grc` du projet pendulum à l'adresse suivante : `https://github.com/remyzorg/pendulum/tree/master/src/preproc`. Dans le chapitre suivant, nous décrivons le processus de transformation qui prend le graphe de flot de contrôle en entrée et qui construit un programme OCaml.

Chapitre 4

Compilation : vers OCaml et le Web

Le graphe étant maintenant dans une forme exécutable séquentiellement, on peut le compiler vers un programme séquentiel. On commence par présenter un mini langage intermédiaire qui permet de capturer une version impérative de la fonction d'instant. On décrira ensuite la représentation de l'environnement d'exécution et de la structure de données qui représente un programme synchrone dans le programme OCaml.

4.1 Compilation du GRC vers un langage intermédiaire : Imp

Le graphe de flot de contrôle ne se compose que de conditions et d'actions en séquence. On propose donc un langage intermédiaire minimal dont la structure est choisie pour faciliter la traduction vers ce dernier et de ce dernier vers OCaml ou n'importe quel langage de programmation composé d'au moins la séquence et une structure de contrôle conditionnelle.

stmt ::= *stmt* **;** *stmt*
| **if** *pred* **then** *stmt* **else** *stmt*
| *action*
| **pause** | **finish**

Figure 4.1 – Grammaire de Imp

On ne donne pas l'algorithme de traduction de la structure du graphe vers Imp qui est très similaire à la transformation en liste chaînée présentée dans la figure 3.5. L'environnement d'exécution d'un programme Imp est strictement égal à l'environnement d'exécution de la version du programme en arbre de flot de contrôle. Étant donnée que les instructions qui lisent et mettent à jour cet environnement sont les mêmes, aucune traduction n'est nécessaire.

4.2 L'environnement d'exécution et la fonction d'instant en OCaml

La sortie de notre compilateur vers OCaml repose sur le module `Parsetree` du compilateur OCaml, qui contient l'ensemble des types nécessaire à l'arbre de syntaxe abstraite du langage à la sortie de la passe d'analyse syntaxique. La valeur construite dans notre compilateur est un `Parsetree.structure_item` [92], c'est-à-dire une définition de la forme `let ... = ... ;;` à la racine d'un module.

Remarque. *Le fait de reposer sur l'arbre de syntaxe abstraite plutôt que sur une autre représentation intermédiaire du langage hôte est une conséquence de notre choix de technologie d'extension de syntaxe PPX, mais il pourrait en être autrement.*

Les structures de contrôle et la séquence sont traduites directement puisqu'elles sont similaires entre le langage source et le langage cible. Les tests et les actions seront eux traduits vers des expressions OCaml qui modifient l'environnement d'exécution par effet de bord. Il est donc nécessaire de construire une représentation de cet environnement d'exécution de Imp. L'environnement d'exécution du graphe est composé de plusieurs structures de données qui doivent apparaître en OCaml : l'environnement des signaux $\mathcal{F}$, l'arbre de sélection $\mathcal{S}$ et l'environnement des labels $\mathcal{E}$. La construction de la représentation de l'environnement d'exécution.

Définition 8 (Fonction de compilation)**.** *La fonction de compilation prend un élément de Imp ou de l'environnement du programme Imp et le transforme vers un arbre de syntaxe abstraite OCaml.*

- *On la note* $a \overset{t}{\Rightarrow} [\alpha]_m$ *où*
  - *a est un élément à compiler dont le type est t, où* $t \in \{env, pred, cont, action, sigs, ev\}$*. On peut traduire un environnement (env), une expression de prédicat (pred), une structure de contrôle (cont), une action (action), un ensemble de signaux (sigs), un évènement (ev), en un arbre de syntaxe abstraite du langage cible.*
  - $[\alpha]_m$ *est le résultat de la compilation où* α *est une valeur qui représente un arbre de syntaxe abstraite OCaml. Le m désigne une expression de macro-transformation.*
- $(\mathcal{S}, \mathcal{F}, \mathcal{E})$ *est un triplet qui contient l'environnement de Imp, avec* $\mathcal{S}$ *l'arbre de sélection,* $\mathcal{F}$ *l'environnement des signaux,* $\mathcal{E}$ *l'environnement des labels d'échappement. Ceux-ci sont passés à la fonction de macro-transformation pour engendrer les structures de données qui les représentent.*

Pour éviter de reposer directement sur les définition de type du module `Parsetree`, les schémas de compilation seront donnés avec la syntaxe concrète entre crochet indicés par *m* pour mettre en avant que c'est une valeur macro-générée et non pas un programme. On pourra échapper les éléments introduit dans cette valeur. Par exemple :

$$\textbf{emit}\ s \overset{action}{\Rightarrow} [\ \texttt{Signal.emit}\ [\![s]\!]_{ident}\ [\![v]\!]_{\overline{m}}\ ]_m$$

Notation (Les échappements de macro-transformation)**.**

- $[\![\_]\!]$ *est un échappement dans l'environnement* $[\_]_m$*. C'est à dire l'introduction d'un morceau d'arbre de syntaxe abstraite construit à partir d'une valeur.*
- $[\![i]\!]_{entier}$, $[\![s]\!]_{ident}$ *est une introduction d'un entier ou d'un identifiant de variable échappé dans l'arbre de syntaxe abstraite construit.*
- $[\![\_]\!]_g$ *est l'introduction d'une valeur d'arbre de syntaxe abstraite échappée calculé à partir de l'application de la fonction de compilation sur un élément du contexte ou une action.*
- $[\![\_]\!]_{\overline{m}}$ *désigne l'insertion d'une expression d'arbre de syntaxe abstraite OCaml qui n'a pas besoin d'être transformée. Dans l'exemple ci-dessus avec* $[\![v]\!]_{\overline{m}}$*, v est stockée dans les structures intermédiaires de pendulum tel quel, sous la forme d'un arbre de syntaxe abstraite du langage hôte. On peut donc la réintroduire sans modification.*

4.2.1 Représentation de l'environnement d'exécution

Représentation de l'environnement des signaux $\mathcal{F}$ en OCaml Un signal est une structure qui contient un état de présence et une valeur. La valeur transportée est construite par une expression OCaml, et a donc

un type dans le système de types. Pour une utilisation la plus cohérente possible avec le modèle de programmation du langage source, on souhaite que le type transporté par un signal puissent être inféré par le système de types d'OCaml, et que l'utilisation du signal soit cohérente avec le type qu'il transporte, notamment dans les instructions atomiques **atom**. Pour atteindre cet objectif, on choisit un type enregistrement paramétré pour les signaux.

```
type 'a signal = {
  mutable value : 'a;
  mutable state : state;
}
```

Les signaux peuvent avoir deux espaces de définition possibles : local ou global, sachant que les signaux lié aux évènements sont considérés comme globaux. Les signaux locaux ne sont accessibles que par les instructions qui sont dans la portée lexicale de la définition. Cependant, la compilation en GRC fait disparaître les informations de portée lexicale, puisqu'à chaque instant le flot de contrôle traverse les instructions actives depuis la racine du programme. Si on veut conserver le pointeur vers un signal local, on doit conserver sa valeur dans un registre global. On doit donc conserver tous les signaux, globaux ou locaux dans le même environnement global, pour chaque instance de programme. Un signal global est initialisé une fois au même moment que l'instance du programme synchrone avec sa valeur initiale. On a donc un registre, avec un pointeur vers le tas qui est immuable et une valeur pointée dans le tas, avec deux champs qui sont modifiables. Un signal local est initialisé lorsque qu'une action **local** est exécutée. Avec cette méthode on ne rencontre pas non plus les problèmes de réincarnation qui font coexister un signal émis lors d'une itération de boucle précédente, mais durant l'instant courant. On a un registre modifiable, sur lequel on écrit à chaque exécution de l'action, qui pointe vers une valeur dans le tas avec deux champs modifiables. On précise, avant de continuer, que pour définir les signaux locaux globalement, le nom de chaque signal doit être unique. Une passe d'alpha-conversion sur le programme synchrone est donc nécessaire pour renommer les signaux en fonction de leur portée lexicale. On peut maintenant définir la représentation de cet environnement global $\mathcal{F}$.

Pour représenter l'environnement $\mathcal{F}$, on a besoin d'une structure qui permet, en conservant les informations de type de chaque signal, d'accéder à chacun d'entre eux pour lire et modifier la valeur transportée depuis le corps du programme. Au moment de construire l'environnement, on a aucune connaissance des types des signaux : la phase de macro-génération de pendulum passe avant la phase de typage. On ne peut donc pas préciser chaque type des éléments de la structure à l'avance. Les types des valeurs doivent donc être inférés par le système de types et la structure de données doit être hétérogène polymorphe en conservant les informations de type. Une telle structure ne peut pas être définie dans le système de types du langage OCaml. La seule méthode possible est donc d'utiliser l'environnement des variables d'OCaml comme registre pour nos signaux. Cette méthode est possible parce que le nombre de registres nécessaires pour conserver l'ensemble des signaux, locaux ou globaux est constant ainsi que leurs types.

De cette façon, on a directement accès aux signaux par leur nom dans le code généré, au même titre que des variables. Les types sont naturellement inférés et l'exécution est plus efficace en temps qu'une recherche dans un dictionnaire pour accéder à la variable. Finalement, on affecte une variable par signal. La fonction de compilation agit sur l'ensemble des signaux $\mathcal{F}$ élément par élément.

- Registre pour un signal global

$$(s, v) \cup \mathcal{F} \overset{env}{\Rightarrow} \left[\texttt{let } [\![s]\!]_{ident} \texttt{ = \{value = } [\![v]\!]_m\texttt{; state = Absent\} in } [\![\mathcal{F}]\!]_g \right]_m$$

- Registre pour un signal local (justification du `magic` dans le paragraphe qui suit)

$$s \cup \mathcal{F} \overset{env}{\Rightarrow} [\ \texttt{let}\ [\![s]\!]_{ident}\texttt{\_rf} = \texttt{ref (Obj.magic 1) in}\ [\![\mathcal{F}]\!]_g\ ]_m$$

- Test de présence d'un signal global

$$\textbf{present}\ s \overset{pred}{\Rightarrow} [\ [\![s]\!]_{ident}\texttt{.state = Present}\ ]_m$$

- Émission d'un signal global

$$\textbf{emit}\ (s,v) \overset{action}{\Rightarrow} [\ \texttt{Signal.emit}\ [\![s]\!]_{ident}\ [\![v]\!]_{\overline{m}}\ ]_m$$

Remarque. *La fonction* `emit` *est définie dans le module* `Signal` *de la bibliothèque d'exécution de pendulum. L'implémentation est la suivante :*

```
let emit s v = s.state <- Present; s.value <- v
```

Il est nécessaire de justifier la définition des signaux locaux ci-dessus. Les signaux locaux n'ont pas de valeur à l'initialisation du programme. Le langage hôte nous force à donner une valeur. On pourrait utiliser un type option, mais une telle représentation risquerait d'alourdir l'utilisation des signaux locaux, alors que la valeur d'un signal local ne pourra jamais être lue non-initialisée par une instruction du programme. On ne peut néanmoins pas donner de preuve de cette information au système de types. On va donc le contourner, en tout sûreté. On choisit pour cela une valeur arbitraire ne posant aucune contrainte de type : `Obj.magic 1` qui a le type `'a`. La fonction `magic` fait partie de la bibliothèque standard non documentée, dont l'utilisation est déconseillée. Elle a le type `'a → 'b`, cela signifie que cette fonction prend une valeur de n'importe quel type et la renvoie avec le type attendu à l'endroit de l'appel. C'est la fonction d'identité non-typée. On peut se permettre de l'utiliser ici parce que le code est généré et le programmeur ne peut en aucun cas manipuler la valeur tant qu'elle n'est pas proprement initialisée.

En l'occurrence, le type inféré pour $[s]_{ident}$`_rf`, qui est une référence est `'_a ref`, où `'_a` est une variable de type non-quantifiée. Quand le système de type détecte une utilisation de `'_a ref` dans la suite de la phase de typage, il décidera de son type définitif, qui est inconnu à la première tentative d'unification. L'utilisation détectée sera forcément le résultat de la compilation d'une action **local**, c'est-à-dire l'initialisation d'un signal local avec sa valeur. On peut tout de suite donner la génération de code de l'action **local** ainsi que du test d'un signal local.

- Initialisation locale d'un signal

$$\textbf{local}\ (s,v) \overset{action}{\Rightarrow} [\ [\![s]\!]_{ident}\texttt{\_rf := \{value =}\ [\![v]\!]_{\overline{m}}\texttt{; state = Absent\}}]_m$$

- Test de présence si $local_{\mathcal{F}}(s)$

$$\textbf{present}\ s \overset{pred}{\Rightarrow} [\ \texttt{(!}[\![s]\!]_{ident}\texttt{).state = Present}]_m$$

- Émission d'un signal $local_{\mathcal{F}}(s)$

$$\textbf{emit}\ (s,v) \overset{pred}{\Rightarrow} [\ \texttt{Signal.emit !}[\![s]\!]_{ident}\ [\![v]\!]_{\overline{m}}\ ]_m$$

En terme d'exécution, il est impossible qu'une lecture ou une écriture du signal soit effectuée avant l'exécution du **local**, sinon, le programme aurait été rejeté plus tôt, lors de l'analyse de portée des signaux. En terme de typage, la référence a une variable de type faiblement polymorphe [36]. En conséquence, un signal ne peut se voir allouer qu'un type par instance du programme, toutes instanciations confondues de ce signal. Le code généré est donc correct vis-à-vis du typage et de l'ordre d'exécution.

Représentation de l'arbre de sélection $\mathcal{S}$ en OCaml On rappelle que $\mathcal{S}$ encore l'état courant du programme sous la forme d'une structure d'arbre. L'arbre est structurellement équivalent à l'arbre de syntaxe abstraite du programme synchrone et chaque nœud représente les instructions non-instantanées par des identificateur uniques. Deux actions du GRC vont modifier cet environnement : **exit** et **enter**. On rappelle que la structure d'arbre est associée à l'utilisation de **exit** : lorsque que l'on désélectionne une instruction, on désélectionne aussi toutes les instructions du sous-arbre. Par contre la sélection n'a de conséquence que sur l'instruction ciblée.

On a donc un ensemble d'entiers de taille bornée que l'on veut pouvoir modifier en place en ajoutant ou retirant des éléments au fur et à mesure. Si on considère que l'on *inline* l'application du **exit** à un sous arbre, on peut faire disparaître la structure arborescente et l'information qu'elle nous donne : les relations d'imbrication entre les instructions. On choisit de représenter la sélection par un tableau de bits indicé par les identifiants uniques des instructions. On utilise le module `Bitset` qui fait parti de la bibliothèque d'exécution de pendulum. La fonction `make` prend comme argument le cardinal de l'ensemble à créer, calculé avec la fonction *Card*.

- Initialisation de l'arbre de sélection

$$\mathcal{S} \overset{env}{\Rightarrow} [\ \texttt{let pendulum\_state = Bitset.make } [\![Card(\mathcal{S})]\!]_{entier} \texttt{ in } [\![\mathcal{E}]\!]_g\ ]_m$$

La suite de l'initialisation de $\mathcal{S}$ est $\mathcal{E}$, l'environnement des labels d'échappement.

On peut maintenant donner la traduction des actions qui modifient $\mathcal{S}$ et qui le lisent. Cette traduction se base aussi sur l'utilisation du module `Bitset`. Pour l'instruction **exit**, on considère que les instructions qui dépendent de i dans $\mathcal{S}$ ont déjà été ajoutées en séquence pendant la traduction de GRC vers Imp.

- Sélection d'une instruction

$$\textbf{enter } i \overset{action}{\Rightarrow} [\ \texttt{Bitset.add pendulum\_state } [\![i]\!]_{entier}\ ]_m$$

- Désélection d'une instruction

$$\textbf{exit } i \overset{action}{\Rightarrow} [\ \texttt{Bitset.remove pendulum\_state } [\![i]\!]_{entier}\ ]_m$$

- Test de sélection

$$\textbf{sel } i \overset{pred}{\Rightarrow} [\ \texttt{Bitset.mem pendulum\_state } [\![i]\!]_{entier}\ ]_m$$

- Test de synchronisation

$$\textbf{sync } (i, j) \overset{pred}{\Rightarrow} [\ [\![\texttt{sel } i]\!]_g \texttt{ || } [\![\texttt{sel } j]\!]_g\ ]_m$$

- Action de synchronisation

$$\textbf{sync } (i, j) \overset{action}{\Rightarrow} [\ \texttt{()}\ ]_m$$

Si plusieurs instructions **exit** ou **enter** se suivent, il est possible de calculer l'union de ces opérations d'ajout dans un ensemble d'entier à la compilation et d'appliquer celui-ci en une seule opération à l'exécution.

Il est important de faire la différence entre le prédicat de synchronisation qui est donné dans la définition originale de GRC et l'action de synchronisation, définie dans la sous-section 3.3.3, qui n'est nécessaire qu'à la structure de l'arbre lors de l'ordonnancement. On notera que l'action de synchronisation est compilée vers l'expression *unit*, c'est-à-dire l'expression neutre en OCaml, car cette instruction ne fait rien.

Représentation de l'environnement des labels $\mathcal{E}$ en OCaml Il existe un nombre fini de labels d'échappement et $\mathcal{E}$ est une association entre un label et son activation. On peut donc utiliser, comme pour l'arbre de sélection, un ensemble d'entiers représentés par un tableau de bits.

- Initialisation de l'environnement

$$\mathcal{E} \overset{env}{\Rightarrow} [\ \texttt{let pendulum\_retcodes = Bitset.make } [\![Card(\mathcal{E})]\!]_{entier} \texttt{ in } [\![\mathcal{F}]\!]_g\ ]_m$$

- Activation d'un code d'échappement

$$\textbf{return } i \overset{action}{\Rightarrow} [\ \texttt{Bitset.add pendulum\_retcodes } [\![i]\!]_{entier}\ ]_m$$

- Test d'un code d'échappement

$$\textbf{code } i \overset{pred}{\Rightarrow} [\ \texttt{Bitset.not\_in pendulum\_retcodes } [\![i]\!]_{entier}\ ]_m$$

Pour le test d'un code d'échappement, le résultat est inversé par rapport à un test de sélection d'une instruction. Dans le graphe, on souhaite emprunter l'arc *vrai* si le code n'est pas activé. C'est pourquoi on utilise la fonction `not_in` plutôt que `mem`.

4.2.2 Macro-génération de la fonction d'instant

La fonction d'instant est construite à partir de la traduction du programme Imp en OCaml, dans une fermeture qui capture les variables représentant l'environnement. On part du programme p_{imp} pour définir la fonction OCaml `react0`. On construit cette fonction à la suite des définitions de variable pour des environnements $\mathcal{S}$ et $\mathcal{F}$ et en particulier lorsqu'il ne reste plus aucun signal dans l'environnement $\mathcal{F}$ courant. On traite la fin de l'instant, comme une exception, d'où le bloc `try with` autour du corps de la fonction de réaction.

- avec $\mathcal{I}$, l'interface de l'instance générée pour le programme synchrone
- et $\mathcal{F}_0$, l'ensemble des signaux d'origine et $\mathcal{F} = \emptyset$

$$\mathcal{F} \overset{env}{\Rightarrow} \left[\begin{array}{l}\texttt{let react0 () =}\\ \quad\texttt{Bitset.remove\_all pendulum\_retcodes;}\\ \quad\texttt{try } [\![p_{imp}]\!]_g \texttt{ with}\\ \quad\texttt{| Pause } \rightarrow \texttt{ Pause | Finish } \rightarrow \texttt{ Finish}\\ \texttt{in } [\![I]\!]_g\end{array}\right]_m$$

On peut ensuite traduire les structures de contrôle de Imp.

- Séquence

$$q\ \textbf{;}\ r \overset{cont}{\Rightarrow} [\ [\![q]\!]_g\ ;\ [\![r]\!]_g\ ]_m$$

- Structure de contrôle conditionnelle

$$\textbf{if } t \textbf{ then } q \textbf{ else } r \overset{cont}{\Rightarrow} [\ \texttt{if } [\![t]\!]_g \texttt{ then } [\![q]\!]_g \texttt{ else } [\![r]\!]_g\ ]_m$$

- Instructions **pause**

$$\textbf{pause} \overset{cont}{\Rightarrow} [\ \texttt{raise Pause}\ ]_m$$

- Instructions **finish**

$$\textbf{finish} \overset{cont}{\Rightarrow} [\ \texttt{raise Finish}\ ]_m$$

Les instructions **pause** et **finish** sont traduites vers le déclenchement d'une exception OCaml, quittant la fonction d'instant.

4.2.3 Interface d'accès à une instance du programme synchrone

Les parties précédentes ont montré la génération de code du cœur d'un programme synchrone : la fonction d'instant avec les structures de contrôle et les actions qui la compose, l'environnement d'exécution du programme. On va maintenant donner les règles de constructions de l'interface du programme synchrone. C'est-à-dire la structure de données qui va comprendre l'appel à la fonction d'instant, mais aussi la mise-à-jour des signaux. La présentation de cette interface de programmation est donnée dans la présentation informelle, partie 2.2.

On rappelle qu'un programme synchrone est une structure d'objet avec deux niveaux : un générateur et une instance. L'instance a une méthode `react` qui ne prend aucun paramètre et qui rend la valeur `() : unit`.

$$I \overset{env}{\Rightarrow} \left[\begin{array}{l} \texttt{object} \\ \quad [\![inputmeths(\mathcal{F})]\!]_{\overline{m}} \\ \quad \texttt{method react = react0 ()} \\ \quad \texttt{method status = Bitset.mem pendulum\_state 0} \\ \texttt{end} \end{array}\right]_m$$

où $inputmeths(\mathcal{F})$ est la génération des méthodes d'accès aux signaux d'entrée, dont l'appel permet d'émettre un signal depuis l'extérieur du programme pour l'instant suivant. Cette génération de code se comporte de la façon suivante :

- $\forall s \in \mathcal{F}_{input}$
  $s \Rightarrow [\ \texttt{method}\ [\![s]\!]_{ident}\ \texttt{v = Signal.emit}\ [\![s]\!]_{ident}\ \texttt{v}\ ]_m$

Remarque. *Lorsqu'on définit une méthode d'objet en OCaml, on peut décider de ne pas lui donner de paramètre, comme* `react` *ici, au contraire des fonctions usuelles qui nécessitent un paramètre (au minimum de type* `unit`*). Par exemple,* `p#react` *est un appel de méthode valide et déclenche l'évaluation de l'expression* `react0 ()`*. Cette précision est importante pour différencier les programmes qui n'ont aucune paramètre des programmes qui prennent un paramètre d'entrée qui est* `() : unit`*.*

4.2.4 Interface d'accès au générateur de programme synchrone

Le code que l'on a généré dans les parties précédentes, qui comprend l'environnement d'exécution, le corps de la fonction d'instant et l'interface d'accès à une instance d'un programme est exécuté à chaque nouvelle initialisation d'instance du programme synchrone. Pour initialiser une instance, on génère le code d'un objet que l'on appelle le générateur. Ce dernier a une méthode `create` qui prend en paramètre les valeurs initiales des signaux, construit l'environnement, et rend comme valeur l'objet interface de l'instance du programme.

$$G \overset{env}{\Rightarrow} \left[\begin{array}{l} \texttt{let}\ [\![p]\!]_{ident}\ \texttt{=} \\ \quad \texttt{object} \\ \quad\quad \texttt{method create}\ [\![args(\mathcal{F}_{inputs})]\!]_{\overline{m}}\ [\![args(\mathcal{F}_{outputs})]\!]_{\overline{m}}\ \texttt{=}\ [\![\Gamma]\!]_g \\ \quad \texttt{end} \end{array}\right]_m$$

On définit le résultat de $args(\mathcal{F})$ de la façon suivante

- Pour les $s_0, \ldots, s_i \in \mathcal{F}_{inputs}$
  $(s_0, \ldots, s_i) \overset{sigs}{\Rightarrow} [\ [\![s_0]\!]_{ident}, \ \ldots, \ [\![s_i]\!]_{ident}\ ]_m$

- Pour les $s_{i+1}, \ldots, s_j \in \mathcal{F}_{ouputs}$
  $(s_{i+1}, \ldots, s_j) \overset{sigs}{\Rightarrow} [\ (\llbracket s_{i+1} \rrbracket_{ident},\ \llbracket s_{i+1} \rrbracket_{ident}\texttt{\_callback}),\ \ldots,\ (\llbracket s_j \rrbracket_{ident},\ \llbracket s_j \rrbracket_{ident}\texttt{\_callback})\ ]_m$

Ces règles signifient que l'on parcourt les signaux de $\mathcal{F}$. On engendre un n-uplet qui contient les signaux d'entrée, et un n-uplet qui contient les signaux de sortie. Les signaux de sortie doivent contenir aussi la fonction de rappel qui a été passée lors de leur initialisation.

4.3 Liaison avec le client Web

Dans la partie précédente, on a exposé le processus de compilation d'un langage impératif simple Imp contenant les actions et les conditions du graphe de flots de contrôle, lui même construit à partir d'un programme synchrone. On a montré comment on passe d'un programme synchrone à son exécution dans un programme hôte généraliste, comme introduit dans la partie 2.2. On va, a posteriori, compléter ces phases de compilation, en montrant comment brancher un programme au navigateur, c'est-à-dire aux évènements et aux éléments du DOM, en compilant les constructions présentées partie 2.3. On rappelle que les constructions de pendulum propres à l'utilisation du DOM dans la figure 4.2

test : := ...
  | *ident*##*event*

header : := ...
  | **element** *ident gather*?;

gather : := **{** (*event* **=** *ocaml-expr*, *ocaml-expr* **;**)*}**;**

stmt : := ...
  | **emit** *ident*##**.***property ocaml-expr*

FIGURE 4.2 – Constructions de pendulum propres au client Web

La traduction de ces constructions a deux niveaux : la première pendant phase de traitement du code synchrone et la seconde pendant la génération du code OCaml. On commence par traduire les éléments, puis la fonction d'émission sur des propriétés d'éléments, et on termine par la fonction de combinaison.

4.3.1 Les éléments et le test de présence d'évènements

On a décrit dans la section 2.3 le comportement des signaux calqués sur un comportement évènementiel. On a vu que les éléments étaient définis globalement en tête du programme et que pour chaque association d'un élément à un évènement, que l'on appelle un comportement évènementiel, on créée un signal qui reflète ce comportement. Il n'est pas nécessaire de décrire en en-tête à quels évènements sont associés les éléments en entrée. C'est par une analyse syntaxique que l'on va inférer l'ensemble des comportements évènementiels (ν, ϵ) et donc l'ensemble des signaux $\nu\#\#\epsilon$.

Si on lit au moins une occurrence d'un texte avec la syntaxe *ident*##*ident*, on ajoute à l'environnement $\mathcal{F}_{input}$ global le signal dont l'identifiant est la concaténation du nom de l'élément et de l'évènement séparé par la chaine "##". On donne $\wedge$ pour l'opération de concaténation de chaînes.

- $\nu\#\#\epsilon \quad \rightsquigarrow \quad \llbracket \nu \wedge \text{"\#\#"} \wedge \epsilon \rrbracket_{ident}$

Après cette transformation du programme pendulum, on a réduit ce dernier aux constructions synchrones atomiques. Le test de présence devient un test de présence usuel sur un identifiant de signal, qui est contenu dans l'environnement. On stocke dans l'environnement $\mathcal{F}_{Event}$ l'ensemble des couples (ν, ϵ), que l'on ne considère plus jusqu'à la génération de code.

Remarque. *Un nom de variable comportant le caractère* `'#'` *est pas accepté par l'analyseur syntaxique d'OCaml. Il est néanmoins possible de générer, via une macro PPX un tel nom de variable. Il n'est donc pas possible de faire référence au signal généré par son nom dans les émissions de signaux.*

Lors de l'étape de génération de code, on construit les fonctions de rappels qui vont déclencher la mise-à-jour du signal ainsi que la réaction du programme. Pour chaque couple (ν, ϵ), on affecte un gestionnaire d'évènement à la propriété $[\![\epsilon]\!]_{ident}$ de l'élément $[\![\nu]\!]_{ident}$. La propriété a le même nom que l'évènement.

$$(\nu, \epsilon) \overset{ev}{\Rightarrow} \left[\begin{array}{l} [\![\nu]\!]_{ident}\texttt{\#\#.}[\![\epsilon]\!]_{ident}\ \texttt{:=} \\ \quad \texttt{Dom\_html.handler (fun ev} \rightarrow \\ \quad\quad \texttt{Signal.emit } [\![\nu \wedge \text{"\#\#"} \wedge \epsilon]\!]_{ident}\ \texttt{ev;} \\ \quad\quad \texttt{react0 ());} \end{array}\right]_m$$

On construit un gestionnaire d'évènement en appelant la fonction `Dom_html.handler`, dont le type est

```
val handler : ((#Dom_html.event t as 'b) → bool t) → ('a, 'b) event_listener
```

Ce type nous dit que `handler` prend une fonction puis rend un gestionnaire d'évènement. Cette fonction prend un paramètre évènement dont le type est hérité de `#Dom_html.event Js.t` et rend un booléen. Le paramètre évènement correspond à un objet du DOM transportant toutes les informations de l'évènement déclenché en fonction du comportement en question : coordonnées de la souris, élément cible, etc. Cette valeur d'évènement est émise dans le signal de l'environnement qui correspond au couple (ν, ϵ) auquel on accède par son nom. On exécute ensuite la fonction de réaction.

4.3.2 Agrégation de la valeur des signaux

La génération de code pour les signaux à comportement évènementiel présentée ici est une version naïve, puisqu'à chaque exécution de la fonction de rappel d'un comportement évènementiel, la fonction de réaction est déclenchée. Cela n'est pas toujours souhaitable, et on verra dans la section 5.1 comment permettre au programmeur de préciser le modèle d'exécution. Pour préparer ces modifications, on présente une nouvelle façon de représenter les signaux.

```
type 'a signal = {
  mutable value : 'a;
  mutable state : state;
  set : a' → 'b → 'a;
}
```

On ajoute à l'ancienne représentation une fonction qui prend une valeur du même type que celle contenue dans le signal : `'a`, et une valeur à agréger, `'b`, et rend un `'a`. Dans cette configuration, les signaux ont un type de valeur d'entrée, et un type de valeur contenue. Si l'utilisateur ne précise pas de fonction d'agrégation, on donne la valeur `(fun _ x → x)` au champ `set`, c'est-à-dire une fonction qui remplace l'ancienne valeur du signal par la nouvelle. On peut alors redéfinir la fonction `Signal.emit` avec l'implémentation suivante :

```
    let emit s v =
      s.state <- Present;
      s.value <- s.set s.value v
```

Conclusion

Dans ce chapitre, nous terminons le processus de compilation débuté dans le le chapitre 3 en transformant le graphe de flot de contrôle en un programme OCaml que l'on peut instrumenter dans une application Web. Le code de cette partie est accessible dans les modules `Grc2ml` et `Ml2ocaml` du projet pendulum à l'adresse suivante : `https://github.com/remyzorg/pendulum/tree/master/src/preproc`. On va utiliser le code engendré dans le chapitre suivante pour montrer la relation entre le modèle d'exécution d'un programme synchrone et celui du navigateur Web.

Chapitre 5

Exécution d'un programme synchrone dans le navigateur

Dans le chapitre 2, on a donné une présentation générale du langage et les constructions étendant le modèle synchrone, grâce auxquelles le programme peut manipuler les interactions du client Web. On a ensuite, dans le chapitre 3, détaillé comment ce type de programme se traduit vers graphe de flot de contrôle, et ensuite vers OCaml dans le chapitre 4. En particulier, on a montré comment faciliter la programmation et l'intégration du synchrone au client Web, en générant automatiquement le lien entre les signaux et les évènements.

Bien que simplifiant l'écriture et la compréhension du programme synchrones dans le client, on souhaiterait préciser le comportement du code faisant la liaison. Le code généré, en particulier, fait dépendre l'exécution de la fonction d'instant du programme synchrone à l'exécution des fonctions de rappel des évènements. Cela signifie, du point de vue du programme synchrone, que son horloge est définie par la séquence des entrées qu'il reçoit, mais ce choix n'est pas toujours souhaitable. On donne donc dans une première section des outils au programmeur pour choisir l'horloge du programme plus précisément. On verra ensuite comment ces décisions peuvent impacter les performances du programme lorsqu'il implique de nombreux évènements dans une seconde section.

5.1 Le navigateur Web comme horloge

On expose premièrement le comportement d'une horloge dirigée implicitement par les évènements, et en quoi ce type de sémantique pourrait s'avérer être inadaptée dans certaines situations. On propose ensuite une approche pour détacher partiellement l'horloge de la réaction au évènements, en l'adaptant au comportement du navigateur.

5.1.1 Une horloge dépendante des évènements

Dans les chapitres précédents, pour simplifier l'approche de la compilation de pendulum et ses constructions spécifiques vers le client Web, on a volontairement mis de côté la définition de l'horloge. On a considéré que ces deux problèmes se traitent séparément. L'horloge d'un programme pendulum du client Web est donc définie, dans les chapitre précédents, par la séquence des occurrences d'évènements que le programme reçoit en entrée. On pourrait aussi l'exprimer différemment : on déclenche une exécution du programme synchrone à chaque fois qu'un évènement dont il dépend apparaît. Partons sur un exemple de

taille réduite,

```
loop (
  present w##onclick
    (emit sp##.textContent !!(w##onclick));
  pause)
```

Dans cet exemple, la réaction du programme synchrone aura lieu à chaque clic de souris. On peut considérer que ce comportement est acceptable pour ce programme, parce qu'il ne dépend que d'un évènement ponctuel. Si un programme implique plusieurs comportements évènementiels dont les occurrences respectives sont temporellement proches, l'idéal est de n'exécuter qu'une seule réaction et pas plusieurs. Exécuter plusieurs réactions du programme est inutilement coûteux alors que toutes les entrées pourraient être considérées pendant le même instant. C'est d'autant plus vrai si le nombre d'évènements est grand et que ces occurrences d'évènements concernent un même comportement évènementiel.

La fréquence d'exécution du programme synchrone ne doit pas être linéaire en le nombre d'entrées qu'il reçoit. Cette fréquence d'exécution est supposée représenter plus ou moins fidèlement l'évolution du temps réel dans un modèle logique. De plus, l'interprète du navigateur s'exécute complètement séquentiellement. L'exécution d'une fonction de rappel, et donc de la fonction d'instant est atomique et bloquante. Une surcharge en temps d'exécution pourrait avoir des conséquences désastreuses sur l'interactivité de l'interface et la capacité du programme à répondre.

Il existe aussi de nombreux cas ou la sémantique du programme pourrait sembler obscure au programmeur. Pour illustrer ce problème, on définit le programme synchrone suivant qui émet un signal tant qu'une touche du clavier précise est enfoncée. Les signaux d'entrée sont les évènements d'appui et de relâchement de la touche.

```
let%sync plex  =
  element field {
    onkeydown = false, fun acc ev → acc || ev##.keyCode = 32;
    onkeyup = false, fun acc ev → acc || ev##.keyCode = 32;
  };
  output out;
  loop (
    present field##onkeydown & !!(field##onkeydown) (
      trap up (
        loop (
          present field##onkeyup & !!(field##onkeyup) (exit up)
        ; emit out
        ; pause))
    ) ; pause
  )
```

Ce programme est décrit par une boucle qui à chaque réaction attend la présence du signal d'appui `field##onkeydown` avec la bonne touche appuyée, quand la valeur contenue dans le signal vaut *vrai*. Dans un nouvel état, on émet à chaque instant le signal `out` et on attend la présence du signal de relâchement `field##onkeyup` avec sa valeur à vrai pour quitter cet état grâce à une construction d'échappement. L'horloge du programme est donc définie par deux évènements. Si rien d'autre n'est fait et qu'on instancie ce programme en le liant à des évènements du DOM automatiquement, le programme ne sera pas exécuté pendant l'intervalle entre les deux interactions parce que rien ne le déclenchera. Il sera exécuté à la première interaction, donc l'appui, qui émettra `out` et à la seconde, le relâchement. Si on lie le signal de sortie

`out` à une fonction d'animation d'un composant, il ne se passera rien parce que l'état continu d'animation ne sera pas exécuté véritablement. Il faut donc pouvoir définir un comportement de déclenchement automatique du programme synchrone à un rythme régulier.

On peut le faire manuellement en utilisant les fonctions d'évènements temporels du DOM, comme `window##setTimeout`, qui permet de repousser l'exécution d'une fonction de rappel. Plutôt que de choisir cette solution, on décide de palier à tout ces problèmes d'interopérabilité et de réactivité entre les évènements et le modèle synchrone par une seule proposition.

5.1.2 Définition d'une horloge indépendante des entrées

La solution serait de faire reposer l'horloge du programme sur un déclenchement qui dépend de l'interprète du navigateur dans lequel le programme s'exécute au lieu de se reposer sur les entrées, comme on pourrait utiliser la fréquence du processeur dans le cas d'un système embarqué. On a vu en introduction, partie 1.1.3, qu'il était possible de borner le rythme d'exécution d'un script à celui des rafraîchissements du navigateur, grâce à la fonction du DOM, `requestAnimationFrame`. Pour rappel, cette fonction repousse l'exécution d'une fermeture passée en paramètre à la prochaine étape de rafraîchissement du navigateur. Dans la bibliothèque de Js_of_ocaml, cette fonction est accessible depuis le module `Dom_html` avec l'interface suivante :

```
class type window = object
  (* ... *)
  method requestAnimationFrame :
      (float → unit) Js.callback →
      Dom_html.animation_frame_request_id Js.meth
  (* ... *)
end
```

D'après son type, on observe qu'elle prend un paramètre une fonction de rappel, et retourne un entier qui identifie de façon unique cette fonction de rappel. La fonction de rappel doit avoir le type `float → unit` où le type du paramètre est un flottant qui indique un horodatage de l'appel de la fonction de rappel. On peut donc en déduire le temps écoulé entre l'ajout d'une fonction à `requestAnimationFrame` et son déclenchement réel. Ce temps peut alors servir à animer les éléments de la page à une certaine fréquence d'affichage. Dans l'exemple suivant, on définit récursivement une fonction `animation` qui permet d'exécuter une animation 24 fois par seconde si le navigateur le permet.

```
let fps : float = 24. in
let tick : float ref = ref 0. in
let interval : float = 1000. /. fps in

(* animation : float → unit *)
let rec animation timestamp =
  window##requestAnimationFrame (Js.wrap_callback animation);
  let delta = timestamp -. !tick in (* calcul du temps écoulé *)
  if delta > interval then begin
    (* mise-a-jour du dernier tick *)
    tick := timestamp -. (mod_float delta interval);
```

```
    (* animation d'un élément de la page *)
  end
```

A chaque itération, la fonction commence par passer sa propre valeur à `requestAnimationFrame`, à la manière d'un appel récursif retardé. On calcule ensuite le temps écoulé depuis la dernière exécution de l'animation. Si ce temps est supérieur à l'intervalle choisit, on peut ré-exécuter l'animation. L'intervalle de temps dépend du nombre d'image par seconde que l'on spécifie pour notre animation. La fonction `requestAnimationFrame` est donc à la fois spécifiée pour retarder une fonction au rythme voulu par le client Web, mais donne aussi les outils pour construire des boucles d'animation avec une fréquence de rafraîchissement bornée.

Les possibilités proposées par cette fonction correspondent à la problématique déclarée dans la partie précédente. À partir de là, on peut proposer une nouvelle fonction de réaction lors de la génération code OCaml pour le programme synchrone. que l'on génère au même niveau que `react0`, et avec cette dernière. On nomme notre nouvelle fonction `animate`. On modifie pour cela la génération du code de `react0`, donnée dans le chapitre compilation, partie 4.2.3.

$$\mathcal{F} \overset{\Gamma}{\Rightarrow} \left[\begin{array}{l}
\texttt{let animated = ref false in} \\
\texttt{let rec react0 () =} \\
\quad \texttt{let react0 () = animate () in} \\
\quad [\![p_{imp}]\!]_g \\
\texttt{in} \\
\texttt{and animate () =} \\
\quad \texttt{if not (!animated) then begin} \\
\quad\quad \texttt{animated := true;} \\
\quad\quad \texttt{Dom_html.window\#\#requestAnimationFrame} \\
\quad\quad\quad \texttt{(Js.wrap_callback} \\
\quad\quad\quad\quad \texttt{(fun _ } \rightarrow \texttt{ animated := false; react0 ()))} \\
\quad \texttt{end} \\
\texttt{in } [\![I]\!]_g
\end{array}\right]_m$$

La fonction `animate` enveloppe la fonction `react0` et consiste à retarder l'exécution de cette dernière au prochain rafraîchissement du navigateur grâce à `requestAnimationFrame`. On place une garde permettant de vérifier si la fonction de réaction est déjà inscrite pour la prochaine mise-à-jour de la page. Si ce n'est pas le cas, on l'y inscrit et on passe le booléen `animated` à *vrai*. Si la garde est vraie, on ne fait rien, ça signifie que la fonction de réaction est déjà en attente d'exécution, dans un temps arbitrairement proche.

La définition proposée ici pour `animate` nous apporte une fonctionnalité inattendue, mais nécessaire : la définition récursive de la fonction de réaction. Jusque là, il n'était pas possible d'obtenir cette fonctionnalité, car la fonction de réaction ne doit pas pouvoir être appelée depuis son propre corps : pour conserver tous les invariants du programme synchrone, son exécution doit être atomique et ne pas être interrompue par une autre exécution qui modifie le même environnement. Néanmoins, `animate` nous propose une version retardée de la fonction de réaction. On peut donc décider, depuis un appel de cette fonction, de relancer une réaction dans un futur proche. Ce comportement est sûr du point de vue de la sémantique du programme synchrone, puisqu'il conserve l'atomicité de `react0`.

Remarque. *Pour empêcher le programmeur d'exécuter un appel récursif instantané de* `react0` *dans le programme synchrone, on cache l'accès à cette fonction dans son propre corps en définissant une nouvelle fonction du même nom qui appelle* `animate` *à la place :* `let react0 = animate in ...`

Cette fonctionnalité est aussi indispensable pour certaines implémentation que nous verrons dans les ex-

périmentations, section 6.2. Néanmoins, le cœur synchrone de pendulum est supposé pouvoir fonctionner sans dépendances avec le client Web, même si c'est le contexte d'exécution désiré. On considère donc que cette couche provoquant les appels retardés à la fonction de réaction est spécifiée optionnellement par le programmeur, pour chaque définition de programme synchrone. On propose la syntaxe suivante :

```
let%sync p ~animate = (* ... *)
```

Remarque. *Cette syntaxe se base sur celle des arguments labellisés en OCaml. Pour les programmes pendulum, il s'agit des options de compilation, et sont définies dans la documentation dans un nombre limité.*

Le fait de spécifier cette option déclenche dans le compilateur la génération du code modifié donnée plus haut. Les appels aux instances de `p` seront donc tous retardés au prochain rafraîchissement du client Web.

L'utilisation de cette option n'implique pas une horloge basée sur le temps réel, mais simplement une contrainte supplémentaire sur la notion d'horloge présenté dans la partie précédente. On peut mutualiser de multiples appels à la fonction de réaction en un seul appel, ainsi que respecter les ressources d'exécution du navigateur en se calant sur son rythme.

5.2 Tests de performances comparés

Dans la section précédente, on a précisé l'interopérabilité entre les évènements du client Web et les programmes synchrones, pour une exécution plus homogène, moins dépendante des évènements. On souhaite maintenant montrer en quoi ces décisions ont un impact sur l'exécution et notamment les performances d'un programme pendulum en temps d'exécution.

Tout d'abord, la notion d'agrégation des changements présentée dans la section précédente est une notion qui est déjà présente dans les communautés de programmation du client Web. Notamment parce que le DOM est une structure de données volatile, où toutes les modifications se font en place, d'où la nécessité de proposer les modifications du DOM comme une reconstruction atomique, à la façon d'une structure de données purement fonctionnelle. Mais aussi parce que modifier le document en place directement à chaque modification force le rafraîchissement du navigateur et peut avoir des conséquences négatives sur le temps d'exécution.

L'approche qui fait autorité dans ce domaine s'appelle le DOM Virtuel. Elle a été popularisée par React [82], la bibliothèque virtual-dom [69], Mercury [90] ou encore Elm [67] qui insiste sur l'efficacité d'une telle méthode [66]. Elle propose de faire passer toutes les modifications du document par un arbre intermédiaire, que l'on dit virtuel et qui est une abstraction de l'original. Ce dernier est reconstruit à chaque rafraîchissement, ce qui semble intuitivement coûteux en temps d'exécution. Ce serait le cas sans le principe de base : la différenciation, ou *diffing*. Au lieu de reconstruire entièrement le document virtuel, l'algorithme de différenciation permet de ne rechercher que les changements entre le document virtuel du rafraîchissement précédent et l'actuel. On peut alors considérer ces différences comme des fonctions de changement d'un état du document vers un autre, et appliquer ces fonctions au document lui même. Si on appelle l'étape de différenciation dans la fonction `requestAnimationFrame`, on peut alors mutualiser en une seule étape la modification du document, et ainsi éviter des rafraîchissements forcés. Cette approche, en plus de posséder ces qualités d'efficacité, et une interface plus modulaire et fonctionnelle au programmeur, peut s'utiliser aussi sans que ce dernier aie à intervenir à bas niveau dans le modèle de concurrence et de modification de la vue client.

Dans pendulum, on choisit finalement une approche qui est très proche du document virtuel, puisque l'ensemble des modifications est appliqué une fois pour toute après la récupération d'un ensemble des entrées du programme au travers des évènements. On peut alors mettre sur le banc d'essai les technologies Web client équivalentes à pendulum et les comparer en termes de performance. Dans une présentation d'Elm [66], la performance est calculée sur la base du traitement d'un grand nombre d'évènements dans un laps de temps très court, et sur le temps de réaction moyen pour appliquer toutes ces modifications. C'est ce canevas que l'on expose dans un premier temps, puis nous présenterons les résultats produits par l'exécution de l'application implémentée en pendulum.

5.2.1 Application TodoMVC

Le canevas de test existe déjà et il s'appelle TodoMVC Benchmark [86]. La documentation spécifie une liste de fonctionnalités pour une application du client Web de liste de tâches ou *todo-list*. Le site Web héberge aussi les implémentations dans de nombreux langages, ou bibliothèque de cette application. Il permet donc de comparer la taille du code, l'expressivité, l'interopérabilité d'un langage avec HTML et autres différences possibles entre les plateformes de développement logiciel client. Les caractéristiques graphiques de l'application sont décrite en CSS. Il est le même pour toutes les implémentations, de même que la structure HTML demandée. L'application se limite au client et stocke les informations dans le navigateur. Aucune requête n'est donc nécessaire : tout se joue dans les interactions avec l'utilisateur.

FIGURE 5.1 – Capture d'écran de l'application TodoMVC

La figure 5.1 présente une capture d'écran d'une exécution de l'application. On a un champ de texte éditable (1), dans lequel on peut entrer au clavier le contenu textuel de l'objet à ajouter dans la liste, et le valider par la touche `Enter`. Les objets, une fois dans la liste peuvent être dans l'état *terminé* (2) si l'utilisateur clique sur l'une des cases à gauche (3) ou *en cours* (4) sinon. On peut repasser d'un état à un autre à volonté en cliquant à nouveau sur les cases. La liste des objets peut ainsi être filtrée en fonction de l'état de ses composants grâce aux boutons (5). On peut supprimer les objets un par un en passant la souris sur la partie droite et les éditer en double-cliquant sur le texte. Il est aussi possible de supprimer

tous les éléments d'un coup (6), ou de tous les changer d'état (7).

Implémentation en pendulum

On implémente les fonctionnalités précédentes en utilisant la programmation synchrone. Le programme est donc partagé entre les fonctions Js_of_ocaml qui servent à créer les éléments du DOM, en particulier les objets de la liste qui vont évoluer dynamiquement, et la partie en pendulum qui doit traiter les évènements qui proviennent à la fois des éléments statiques, mais aussi des évènements dynamiques. On utilise un environnement pour représenter ce que l'on appelle le modèle, c'est-à-dire la représentation des objets de la liste en mémoire. On s'attarde en particulier sur deux fonctionnalités qui présentent différentes méthodes de programmation en pendulum.

1. Filtrage des objets
2. Ajout d'un objet

La 1 est simple et utilise les constructions du modèle Web-synchrone proposé par le langage. L'ajout d'un objet à la liste nécessite des fonctionnalités d'animation décrites dans la section précédente, et en particulier la définition récursive de la fonction d'instant. L'ensemble du programme synchrone se présente sous la forme d'une boucle qui contient, en parallèle les tests de présence des signaux provenant des différents comportement évènementiels.

```
let%sync todolist_machine ~print:pdf =
  element all, compl, active;

  element newitem {
    onkeydown = [], fun acc ev →
        if ev##.keyCode = 13 && newitem##.value##.length > 0
        then newitem##.value :: acc else acc
  };

  input selected_items (fun l id → id :: l);
  input deleted_items (fun l id → id :: l);

  let visibility = None in
  let tasks = Hashtbl.create 19 in

  (* etc *)
  loop (
    (* Signal provenant du champ de texte *)
    ||
    (* Signaux provenant des boutons de filtrage *)
    ||
    (* etc *)
    ; pause
  )
```

On donne la structure du programme et les instructions qui correspondent aux fonctionnalités que l'on souhaite détailler. Les éléments en entrée du programme sont les boutons en (5) et le champ de texte `newitem` que l'on détaillera plus tard. Le signal local `visibility` encode les changements d'état du filtrage.

Le signal local `tasks` est utilisé comme une variable locale et contient la représentation de l'ensemble des objets de la liste en mémoire. Les signaux `selected_items` et `deleted_items` sont des signaux de listes qui agrègent les éléments cliqués et les éléments supprimés. La boucle vérifie donc à chaque instant si l'un des signaux est présent.

Le filtrage des objets se fait en réaction au clic de l'utilisateur sur les boutons en (5) de la figure 5.1. On met simplement en parallèle les tests de présence sur les comportements évènementiels liés au clique. On mutualise le tout dans le signal local `visibility`, qui une fois présent déclenche l'appel à la fonction Js_of_ocaml nommée `View.change_visibility`.

```
    (* ... *)

    || present all##onclick (emit visibility None)
    || present compl##onclick (emit visibility (Some true))
    || present active##onclick (emit visibility (Some false))

    || present visibility !(View.change_visiblity !!tasks
                              (all, compl, active) !!visibility)
    (* ... *)
```

Dans le code ci-dessus, on observe qu'on lui passe les trois boutons pour changer leur aspect visuel, ainsi que l'ensemble des tâches. La fonction parcourt simplement les éléments et modifie leur CSS pour qu'elles soient visibles ou non selon la valeur du signal `visibility`.

Ajout d'un objet On rappelle que l'élément correspondant au champ de texte pour ajouter un nouvel objet est donné en entrée du programme sous le nom `newitem`. Il est défini de pair avec une fonction d'agrégation sur l'évènement `onkeydown`. Cette fonction agrège dans une liste la valeur contenue dans `newitem` si le caractère capturé par l'évènement a le code 13, c'est-à-dire la touche *retour chariot* et que cette valeur n'est pas la chaîne vide. Le code pour la définition est le suivant,

```
element newitem {
  onkeydown = [], fun acc ev →
      if ev##.keyCode = 13 && newitem##.value##.length > 0
      then newitem##.value :: acc else acc
};
```

On en déduit le type du signal :

```
- newitem##onkeydown : (string list) signal
```

Celui-ci une liste de chaîne de caractères des mots à ajouter dans la liste de tâches. On peut ensuite gérer ce signal avec un test de présence où l'on vérifie aussi que la liste des mots ajoutés n'est pas vide. Si ces conditions sont réunies, on peut appeler la fonction `View.create_items` qui se charge de modifier tasks et créer les composants graphiques dans la vue de la liste.

```
    || present (newit##onkeydown & !!(newit##onkeydown) <> []) (
        (View.create_items !!tasks animate !!items_ul
           deleted_items selected_items !!(newit##onkeydown));
      emit newit##.value (Js.string ""))
```

Les éléments graphiques du DOM créés dynamiquement par cette fonction doivent aussi réagir aux actions de l'utilisateur. Si on clic sur une case dans la liste entre deux instants, le programme synchrone doit recevoir une information en entrée précisant que d'un objet de la liste a été sélectionné. On utilise pour cela un signal `input` que l'on passe à la fonction de construction des éléments graphiques `create_items`. Cette fonction, en initialisant l'élément graphique dans le DOM, lui affecte une fonction de rappel liée au clic de la case. On peut donner la forme suivante à la fonction de rappel.

```
let select_callback evt =
  Signal.emit select_s id; animate ()
```

C'est cette fonction de rappel qui émet le signal entre les deux instants, et déclenche une réaction retardée du programme synchrone. Pour cette raison, on doit passer en paramètre de `create_items` les deux signaux `selected_items`, `deleted_items` et la fonction de réaction `animate`. On remarquera que la valeur de ces signaux est une agrégation des émissions. Si plusieurs objets de la liste sont sélectionnés ou supprimés entre deux instants, les deux signaux contiendront une liste des éléments graphiques respectivement à supprimer ou sélectionner.

5.2.2 Comparaison des performances en temps d'exécution des implémentations

Maintenant que l'on a donné des détails sur les fonctionnalités de cette application et son implémentation en pendulum, nous allons étudier les résultats qu'elle produit dans son intégration à des tests de performance comparatifs. Le but recherché dans cette intégration est de déterminer si un programme pendulum comporte ou non un surcoût en temps d'exécution par rapport à d'autres techniques de programmation. Il n'y a pas beaucoup de tests existants qui mesurent le temps entre l'apparition des évènements et la fin de l'affichage, mais nous avons choisis une méthode préexistante pour cette situation particulière où l'utilisation de pendulum prend son importance. Cette méthode est produite par les auteurs de Mercury [73], une plateforme de développement pour le client Web à destination des développeurs et utilisateurs de ce type d'outils, sur la base du modèle TodoMVC. Le test prend la place de l'utilisateur d'une liste de tâche, et génère un grand nombre d'évènements pour mettre à jour la liste. Dans notre cas de test, on provoque dans cet ordre :

- 100 ajouts d'objet
- Sélection de chaque objet (un évènements par objet)
- Suppression de chaque objet (un évènement par objet)

Pour chaque implémentation, on lance l'application et on relance cet enchaînement plusieurs fois pour obtenir un temps moyen d'exécution pour terminer un enchaînement d'ajouts, sélections et suppressions de cents objets. La spécificité de ce test est qu'il génère tous les évènements de façon atomique. Son comportement diffère donc de celui d'un utilisateur sur ce programme en particulier. Il est effectivement impossible qu'un utilisateur produise un nombre d'évènements aussi grand en une si courte période de temps avec des périphériques standards tels qu'une souris et un clavier. Néanmoins, il existe des situations où une application Web reçoit un grand nombre de sollicitations externes dans un laps de temps très court,

lorsqu'il y a une accumulation de d'évènements de différentes sources ou des résultats de requêtes de différents serveurs par exemple. On teste donc là une situation de stress en performances qui existe en réalité.

On présente dans la figure 5.2 le temps d'exécution moyen de différentes implémentations pour ce test, incluant celles en Js_of_ocaml avec React (OCaml) et pendulum.

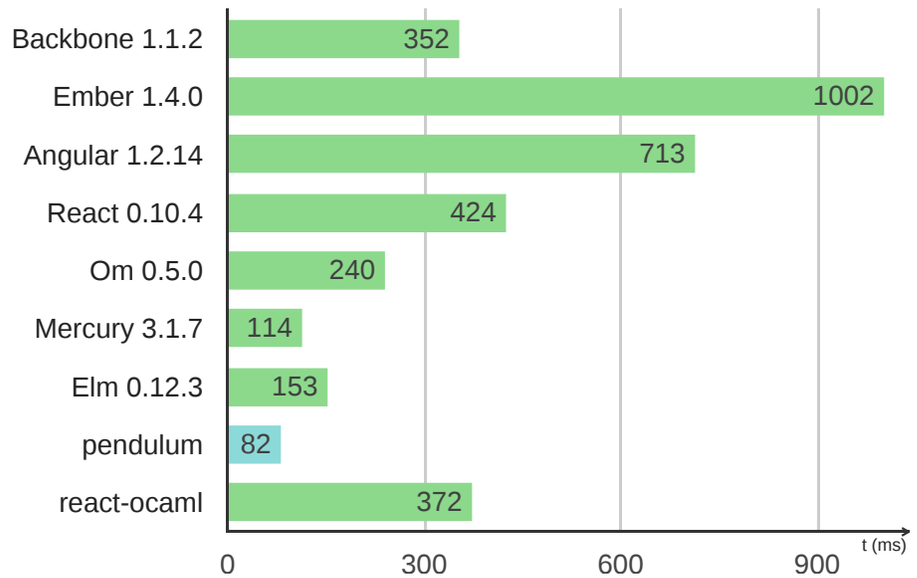


FIGURE 5.2 – Comparaison des temps d'exécution moyen en ms, sur 30 exécutions

Ce test a été effectué avec le navigateur Google Chrome version 51 s'exécutant sur un processeur Intel Core i7-4690. On note que les implémentations en Mercury, Elm et pendulum bénéficie de temps d'exécution inférieurs aux autres implémentations. Cette différence est due à l'agrégation des modifications sur le DOM en une seule passe pour les cent évènements à chaque étape. Pour le code pendulum tout se fait dans la réaction à la présence aux signaux `newitem##onkeydown`, `selected_items` et `deleted_items` qui contiennent dans leurs valeurs respectives les listes des cent modifications pour chaque situation, parcourues itérativement.

On peut considérer que ce test est biaisé au sens où il présente exactement la problématique qu'il est aisé de résoudre dans une programmation sur un temps logique. Néanmoins, toute proportion gardée, on peut en déduire que les méthodes de compilation utilisées dans l'implémentation, et le code généré ne produisent pas de surcoût en temps d'exécution. A l'inverse, ils permettent une forme d'optimisation qui n'oblige pas le programmeur à s'impliquer dans des modifications de bas niveau et qui agissent au contraire sur du code écrit « naturellement ». On peut aussi déduire de ce test une bonne réaction du modèle synchrone à un grand nombre d'évènements du même type et que l'utilisation de pendulum produit un programme efficace s'il est confronté à ce genre de problématiques.

Conclusion

La corrélation constatée entre l'exécution synchrone où le temps est discret, basé sur une horloge, et le moteur des évènements du navigateur nous a permis de proposer une nouvelle version de la compilation d'un programme pendulum dans un contexte de client Web. Cet ajout permet de faire des appels récursifs au programme respectant le temps logique synchrone et de traiter les déclenchements asynchrone de la page Web, comme les requêtes. Cette approche permet aussi une exécution efficace dans le traitement d'un grand nombre d'évènements, grâce à l'accumulation des modifications en une seule écriture sur le document. Dans la partie suivante, on se sert de ces ajouts pour proposer sous la forme de tutoriels, différentes implémentations pour des programmes de moyenne envergure.

Chapitre 6

Applications

Pour montrer les possibilités de pendulum dans la programmation client Web, on exhibe une série d'exemples qui apportent chacun un aspect différent du problème et comment utiliser la programmation synchrone pour le résoudre.

Le premier exemple est une interface multimédia basée sur le lecteur audio-vidéo HTML5, dont les contrôles (lecteur, pause, etc.) sont manipulés par le langage synchrone. Cet exemple montre l'intérêt de la communication entre les tâches au travers des signaux. L'exemple suivant expose la gestion des appels aux requêtes HTTP avec AJAX depuis un programme synchrone, où l'on insiste sur les appels asynchrones qui . On présente ensuite la programmation d'un jeu vidéo en 2D, dans un canevas HTML5 pour expérimenter une application avec un rafraîchissement rapide et plusieurs évènements simultanés. On terminera cette suite d'exemples par l'interopérabilité entre pendulum et Eliom notamment pour montrer des problématiques de développement sur terminal mobile mais aussi l'interopérabilité entre la programmation réactive synchrone et programmation réactive fonctionnelle.

6.1 Lecteur multimédia

Le langage HTML5 inclut des balises de lecteurs multimédia, comme `<video/>` et `<audio/>`. Les contrôles du lecteur comme le bouton de lecture et pause, barre d'avancement et de modification du temps de lecture, sont déjà implémentés et intégrés avec la propriété HTML `controls`. Au lieu d'utiliser ces contrôles par défaut, on programme nos propres composants et on gère leurs comportements respectifs avec un programme synchrone. Le programme complet se trouve à l'adresse suivante : `https://github.com/remyzorg/pendulum/tree/master/examples/player`. On souhaite implémenter les trois fonctionnalités suivantes :

- Bouton de lecture/pause
- Affichage du temps courant dans une barre de progression
- Interaction avec la barre de progression pour changer le temps courant de la lecture

Dans notre exemple, on présente avec une balise `<audio/>` mais seule la balise change, et le programme serait le même avec une balise `<video/>`. On commence ce programme par le code en HTML qui décrit la structure de la page. On utilise notre balise média qui contient une balise `<source/>` qui décrit le chemin vers la ressource et son type de fichier. On a ensuite un bouton et une balise `<input/>` sous forme de barre de progression. On peut observer le résultat de ce code dans la figure 6.1.

```
<audio id="media">
  <source src="aerosmith.mp3" type="audio/mp3">
</audio>
<button id="playbtn">Play</button>
<input type="range" id="progress" value="0"/>
```

FIGURE 6.1 – Capture d'écran des contrôles du lecteur

On décrit maintenant le programme synchrone orchestrant notre lecteur audio que l'on nomme `reactive_player` avec trois éléments en arguments, correspondant aux composant HTML introduits ci-dessus. On ajoute aussi un signal local qui représente l'état courant de la lecture, `true` si elle est en cours et `false` si elle est sur pause.

```
let%sync reactive_player =
  element playbtn;
  element media;
  element progress;

  let state = media##.autoplay in
  loop (
    present playbtn##onclick (
      emit state (not !!state);
      !(update_state !!state media playbtn);
      !(update_btn_content !!state playbtn);
    )
  ; pause)
```

Le code qui suit les arguments, est une boucle infinie qui va tester à chaque instant la présence du clic sur le bouton lecture, et inverser la valeur du signal `state` le cas échéant. On exécute aussi la fonction du langage hôte `update_btn_content` qui modifie l'aspect du bouton avec les chaînes de caractères `"Play"` ou `"Pause"`. Puis on exécute la fonction `update_state` qui elle même exécute les fonctions `play()` et `pause()` du DOM pour changer l'état réel de la lecture de la balise.

On veut maintenant capturer les mises-à-jour de progression de la lecture du média. On met ce code en parallèle à la suite du précédent. L'API du DOM précise que l'évènement `timeupdate` ciblant les éléments audio et vidéo peut être utilisé dans ce but. On utilise l'opérateur de liaison évènementiel pour le représenter par un signal.

```
  ||
  loop (
    present media##ontimeupdate (
      !(update_slider progress media)
```

```
    ); pause
  )
```

La fonction `update_slide` modifie la propriété `value` de l'élément du `progress` pour que le curseur de la barre soit à la coordonnée correspondant au temps courant de la lecture. Si on veut interagir et déplacer le curseur de la barre de progression pour modifier le temps courant, on doit éviter que l'évènement de lecture ne déclenche le déplacement à ce moment là.

FIGURE 6.2 – Bug : le curseur saute pendant le déplacement

Dans le cas contraire, on aurait le comportement décrit dans la figure 6.2 où le curseur se trouve alternativement sous le curseur de la souris, et à la coordonnée où `update_slide` l'a placé. On remplace donc le code par le suivant où l'on a ajouté un signal local `seeking` dont la spécification est qu'il est présent lorsque l'utilisateur déplace le curseur.

```
  ||
  let no = () in
  loop (
    present media##ontimeupdate (
      present seeking
       (* then *) nothing
       (* else *) !(update_slider progress media)
  ); pause)
```

On utilise deux tests qui expriment qu'à chaque mise-à-jour de la vidéo, on teste si `seeking` est présent. Si oui on exécute `nothing`, sinon on exécute la mise-à-jour de `progress`.

On utilise ce signal dans la prochaine tâche parallèle qui consiste à gérer l'interaction avec `progress`. Cette dernière tâche met à jour le temps courant de la lecture, en fonction des coordonnées à laquelle le curseur est déplacé. On utilise un signal basé sur les évènements `onmousedown` et `onmouseup`. Entre ces deux évènements, on sait que l'utilisateur déplace le curseur de `progress` et c'est l'état dans lequel on doit émettre `seeking`. On ajoute le code suivant en parallèle du précédent.

```
  ||
  loop (
    present progress##onmousedown (
      trap release (
        loop (
          emit seeking;
          present progress##onmouseup (
            !(update_media media progress);
            exit release
          ); pause))
  ); pause)
```

Cette tâche est définie comme une boucle infinie qui attend que le bouton de la souris soit enfoncé `onmousedown`. Lorsque c'est le cas on rentre dans un état défini par un `trap` et une boucle. Dans cet état, on émet à chaque instant le signal `seeking` pour communiquer à l'autre tâche de bloquer la mise-à-jour de `progress`. On test aussi à chaque instant si le signal de relâchement du bouton de la souris est présent. Si c'est le cas on quitte cet état grâce au label d'échappement `release` et on retourne à l'état où l'on attend le signal basé sur `onmousedown`.

Dans cet exemple on peut observer les avantages de la programmation synchrone et de l'opérateur parallèle pour ajouter des comportements au fur et à mesure sans se poser de question d'ordonnancement. Cela autorise à construire le programme de façon incrémentale. On peut voir aussi à quel point il est facile de faire communiquer deux comportements au travers des signaux et de bloquer l'un grâce à l'autre. On voit aussi qu'il est aisé de séparer la modification pure du DOM par effet de bord et le programme concurrent.

6.2 Gestion des communications asynchrones

En JavaScript, on utilise la méthode AJAX pour effectuer des échanges interactifs avec le serveur sans mettre à jour la page. AJAX est l'acronyme de *asynchronous JavaScript and XML*, où asynchrone qualifie le modèle de concurrence utilisé pour démarrer une requête et recevoir la réponse. XML est supposé être le format des données échangées, mais ce dernier peut en vérité être plus large. Le modèle de concurrence pour les communications suit donc les standards du DOM et utilise des évènements et des fonctions de rappel. Une requête est un objet du langage :

```
var oReq = new XMLHttpRequest();
oReq.addEventListener("load", reqListener);
oReq.open("GET", "http://www.example.org/example.txt");
oReq.send();
```

Dans ce code, `reqListener` est la fonction de rappel qui s'exécute quand la réponse de la requête est arrivée, déclenchée par l'évènement `"load"`. Dans le contexte de Js_of_ocaml, les requêtes s'utilisent d'une façon similaire. On définit par exemple la fonction `get`. Cette fonction prend en paramètre une chaîne de caractères et une fonction de rappel qui prend en paramètre la réponse de la requête au format textuel. Son type et sa déclaration s'écrivent comme suit :

```
val get : string → (string → unit) → unit

let get url complete =
  let rq = XmlHttpRequest.create () in
  rq##.onload := Dom.handler
    (fun _ → complete (Js.to_string rq##.responseText); Js._true) ;
    rq##_open (Js.string "GET") (Js.string url) Js._true;
  rq##send Js.null
```

La présence de typage introduit des différences entre ces deux programmes : l'évènement est un champ typé de l'objet `rq` et non pas une chaîne de caractères, une traduction est nécessaire entre les chaînes de caractères OCaml et Js_of_ocaml, etc. Malgré tout, si on connait l'API du DOM, on retrouve les objets et méthodes de ce dernier.

On peut alors utiliser cette fonction dans un programme synchrone. On peut démarrer une exécution asynchrone depuis ce programme via l'instruction atomique `!()`. La fonction de rappel passée doit à la fois émettre le signal à l'instant suivant et déclencher l'exécution d'un instant.

```
let%sync fetch ~animate =
  element text;
  output complete;

  loop (
    present text##onchange (
      !(get text##.value
          (fun v →
            Signal.emit complete v
            ; animate ()
          ))
    ) ; pause
  )
```

On utilise le déclenchement d'instant différé donné par la fonction `animate`, définie dans le chapitre précédent, qui nous donne une version récursive sûre de la fonction de réaction. La fonction `Signal.emit` ne met pas à jour le signal dans l'instant courant, car la fonction anonyme que l'on utilise comme fonction de rappel de l'évènement ne peut pas être déclenchée tant que l'instant courant est en cours d'exécution, les fonctions de rappel en JavaScript étant atomiques. Le résultat de la requête se trouve alors dans la valeur transportée par le signal `complete`.

6.3 Intégration avec Eliom

Eliom [4] est la plateforme de développement du projet OCsigen qui agrège les différentes technologies décrites en Introduction, dans la sous-section 1.2.1. Cette plateforme permet le développement d'une application Web et mobile de bout en bout. Via TyXML, Eliom propose la construction de composants HTML persistants et statiquement typés, respectant les règles de sémantique du standard. En particulier, on est certain que le HTML généré par Eliom est bien formé. Eliom possède un modèle de concurrence coopératif, Lwt, utilisé par l'interface de programmation pour toutes les communications et appels asynchrones, ainsi qu'un modèle de programmation réactive que nous allons introduire par la suite. Il est intéressant d'incorporer la programmation synchrone-réactive à une plateforme aboutie proposant des paradigmes modernes tels que le multitier avec des communications implicites entre client et serveur.

6.3.1 Les signaux de sortie réactifs

L'utilisation de valeurs immuables via TyXML pour décrire la structure de l'interface est une approche ayant une saveur très fonctionnelle qui permet aussi d'éviter de nombreuses erreurs et conserver un aspect déclaratif au document. Il arrive néanmoins un moment où ces éléments doivent être mis-à-jour pour que le document puisse interagir. Il est alors possible d'accéder à une référence mutable au format DOM d'un élément HTML à travers la bibliothèque Js_of_ocaml, et de le modifier par effet de bord. Le format DOM n'est accessible que côté client, par contre la construction des éléments HTML est accessible côté client et côté serveur.

```
open Html.F

let txt_elt = span
let update _ =
  let dom_txt = To_dom.of_span txt_elt in
  dom_txt##.textContent :=
    Js.(string (to_string dom_txt##textContent ^ "Hello")

let btn1 = button
  ~a:[a_onclick update]
  ["Submit"]
```

On est forcé d'abandonner les principes de la programmation fonctionnelle et on sépare en deux parties la construction de l'élément et son comportement. Au lieu de localiser le code qui décrit le comportement de l'élément `txt_elt` à l'endroit où il est construit, on décrit son comportement dans la description d'un autre élément de l'interface qui est le bouton. On pourrait choisir une autre alternative qui améliore la composition du programme où chaque élément est décrit à part et d'un seul bloc. C'est l'approche que propose l'association des bibliothèques React et TyXML.

```
open Html.F
let s, set_s = React.S.create [] in
let txt_elt =
  R.node (React.S.map (fun s → span (List.map pcdata s)) s)
in
let b = button ~a:[a_onclick (fun _ →
    set_s ("Hello " :: React.S.value s))] ["add"]
in
div [txt_elt; b; reset]
```

Dans le code ci-dessus, on construit un signal `s` de type `string list React.signal`. Ensuite on utilise la fonction R.node qui permet de construire un élément HTML par application d'un fonction sur `s`. On peut ensuite construire un bouton qui, une fois cliqué, va mettre à jour le signal et par propagation, `txt_elt`. On a une description explicite de la façon dont se met à jour un composant en fonction d'une ou plusieurs entrées. On est ensuite libre de décrire d'autres interactions qui mettent à jour le signal `s` ou qui réagissent à sa modification : le modèle est composable. On peut par exemple construire après coup le bouton suivant qui met à jour le signal et donc l'affichage. Celui-ci permet de remettre à zéro la liste de chaînes de caractères.

```
let reset = button ~a:[a_onclick (fun _ → set_s [])] ["reset"] in
```

La mise-à-jour d'un composant et sa définition est une partie de l'application qu'il est plus commode de décrire en programmation réactive fonctionnelle et qui n'est pas capturée par le modèle de pendulum. Un programme synchrone définie une machine à état, mais pas une composition de valeurs. Pour modifier des valeurs, le programme synchrone agit par effet de bord. A l'inverse la programme réactive fonctionnelle propose une vision persistante des données. Par contre, la programmation réactive synchrone est très efficace pour décrire une vision globale et la multiplicité des interactions, lorsqu'il ne s'agit pas simplement de connecter une entrée à une sortie.

Si chacun des deux modèles peut résoudre une partie du problème plus efficacement, on peut les associer et profiter du meilleur des deux approches. Pour cela, on ajoute à pendulum une interopérabilité avec React se situant au niveau des définitions de signaux en en-tête. Pour cela, on donne une nouvelle catégorie de signaux de sortie, définis par le mot clef `react`. Au lieu de se présenter comme interface de sortie une fonction de rappel à la façon des **`output`**, ils présentent un signal fonctionnel réactif provenant de la bibliothèque React, c'est-à-dire une valeur de type `'a React.signal`. Il est alors possible de lier n'importe quel composant réactif à ce signal comme ceux du module `R` présenté au dessus. Supposons que l'on créer un signal de sortie FRP - `react a;` - pour le programme `p`, le type du programme est

```
val p : < create : 'a →
  < a : 'a React.signal ; react : unit > >
```

Ici, la méthode `a` est un accesseur du signal FRP. Pour une illustration plus complète, on donne une nouvelle version du programme de la section 2.3 qui affiche les coordonnées de la souris. Au lieu d'écrire l'émission directement dans la propriété de l'élément, on déclare un signal `react write`.

```
let%sync mouse_loc3 =
  element w {
    onmousemove = "", (fun x ev →
      sprintf "%d,%d" ev##.clientX ev##.clientY);
  };
  react write;
  loop (
    present w##onmousemove
      (emit write !!(w##onmousemove))
  ; pause
  )
```

Il n'y a pas d'autre différence entre `mouse_loc3` et ses versions précédentes. On remarque néanmoins qu'un signal FRP s'utilise de la même façon qu'un signal de sortie standard. La vraie différence se situe dans l'utilisation du signal et l'interface de programmation fournie par `mouse_loc3`. Comme précédemment, ce programme a besoin des valeurs d'initialisation. Effectivement, un signal React se construit à partir d'une valeur initiale, de façon similaire aux signaux de pendulum.

```
let onload _ =
  let m = mouse_loc3#create window ("") in
  let sp5 = R.node (React.S.map (fun (x : string) → span [pcdata x]) m#write) in
  Dom.appendChild document##.body (Tyxml_js.To_dom.of_span sp5);
  Js._false
```

Une fois l'instance `m` construite, on peut lier le signal React `m#write` à autant de composants du DOM que nécessaire. Ici on le lie à un élément `R.span` que l'on incruste ensuite dans l'élément `body` de la page.

Cette extension au langage nécessite une modification légère du compilateur de pendulum. consistante à appeler un constructeur de signaux React à l'initialisation, et à utiliser la fonction de mise-à-jour du signal comme fonction de rappel de signal de sortie. Cela facilite l'écriture de programme pendulum en profitant des avantages d'autres paradigmes. De plus, de nombreux composants d'Eliom utilisent une interface de programmation réactive. On améliore donc l'interopérabilité entre Eliom et pendulum.

6.3.2 Application mobile

La programmation d'applications adaptée à des terminaux mobiles n'est aujourd'hui plus seulement un enjeu futur, mais une réalité, et les technologies doivent s'y adapter, en particulier celle du Web. Un terminal mobile, de par sa dimension nomade perçoit des interactions différentes d'un navigateur s'exécutant sur un terminal fixe. Les applications mobiles utilisent largement les capteurs tels que la géolocalisation, la boussole, l'accéléromètre ou la caméra. Les écrans tactiles, largement répandus dans les terminaux mobiles proposent aussi des interactions différentes sur le Web, et demandent une refonte des règles de navigation. Les navigateurs prennent en compte ces nouvelles fonctionnalités au fur et à mesure et l'API du DOM est ensuite augmentée en conséquence. L'objet global `window.navigator`, en particulier, contient des informations sur le matériel, comme `window.navigator.geolocation` [50] qui permet d'obtenir les coordonnées GPS du terminal à intervalle régulier et d'une précision variable.

Pour montrer l'intérêt de pendulum pour programmer ce type d'applications, on donne un exemple contenant plusieurs interactions. On utilise Eliom comme contexte de programmation Web, qui apporte l'aspect multitier en OCaml et la base d'application mobile fournie par Ocsigen-Start [80]. Nous appelons cet exemple *Cartes*.

Cartes est une application de cartographie pour terminaux mobiles et Web. Elle se base sur l'interface de programmation du service GoogleMaps [72] pour afficher une carte dans la fenêtre de l'utilisateur. Une interface de programmation est proposée par la bibliothèque ocaml-googlemaps [78]. Celle-ci utilise la géolocalisation du terminal client pour marquer la position de l'utilisateur sur cette carte. L'application a deux modes.

- Dans le premier, qui est le mode par défaut, la fenêtre est centrée sur le marqueur de l'utilisateur au long de ses déplacements et l'affichage du marqueur se met à jour en fonction.
- Dans le second mode, l'utilisateur peut choisir de rechercher un lieu sur la carte en sélectionnant un champ de texte. L'écran est alors centré sur les résultats de la recherche et ne suit plus la localisation du terminal. L'utilisateur peut choisir de centrer la fenêtre sur lui à nouveau en appuyant sur un bouton.

Le projet contenant cette application se trouve à l'adresse suivante : `https://github.com/remyzorg/pdldemo`.

Structure de la page et initialisation du code

Le prototype d'application Ocsigen-Start se compose de différentes pages au travers desquelles on peut naviguer via un panneau latéral. Chaque page est alors décrite par un module contenant une fonction `page`. On crée donc notre module `Demo_map`, dans lequel on écrit l'ensemble du code et la fonction `Demo_map.page`.

```
let%shared page () =
  let searchbox = input ~a:[ a_input_type `Text ] () in
  let map = div [] in
  let locate = button ["Locate"] in
  let _ = [%client
    Lwt_js_events.async (fun () →
    (* initialisation de la carte *)
    );] in
  Lwt.return [ searchbox ; locate; map ]
```

On donne tout d'abord la structure de notre document qui se compose d'un élément graphique contenant la carte (`map`), d'une barre de recherche (`searchbox`) et d'un bouton de localisation (`locate`). Le bloc client qui suit les définitions est exécuté de façon asynchrone au chargement de la page et c'est là que l'on initialise la carte et la bibliothèque GoogleMaps. On notera que la fonction `page` peut être exécutée plusieurs fois sans rafraîchissement de la page HTML, lors d'un changement d'onglet par exemple. On donne ensuite quatre fonctions dans un bloc de code client en utilisant l'annotation `[%%client ]`.

```
[%%client
open Googlemaps

let onmapsload (f : unit → unit) : unit =  ...
let start_watching_loc (f : int * int → unit )  =  ...

let create_map (lat,lng) map =
  let center = LatLng.new_lat_lng ~lat ~lng in
  let opts = MapOptions.create ~zoom:15 ~center () in
  Map.new_map (To_dom.of_div map) ~opts () in

let create_mymarker map (lat,lng) =
  let position = LatLng.new_lat_lng ~lat ~lng in
  let opts = MarkerOptions.create
      ~position
      ~title:("Moi")
      ~draggable:false
      ~map:map () in
  Marker.new_marker ~opts ()
]
```

La première fonction, `onmapsload f`, enregistre la fonction `f` comme fonction de rappel du chargement de la bibliothèque GoogleMaps, qui est sous forme de script distant. La seconde fonction, `onwatchingloc f` prend en paramètre une fonction de rappel `f` qu'elle appelle lorsque que la géolocalisation du terminal est mise à jour. La troisième fonction, `createmap` construit une valeur de type `Map.t` avec un niveau de zoom de 15, c'est-à-dire à l'échelle d'un ensemble de rues. La fonction de création `new_map` prend en paramètre l'élément HTML dans lequel la carte est créée. La dernière fonction construit un marqueur à la position donnée sur la carte donnée.

Création de la carte

Le premier enjeu est d'initialiser le contexte de calcul du programme : une carte. Pour cela, il y a plusieurs ressources auxquelles on doit accéder dans un ordre précis en s'assurant qu'elles existent.

- La page doit être chargée pour pouvoir la modifier or le script peut s'exécuter avant le chargement complet du document.
- Pour intégrer la carte dans le document, la bibliothèque de cartographie doit être téléchargée et le script exécuté.
- La bibliothèque ne doit être chargée qu'une fois et non pas à chaque chargement de page.
- La carte a besoin de coordonnées pour s'afficher. On les récupère via l'API du DOM en Js_of_ocaml avec la fonction `Geolocation.geolocation##watchPosition`.

On va représenter ces différents états d'avancement dans le programme synchrone pour respecter ces contraintes. On crée ce dernier en l'appelant `carto`. Le programme a un signal d'entrée en argument, `page_loading` qui est émis lorsque la page se charge. La valeur qu'il transporte a le type ci-dessous, qui transporte les éléments dont on a besoin lors du rechargement de l'onglet.

```
type%client page_load = {
  map : Html_types.div Eliom_content.Html.D.elt;
}
```

Pour l'instant, on a uniquement la balise qui contient la carte. Le contenu de la page se trouve dans des onglets, donc ce dernier se charge dynamiquement lorsque l'onglet est sélectionné. Ce n'est pas le cas de la bibliothèque Googlemaps qui reste présente lorsque du changement d'onglet. On définit ensuite deux signaux locaux `location` et `map_loaded`. Après la définition des signaux, on exécute deux appels asynchrones. Le premier se déclenche au chargement de la bibliothèque de cartographie et il émet le signal `map_loaded`. Le second appel asynchrone ré-exécute sa fonction de rappel à chaque fois qu'une nouvelle géolocalisation est trouvée pour émettre le signal `location`. On notera que ces émissions des signaux ne se font pas pendant mais entre deux instants, lorsque le navigateur reprend la main sur l'exécution pour gérer la pile des évènements.

```
let%sync carto ~animate =
  input (page_loading : page_load);

  let location = 0., 0. in
  let map_loaded = () in

  !(onmapsload (fun () →
      Pendulum.Signal.emit map_loading ();
      animate () ));
  !(start_watching_loc (fun loc →
      Pendulum.Signal.emit location loc;
      animate () ));
```

Ces deux fonctions vont s'exécuter au premier chargement du programme `carto`. La suite consiste à gérer ces trois signaux. Dans la suite du code, ci-dessous, on utilise le mot clef **`await`** que l'on compose en parallèle, permettant de bloquer sur cette instruction pendant de multiples instants tant que les trois signaux, `page_loading`, `map_loaded` et `location` n'ont pas été présents au moins une fois. Cela permet néanmoins de considérer chaque signal à des instants potentiellement différents. On imprime ensuite sur la console du navigateur que le chargement est effectué. Dans le code, suivant les marqueurs `(* < 1 > *)` indique la localisation des ajouts de code au fur et à mesure de cette partie.

```
  await page_loading || await map_loaded || await location;
  !(print_endline "Map loaded");
  loop ( trap reload (
    let map = create_map !!location (!!page_loading).map in
    let mymarker = create_mymarker !!map !!location in
    (* < 1 > *)

    loop ( present page_loading ( exit reload ))
```

```
    ||
    loop ( present location (
      !(update_location !!map !!mymarker !!location)
    ))
    (* < 2 > *)
  ))
```

On entre ensuite dans un bloc `loop` associé à un bloc `trap` nous permettant de réinitialiser son comportement sous certaines conditions. En l'occurrence, lorsque la page est rechargée, on veut recharger la carte et le marqueur, en se basant sur `page_loading`.

Dans ce double bloc, on crée deux signaux locaux : la carte et le marqueur de position de l'utilisateur en utilisant `create_map` qui construit la carte à partir de la valeur contenue dans `page_loading`, puis le marqueur à partir de la carte. Pour terminer cette partie, on déclare deux boucles qui s'exécutent en parallèle. L'une capture l'émission du signal `page_loading`, et effectue un "reset" le cas échéant. L'autre boucle met à jour la carte et le marqueur courant si une nouvelle localisation est trouvée.

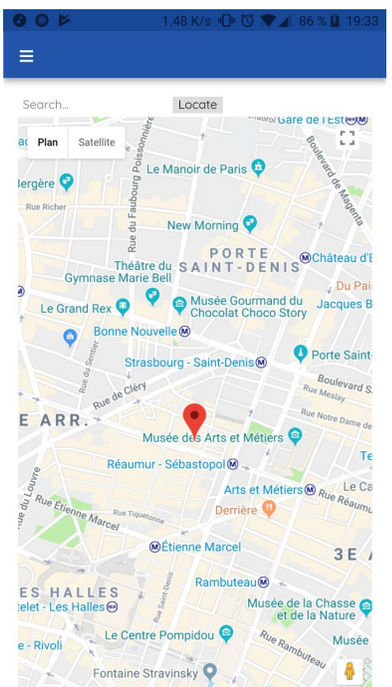


FIGURE 6.3 – Capture d'écran de en localisation

A la fin, on obtient une carte dont le centre est replacé à la position de l'utilisateur et avec un marqueur sur cette position, comme on peut l'observer figure 6.3

Rechercher sur la carte

On souhaite maintenant ajouter à notre carte une barre de texte dans laquelle taper le nom d'un lieu à rechercher grâce à l'API Googlemaps Places. Le champ de texte propose alors des suggestions pour compléter l'entrée courante et une fois validé nous donne accès aux lieux correspondants et leur géolocalisation. On utilise le module `SearchBox` qui permet de construire ce type d'élément. On augmente notre type `page_load` pour y ajouter la balise `input` qui représentera la barre de recherche.

```
type%client page_load = {
  map : Html_types.div Eliom_content.Html.D.elt;
  search_inp : Html_types.input Eliom_content.Html.D.elt;
}
```

On donne ensuite la fonction d'initialisation de la barre de recherche.

```
[%%client
  let init_searchbox elt f =
    let inp = To_dom.of_element elt in
    let sb = SearchBox.new_search_box inp () in
    ignore @@ Event.add_listener (SearchBox.t_to_js sb) "places_changed" (fun _ →
        f @@ SearchBox.get_places sb);
    sb
]
```

On utilise l'évènement `"places_changed"`, fourni par la bibliothèque de cartographie. Comme précédemment on construit cette fonction comme prenant une fonction de rappel en paramètre. On peut alors utiliser cette fonction pour construire cet élément et lui donner comme fonction de rappel la mise-à-jour du signal et le déclenchement du programme courant. La définition qui suit est écrite à l'emplacement du symbole `< 1 >` dans le programme synchrone.

```
    let places_changed = [] in
    let markers = [] in
    let searchbox = init_searchbox (!!page_loading).search_inp
        (fun places →
           Pendulum.Signal.emit places_changed places ;
           animate ()) in
```

Une fois les signaux construits, on peut utiliser le signal `places_changed` pour détecter la mise-à-jour de la recherche. On ajoute le code suivant en parallèle des autres à l'emplacement `< 2 >`.

```
    || loop ( present places_changed (
         !(remove_markers !!markers);
          emit markers (generate_markers !!map !!places_changed)))
```

On récupère la valeur stockée dans le signal et on construit les marqueurs correspondant à la liste de lieux sur la carte en ayant effacé préalablement les marqueurs précédents, d'où les deux fonctions suivantes, que l'on définit en dehors du programme synchrone.

```
[%%client
  let remove_markers ms =
    List.iter (fun m → Marker.set_map m None) ms
```

Supprimer un marqueur dans cette bibliothèque consiste à lui retirer l'affectation à une carte. On parcours la liste des lieux grâce à un itérateur qui génère deux valeurs par accumulation : la liste des marqueurs correspondant aux lieux et le rectangle qui englobe l'ensemble de ces lieux. Pour étendre le rectangle on fait appel à la fonction `extend`. On rend ensuite les nouveaux marqueurs qui seront stockés dans le signal `markers`. Ce code est dans la fonction `generate_markers` dont le type et la définition sont ci-dessous.

```
val generate_markers : Map.t → PlaceResult.t list → Marker.t list
```

```
  let generate_markers map places =
    let bds, markers = List.fold_left (fun (b, mrks) p →
      let position = PlaceGeometry.location @@ PlaceResult.geometry p in
      let bounds = match b with
      | None → Some (LatLngBounds.new_lat_lng_bounds ~sw:position ~ne:position)
      | Some b → Some (LatLngBounds.extend b position)
      in
      let opts = MarkerOptions.create
          ~position
          ~title:(PlaceResult.name p)
          ~map ()
      in bounds, Marker.new_marker ~opts () :: mrks
    ) (None, []) places
    in
    Map.set_zoom map (min (Map.get_zoom map) 16);
    Eliom_lib.Option.iter (Map.fit_bounds map) bds;
    markers
]
```

On observe le résultat de ce code dans la figure 6.4. Où la recherche `italiens à proximité de Jussieu` a rendu une liste de lieux.

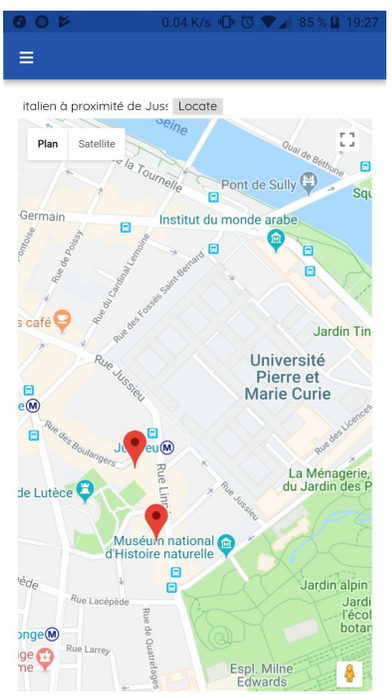


FIGURE 6.4 – Capture d'écran en recherche

Bouton de localisation

On veut maintenant instaurer deux états de visualisation de la carte, le premier où l'on suit la localisation de l'utilisateur et le deuxième où l'on recherche. On passe de l'un à l'autre en utilisant soit la barre de recherche, soit un bouton `locate` que l'on va ajouter dans cette partie. On augmente à nouveau le type `page_loaded` avec notre bouton.

```
type%client page_load = {
  map : Html_types.div Eliom_content.Html.D.elt;
  search_inp : Html_types.input Eliom_content.Html.D.elt;
  locate_btn : Html_types.button Eliom_content.Html.D.elt;
}
```

On ajoute ensuite deux définitions locales à l'emplacement `< 1 >`, ainsi que le comportement du bouton. Lorsque qu'il est cliqué, le signal `locate` doit être émis.

```
    let locate = () in
    let searching = () in
    !((To_dom.of_button (!!page_loading).locate_btn)##.onclick :=
         Dom_html.handler (fun _ → Pendulum.Signal.emit locate ();
         animate (); Js._true ));
```

On ajoute ensuite une tâche parallèle et on modifie une des tâches écrites précédemment. On redonne l'ensemble des tâches exécutées en parallèle pour simplifier la lecture. Les deux premières n'ont subi aucune modifications par rapport au code précédent. La troisième permet de faire changer le centre de la carte au moment du clic sur le bouton en émettant le signal `location` avec sa propre valeur.

```
    loop ( present page_loading ( exit reload ))
    ||
    loop ( present places_changed (
      !(remove_markers !!markers);
      emit markers (generate_markers !!map !!places_changed)))
    ||
    loop ( present locate ( emit location !!location); pause)
    ||
    loop ( present places_changed (
      trap t (loop (present locate (exit t); emit searching;))))
    ||
    loop ( present location (
      present searching nothing !(update_location !!map !!mymarker !!location)
      ))
```

La quatrième tâche est nouvelle et permet d'émettre le signal `searching` dès que `places_changed` est émis, tant que le signal `locate` n'est pas présent. Ceci maintient l'état de recherche. La dernière tâche est une modification du code de la partie précédente. On y teste la présente du signal `searching` avant de mettre à jour la localisation sur la carte. Si une nouvelle géolocalisation a été trouvée mais que ce dernier est présent, on ne bouge pas le centre de la carte.

Lancement du programme

On crée l'instance de notre programme `carto` avec des éléments vides.

```
let%client carto =
  carto#create ({map = div []; search_inp = input (); locate_btn = button []})
```

Et on donne notre nouvelle définition de la fonction page, qui contient la déclaration des différents éléments HTML utilisés ainsi que le bloc client exécuté à chaque fois que l'onglet est rechargé.

```
let%shared page () =
  let search_inp = input ~a:[a_input_type `Text; ] () in
  let map = div [] in
  let locate_btn = button [pcdata "Locate"] in
  let page = [div [search_inp; locate_btn] ; map] in
  let _ : unit Eliom_client_value.t = [%client
    Lwt_js_events.async (fun () →
      p#page_loading ({map = ~%map;
                       search_inp = ~%search_inp;
                       locate_btn = ~%locate_btn });
      p#react;
      Lwt.return ()
    )]
  in
  Lwt.return page
```

Le bloc client en question ne fait qu'une émission du signal `page_loading` avec les différents éléments, la carte, le bloc de texte et le bouton.

On a vu que notre programme contenait de multiples interactions. Les boutons, la géolocalisation, ou l'accès à des ressources distantes telles que du code. Il a été possible d'exprimer toutes ces relations de causalité dans un programme concis et exhaustif en gérant aussi le chargement des onglets et la réinitialisation de l'interface cliente.

Conclusion

Cette partie nous a permis de mettre en avant les atouts de la programmation synchrone en résolvant des problématiques inhérentes au client Web. Les interfaces présentées sont complexes compte tenu de la taille des programmes qui les manipulent, comme la cartographie, ou la lecture multimédia. On a montré que pendulum était capable d'utiliser les appels asynchrones dont la fonction de rappel répond au travers des signaux. Le langage peut aussi profiter à une application mobile en interopérant, autant en termes de technologie que de paradigme de programmation, avec la plateforme Eliom.

Chapitre 7

Conclusion et perspectives

En introduction, nous avons soulevé les problématiques inhérentes à la programmation Web dans sa forme la plus répandue. Nous avons en particulier abordé les problèmes liés à son langage central, JavaScript, ainsi qu'au modèle de concurrence du navigateur : la programmation évènementielle. Nous avons en particulier mis l'accent sur la connexion entre le programme et le monde extérieur, que l'on appelle les interactions : les communications avec un serveur, les capteurs ou les actions de l'utilisateur. De notre point de vue, les méthodes de programmation les plus répandues ne permettent pas d'installer un contexte de sûreté pour le programmeur et ne permettent pas d'exprimer les réactions aux interactions dans toute leur complexité, en particulier quand ces dernières créent des relations de dépendance forte entre elles et avec les données qu'elles manipulent.

Dans la suite de cette introduction, nous avons donné une vue d'ensemble de notre perception de la programmation Web aujourd'hui. Nous avons d'abord présenté les techniques issues du standard et leur modèle d'exécution, ainsi que le modèle de concurrence du client Web : la programmation évènementielle avec fonctions de rappel. Ont été présentées ensuite les techniques modernes qui tentent d'améliorer cette condition, en particulier en termes de sûreté grâce au typage ou à des sémantiques de langages plus précises, tant au niveau du modèle de concurrence que de la gestion des interactions. Sur cette base, nous avons présenté notre propre solution au problème de sûreté et d'expressivité soulevé, en proposant un langage, pendulum, inspiré du modèle de programmation réactif-synchrone. Ce langage se place dans un cadre typé statiquement en proposant une extension d'OCaml, langage pouvant être compilé vers JavaScript.

Le chapitre 2 présente le langage et ses constructions. Le modèle d'exécution réactif-synchrone, étant fondamentalement différent de la programmation généraliste usuelle, on montre les échanges entre les niveaux de programmation : pendulum avec OCaml pour l'interface de programmation et le lancement d'un programme synchrone, puis la partie synchrone avec les évènements du Web. On propose alors une meilleure interopérabilité avec les évènements, qui deviennent des valeurs à part entière du monde synchrone sous la forme de signaux. On pourrait imaginer remplacer l'utilisation du modèle évènementiel par le modèle synchrone comme référence de programmation concurrente, à partir du moment où le programmeur n'a plus besoin de manipuler explicitement les évènements du navigateur. On peut donc donner au programmeur le sentiment de manipuler ses outils habituels sous une nouvelle forme, plus sûre et plus expressive, par la capacité de composer les programmes s'exécutant en parallèle.

Les chapitres 3 et 4 montrent la façon dont nous avons mis en œuvre ce langage. De la source synchrone au code OCaml agrémenté de JavaScript, on utilise une technique existante de compilation prévue pour Esterel dont la structure intermédiaire est un graphe de flot de contrôle des instants possibles du programme. Le graphe de flot de contrôle nécessite un ordonnancement statique pour rendre linéaire tous

les chemins de tâches exécutées en parallèle. Nous proposons un algorithme pour résoudre ce problème qui s'inspire du travail de Edwards [27] en travaillant par décomposition des nœuds du graphe. Pour finir, la compilation des éléments spécifiques au Web utilise les noms d'évènements donnés par le programme dans le programme synchrone pour générer les gestionnaires d'évènements correspondants et les lier automatiquement à des signaux.

Dans le chapitre 5, nous avons creusé le sujet de l'exécution d'un programme synchrone sur le modèle d'exécution du navigateur. Nous avons tout d'abord montré comment proposer une interface de programmation récursive pour un programme synchrone sans casser la cohérence de la fonction d'instant en reportant l'exécution de l'itération suivante. En utilisant ce principe, on montre comment un programme pendulum peut automatiquement s'adapter à la fréquence de rafraîchissement d'une page Web, permettant à celui-ci d'être plus efficace en vitesse d'exécution sur notre ensemble de tests qu'un programme écrit à la main en JavaScript non-optimisé. Nos tests ont mis en avant que certains projets d'atelier de programmation de la communauté Web ne prenaient pas en compte ces soucis d'efficacité de façon de automatique, car le programme pendulum les dépasse en vitesse d'exécution.

Dans le dernier chapitre, nous avons proposé différents exemples de programmation en pendulum en mettant en avant son expressivité. Le langage nous a permis d'écrire plusieurs applications et en particulier une de cartographie sur mobile, en gérant les interactions à travers un programme synchrone. D'autres exemples montrent aussi la manipulation de composants multimédia envoyant des informations régulières sur leur état d'avancement, ou encore l'appel à des fonctions réagissant de manière asynchrone comme les requêtes HTTP.

L'écriture du prototype de pendulum a occupé une partie de ce travail (4000 lignes de code pour le cœur du logiciel). Le projet contient l'ensemble des sujets abordés dans ce manuscrit, comme la syntaxe, la compilation et les exemples. Ce prototype est distribué sous licence libre via la plateforme Github [1]. Il peut être installé et lié à un projet en quelques lignes de commande, interopérable sans effort de la part du programmeur avec les technologies Web en OCaml. En plus de l'utilisation orientée vers la programmation Web, l'implémentation est réalisée de telle sorte que le compilateur puisse être réutilisable pour compiler un langage synchrone vers un langage impératif quelque soit le langage hôte.

Il reste néanmoins encore des efforts à fournir pour faire de notre prototype une technologie industrielle du Web, mais nous avons plusieurs idées quand aux possibilités futures de notre réalisation.

Travaux futurs et perspectives pour Pendulum

Une technique avancée de compilation partagée L'approche statique de compilation choisie apporte malgré tout certains problèmes notamment au niveau de l'exécution entrelacée des programmes externes. Nous n'avons présenté que rapidement l'instruction `run` de pendulum dans ce travail, permettant d'appeler des programmes synchrones externes. Effectivement, le résultat en ce qui concerne cette instruction n'est pas abouti, même si nous avons approché certaines solutions. La technique de compilation GRC n'est pas adaptée à la compilation séparée [51] et l'utilisation du `run` est donc bornée à des sous-programmes dont les instants sont exécutés les un après les autres, avec l'impossibilité de décomposer ces instants. Néanmoins, en poussant plus loin les méthodes de compilation et les outils, il serait possible d'introduire un programme synchrone dans un autre par expansion en ligne [2] et de compiler le tout. Cela ne règlerait cependant pas le problème de programmes OCaml déjà compilés en code machine ou octet et liés statiquement, ou pire, à des programmes OCaml compilés en JavaScript puis liés dynamiquement dans le

1. https://github.com/remyzorg/pendulum
2. intégration ou *inlining* en anglais

client. En utilisant un format de description de la représentation GRC pour accompagner les programmes déjà compilés on transmettrait l'information, à la façon des `.cmi` en OCaml, au compilateur pendulum.

Des programmes synchrones client-server En s'inspirant des techniques utilisées dans Eliom avec les signaux partagés, il serait possible de construire une version du programme synchrone sur le client et une autre sur le serveur. L'état d'un programme synchrone étant très simple à représenter, l'envoi de l'état du programme vers le client ne poserait pas de problèmes. En calculant un instant du programme synchrone côté serveur, on génèrerait un le contenu du document HTML en conservant le même programme synchrone dans le client. Cela permettrait de factoriser le code.

Un aspect flot de données Notre proposition de langage pendulum se concentre en majorité sur les interactions et non pas sur la transformation des données. L'interopérabilité avec la bibliothèque OCaml-React permet de palier partiellement à ce problème, mais il serait intéressant de définir des signaux par composition à la manière des flots de données, sans sortir d'un programme pendulum comme on le fait actuellement. Le modèle flot de données permettrait de factoriser et faciliter l'écriture de code.

Plus de dynamicité dans l'interopérabilité avec les évènements La proposition que nous avons faite de sucre syntaxique sur les évènements, permet une interopérabilité entre les deux niveaux de concurrence. Néanmoins, pour une plus grande expressivité de programmation, on souhaiterait considérer les signaux basés sur des listes de composants et générer automatiquement l'ensemble des gestionnaires d'évènements. Dans le Web, il est souvent question d'une liste d'éléments dans une page et une interaction avec l'un de ces éléments modifie l'état de la liste. La possibilité de générer des gestionnaires d'évènements à partir de signaux locaux, ce qui n'est pas le cas actuellement où, aiderait aussi en ce sens. Les gestionnaires d'évènements sont actuellement initialisés une fois à la construction de l'instance du programme.

Les expérimentations faites avec pendulum nous montrent que la programmation synchrone est un modèle adapté pour décrire de façon précise ce qui se passe dans une application Web soumise à des interactions nombreuses. Certains comportements doivent en inhiber d'autres, certains sont interrompus et on souhaite parfois en ajouter par composition de programmes. Ces besoins sont capturés par les différentes constructions des langages synchrones. En plus de cet aspect portant sur l'expressivité, ces expérimentations ont permis de mettre en avant la corrélation entre le modèle d'exécution du navigateur et la notion d'horloge et d'instant. On a pu tirer de cette constatation un gain en vitesse d'exécution dans la gestion des évènements de façon naturelle dans notre exemple de liste de tâches dont les performances sont testées. D'une façon plus générale, ce travail montre aussi comment exécuter un langage synchrone dans un contexte d'exécution impératif, et comment il peut interopérer avec un modèle de programmation évènementiel. Nous apportons une autre ouverture à la programmation synchrone qui diffère des applications orientées vers le logiciel certifié, en montrant que ce modèle de programmation pourrait apporter de nouvelles clefs au modèle hégémonique du Web actuel centré sur des technologies certes dynamiques mais peu sûres.

Les communautés Web sont actuellement riches de langages et de paradigmes de programmations variés. Il est néanmoins clair que certaines technologies issues de la recherche peinent à trouver leur public malgré des langages très élaborés et de véritables solutions aux problèmes du programmeur Web moyen. L'utilisation de certaines de ces technologies par l'industrie pousse la communauté Web à s'y intéresser.

On a vu comment l'exemple du langage ReasonML, pourtant qu'une syntaxe alternative pour OCaml, a pu créer un engouement envers les langages statiquement typés fonctionnels, en s'associant avec le compilateur BuckleScript pour la partie JavaScript. Cette voie serait peut être à emprunter avec pendulum pour proposer au programmeur Web, et non au programmeur OCaml, la programmation synchrone.

Bibliographie

[1] Jean-Raymond Abrial, [**The B-book : Assigning Programs to Meanings**], Cambridge University Press, 1996.

[2] Sameh El-Ansary et al., [**Overcoming the Multiplicity of Languages and Technologies for Web-Based Development Using a Multi-paradigm Approach**], *Multiparadigm Programming in Mozart/Oz*, sous la dir. de Peter Van Roy, Springer, 2005, 113-124.

[3] Joe Armstrong, [**Programming Erlang : Software for a Concurrent World**], Pragmatic Bookshelf, 2007.

[4] Vincent Balat, [**Ocsigen : Typing Web Interaction With Objective Caml**], *Proceedings of the ACM Workshop on ML*, 2006, 84-94.

[5] Vincent Balat, [**Rethinking Traditional Web Interaction**], *Proceedings of the Eighth International Conference on Internet and Web Applications and Services*, Rome, Italy, 2013.

[6] Benjamin Canou and Emmanuel Chailloux and Vincent Botbol, [**Static typing & JavaScript libraries : towards a more considerate relationship**], *Proceedings of the 22nd International Conference on World Wide Web (WWW'13)*, 2013.

[7] Gérard Berry, [**Esterel V7 : From Verified Formal Specification to Efficient Industrial Designs**], *Proceedings of the 8th International Conference on Theory and Practice of Software Conference on Fundamental Approaches to Software Engineering*, FASE'05, Edinburgh, UK : Springer-Verlag, 2005.

[8] Gérard Berry, [**The Esterel v5 Language Primer - Version v5_91**], Centre de Mathématiques Appliquées, Ecole des Mines et INRIA, 2000.

[9] Gérard Berry, [**The Foundations of Esterel**], *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, 2000, 425-454.

[10] Gérard Berry et Manuel Serrano, [**Hop and HipHop : Multitier Web Orchestration**], *In Proceedings of the 10th International Conference on Distributed Computing and Internet Technology, ICDCIT 2014, Bhubaneswar, India*, 2014, 1-13.

[11] Frédéric Boussinot, [**FairThreads : mixing cooperative and preemptive threads in C**], *Concurrency and Computation : Practice and Experience* 18.5 2006, 445-469.

[12] Frédéric Boussinot, [**Reactive C : An Extension of C to Program Reactive Systems**], *Softw., Pract. Exper.* 21.4 1991, 401-428.

[13] Frédéric Boussinot, [**SugarCubes Implementation of Causality**], rapp. tech. RR-3487, INRIA, sept. 1998.

[14] Frédéric Boussinot et Robert de Simone, [**The SL Synchronous Language**], *IEEE Trans. Software Eng.* 22.4 1996, 256-266.

[15] Frédéric Boussinot et Jean-Ferdy Susini, [**The sugarCubes Tool Box : A Reactive Java Framework**], *Softw. Pract. Exper.* 28.14 déc. 1998.

[16] Benjamin Canou, [**Programmation Web Typée. (Typed Web Programming)**], Thèse de doctorat, Université Pierre et Marie Curie, Paris, France, 2011.

[17] Benjamin Canou, Vincent Balat et Emmanuel Chailloux, [**O'Browser : Objective Caml on Browsers**], *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada*, 2008, 69-78.

[18] Paul Caspi, Grégoire Hamon et Marc Pouzet, [**Lucid Synchrone, un langage de programmation des systèmes réactifs**], *Systèmes Temps-réel : Techniques de Description et de Vérification – Théorie et Outils*, t. 1, Hermès, 2006, 217-260.

[19] Paul Caspi et al., [**Lustre : A Declarative Language for Programming Synchronous Systems**], *Proceedings of the ACM Symposium on Principles of Programming Languages, POPL, Munich, Germany*, 1987, 178-188.

[20] Adam Chlipala, [**Ur/Web : A Simple Model for Programming the Web**], *POPL : Proceedings of the ACM Symposium on Principles of Programming Languages*, Mumbai, India, jan. 2015.

[21] Etienne Closse et al., [**Saxo–rt : Interpreting Esterel Semantic on a Sequential Execution Structure**], *Electronic Notes in Theoretical Computer Science*, Synchronous Languages, Applications, and Programming, SLAP'2002 (Satellite Event of ETAPS 2002), 80-94.

[22] Ezra Cooper et al., [**Links : Web Programming Without Tiers**], *Proceedings of 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006), Amsterdam, The Netherlands*, Springer Berlin Heidelberg, nov. 2006, 266-296.

[23] Thierry Coquand et Christine Paulin, [**Inductively Defined Types**], *Proceedings of the International Conference on Computer Logic*, COLOG-88, Tallinn, USSR : Springer-Verlag New York, Inc., 1990, 50-66.

[24] Evan Czaplicki et Stephen Chong, [**Asynchronous Functional Reactive Programming for GUIs**], *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, Seattle, Washington, USA : ACM, 2013, 411-422.

[25] ECMA International, [**Standard ECMA-262 - ECMAScript Language Specification**], 8e éd., juin 2017.

[26] Stephen A. Edwards, [**An Esterel compiler for large control-dominated systems**], *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.2 fév. 2002, 169-183.

[27] Stephen A. Edwards et Jia Zeng, [**Code Generation in the Columbia Esterel Compiler**], *The European Association for Signal Processing (EURASIP), Journal on Embedded Systems* 1 fév. 2007, 052651.

[28] Rémy El Sibaïe et Emmanuel Chailloux, [**Pendulum : une extension réactive pour la programmation Web en OCaml**], *Actes des Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA)*, Saint-Malo, France, jan. 2016.

[29] Rémy El Sibaïe et Emmanuel Chailloux, [**Synchronous Web Programming**], *Proceedings of the International Workshop on Reactive and Event-Based Languages and Systems (REBLS)*, Amsterdam, Netherlands, oct. 2016.

[30] Rémy El Sibaïe et Jean-Christophe Filliâtre, [**combine : une bibliothèque OCaml pour la combinatoire**], *Actes des Vingt-cinquièmes Journées Francophones des Langages Applicatifs (JFLA)*, Aussois, France, fév. 2013.

[31] Conal Elliott et Paul Hudak, [**Functional Reactive Animation**], *Proceedings of International Conference on Functional Programming (ICFP'97)*, Amsterdam, The Netherlands, juin 1997, 263-273.

[32] Esterel Technologies, [**Scade Language Primer**], rapp. tech., 2014.

[33] Steve Faulkner et al., [**HTML5**], W3C Recommendation, http ://www.w3.org/TR/2014/REC-html5-20141028/, W3C, oct. 2014.

[34] Jean-Christophe Filliâtre et Sylvain Conchon, [**Type-safe modular hash-consing**], *Proceedings of the 2006 workshop on ML*, ACM Press, 2006, 12-19.

[35] Philippa A. Gardner et al., [**Local Hoare Reasoning About DOM**], *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, Vancouver, Canada : ACM, 2008, 261-270.

[36] Jacques Garrigue, [**Relaxing the Value Restriction**], *Proceedings of the International Symposium On Functional And Logic Programming, Nara*, Springer-Verlag, 2003, 196-213.

[37] Thierry Gautier et Paul Le Guernic, [**SIGNAL : A declarative language for synchronous programming of real-time systems**], *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture (FPCA'87), Portland*, sept. 1987, 257-277.

[38] Aryeh Gregor et al., [**W3C DOM4**], W3C Recommendation, http ://www.w3.org/TR/2015/REC-dom-20151119/, W3C, nov. 2015.

[39] DOM Working Group, [**Document Object Model (DOM) Level 2 Events Specification**], W3C Recommendation, W3C, nov. 2000.

[40] Arjun Guha, Claudiu Saftoiu et Shriram Krishnamurthi, [**The Essence of JavaScript**], *CoRR* abs/1510.00925 2015.

[41] Nicolas Halbwachs et Pascal Raymond, [**Validation of Synchronous Reactive Systems : From Formal Verification to Automatic Testing**], *Proceedings of the 5th Asian Computing Science Conference (ASIAN'99)*, Phuket, Thailand : Springer Berlin Heidelberg, 1999, 1-12.

[42] Gilles Kahn, [**The Semantics of Simple Language for Parallel Programming**], *Proceedings of the International Federation for Information Processing (IFIP) Congress*, 1974, 471-475.

[43] Oleg Kiselyov, [**The Design and Implementation of BER MetaOCaml**], *Proceedings of the International Conference on Functional and Logic Programming (FLOPS) 2014*, sous la dir. de Michael Codish et Eijiro Sumii, Kanazawa, Japan : Springer International Publishing, juin 2014, 86-102.

[44] Xavier Leroy et al., [**The OCaml System Release 4.05 : Documentation And User's Manual**], rapp. tech., Inria, juil. 2017.

[45] Sergio Maffeis, John C. Mitchell et Ankur Taly, [**An Operational Semantics for JavaScript**], *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, 2008.

[46] Ingo Maier et Martin Odersky, [**Deprecating the Observer Pattern with Scala.React**], rapp. tech. EPFL-REPORT-176887, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2012.

[47] Louis Mandel, [**Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive**], Thèse de doctorat, Université Pierre et Marie Curie, Paris, France, 2006.

[48] Leo A. MEYEROVICH et al., [**Flapjax : A Programming Language for Ajax Applications**], *SIGPLAN Not.* 44.10 oct. 2009.

[49] Robin MILNER, Mads TOFTE et David MACQUEEN, [**The Definition of Standard ML**], MIT Press, 1997.

[50] Andrei POPESCU, [**W3C Geolocation**], W3C Recommendation, https ://www.w3.org/TR/geolocation-API/, W3C, nov. 2017.

[51] Dumitru POTOP-BUTUCARU, [**Compiling Esterel**], Springer, 2007.

[52] Gabriel RADANNE, [**Tierless Web programming in ML**], Thèse de doctorat, Université Paris-Diderot, 2017.

[53] Gabriel RADANNE, Jérôme VOUILLON et Vincent BALAT, [**Eliom : A core ML language for Tierless Web programming**], *Proceedings of The Asian Symposium on Programming Languages and Systems (APLAS)*, Hanoi, Vietnam, nov. 2016.

[54] Peter Van ROY, [**Programming Paradigms for Dummies : What Every Programmer Should Know**], *New Computational Paradigms for Computer Music*, IRCAM/Delatour, 2009.

[55] Bernard P. SERPETTE, Pascal MANOURY et Emmanuel CHAILLOUX, [**Unification des couleurs dans un λ-calcul polychrome**], *Actes des Vingt-cinquièmes Journées Francophones des Langages Applicatifs, Fréjus (JFLA)*, jan. 2014, 65-76.

[56] Manuel SERRANO, [**HOP : an environment for developing web 2.0 applications**], *International Lisp Conference, ILC 2007, Cambridge, Apr 1-4, 2007*, 2007, 6.

[57] Manuel SERRANO et Vincent PRUNET, [**A Glimpse of Hopjs**], *Proceedings of the International Conference on Functional Programming (ICFP'16)*, ACM, Nara, Japan, sept. 2016.

[58] Jeremy G. SIEK, [**Gradual Typing for Functional Languages**], *In Scheme and Functional Programming Workshop*, 2006, 81-92.

[59] Steven VAROUMAS, Benoît VAUGON et Emmanuel CHAILLOUX, [**Concurrent Programming of Microcontrollers, a Virtual Machine Approach**], *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, jan. 2016, 711-720.

[60] Jérôme VOUILLON, [**Lwt : a Cooperative Thread Library**], *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, 2008, 3-12.

[61] Jérôme VOUILLON et Vincent BALAT, [**From Bytecode to JavaScript : the Js_of_ocaml Compiler.**] *Softw., Pract. Exper.* 2014.

Ressources Web

[62] [**AngularJS**], https://angular.io/, Google, Inc.

[63] [**backbone.js**], http://backbonejs.org/.

[64] [**BuckleScript**], https://bucklescript.github.io/, Bloomberg.

[65] Daniel Bunzli, [**React**], http://erratique.ch/logiciel/react.

[66] Evan Czaplicki, [**Blazing Fast HTML**], http://elm-lang.org/blog/blazing-fast-html, 2014.

[67] Evan Czaplicki, [**Elm**], http://elm-lang.org/.

[68] [**Ember.js**], https://emberjs.com.

[69] Matt Esch, [**A Virtual DOM and diffing algorithm**], https://github.com/Matt-Esch/virtual-dom, 2014.

[70] [**Flow**], http://flowtype.org, Facebook.

[71] JQuery Foundation, [**JQuery**], https://jquery.com/.

[72] [**Google Maps Javascript API**], https://developers.google.com/maps/documentation/javascript/, Google, Inc.

[73] Matt Esch Jake Verbaten, [**Mercury performances benchmarks**], https://github.com/Raynos/mercury-perf, 2014.

[74] [**JavaScriptCore**], https://developer.apple.com/reference/javascriptcore, Apple.

[75] Inc Joyent, [**Node.js**], http://nodejs.org.

[76] Microsoft, [**ChakraCore**], https://github.com/Microsoft/ChakraCore/.

[77] Mozilla Developer Network, [**SpiderMonkey**], https://developer.mozilla.org/fr/docs/SpiderMonkey.

[78] [**ocaml-googlemaps**], https://github.com/besport/ocaml-googlemaps, BeSport, Inc.

[79] Ocsigen, [**js_of_ocaml manual**], http ://ocsigen.org/js_of_ocaml/manual/, 2015.

[80] [**Ocsigen Start**], https://github.com/ocsigen/ocsigen-start, Ocsigen.

[81] [**PgOCaml**], http://pgocaml.forge.ocamlcore.org/, Dario Texeira.

[82] [**React**], http://facebook.github.io/react, Facebook.

[83] [**ReasonML**], https://reasonml.github.io/, Facebook.

[84] [**Redux**], https://redux.js.org/, 2014.

[85] [**scala.js**], https://www.scala-js.org/.

[86] TasteJS, [**TodoMVC**], http://todomvc.com/, 2014.

[87] [**Tiobe Index**], https://www.tiobe.com/tiobe-index/.

[88] [**Typescript**], https://www.typescriptlang.org/.

[89] [**V8**], https://developers.google.com/v8/, Google, Inc.

[90] Jake Verbaten, [**Mercury, A truly modular frontend framework**], https://github.com/Raynos/mercury, 2014.

[91] Whitequark, [**Deriving**], https://github.com/whitequark/ppx_deriving.

[92] Xavier Leroy, [**Module Parsetree**], https ://github.com/ocaml/ocaml/blob/trunk/parsing/parsetree.mli, version 4.05.0.

Table des figures

Liste des acronymes

API	Application Programming Interface
AST	Abstract Syntax Tree
CEC	Columbia Esterel Compiler
CFG	Control Flow Graph
CSS	Cascaded Stylesheets
DOM	Document Object Model
DSL	Domain Specific Language
ECMA	European Computer Manufacturers Association
FRP	Functional Reactive Programming
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfert Protocol
MVC	Model View Controlleur
PHP	PHP : Hypertext Preprocessor
SQL	Structured Query Language
URI	Uniform Resource Identifier
W3C	Worldwide Web Consortium
XML	Extensible Markup Language

Annexe A

Syntaxe complète de Pendulum

prog : := *header* stmt*	*programme*
header : := (**input** | **output**) *ident ocaml-expr* ;	*arguments*
| **element** *ident gather* ? ;	*élément du DOM*
stmt : := **emit** *ident ocaml-expr*	*émission d'un signal valué*
| **emit** *ident*##.*property ocaml-expr*	*émission sur une propriété*
| **nothing**	*ne rien faire*
| **pause**	*attendre l'instant suivant*
| **present** *test stmt stmt*	*branchement conditionnel*
| **loop** *stmt*	*boucle infinie*
| **exit** *label*	*échappement de bloc*
| *stmt* **||** *stmt*	*parallèle*
| *stmt* **;** *stmt*	*séquence*
| **let** *ident* **=** *ocaml-expr* **in** *stmt*	*signal local*
| **trap** *ident stmt*	*bloc interruptible*
| **!** *ocaml-expr*	*expression hôte atomique et instantanée*
| **suspend** *test stmt*	*bloc suspendu*
test : := *ident*	*expression de test de présence d'un signal*
| *ident*##*event*	*expression de test de présence d'un signal évènementiel*
gather : :=	*agrégation de valeurs des signaux*
| **{** (*event* **=** *ocaml-expr*, *ocaml-expr* **;**)+**}** ;	

Annexe B

Code de l'application Cartes

```
let%client onmapsload f =
  let callback = Js.wrap_callback f in
  let callback_name = "googlemapsloaded" in
  Js.export callback_name callback ;
  let script = Dom_html.createScript Dom_html.document in
  script##.defer := Js._true ;
  script##.async := Js._true ;
  script##._type := Js.string "text/javascript";
  let key = !Pdldemo_config.googlemaps_api_key in
  print_endline key;
  script##.src :=
    Js.string @@
    "https://maps.googleapis.com/maps/api/js?\
     key=" ^ key ^ "\
             &libraries=geometry,drawing,places\
             &callback=" ^ callback_name ;
  Dom.appendChild Dom_html.document##.body script

let%client start_watching_loc each_location =
  if Geolocation.is_supported() then begin
    let geo = Geolocation.geolocation in
    let options = Geolocation.empty_position_options () in
    options##.enableHighAccuracy := true;
    let f_error e = Firebug.console##debug e in
    let f_success e = each_location (e##.coords##.latitude, e##.coords##.longitude) in
    geo##getCurrentPosition (Js.wrap_callback f_success) (Js.wrap_callback f_error) options;
   ignore @@ geo##watchPosition (Js.wrap_callback f_success) (Js.wrap_callback f_error) options;
  end


  let%client _ : int = onmapsload


[%%client
  open Googlemaps

(*
```

```
48.8455669,2.3541834
*)

  let create_mymarker map (lat,lng) =
    let position = LatLng.new_lat_lng ~lat ~lng in
    let opts = MarkerOptions.create
        ~position
        ~title:("Mon point")
        ~draggable:false
        ~map:map () in
    Marker.new_marker ~opts ()

  let create_map (lat,lng) map =
    let opts = MapOptions.create ~zoom:15 () in
    let center = LatLng.new_lat_lng
        ~lat ~lng
    in
    let m = Map.new_map (To_dom.of_div map) ~opts () in
    Map.set_center m center;
    m

  let update_location map marker (lat,lng) =
    Firebug.console##debug (lat,lng);
    let center = LatLng.new_lat_lng
        ~lat ~lng
    in
    Marker.set_position marker center;
    Map.set_center map center

  let init_searchbox elt f =
    let inp = To_dom.of_element elt in
    let sb = SearchBox.new_search_box inp () in
    ignore @@ Event.add_listener (SearchBox.t_to_js sb) "places_changed" (fun _ ->
      f @@ SearchBox.get_places sb
      );
    sb

  let remove_markers ms =
    List.iter (fun m -> Marker.set_map m None) ms

  let generate_markers map places =
    let bds, markers = List.fold_left (fun (b, mrks) p ->
      let position = PlaceGeometry.location @@ PlaceResult.geometry p in
      let bounds = match b with
      | None -> Some (LatLngBounds.new_lat_lng_bounds ~sw:position ~ne:position)
      | Some b -> Some (LatLngBounds.extend b position)
      in
      let opts = MarkerOptions.create
          ~position
          ~title:(PlaceResult.name p)
          ~map ()
      in bounds, Marker.new_marker ~opts () :: mrks
```

```
    ) (None, []) places
    in
    Eliom_lib.Option.iter (Map.fit_bounds map) bds;
    Map.set_zoom map (min (Map.get_zoom map) 16);
    markers


]

type%client page_load = {
  map : Html_types.div Eliom_content.Html.D.elt;
  search_input : Html_types.input Eliom_content.Html.D.elt;
  locate_btn : Html_types.button Eliom_content.Html.D.elt;
}



  let%sync p ~animate =
    input (page_loading : page_load);

    let location = 0.,0. in
    let maps_loading = () in
    begin
      !(onmapsload (fun () ->
        Pendulum.Signal.set_present_value maps_loading ()
      ; animate () ))
    ; !(start_watching_loc (fun loc ->
        Pendulum.Signal.set_present_value location loc
      ; animate ()))
    ; !()
    ; (await page_loading || await maps_loading || await location)
    ; !(print_endline "Maps loaded")
    ; loop ( trap reload (
      let map = create_map !!location (!!page_loading).map in
      let mymarker = create_mymarker !!map !!location in
      let places_changed = [] in
      let searchbox = init_searchbox (!!page_loading).search_input
          (fun places ->
             Pendulum.Signal.set_present_value places_changed places ;
             animate ()) in
      let locate = () in
      let searching = () in
      let markers = [] in
      let locate = () in
         loop ( present page_loading ( exit reload ))
      || loop ( present places_changed (
                !(remove_markers !!markers);
                emit markers (generate_markers !!map !!places_changed)))
      || loop ( present places_changed (
                trap t (loop (present locate (exit t); emit searching;))))
      || loop ( present location (
                present searching nothing
```

```
                !(update_location !!map !!mymarker !!location)
    ))
  ))


[%%client

  let%sync p ~animate =
    input (page_loading : page_load);

    let location = 0.,0. in
    let maps_loading = () in

    begin
      !(onmapsload (fun () ->
        Pendulum.Signal.set_present_value maps_loading ()
      ; animate () ))
    ; !(start_watching_loc (fun loc ->
        Pendulum.Signal.set_present_value location loc
      ; animate ()))
    ; !()

    ; (await page_loading || await maps_loading || await location)
    ; !(print_endline "Maps loaded")
    ; let map = create_map !!location (!!page_loading).map in
      let mymarker = create_mymarker !!map !!location in
      let places_changed = [] in
      let searchbox = init_searchbox (!!page_loading).search_input (fun places ->
        Pendulum.Signal.set_present_value places_changed places ;
        animate ())
      in
      let locate = () in
      !((To_dom.of_button (!!page_loading).locate_btn)##.onclick :=
          Dom_html.handler (fun _ ->
            Pendulum.Signal.set_present_value locate ();
            animate (); Js._true
          ));
      loop ( present page_loading (
        !((To_dom.of_button (!!page_loading).locate_btn)##.onclick :=
            Dom_html.handler (fun _ ->
              Pendulum.Signal.set_present_value locate ();
              animate (); Js._true
            ));
        emit map (create_map !!location (!!page_loading).map));
        emit mymarker (create_mymarker !!map !!location);
        emit searchbox (init_searchbox (!!page_loading).search_input (fun places ->
          Pendulum.Signal.set_present_value places_changed places;
          animate ()))
      )
      ||
      let searching = () in
      let markers = [] in
```

```
      loop ( present places_changed (
        !(remove_markers !!markers);
        emit markers (generate_markers !!map !!places_changed)))
      ||
      loop ( present places_changed (
        trap t (loop (present locate (exit t); emit searching;))))
      ||
      loop ( present location (
        present searching nothing (
          !(update_location !!map !!mymarker !!location)
        )
      ))
      ||
      loop ( present locate ( emit location !!location); pause)
    end

]



let%client p =
  p#create ({map = div []; search_input = input ();
             locate_btn = button []})



(* Page for this demo *)
let%shared page () =

  let search_input =
    Eliom_content.Html.D.input
      ~a:[a_class ["pac-input"];
          a_input_type `Text; a_placeholder "Search..."] ()
  in

  let map = Eliom_content.Html.D.div ~a:[a_id "map"; a_class ["map"]] [] in
  let locate_btn = button [pcdata "Locate"] in

  let marker_text_input = input ~a:[ a_input_type `Text ] () in
  let marker_btn_input =
    button ~a:[ a_class [ "marker-btn-input" ]][ pcdata "Search" ]
  in
  let marker_btn_cancel =
    button ~a:[ a_class [ "marker-btn-cancel" ]][ pcdata "Search" ]
  in
  let marker_input =
    div ~a:[ a_id "marker-input"; a_class [ "marker-input" ] ]
      [ marker_text_input
      ; marker_btn_input
      ; marker_btn_cancel
      ]
  in
```

```
let page = div
    [ marker_input
    ; div [search_input; locate_btn]
    ; map
    ]
in

let _ : unit Eliom_client_value.t = [%client
  Lwt_js_events.async (fun () ->
    p#page_loading ({map = ~%map; search_input = ~%search_input;
                     locate_btn = ~%locate_btn });
    p#react;
    Lwt.return ()
  )

] in
Lwt.return [ page ]
```

Annexe C

Code de l'application Lecteur media

```
open Firebug

(* Jsoo boilerplate code *)

module Dom_html = struct
  include Dom_html
  open Js
  class type _mediaElement = object
    inherit mediaElement
    method onprogress : (_mediaElement t, mouseEvent t) event_listener writeonly_prop
    method ontimeupdate : (_mediaElement t, mouseEvent t) event_listener writeonly_prop
    method onplay : (_mediaElement t, mouseEvent t) event_listener writeonly_prop
    method onpause : (_mediaElement t, mouseEvent t) event_listener writeonly_prop
    method onloadeddata : (_mediaElement t, mouseEvent t) event_listener writeonly_prop
  end
  module Coerce = struct
    include CoerceTo
    let unsafeCoerce tag (e : #element t) = Js.some (Js.Unsafe.coerce e)
    let media :  #element t -> _mediaElement t opt = fun e -> unsafeCoerce "media" e
  end
end

let error f = Printf.ksprintf
    (fun s -> Firebug.console##error (Js.string s); failwith s) f
let debug f = Printf.ksprintf
    (fun s -> Firebug.console##log(Js.string s)) f
let alert f = Printf.ksprintf
    (fun s -> Dom_html.window##alert(Js.string s); failwith s) f

let (@>) s coerce =
  Js.Opt.get (coerce @@ Dom_html.getElementById s)
    (fun () -> error "can't find element %s" s)

let str s = Js.some @@ Js.string s

(* ======================== *)
```

```
let ftime_to_min_sec t =
  let sec = int_of_float t in
  let min = sec / 60 in
  let sec = sec mod 60 in
  (min, sec)

let max_slide = 1000.

let set_visible b elt =
  let visibility = if b then "visible" else "hidden"
  in elt##.style##.visibility := Js.string visibility

(* Updates the progress bar value proportionnaly to current time *)
let update_slider slider media =
  slider##.value := Js.string @@ Format.sprintf "%0.f" (
      if media##.duration = 0. then 0.
      else media##.currentTime /. media##.duration *. max_slide)


(* Updates the time of the current position of the cursor over it *)
let update_slider_value slider_value media slider =
  let min, sec = ftime_to_min_sec @@
    ((Js.parseFloat slider##.value) /. max_slide *. media##.duration)
  in
  set_visible true slider_value;
  let padding =
    Format.sprintf "%0.fpx" (Js.parseFloat slider##.value /. max_slide *.
                             (float_of_int slider##.scrollWidth))
  in
  slider_value##.style##.marginLeft := Js.string padding;
  slider_value##.textContent := Js.some @@ Js.string @@ Format.sprintf "%2dmin%2ds" min sec


(* Sets the current time of the media tag in proportion *)
let update_media media slider =
  media##.currentTime := (Js.parseFloat slider##.value) /. max_slide *. media##.duration


(* Apply the state switching to the video by calling pause/play action *)
let update_state state media button =
  if state then media##play else media##pause


(* Update the displayed current time *)
let update_time_a media time_a =
  let cmin, csec = ftime_to_min_sec media##.currentTime in
  let tmin, tsec = ftime_to_min_sec media##.duration in
  time_a##.textContent := str @@ Format.sprintf "%2d:%0d / %0d:%0d" cmin csec tmin tsec

(* Switch the text on the button *)
```

```
let update_content elt b =
  elt##.textContent := Js.some @@ Js.string @@ if b then "Pause" else "Play"


open Dom_html

(* The reactive program of the player *)
let%sync reactive_player ~animate ~obj =
  element (play_pause : buttonElement Js.t);
  element (progress_bar : inputElement Js.t);
  element (media : _mediaElement Js.t);
  element (time_a : anchorElement Js.t);
  element (slider_value : anchorElement Js.t);

  let seeking = () in
 let state = Js.to_bool media##.autoplay in (* carry the state of the video (true if playing) *)

  !(update_content play_pause (pre state)); (* update the button at start-up *)

  loop ( (* handle the different possibles actions  : *)
       present play_pause##onclick (emit state (not !!state))
    || present state !(update_state !!state media play_pause)
    || present state !(update_content play_pause !!state)
  ; pause)

  || loop ( (* while the user moves the cursor, display the time over it *)
    present progress_bar##oninput
      !(update_slider_value slider_value media progress_bar);
    pause
  )

  || loop (
    await progress_bar##onmousedown;            (* When mouse button is down *)
    trap t' (                                   (* open an escape block with exception t' *)
      loop (                                        (* every instant,  *)
        emit seeking;                             (* emits the blocking signal *)
        present progress_bar##onmouseup (             (* if the button is released*)
         !(set_visible false slider_value;     (* stop displaying the time over the cursor *)
            update_media media progress_bar);       (* set the current time of the video *)
        exit t'                             (* leave the escape block and stop this behavior *)
          ); pause)); pause)

  || loop (
    present media##ontimeupdate (               (* everying instants the video updates *)
     present seeking nothing                (* if the blocking signal is present, do nothing *)
      !(update_slider progress_bar media)     (* else, update the progress bar with the current time of the video*)
       ||
      !(update_time_a media time_a)           (* update the display of the current time *)
    ); pause)
```

```
let main _ =
  let open Dom_html in
  let play_button = "reactiveplayer_play" @> Coerce.button in
  let progress_bar = "reactiveplayer_progress" @> Coerce.input in
  let media = "reactiveplayer_media" @> Coerce.media in
  let time = "reactiveplayer_timetxt" @> Coerce.a in
  let range_value = "reactiveplayer_range_value" @> Coerce.a in

  (* Initialize the player program, with the js elements as inputs
     setters function have '_' to avoid the unused warning, as long
     as we don't use them :
     Signals and _react function are implicitely triggered by the generated code
     of the program
  *)
  reactive_player#create (play_button, progress_bar, media, time, range_value);
  Js._false


let () = Dom_html.(window##.onload := handler main)
```

Résumé

Le but de cette thèse est d'apporter de nouvelles possibilités au domaine de la programmation Web, dont les technologies répandues ne capturent pas toutes les problématiques engendrées par les interactions dans une application. Notre solution est un langage, Pendulum, inspiré de la programmation synchrone réactive en Esterel et se présentant comme une extension à OCaml. Il permet de gagner en sûreté et en expressivité en particulier dans la gestion d'interaction multiples.

Dans une première partie, nous présentons notre perception de la programmation Web d'aujourd'hui en partant du standard pour aller vers les technologies plus modernes qui tentent de subvenir aux besoins des programmes par d'autres approches, notamment la programmation multitier et les modèles de concurrence en flot de données. Dans une seconde partie, nous introduisons le langage Pendulum et ses constructions, ce qu'il propose comme interopérabilité avec le client Web le différenciant d'autres langages synchrones, et l'interface de programmation qui le connecte avec le langage hôte.

Dans les parties trois et quatre, nous présentons la méthode de compilation utilisée, GRC, pour GraphCode, qui produit un graphe de flot de contrôle à partir du programme synchrone source. On revient sur la structure du GRC, les règles permettant de le construire, ainsi que notre méthode d'ordonnancement statique. Nous décrivons ensuite la génération de l'environnement d'exécution d'un programme synchrone dans le programme hôte.

Dans une cinquième partie, nous montrons l'intérêt de la programmation synchrone dans le client et en quoi son modèle d'exécution s'adapte naturellement à celui du navigateur Web. Nous montrons qu'il est possible de profiter de cet avantage pour réagir aux évènements plus efficacement sans efforts d'optimisation.

Avant de conclure, nous présentons de multiples exemples implémentés en Pendulum pour mettre en avant les qualités d'expressivité et de sûreté de la programmation synchrone sur différentes problématiques impliquant du multimédia et des interactions.

Abstract

The goal of this thesis is to bring new capabilities to Web programming, whose languages, frameworks don't handle all the problematics raised by interactions in a Web application. Our solution is a programming language, Pendulum, taking its roots in synchronous reactive model *à la* Esterel. It brings safety and expressiveness, especially when handling multiple interactions.

In the first chapter, we give our point of view on what is Web programming today, from the standard to the newest frameworks trying to fill programers needs by other approaches, like multitier programming or dataflow programming. In the second chapter, we introduce Pendulum and its instructions, its interface with the host language, and what it brings to both synchronous and Web programming.

In the third and fourth chapter, we present the compilation method, GRC a.k.a GraphCode, that produces a control flow graph from the source code. In the first part, we insist mainly on GRC structure, the rules describing its creation and our technic to linearize parallel branches. Then, we describe the how to initialize synchronous execution environment in OCaml.

In the fifth chapter, we show why it is a benefit to use synchronous programming in client programming and how its execution model can natively match the Web browser execution model. We use those ideas to show how a synchronous program can be fast to handle events without optimisation attempt.

Before we conclude, we detail several examples implemented with our language to show how expressive and safe synchronous programming can be on diverse programs, implying multimedia and interactions.