Utilisation maximale des cores : expérimentation avec FunLoft

Pejman Attar,
sous la direction de Frédéric Boussinot,
INRIA Sophia-Antipolis

Rapport de stage de Master 2 (MPRI)

Avril - Août 2010

Fiche de synthèse

Le contexte général

FunLoft est un langage expérimental pour la concurrence, développé à l'origine par Boussinot. FunLoft a une sémantique simple et bénéficie d'un compilateur qui contrôle les ressources (CPU et mémoire), ce qui permet d'interdire des programmes qui peuvent créer des problèmes comme les fuites de mémoire. L'aspect important de FunLoft est le bénéfice d'un vrai parallélisme dans les machines multicores et l'utilisation massive des threads coopératifs.

L'utilisation des ressources est l'un des problèmes majeurs des langages concurrents. Il existe plusieurs approches :

- l'approche des langages standards, comme C, qui utilisent des threads natifs et permettent au système de gérer les threads de façon optimale
- l'approche plus fine qui consiste à gérer l'utilisation des CPU(cores) par un mécanisme interne.

FunLoft appartient à la deuxième catégorie, il essaie d'avoir un contrôle plus fin sur les cores disponibles en créant des schedulers statiques au début de chaque programme (en utilisant différentes bibliothèques on peut avoir ce contrôle tant bien que mal).

Le problème étudié

FunLoft est à l'heure actuelle un prototype opérationnel. Avant mon stage, Boussinot avait déjà réalisé quelques benchmarks partiels pour avoir une idée sur l'efficacité de son implementation[1] et continue l'evalution de son langage sur ses impressions obtenues à partir de ses résultats. Cependant dans FunLoft il y a certains points qui manquent.

Plus précisément, dans le cas de définition statique des schedulers même si le programme qu'on écrit pour une machine est optimal du point de vue de l'utilisation des ressources, en changeant le nombre de cores le programme risque de ne plus être optimal. Comme on l'a précisé auparavant, il existe différentes approches pour résoudre ce problème. Parmi ces solutions, une solution est de laisser le système tout gérer ce qui n'est pas toujours convenable, ou bien avoir un contrôle sur le système à l'aide des bibliothèques de threads coopératifs. Le but de notre travail est de trouver un modèle simple et adapté afin de le réaliser dans un cadre plus général.

La contribution proposée

Le travail décrit se compose en de deux parties distinctes : premièrement créer un modèle proche de FunLoft. Deuxièment vérifier des propriétés demandées pour ce langage à l'aide de sémantiques et de typages. J'ai aussi

prouvé la correction du typage et que la sémantique admet un point fixe(à chaque instant).

DSLM : FunLoft dispose déjà d'une sémantique simple mais comme je l'ai précisé auparavant, dans son état actuel ne permet pas d'avoir des programmes portables qui utilisent le maximum de ressources. C'est la raison pour laquelle on a essayé d'avoir un autre modèle (plus proche de SugarCubes[2, 3]) qui nous permettra d'obtenir le résultat recherché.

Preuve de Correction : la sémantique de DSLM permet de vérifier certaines propriétés comme l'absence de boucle instantanée mais cela n'est pas suffisant. On a besoin de vérifier l'absence de data-race à l'aide d'un système de type.

Les arguments en faveur de sa validité

En ce qui concerne la partie sémantique, l'expressivité du langage conçu et la simplicité de la sémantique naïve adoptée nous montrent sa généralité. Grâce à cette sémantique les preuves de correction de typage et aussi d'existence de point fixe sont simplifiées.

Cette sémantique simple a pu assurer, en plus de l'expressivité de FunLoft, aussi bien les propriétés qu'on cherchait au niveau du contrôle de ressources que l'utilisation des ressources (cores) disponibles.

Le bilan et les perspectives

J'ai prouvé que le typage effectué pour DSLM est correct et que la sémantique qu'on a créée admet un point fixe au cours de chaque instant. On essaiera de montrer toutes ces propriétés à l'aide de Coq de façon plus formelle. Un article sur la partie théorique est en cours de rédaction.

Il faudra tout d'abord implémenter le langage DSLM en FunLoft, le tester et le comparer avec d'autres langages concurrents comme CPC. Pour cela il faut faire des implémentations en FunLoft et puis s'assurer que ces implémentations vérifient les propriétés annoncées.

L'autre aspect important est la sécurité. Par exemple, quand on a des liens sur une donnée confidentielle, ne pas la réveler aux autres participants et trouver la bonne approche pour faire ces vérifications sur nos programmes.

Enfin, FunLoft ne fournit pour l'instant aucun mécanisme pour interagir avec d'autres langages Ce qui peut être très intéressant à implémenter dans le cas d'un système distribué et permettre la communication entre 2 systèmes. Le travail de Melliar-Smith sur le sujet peut être un bon point de départ.

Rapport de stage

1 contexte

1.1 Introduction

La programmation synchrone simplifie la concurrence, comparée à l'approche standard basée exclusivement sur des threads préemptifs (pthread[4] ou threads de Java). Cette simplification est le résultat d'une sémantique plus simple et propre, dans laquelle le nombre des entrelacements des calculs parallèles est réduit. Cependant, les langages synchrones standards introduisent des nouvelles problématiques (non-terminasion des instants) ainsi que des problèmes majeurs face à la création dynamique (de threads, composants, signaux, etc.). En general, ils ne sont pas en mesure d'utiliser pleinement le vrai parallélisme, comme celui fournit par les machines multicores. Donc on propose un modèles synchrone qui résoud les problèmes précèdents de la façon suivante :

On utilise la variante réactive de l'approche synchrone, qui est capable d'exprimer la création dynamique. Fondamentalement, dans ce modèle "les cycles de causalité" sont éliminés par construction, en interdisant la réaction immédiate à l'absence. Plusieurs modèles liés à cette approche sont basés sur cette variante (e.g. Reactive-C[5], SugarCube[2], ReactiveML[6]).

On introduit une notion de *site*, pour combiner l'aspect synchrone avec l'asynchrone des systèmes et pour les composer dans le même cadre. Cette combinaison est inspirée par le modèle des FairThreads[7] qui mixe les threads coopératifs et les threads préemptifs. Les sites peuvent être exécutés par un ou plusieurs cores, ce qui nous donne la possibilité d'exploiter pleinement les machines multicores. Chaque site contient un ensemble d'agents qui sont exécutés de façon synchrone (ils partagent la même notion de temps).

On introduit un concept d'agent, dans lequel un agent peut transporter son état d'un site à un autre en gardant sa mémoire intacte. Chaque agent contient un script (sous forme de mise en parallèle d'un ensemble de scripts).

On type tous nos programmes pour s'assurer qu'il n'y a pas de data-race entre deux agents différents. Donc deux entités parallèles n'appartenant pas au même agent n'ont pas d'interference. Cette absence d'interference est basée fortement sur la proprieté de séparation de mémoire qui est verifiée d'une façon statique dans notre modèle.

On va traduire tous nos programme en FunLoft pour bénéficier des fortes vérifications fournies par ce langage. En compilant un programme en FunLoft, on s'assure qu'il n'y a pas de problème des langages synchrone (les

boucles instantanées et la non-terminaison d'un instant).

L'approche qu'on propose est une alternative à l'utilisation des locks pour la protection de mémoire. Donc la programmation est simplifiée. En comparant avec le modèle standard, basé sur utilisation des locks dans le contexte préemptif. Le code suivant qui est considéré par le programmeur comme un programme atomique(e.g. the "swap" code temp=x ;x=y ;y=temp ;) reste atomique dans notre approche, et sans possibilité d'interference, dans tous les contextes, en particulier dans le cas parallèle. On peut dire que notre approche rend à la fois la sémantique et la programmation plus simples.

Durant mon stage j'ai créé un modèle proche des langages de scripts qui est appelé DSLM(Dynamic Script Language with Memory), basé sur cette approche. On va d'abord regarder l'historique des langages concurents et l'utilisation des ressources, puis une description informelle de notre modèle. Par la suite, on va voir la sémantique, l'implementation, le système de type, les preuves, et pour finir le bilan et perspective.

1.2 Historique de controverse

Dès le début de l'informatique le parallélisme a été pris en compte dans les systèmes et les langages à commencer par C. Cependant tous les problèmes concernant le parallélisme ont été poussés vers les systémes d'exploitation et donc on demandait aux experts de ce domaine de résoudre ces problèmes. En commercialisant des machines multi-cores pour l'utilisation publique, le problème ne concerne plus seulement les spécialistes mais aussi les programmeurs.

Pour les programmes concurrents, il y a eu plusieurs approches differentes. Le premier consiste à utiliser des threads préemptivfs. Le problème d'utilisation des threads préemptifs est qu'il y a une limitation du nombre de threads par systèmes ce qui nous empêche d'avoir des composants parallèles en nombre important. De plus, pour pouvoir programmer correctement dans ce cas, il faut utiliser des locks ce qui rend la programmation désagréable ce qui est le cas pour certains langages comme C avec pthread, ou bien Java.

La deuxième approche possible, est l'utilisation des threads cooperatifs en utilisant des bibliothèques appropriées. Cela nous permet de pouvoir utiliser un nombre important de composants parallèles. La plupart des langages qui utilisent cette approche laissent le système gérer l'utilisation des ressources comme CPC[8, 9]. D'autres langages comme FunLoft[10] qui s'interessent à ce probléme et essaie d'avoir un contrôle plus fin. C'est sur ce type de langage que notre modèle est basé.

Il existe d'autres approches comme celles des langages synchrone dont

on a parlé au début, ou bien encore, d'autres langages comme ULM de Boudol [11] qui mélange plusieurs approches differentes. Une partie de notre modèle est inspirée de ce modèle et aussi du modèle d'Abadi[12].

2 FunLoft

2.1 Principes

FunLoft[10] est un langage expérimental pour la concurrence, développé à l'origine par Boussinot. L'objectif de FunLoft est :

- d'avoir un sémantique simple et claire qui permet d'écrire des programmes concurrents d'une façon simple.
- fournir un langage sûr dans lequel, par exemple, les data-races sont impossibles
- contrôler les ressources (CPU et mémoire) ; par exemple des fuites de mémoire ne peuvent pas être produites dans un programme FunLoft
- FunLoft a une implémentation efficace et peut gérer un nombre important de composants concurrents.
- de bénéficier d'un vrai parallélisme offert par les machines multicores.

FunLoft est issu de l'expérience de Boussinot en tant que programmeur. En écrivant des programmes dans les différents langages asynchrones avec parallélisme. il a constaté qu'on peut obtenir des résultats indésirables et qu'on n'arrive pas à contrôler nos programmes de façon efficace[13, 14]. Donc, il propose FunLoft en tant qu'un langage proche de ML qui offre la possibilité d'utiliser le parallélisme de façon simple, efficace et sûre.

Nous pensons que FunLoft est efficace et simple à utiliser d'une part parce qu'on arrive à utiliser un nombre important de composants concurrents et d'autre part par ce que tous les programmes écrits et acceptés par FunLoft sont sûr ce qui signifie qu'on ne peut pas créer de problèmes comme les fuites mémoire au cours de leur exécution. De plus, on peut utiliser les ressources disponibles de façon optimale pour une machine précise.

2.2 Fonctionnement

Le fonctionnement de FunLoft a été décrit en détail par Boussinot[15] ; ci-dessous un résumé des idées principales :

Pour transformer un programme écrit en FunLoft en un programme écrit en C, le compilateur de FunLoft effectue une série de transformations et de vérifications. D'abord on essaie de passer d'un programme écrit en FunLoft

à un programme écrit en Loft et en même temps de vérifier le contrôle des ressources pour empêcher les boucles instantanées et les fuites mémoire.

Les premières étapes consistent à vérifier le typage pour empêcher d'avoir des données incohérentes.

Ensuite, il faut faire le contrôle des ressources. Pour cela on a besoin de vérifier que les fonctions se terminent bien, qu'on n'a pas de boucles instantanées et que chaque programme recevera le contrôle à son tour.

Après avoir fait tous les contrôles nécessaires, on transforme notre programme en Loft[16] et, à l'aide du compilateur de Loft on transforme notre programme en C. La transformation d'un programme en Loft vers C est expliquée en détail par Boussinot.

2.3 Contrôle des ressources et réactivité

Les deux objectifs principaux de FunLoft sont le contrôle des ressources et la réactivité. Pour obtenir ces propriétés, on a besoin de faire des tests supplémentaires et s'assurer que chaque programme écrit en FunLoft les vérifie.

Contrôle des ressources : pour contrôler les ressources on a d'abord besoin de vérifier que toutes nos fonctions se terminent. De plus, on interdit la création des références sur les fonctions ainsi que les fonctions anonymes.

Réactivité : Un autre aspect de FunLoft qu'il faut vérifier est la réactivité. Le premier point à vérifier est l'absence des boucles instantanées et l'absence de module récursif.

2.4 Utilisation des ressources(schedulers)

FunLoft pour pouvoir utiliser les ressources de façon optimale propose de définir le nombre de schedulers nécessaires au programmeur de façon statique au début du programme. De cette façon on espère pouvoir maximiser l'utilisation des ressources. Le seul inconvénient de cette approche apparaît lorque le nombre de cores change. Dans ce cas notre programme n'utilisera plus tous les cores disponibles. sectionDSLM

2.5 Description informelle du modèle

En DSLM, un programme est composé de différents sites indépendants. Chacun de ces sites est composé de différents agents et chaque agent contient un script (un ensemble de scripts parallèles). Chaque script est composé d'expressions et d'instructions. Chaque expression doit finir son exécution

en un instant. Les instructions peuvent prendre un ou plusieurs instants pour finir (ou ne jamais se terminer).

Les fonctions ne sont pas définies dans notre modèle et on utilise celles de FunLoft. Dans ce qui va suivre, le terme fonction va être réservé aux fonctions instantanées (f) et les fonctions non-instantanées seront systématiquement appelées des modules (m). Les fonctions et les modules ne peuvent prendre que des types basiques comme arguments (*int*, *bool*, *string*, *etc.*) et ils doivent retourner des valeurs basiques également(les modules ne retournent rien).

2.6 Description informelle du langage

Les scripts et leur sémantique informelle sont définis par :

- let x = e in s end, $e_1 := e_2$, $!e$, $s_1; s_2$, `if` e `then` s_1 `else` s_2 `end` s'évaluent de manière tout à fait classique.
- $f(\vec{e})$ est la fonction externe f où il y a que les types basiques comme arguments. Son exécution doit être instantanée : notre modèle impose que son exécution ne prend pas plusieurs instants. Ceci est assuré par FunLoft.
- `launch` m $(\vec{e})$ lance le module externe m de FunLoft. Les modules comme les fonctions ne doivent avoir que des paramètres de types basiques. L'exécution d'un module peut prendre plusieurs instants ou ne jamais se terminer.
- `cooperate` termine l'exécution d'un script pour l'instant courant, s'il est permis de continuer (i.e. n'a pas été tué par un watching), l'exécution reprendra à l'instant suivant.
- $s_1 \,||\, s_2$ exécute les 2 scripts s_1 et s_2 en parallèle. Le script parallèle se termine quand les deux scripts finissent leur exécution.
- `while` e `do` s `end`, exécute s tant que e est vrai. L'exécution de s reprend dès qu'il se termine, sauf pour les terminaisons instantanées, dans ce cas, la boucle attend l'instant suivant pour se reéxecuter. De cette façon, il n'y a pas de boucle instantanée qui peut tourner à l'infini durant le même instant.
- `repeat` e `do` s `end`, exécute n fois s, où n est la valeur de e.
- `generate` ev `with` e, génère l'événement ev avec la valeur de e.
- `await` ev arrête l'exécution tant que l'événement ev n'est pas généré. L'exécution du script reprend dès que ev est généré.
- `getall` ev `with` e attend la fin de l'instant pour récupérer toutes les valeurs de l'événement ev et met le résultat dans la location e.
- `do` s `watching` ev, exécute le script s tant que l'événement ev n'est pas généré. L'exécution de s est annulée quand ev est généré. l'exécution

de *watching* se termine normalement quand s termine sans qu'il soit annulé.

- `letagent` $Ag'_{site}\{\ldots, x_i = e_i, \ldots\}$ `with` s `end`, crée l'agent Ag et lance l'exécution du script s dans ce nouvel agent avec des champs prédéfinis x_i où chacun d'entre eux est associé à la valeur v_i.
- `this`.e, permet d'accéder aux champs de l'agent auquel on appartient. Si on essaie d'accéder à quelque chose qui n'existe pas on obtient *unit*.
- `migrate` Ag `to` *site* migre l'agent avec le nom Ag au site distant *site*. Dans le cas d'incohérence, la migration ne se fera pas (par exemple, deux ordres de migrations à un agent pour des schedulers différents).
- `drop` s `in` *site* : Ag, lance le script s dans l'agent Ag qui appartient au site *site*. L'exécution du script continue immédiatement sans attendre la terminaison de s. Dans le cas d'incohérence la migration ne se fera pas.

2.7 Les fonctions et les modules

Les fonctions et les modules n'appartenant pas à DSLM, on utilise ceux de FunLoft. Grâce au système de typage de FunLoft, les fonctions sont toujours bien typées et se terminent dans un instant. Pour la même raison on n'a pas de *boucle instantanée* dans les modules ce qui signifie que les scripts ne peuvent pas empêcher le passage du site à l'instant suivant.

Les modules ont une autre particularité : un script qui lance un module doit attendre la fin d'exécution de ce module.

2.8 Mémoire

Dans notre modèle, la mémoire est un ensemble de locations qui contiennent des valeurs basiques ou des reférences sur d'autres locations. Il faut savoir que notre mémoire est divisée en parties disjointes :

$$M = M_{Ag_1} \oplus ... \oplus M_{Ag_n}$$

Cette particularité de la mémoire nous permet de prouver des propriétés importantes de façon plus simple.

On peut créer des locations avec l'instruction $let\, x = e$ ou bien avec la création d'agent ; on peut le modifier avec $e_1 = e_2$ et on peut y accéder avec $!e$ ou bien par *this*.e.

2.9 Script, Agents, Sites

Les sites sont asynchrones (i.e. chaque site a sa propre notion de temps). Au contraire, les agents et les scripts sont exécutés de façon synchrone sur un site : ils partagent la même notion de temps et ils procèdent à la même vitesse.

Chaque site a son ensemble d'événements ce qui veut dire que chaque agent appartenant à un site particulier peut seulement accéder aux événements de ce site (même chose pour les scripts).

L'instruction *drop* est le moyen par lequel un script peut influencer un site ou agent distant. Si rien ne reste après l'instruction on peut voir *drop* comme la migration de ce scripts.

La création des sites n'est pas spécifiée dans notre modèle. On peut supposer qu'un ensemble de sites est défini. Dans chaque script de la forme `drop` *s* `in` *site* : *Ag* et `migrate` *Ag* `to` *site*, on suppose que *site* est bien défini.

2.10 Atomicité

On définit des scripts *atomiques* de la façon suivant : les expressions (e :=e, !e, etc.), `generate` *ev*`with`*e* sont atomiques. Par ailleurs, une séquence $(s_1; s_2)$ est atomique si s_1 et s_2 sont atomiques. Notre modèle demande que l'exécution d'un script atomique n'ait aucune interférence avec d'autres scripts. On considère l'exemple suivant :

```
while true do
  print "0";print "0"
   ||
  print "1";print "1"
end
```

implique automatiquement que 0 et 1 viennent toujours par paires : une sortie ne doit jamais contenir une sous-chaîne 010 ou 101.

2.11 Evénements

Les événements sont valués et ne peuvent avoir que des valeurs basiques. Ils peuvent être présents ou absents au cours d'un instant. Les événements ne sont pas partagés entre les sites. Une fois qu'un événement est généré au cours d'un instant, il reste présent jusqu'à la fin de cet instant. Les événements sont remis à l'état absent au début de chaque instant.

Les événements sont des couples de la forme $(ev, \vec{v})$ où ev est une chaîne de caractères qui identifie l'événement et $\vec{v}$ est le vecteur qui contient les valeurs associées à cet événement.

2.12 Les vérifications statiques

Notre modèle demande que plusieurs propriétés soient vérifiées :

- Aucun site ne peut passer à l'instant suivant (propriété de réactivité). Ceci veut dire que les fonctions, les accès mémoire, les affectations, etc. finissent toujours leur exécution.
- Aucune data-race ne peut être produite entre deux scripts exécutés par deux agents distincts. Ceci signifie que les fonctions, tasks, locations sont protégées contre ces interférences(propriete d'absence d'interférence).
- Aucun module n'a de boucle instantanée. Cette propriété est vérifiée par FunLoft.
- Toutes les fonctions terminent en un instant(garanti par FunLoft).

2.13 Syntaxe

On assume les ensembles (infinis, disjoints) suivants :

- x ∈ VariableName, Les noms des variables ;
- f ∈ FunName, Les noms des fonctions ;
- m ∈ ModuleName, Les noms des modules ;
- Ag ∈ AgentName, Les noms des agents ;
- site ∈ SiteName, Les noms des sites ;

2.13.1 Valeurs

Entre les valeurs, on distingue les valeurs de types basiques, les locations et les événements :

$$v \in Val = Basic|Loc|Events$$
$$Basic = bool + int + string + unit$$
$$l \in Loc$$
$$st \in Stores = Loc \to Val$$

Les valeurs basiques sont les booleans, les entiers, les chaînes de caractères, et la valeur *unit*.

Les locations sont membres d'un ensemble infini de *Loc* où l est une référence sur une valeur basique ou bien sur une autre location. Une référence est une fonction partielle qui associe une valeur à une location(st).

A cause de la substitution, les locations peuvent apparaître dans notre modèle mais elles sont cachées dans notre sémantique.

Events est l'ensemble des événements comme on les a définis au paravant dans la section consacrée aux événements.

2.13.2 Expressions

Le syntaxe des expressions est définie par :

$$
\begin{array}{rl}
e \in Exp ::= & \\
 & x \,|\, v \,|\, f(\vec{e}) \\
| & !e \,|\, e := e \,|\, this.e
\end{array}
$$

On a déjà expliqué chacune des expressions dans la section précédente. On rappelle que l'évaluation de chaque expression ne doit pas prendre plus d'un instant.

2.13.3 Scripts

Les scripts sont différents des expressions. Ils peuvent prendre plusieurs instants pour que leur évaluation finisse(ou même ne jamais se terminer). Ici on trouve la syntaxe des scripts dans notre modèle :

$$
\begin{array}{ll}
\multicolumn{2}{l}{s \in Script ::=} \\
 & Exp \\
| & s; s \\
| & s \,||\, s \\
| & \texttt{let}\, x = e\, \texttt{in}\, s\, \texttt{end} \\
| & \texttt{if}\, e\, \texttt{then}\, s\, \texttt{else}\, s\, \texttt{end} \\
| & \texttt{while}\, e\, \texttt{do}\, s\, \texttt{end} \,|\, \texttt{repeat}\, e\, \texttt{do}\, s\, \texttt{end} \\
| & \texttt{cooperate} \\
| & \texttt{launch}\, m\, (\vec{e}) \\
| & \texttt{generate}\, ev\, \texttt{with}\, e \,|\, \texttt{await}\, ev \,|\, \texttt{getall}\, ev\, \texttt{with}\, e \,|\, \texttt{do}\, s\, \texttt{watching}\, ev \\
| & \texttt{drop}\, s\, \texttt{in}\, Ag : site \,|\, \texttt{migrate}\, Ag\, \texttt{to}\, site \,| \\
| & \texttt{letagent} Ag_{site} x_1 = v_1, ..., x_n = v_n \texttt{in} s \texttt{end}
\end{array}
$$

2.14 Sémantique

Notre sémantique est une sémantique small step. On a deux formes d'exécution, celles pour les expressions et celles pour les scripts.

Ici on trouve le format général de smallstep pour les expressions :

$$\langle e, M, E, Ag \rangle \rightarrow \langle e', M', E' \rangle$$

- e est l'expression qui doit être réécrite ;
- M est la mémoire qui contient les variables créées ;
- E est l'ensemble des événements présents ;
- Ag est l'agent auquel e appartient ;
- e' est le résultat d'évaluation de e ;
- M' est la nouvelle mémoire après l'évaluation de e ;
- E' est le nouvel environnement après l'évaluation de e ;

Ceci est le format général de notre sémantique small step pour les scripts :

$$\langle s, M, E, Ag, eoi \rangle \xrightarrow{b} \langle s', M', E', D, move \rangle$$

M, M', E, E' et Ag sont les mêmes que dans le cas des expressions.

- s est le script qui doit être réécrit ;
- *eoi* est le boolean qui nous signale la fin de l'instant ; s'il est vrai alors c'est la fin de l'instant et faux autrement ;
- b est soit vrai(tt) ou faux (*ff*), vrai (tt) quand s' est complètement terminé et faux (*ff*) quand s' n'est pas encore terminé. $\perp$ veut dire qu'on attend que quelque chose se produise avant la fin de l'instant :

$$b \coloneqq \mathit{ff} \mid tt \mid \perp$$

- s' est le script résiduel (ce qui reste à exécuter au cours du prochain instant).
- D est l'ensemble des scripts et des agents qui veulent migrer qui sont sous la forme *site* : $Ag \downarrow s$ quand c'est un script et site $\Downarrow Ag$ quand c'est un agent, où site est un nom de site, Ag est un nom d'agent et s est un script.
- *move* est un boolean qui nous dit si un événement est généré. Dans ce cas, on doit évaluer les scripts qui attendent cet événement.

Vous pouvez trouver toutes les règles de sémantique dans l'annexe (B). Ici, je tenterai d'expliquer quelques règles qui peuvent ne pas paraître très claires.

Parmi toutes les règles, la plus compliquée est celle de l'exécution parallèle.

$$\frac{\begin{array}{c}\langle s_{i_1}, M, E, Ag, \mathtt{eoi}\rangle \overset{b_1}{\rightarrow} \langle s'_{i_1}, M_1, E_1, D_1, \mathtt{m}\rangle \\ \langle s_{i_2}, M_1, E_1, Ag, \mathtt{eoi}\rangle \overset{b_2}{\rightarrow} \langle s'_{i_2}, M_2, E_2, D_2, \mathtt{m'}\rangle\end{array}}{\langle s_1 \,||\, s_2, M, E, Ag, \mathtt{eoi}\rangle \overset{b_1 \wedge b_2}{\rightarrow} \langle s'_1 \,||\, s'_2, M_2, E_2, D_1 \cup D_2, \mathtt{m} \vee \mathtt{m'}\rangle} \quad (1)$$

On évalue les deux scripts en parallèle, s_1 et s_2. Si l'exécution se termine pour les 2 branches (b n'est pas $\perp$ pour aucun des deux), alors on a fini l'exécution du script parallèle. L'ordre de l'exécution est totalement indéterministe.

$$\frac{\begin{array}{c}\langle s_{i_1}, M, E, Ag, \mathtt{eoi}\rangle \overset{b_1}{\rightarrow} \langle s'_{i_1}, M_1, E_1, D_1, \mathtt{m}\rangle \\ \langle s_{i_2}, M_1, E_1, Ag, \mathtt{eoi}\rangle \overset{b_2}{\rightarrow} \langle s'_{i_2}, M_2, E_2, D_2, \mathtt{m'}\rangle\end{array}}{\langle s_1 \,||\, s_2, M, E, Ag, \mathtt{eoi}\rangle \overset{\perp}{\rightarrow} \langle s'_1 \,||\, s'_2, M_2, E_2, D_1 \cup D_2, \mathtt{m} \vee \mathtt{m'}\rangle} \quad (2)$$

Ce cas est celui où l'un des scripts doit attendre. On s'évalue en $\perp$ et dans ce cas on sait qu'on doit reprendre l'exécution à un moment ou un autre. Ici aussi l'ordre de l'exécution est totalement arbitraire.

Dans le cas de watching :

$$\frac{ev \notin E \;\; \langle s, M, E, Ag, \mathtt{eoi}\rangle \overset{\perp}{\rightarrow} \langle s', M', E', D, \mathtt{m}\rangle}{\langle \mathtt{do}\ s\ \mathtt{watching}\ ev, M, E, Ag, \mathtt{eoi}\rangle \overset{\perp}{\rightarrow} \langle \mathtt{do}\ s'\ \mathtt{watching}\ ev, M', E', D, \mathtt{m}\rangle} \quad (3)$$

On a fait un choix : même si l'événement est présent on exécute encore une fois notre script et après on termine son exécution. Ce choix est discutable mais c'est celui qui est le plus proche de FunLoft.

2.15 Exécution d'un site

Pour l'exécution d'un site on a besoin de quelque notions supplémentaires. Un site est un ensemble de couples agent et état, ou un état signale

l'état de l'exécution de notre agent. Il peut être vrai (tt), faux (ff), $\perp$ ou bien $\top$. (tt) signifie que l'exécution est totalement terminée, (ff) signifie que l'exécution est terminée mais il reste une partie de l'exécution pour l'instant suivant. ($\perp$) signifie que l'exécution attend qu'un événement se produise. Et ($\top$) signifie qu'il faut exécuter cet agent.

E est l'ensemble des événements produits au cours de l'instant. M est l'ensemble des mémoires qui appartiennent à chaque agent.

$$site = \{[(Ag_1 : s_1, state_1), \ldots, (Ag_n : s_n, state_n)], E, M\}$$

2.15.1 Absorbtion d'un script ou agent

– Règle d'absorbtion d'agent :

$$\begin{aligned} &\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\}, site \Downarrow Ag_j : s_j \\ &\Rightarrow \{[\ldots, (Ag_i : s_i, \top), \ldots, (Ag_j : s_j, \top)], E, M \oplus M_{Ag_j}\} \end{aligned} \tag{4}$$

Ici on est dans le site $site$ et on essaie d'absorber l'agent Ag_j.

– Règle d'absorbtion de script :

$$\begin{aligned} &\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\}, site : Ag_i \downarrow s_j \\ &\qquad\Rightarrow \{[\ldots, (Ag_i : s_i \,||\, s_j, \top), \ldots], E, M\} \end{aligned} \tag{5}$$

2.15.2 Exécution d'un site

On divise l'exécution de site en deux parties, l'exécution au cours d'un instant et l'exécution à la fin d'un instant. La première fois qu'on a commencé l'exécution, l'état($state$) de chaque agent est $\top$.

Quand quelque chose se passe (génération d'un événement) tous les agents qui attendaient vont être remis en marche pour qu'ils puissent être reexécutés.

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \texttt{true}\rangle}{\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\} \Rightarrow \delta(\{[\ldots, (Ag_i : s'_i, b), \ldots], E', M'\})} \tag{6}$$

La fonction δ change tous les $\bot$ en $\top$. Bien sûr ce n'est pas la façon la plus optimale mais notre but n'est pas de trouver une méthode optimale(ceci concerne la partie implémentation).

Autrement, quand aucun événement n'est signalé :

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s_i', M', E', D, \texttt{false}\rangle}{\{[\ldots,(Ag_i : s_i, \top),\ldots], E, M\} \Rightarrow \{[\ldots,(Ag_i : s_i', b),\ldots], E', M'\}} \tag{7}$$

Pour la fin d'instant, on a besoin d'un mécanisme particulier. On a la forme de sémantique suivante :

$$\langle s, M, E, Ag, p\rangle \to \langle s, M\rangle$$

où s, s', M, E et Ag sont ceux qu'on a déjà vu dans la partie auparavant et p peut être soit $\bot$, *tt* ou bien *ff*.

Deux comportements changent par rapport à p :

$$\frac{\langle s, M, E, Ag, \texttt{true}\rangle \xrightarrow{b} \langle s', M', E', D, move\rangle}{\langle s, M, E, Ag, \bot\rangle \to \langle s', M'\rangle} \tag{8}$$

cela signifie que le script était en attente donc il y a deux cas possibles :

- soit on est en attente d'un événement qui ne s'est pas produit encore ; alors on s'écrit en nous-même
- ou bien on le reçoit et on passe à l'instruction suivante.
- soit on est dans le cas de *getall* ; dans ce cas on prend toutes les valeurs associées à cet événement.

Si le script n'a pas été en attente :

$$\frac{p \neq \bot}{\langle s, M, E, Ag, p\rangle \to \langle s, M\rangle} \tag{9}$$

La règle d'exécution à la fin de l'instant est la suivante :

$$\frac{\forall i : state_i \neq \top s.t. \langle s_i, M, E, Ag, state_i \rangle \rightarrow \langle s'_i, M'_i \rangle}{\{[\ldots, (Ag_i : s_i, state_i), \ldots], E, M\} \Rightarrow \{[\ldots, (Ag_i : s'_i, \top), \ldots], E, \oplus M'_i\}} \quad (10)$$

Dans ce cas, on est à la fin de l'instant et ce qui reste à faire est d'exécuter encore une fois les scripts qui étaient en attente. On a besoin de cette exécution pour récupérer les valeurs attendues par *getall* et on va remettre tous les agents dans leurs état initial($\top$).

Dans notre sémantique d'exécution de site, on est sûr que durant chaque instant chaque agent a été exécuté au moins une fois et qu'un instant ne dure jamais à l'infini.

2.16 Implémentation

D'après la sémantique qu'on a proposé, on n'arrive pas à voir comment utiliser les cores de façon maximale. On a préféré garder cette partie cachée au point de vu de l'utilisateur, pour faciliter la programmation dans notre modèle.

Pour maximiser l'utilisation des cores, on utilise la contraction et expansion des sites à l'aide de la notion de scheduler synchronisé. Une scheduler synchronisé est un ensemble de schedulers qui s'exécutent à la même vitesse (partage la même notion du temps) mais qui peuvent être exécutés sur de cores distincts.

Ainsi un site devient un ensemble de schedulers synchronisés non-vide, où Chaque scheduler exécute un ensemble d'agents. Alors un site sera :

$$site = \{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, state_{i_j}), \ldots|\!), \ldots], E, M\}$$

Où $state_i$, Ag_{i_j}, s_{i_j}, E et M ont la même définition qu'auparavant dans la partie 2.15($state_i \in \{\bot, \top, tt, \mathit{ff}\}$) et $sched_i$ est un scheduler appartenant à notre site et les agents entre $(\!|$ et $|\!)$ appartiennent au même scheduler. Les schedulers appartenant au même site sont des schedulers synchronisé (ceux entre [et]).

2.16.1 Exécution d'un site avec des schedulers

L'exécution de notre nouvelle définition de site ressemble à l'exécution d'un site dans la section 2.15 avec des schedulers en plus. On a toujours trois cas que voici :

– Génération d'événement pendant l'exécution du script :

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \texttt{true}\rangle}{\begin{array}{l}\{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, \top), \ldots|\!), \ldots], E, M\} \\ \Rightarrow \delta(\{[\ldots, sched_i := (\!|\ldots, (Ag_{i_j} : s'_{i_j}, b), \ldots|\!), \ldots], E', M'\})\end{array}} \quad (11)$$

– Rien n'a été généré par le script au cours de l'exécution :

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \texttt{false}\rangle}{\begin{array}{l}\{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, \top), \ldots|\!), \ldots], E, M\} \\ \Rightarrow \{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s'_{i_j}, b), \ldots|\!), \ldots], E', M'\}\end{array}} \quad (12)$$

– Fin d'instant :

$$\frac{\forall i : state_i \neq \top \mid \langle s_i, M, E, Ag, state_i\rangle \rightarrow \langle s'_i, M'_i\rangle}{\begin{array}{l}\{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, \top), \ldots|\!), \ldots], E, M\} \\ \Rightarrow \{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s'_{i_j}, \top), \ldots|\!), \ldots], E, \oplus_{i=1\ldots n} M'_i\}\end{array}} \quad (13)$$

Où s,Ag,$\bot$,$\top$ et etc. sont les mêmes qu'auparavant et $sched_i$ est le schedulers i auquel Ag_{i_j} appartient.

2.16.2 Expansion et contraction d'un site

Un site peut contenir un ou plusieurs schedulers synchronisés et on a conçu un mécanisme qui nous permet d'ajouter des nouveaux scheduler à notre site (contraction). Et si on n'a plus besoin d'un scheduleur (i.e. il n'exécute plus aucun agent), on libère le scheduler et on rétracte le site.

Voici les deux régles pour la contraction et l'expansion des sites :

– Expansion d'un site :

$$\begin{array}{c}\{[sched = (\!|(Ag : s, state), (Ag' : s', state'), \ldots|\!), \ldots], E, M\} \\ \rightsquigarrow \{[sched = (\!|(Ag : s, state), \ldots\}, sched' = (\!|(Ag' : s', state')|\!), \ldots], E, M\}\end{array} \quad (14)$$

– Contraction d'un site :

$$\{[sched = (\!|(Ag : s, state), \ldots|\!), sched' = (\!|(Ag' : s', state')|\!), \ldots], E, M\}$$
$$\rightsquigarrow \{, [sched = (\!|(Ag : s, state), (Ag' : s', state'), \ldots|\!), \ldots], E, M\} \quad (15)$$

On a remarqué qu'ajouts et suppressions des schedulers ne suffisent pas à notre propos et il nous faut un mécanisme pour partager la charge entre les schedulers. Pour cela, on a proposé *la migration automatique*. En d'autre terme, chaque agent peut migrer d'un scheduler à un autre, sans que cela soit visible au point de vue de l'utilisateur, pour partager la charge entre les schedulers appartenant au même site.

$$\{[\ldots, sched = (\!|(Ag : s, state), \ldots|\!), sched' = (\!|(Ag' : s', state'), \ldots|\!), \ldots], E, M\}$$
$$\rightsquigarrow \{[\ldots, sched = (\!|(Ag : s, state), (Ag' : s', state'), \ldots|\!), sched' = (\!|\ldots|\!), \ldots], E, M\} \quad (16)$$

Cette migration automatique ne pose aucun problème à l'intégralité des programmes écrits dans notre modèle, parce que chaque agent garde son état en migrant et que les événements sont partagés entre les agents de même site.

2.17 Typage

Pour pouvoir contrôler les ressources et les références des agents et des scripts, on a besoin d'un système de typage qui nous permet de vérifier cela de façon statique. Il faut signaler que dans le cas de génération des événements et attente d'un événement, on n'a pas besoin de vérifier le type des événements vu qu'ils sont traités comme des chaînes de caractères.

Un type est soit un type basique(int, bool, etc.), ou bien une référence sur ce type :

$$\tau ::= string \mid bool \mid int \mid unit \mid \mathtt{ref}_{Ag}\, \tau$$

où ref_{Ag} signifie que le type de référence appartient à la mémoire de l'agent Ag.

Un environnement de type (Γ) est un ensemble (qui peut être vide) d'éléments de la forme x :τ où x est une variable et τ sera son type.

$$\Gamma ::= x_1 : \tau_1, \cdots, x_n : \tau_n$$

Ci-dessous la forme générale de typage :

$$Ag, \Gamma \vdash s : \tau$$

- Ag est l'agent dans lequel on exécute s ;
- Γ est l'environnement de typage ;
- s est le script à typer ;
- τ est le type du script s ;

Vous trouverez les règles de typage dans l'annexe(G). On doit être averti que dans les règles de typage on a besoin des sites parce qu'on a besoin des types de nos événements et que chaque événement ne peut avoir qu'un type particulier.

Pour que les lecteurs puissent mieux comprendre le typage et ainsi que la sémantique on a donné quelques exemples dans l'annexe(H) pour clarifier les choses.

2.18 Preuve de correction du typage

Pour montrer que notre typage est correct, on a besoin de prouver que si un programme est correctement typé[17, 18], alors ce programme n'admet pas de data-race.

preuve : vous trouverez la preuve complète dans l'annexe. Ici, je ne donnerai que l'idée de la preuve. L'idée générale est : il y a, par définition, une séparation de mémoire entre les agents. Cette forte propriété nous permet de prouver ce théorème de façon simple. Il suffit de vérifier que si on fait une migration de script, on n'accède pas à la mémoire d'un autre agent (La migration d'agent ne pose aucun problème au niveau de data-race, vu qu'un agent emporte avec lui sa mémoire quand il migre). Donc il suffit de vérifier, en cas de migration d'un script, qu'on n'accède jamais à la mémoire d'un autre agent à part celui auquel on appartient. on vérifie les accés mémoire ($!e$,$e_1 := e_2$, etc.) et grâce à notre typage et à la séparation de la mémoire, on est sûr que cela ne risque pas d'arriver.

2.19 Preuve d'existence d'un point fixe

Pour prouver l'existence de point fixe au cours d'un instant, il suffit de montrer que tous nos programmes se terminent au bout d'un temps borné. Pour prouver cela on a besoin de regarder les cas où le programme peut être en boucle infinie ou bien un script ne laissera pas la main aux autres. On remarque que les seuls cas qui peuvent arriver sont les boucles *while* et dans

les fonctions ou bien les modules. On sait que notre sémantique ne permet pas de boucle instantanée, et qu'à l'aide de FunLoft, on est assuré qu'aucun des modules, ni des fonctions ne peut durer un temps non-borné au cours d'un même instant. Donc on sait que tout nos programme s'arrêteront au bout d'un temps borné (la preuve complète se trouve en annexeJ).

3 Conclusion

3.1 Bilan

On a réussi à trouver un modèle réactif synchrone à l'aide de FunLoft, avec la sémantique, le typage, etc. On a réussi à résoudre une partie des problèmes des programmes synchrones, comme la difficulté d'utiliser des composantes concurrentes tout en utilisant toutes les ressources (cores) disponibles à l'aide des mécanismes comme les agents et les sites. J'ai aussi prouvé la correction de notre typage ainsi que notre sémantique admet un point fixe à chaque instant. Un article sur la partie théorique est en cours de rédaction.

3.2 Perspective

Il faudra tout d'abord implémenter le langage DSLM en FunLoft(ou un autre language synchone qui permet d'avoir les mêmes propriétés), tester et le comparer avec d'autres langages concurrents comme CPC ou mm̂e lwt[19]. Pour cela il faut faire des implémentations en FunLoft et puis s'assurer qu'après ces implementations les mêmes propriétés sont vérifiées.

L'autre aspect important est la sécurité. Par exemple, quand on a des liens sur une donnée confidentielle, ne pas la révéler aux autres participants et trouver la bonne approche pour faire ces vérifications sur nos programmes.

Enfin, FunLoft ne fournit pour l'instant aucun mécanisme pour interagir avec d'autres langages. Ce qui peut être très intéressant dans le cas des systèmes distribué. Le travail de Melliar-Smith[20] sur le sujet peut être un bon point de départ.

Références

[1] Frédéric Boussinot. A benchmark for multicore machines. 2007. http://hal.inria.fr/inria-00174843.

[2] Frédéric Boussinot and J-F Susini. The sugarcubes tool box - a reactive java framework. *Software Practice and Experience*, 1998.

[3] Frédéric Boussinot and J-F Susini. Java threads and sugarcubes. *Software Practice and Experience*, 2000.

[4] B Lewis and DJ Berg. *Multithreaded Programming with Pthreads*. ACM, 1998.

[5] Frédéric Boussinot. Reactive c : An extension of c to program reactive systems. *Software Practice and Experience*, 1991.

[6] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[7] Frédéric Boussinot. Fairthreads : mixing cooperative and preemptive threads in c. *Concurrency and Computation : Practice and Experience*, 2003.

[8] Juliusz Chroboczek. Continuation passing for c : A space-efficient implementation of concurrency. *Technical report, PPS, UniversitÃ© de Paris 7*, 2005.

[9] Juliusz Chroboczek. The cpc manual, preliminary edition, 2008. http://www.pps.jussieu.fr/~jch/software/cpc/cpc-manual.pdf.

[10] Frédéric Boussinot and Frédéric Dabrowski. Safe reactive programming : the funloft proposal. *Proc. of MULTIPROG – First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[11] Gérard Boudol. Ulm, a core programming model for global computing. *In Proc. of ESOP,Springer LNCS 2986*, 2004.

[12] Martín Abadi and Gordon Plotkin. A model of cooperative threads. *In Proc. of the 36th Annual ACM Symposium on Principles of Programming Languages*, 2009.

[13] Frédéric Boussinot and Frédéric Dabrowski. Cooperative threads and preemptive computations. *Proc. of TV'06 – Multithreading in Hardware and Software : Formal Approaches to Design and Verification*, 2006.

[14] Frédéric Dabrowski. Programmation réactive synchrone : langages et contrôle des ressources. 2007. PhD thesis.

[15] Frédéric Boussinot and Frédéric Dabrowski. Formalisation of funloft. 2007. http://hal.inria.fr/inria-00183242.

[16] Frédéric Boussinot. Loft. 1996. http://www-sop.inria.fr/mimosa/rp/LOFT.

[17] M. Felleisen and AK. Wright. A syntactic approach to type soundness. *Information and Computation*, 1994.

[18] Dennis Volpano, Geoffrey Smith, and Cynthia Ivrine. A sound type system for secure flow. *Journal of Computer Security, IOS Press*, 2009.

[19] Christophe Deleuze. Light weight concurrency in ocaml. 2010. http://hal.archives-ouvertes.fr/docs/00/49/32/13/PDF/lwc.pdf.

[20] P.M. Melliar-Smith, Louise E. Moser, and Agrawala Vivek. *Broadcast Protocols for Distributed Systems*. 1990.

Annexes

A Syntaxe

A.1 Syntaxe des expressions

$$
\begin{array}{ll}
e \in Exp & ::= \\
 & x \,|\, v \,|\, f(\vec{e}) \\
| & !e \,|\, e := e \,|\, this.e
\end{array}
$$

A.2 Syntaxe des Scripts

$$
\begin{array}{ll}
s \in Script & ::= \\
 & Exp \\
| & s; s \\
| & s \,||\, s \\
| & \texttt{let}\; x \;=\; e \;\texttt{in}\; s \;\texttt{end} \\
| & \texttt{if}\; e \;\texttt{then}\; s \;\texttt{else}\; s \;\texttt{end} \\
| & \texttt{while}\; e \;\texttt{do}\; s \;\texttt{end} \,|\, \texttt{repeat}\; e \;\texttt{do}\; s \;\texttt{end} \\
| & \texttt{cooperate} \\
| & \texttt{launch}\; m\; (\overrightarrow{e}) \\
| & \texttt{generate}\; ev \;\texttt{with}\; e \,|\, \texttt{await}\; ev \,|\, \texttt{getall}\; ev \;\texttt{with}\; e \,|\, \texttt{do}\; s \;\texttt{watching}\; ev \\
| & \texttt{drop}\; s \;\texttt{in}\; Ag : site \,|\, \texttt{migrate}\; Ag \;\texttt{to}\; site \,| \\
| & \texttt{letagent} Ag_{site} x_1 = v_1, ..., x_n = v_n \texttt{in} s \texttt{end}
\end{array}
$$

B Sémantique

B.0.1 Expressions

- Les valeurs

$$\langle v, M, E, Ag \rangle \rightarrow \langle v, M, E \rangle \tag{17}$$

– référence

$$\frac{\langle e, M, E, Ag\rangle \rightarrow \langle v, M', E'\rangle \;\; l \textit{ is fresh}}{\langle \texttt{ref}\, e, M, E, Ag\rangle \rightarrow \langle l, M'[l \leftarrow v], E'\rangle} \tag{18}$$

– Accès à la mémoire

$$\frac{\langle e, M, E, Ag\rangle \rightarrow \langle l, M', E'\rangle}{\langle !e, M, E, Ag\rangle \rightarrow \langle M'[l], M', E'\rangle} \tag{19}$$

où $M'[l]$ retourne la velure de référence de l dans la mémoire de M' (l appartient à la memoire de l'agent Ag).

– Les points d'entrées d'agents

$$\frac{\langle e, M, E, Ag\rangle \rightarrow \langle l, M', E'\rangle}{\langle \texttt{this.}\, e, M, E, Ag\rangle \rightarrow \langle M'[Ag][l], M', E'\rangle} \tag{20}$$

où $M'[Ag][l]$ retrourne la valeur de la référence l qui est l'un des points d'entrée de l'agent dans lequel on est en train de s'exécuter.

– appelle de function :

$$\frac{\langle \overrightarrow{e}, M, E, Ag\rangle \rightarrow \langle \overrightarrow{v}, M_1, E_1\rangle}{\langle f(\overrightarrow{e}), M, E, Ag\rangle \rightarrow \langle v', M_1, E_1\rangle} \; v' = \ulcorner f \urcorner(\overrightarrow{v}) \tag{21}$$

où $\ulcorner f \urcorner(\overrightarrow{v})$ veut dire : l'evaluation de la fonction f dans FunLoft avec $\overrightarrow{e}$ ($\overrightarrow{e} = (e_1, e_2, ..., e_n)$) évalue de gauche à droit.

B.0.2 Scripts

Voici la sémantique small step pour les scripts :

– Affectation

$$\frac{\langle e_1, M_1, E_1, Ag\rangle \rightarrow \langle l, M_2, E_2\rangle \;\; \langle e_2, M, E, Ag\rangle \rightarrow \langle v, M_1, E_1\rangle}{\langle e_1 := e_2, M, E\rangle Ag\texttt{eoi} \rightarrow \langle \texttt{unit}, M_2[l \leftarrow v], E_2\rangle} \tag{22}$$

– Déclaration de variables

$$\frac{\langle e, M, E, Ag\rangle \rightarrow \langle v, M_1, E_1\rangle \;\; \langle s[x\backslash v], M_1, E_1, Ag, \texttt{eoi}\rangle \xrightarrow{b} \langle s', M_2, E_2, D, \texttt{m}\rangle}{\langle \texttt{let}\, x \;=\; e\, \texttt{in}\, s\, \texttt{end}, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{b} \langle s', M_2, E_2, D, \texttt{m}\rangle} \tag{23}$$

– Séquence

$$\frac{\langle s_1, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_1, M', E', D, \mathtt{m}\rangle \;\; b \in \{ff, \bot\}}{\langle s_1; s_2, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_1; s_2, M', E', D, \mathtt{m}\rangle} \tag{24}$$

$$\frac{\langle s_1, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{tt} \langle s'_1, M_1, E_1, D_1, \mathtt{m}\rangle \;\; \langle s_2, M_1, E_1, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_2, M_2, E_2, D_2, \mathtt{m}'\rangle}{\langle s_1; s_2, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_2, M_2, E_2, D_1 \cup D_2, \mathtt{m} \vee \mathtt{m}'\rangle} \tag{25}$$

– Conditionnelle

$$\frac{\langle e, M, E, Ag\rangle \to \langle true, M_1, E_1\rangle \;\; \langle s_1, M_1, E_1, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_1, M_2, E_2, D, \mathtt{m}\rangle}{\langle \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_1, M_2, E_2, D, \mathtt{m}\rangle} \tag{26}$$

$$\frac{\langle e, M, E, Ag\rangle \to \langle false, M_1, E_1\rangle \;\; \langle s_2, M_1, E_1, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_2, M_2, E_2, D, \mathtt{m}\rangle}{\langle \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'_2, M_2, E_2, D, \mathtt{m}\rangle} \tag{27}$$

– While

$$\frac{\langle e, M, E, Ag\rangle \to \langle \mathtt{true}, M_1, E_1\rangle \;\; \langle s, M_1, E_1, Ag, \mathtt{eoi}\rangle \xrightarrow{tt} \langle s', M_2, E_2, D, \mathtt{m}\rangle}{\langle \mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{end}, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{ff} \langle \mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{end}, M_2, E_2, D, \mathtt{m}\rangle} \tag{28}$$

$$\frac{\langle e, M, E, Ag\rangle \to \langle \mathtt{true}, M_1, E_1\rangle \;\; \langle s, M_1, E_1, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s', M_2, E_2, D, \mathtt{m}\rangle \;\; b \in \{ff, \bot\}}{\langle \mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{end}, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{b} \langle s'; \mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{end}, M_2, E_2, D, \mathtt{m}\rangle} \tag{29}$$

$$\frac{\langle e, M, E, Ag\rangle Ag \to \langle \mathtt{false}, M_1, E_1\rangle}{\langle \mathtt{while}\ e\ \mathtt{do}\ s\ \mathtt{end}, M, E, Ag, \mathtt{eoi}\rangle \xrightarrow{tt} \langle \mathtt{unit}, M_2, E_2, D, \mathtt{m}\rangle} \tag{30}$$

– Repeat

$$\frac{\langle e, M, E, Ag\rangle \rightarrow \langle n, M_1, E_1\rangle \ \langle \overbrace{s; \ldots; s}^{n\ times}, M_1, E_1, \texttt{eoi}\rangle \xrightarrow{b} \langle s', M_2, E_2, D, \texttt{m}\rangle}{\langle \texttt{repeat}\ e\ \texttt{do}\ s\ \texttt{end}, M, E, \texttt{eoi}\rangle \xrightarrow{b} \langle s', M_2, E_2, D, \texttt{m}\rangle} \quad (31)$$

– Parallèle

$$\frac{\langle s_{i_1}, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{b_1} \langle s'_{i_1}, M_1, E_1, D_1, \texttt{m}\rangle \ \langle s_{i_2}, M_1, E_1, Ag, \texttt{eoi}\rangle \xrightarrow{b_2} \langle s'_{i_2}, M_2, E_2, D_2, \texttt{m}'\rangle}{\langle s_1 \,||\, s_2, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{b_1 \wedge b_2} \langle s'_1 \,||\, s'_2, M_2, E_2, D_1 \cup D_2, \texttt{m} \vee \texttt{m}'\rangle} \quad (32)$$

où $b_i \in \{ff,\ tt\}$ et $i_1 = 1 \wedge i_2 = 2$ ou bien $i_1 = 2 \wedge i_2 = 1$ (i_1 et i_2 sont choisie de façon totalement arbitraire).

$$\frac{\langle s_{i_1}, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{b_1} \langle s'_{i_1}, M_1, E_1, D_1, \texttt{m}\rangle \ \langle s_{i_2}, M_1, E_1, Ag, \texttt{eoi}\rangle \xrightarrow{b_2} \langle s'_{i_2}, M_2, E_2, D_2, \texttt{m}'\rangle}{\langle s_1 \,||\, s_2, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{\perp} \langle s'_1 \,||\, s'_2, M_2, E_2, D_1 \cup D_2, \texttt{m} \vee \texttt{m}'\rangle} \quad (33)$$

ici b_1 ou b_2 (ou bien tous les deux) sont égaux à $\perp$ et $i_1 = 1 \wedge i_2 = 2$ ou $i_1 = 2 \wedge i_2 = 1$ (i_1 et i_2 sont choisis de façon non-déterministe).

– Cooperate

$$\langle \texttt{cooperate}, M, E, Ag, _\rangle \xrightarrow{ff} \langle \texttt{unit}, M, E, \emptyset, \texttt{false}\rangle \quad (34)$$

– Launch

$$\frac{\langle \vec{e}, M, E, Ag\rangle \rightarrow \langle \vec{e'}, M', E'\rangle \ \ \vec{e} = (e_1, \ldots, e_n)\ \ ev\ is\ a\ fresh\ event}{\langle \texttt{launch}\ m\ (ev, \vec{e}), M, E, Ag, \texttt{eoi}\rangle \xrightarrow{\perp} \langle \texttt{await}\ ev, M', E', \emptyset, false\rangle} \quad (35)$$

– Generate

$$\frac{\langle e, M, E, Ag\rangle \rightarrow \langle v, M', E'\rangle}{\langle \texttt{generate}\ ev\ \texttt{with}\ e, M, E, Ag, _\rangle \xrightarrow{tt} \langle \texttt{unit}, M', E' \cup \{(ev, v)\}, \emptyset, \texttt{true}\rangle} \quad (36)$$

c'est le seul cas où *move* est mis à true.

– GetAll

$$\langle \texttt{getall}\, ev\, \texttt{with}\, e, M, E, Ag, \texttt{false}\rangle \xrightarrow{\perp} \langle \texttt{getall}\, ev\, \texttt{with}\, e, M, E, \emptyset, \texttt{false}\rangle \quad (37)$$

$$\frac{E[ev] = \overrightarrow{v} \quad \langle e, M, E, Ag\rangle \rightarrow \langle l, M', E'\rangle}{\langle \texttt{getall}\, ev\, \texttt{with}\, e, M, E, Ag, \texttt{true}\rangle \xrightarrow{tt} \langle \texttt{unit}, M[l \leftarrow \overrightarrow{v}], E, \emptyset, \texttt{false}\rangle} \quad (38)$$

$\overrightarrow{ev}$ est la valeur de l'événement ev. Ce peut être une liste vide quand aucun événement n'a été généré durant l'instant (currant), et sinon c'est la liste des valeurs de ev.

– Await

$$\frac{ev \in E}{\langle \texttt{await}\, ev, M, E, Ag, _\rangle \xrightarrow{tt} \langle \texttt{unit}, M, E, \emptyset, \texttt{false}\rangle} \quad (39)$$

$$\frac{ev \notin E}{\langle \texttt{await}\, ev, M, E, Ag, \texttt{true}\rangle \xrightarrow{ff} \langle \texttt{await}\, ev, M, E, \emptyset, \texttt{false}\rangle} \quad (40)$$

$$\frac{ev \notin E}{\langle \texttt{await}\, ev, M, E, Ag, \texttt{false}\rangle \xrightarrow{\perp} \langle \texttt{await}\, ev, M, E, \emptyset, \texttt{false}\rangle} \quad (41)$$

– Watching

$$\frac{\langle s, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{tt} \langle s', M', E', D, \texttt{m}\rangle}{\langle \texttt{do}\, s\, \texttt{watching}\, ev, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{ff} \langle \texttt{unit}, M', E', D, \texttt{m}\rangle} \quad (42)$$

$$\frac{ev \in E \quad \langle s, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{ff} \langle s', M', E', D, \texttt{m}\rangle}{\langle \texttt{do}\, s\, \texttt{watching}\, ev, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{ff} \langle \texttt{unit}, M', E', D, \texttt{m}\rangle} \quad (43)$$

$$\frac{ev \notin E \ \langle s, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{ff} \langle s', M', E', D, \texttt{m}\rangle}{\langle \texttt{do}\ s\ \texttt{watching}\ ev, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{ff} \langle \texttt{do}\ s'\ \texttt{watching}\ ev, M', E', D, \texttt{m}\rangle} \quad (44)$$

$$\frac{ev \notin E \ \langle s, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{\perp} \langle s', M', E', D, \texttt{m}\rangle}{\langle \texttt{do}\ s\ \texttt{watching}\ ev, M, E, Ag, \texttt{eoi}\rangle \xrightarrow{\perp} \langle \texttt{do}\ s'\ \texttt{watching}\ ev, M', E', D, \texttt{m}\rangle} \quad (45)$$

- Création d'agent

$$\frac{\langle e_i, M, E, Ag\rangle \rightarrow \langle v_i, M', E'\rangle \ E_{Ag'_{site}} = [\ldots, x_i \leftarrow v_i, \ldots]}{\langle \texttt{letagent}\ Ag'_{site}\{\ldots, x_i = e_i, \ldots\}\ \texttt{with}\ s\ \texttt{end}, M, E, Ag, _\rangle \xrightarrow{tt} \langle \texttt{unit}, M, E \oplus E_{Ag'_{site}}, \{site \Downarrow Ag' : s\}, \texttt{false}\rangle} \quad (46)$$

où le script s est attaché à l'agent Ag'.

- Drop

$$\langle \texttt{drop}\ s\ \texttt{in}\ site : Ag', M, E, Ag, _\rangle \xrightarrow{tt} \langle \texttt{unit}, M, E, \{site : Ag' \downarrow s\}, \texttt{false}\rangle \quad (47)$$

- Migration

$$\langle \texttt{migrate}\ Ag'\ \texttt{to}\ site, M, E, Ag, _\rangle \xrightarrow{tt} \langle \texttt{unit}, M, E, \{site \Downarrow Ag' : s\}, \texttt{false}\rangle \quad (48)$$

  S'il y a une incohérence dans la migration alors cette migration ne sera pas effectuée.

C Absorbtion d'agents et de scripts

- Règle d'absorbtion d'agent :

$$\begin{aligned} &\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\}, site \Downarrow Ag_j : s_j \\ &\Rightarrow \{[\ldots, (Ag_i : s_i, \top), \ldots, (Ag_j : s_j, \top)], E, M \oplus M_{Ag_j}\} \end{aligned} \quad (49)$$

- Règle d'absorbtion de script :

$$\begin{aligned} &\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\}, site : Ag_i \downarrow s_j \\ &\qquad\qquad \Rightarrow \{[\ldots, (Ag_i : s_i \,||\, s_j, \top), \ldots], E, M\} \end{aligned} \quad (50)$$

D Exécution d'un site au point de vue de l'utilisateur

– Le script qui a été exécuté a généré un événement

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \texttt{true}\rangle}{\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\} \Rightarrow \delta(\{[\ldots, (Ag_i : s'_i, b), \ldots], E', M'\})} \tag{51}$$

– Le script n'a pas généré d'événement

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \texttt{false}\rangle}{\{[\ldots, (Ag_i : s_i, \top), \ldots], E, M\} \Rightarrow \{[\ldots, (Ag_i : s'_i, b), \ldots], E', M'\}} \tag{52}$$

D.1 Fin d'instant

– Dans le cas d'attente de script (*await* ou bien *getAll*)

$$\frac{\langle s, M, E, Ag, \texttt{true}\rangle \xrightarrow{b} \langle s', M', E', D, move\rangle}{\langle s, M, E, Ag, \bot\rangle \rightarrow \langle s', M'\rangle} \tag{53}$$

– Si le script n'a pas été en attente

$$\frac{p \neq \bot}{\langle s, M, E, Ag, p\rangle \rightarrow \langle s, M\rangle} \tag{54}$$

– La règle d'exécution à la fin de l'instant est la suivante :

$$\frac{\forall i : state_i \neq \top s.t. \langle s_i, M, E, Ag, state_i\rangle \rightarrow \langle s'_i, M'_i\rangle}{\{[\ldots, (Ag_i : s_i, state_i), \ldots], E, M\} \Rightarrow \{[\ldots, (Ag_i : s'_i, \top), \ldots], E, \oplus M'_i\}} \tag{55}$$

E Implementation

– Un site est :

$site = \{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, state_{i_j}), \ldots|\!), \ldots], E, M\}$

E.1 Exécution d'un site

– Le script qui a été exécuté a généré un événement

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \mathtt{true}\rangle}{\begin{array}{l}\{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, \top), \ldots|\!), \ldots], E, M\} \\ \Rightarrow \delta(\{[\ldots, sched_i := (\!|\ldots, (Ag_{i_j} : s'_{i_j}, b), \ldots|\!), \ldots], E', M'\})\end{array}} \quad (56)$$

– Rien n'a été généré par le script au cours de l'exécution :

$$\frac{\langle s_i, M, E, Ag_i, _\rangle \xrightarrow{b} \langle s'_i, M', E', D, \mathtt{false}\rangle}{\begin{array}{l}\{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, \top), \ldots|\!), \ldots], E, M\} \\ \Rightarrow \{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s'_{i_j}, b), \ldots|\!), \ldots], E', M'\}\end{array}} \quad (57)$$

– Fin d'instant :

$$\frac{\forall i : state_i \neq \top \mid \langle s_i, M, E, Ag, state_i\rangle \rightarrow \langle s'_i, M'_i\rangle}{\begin{array}{l}\{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s_{i_j}, \top), \ldots|\!), \ldots], E, M\} \\ \Rightarrow \{[\ldots, sched_i = (\!|\ldots, (Ag_{i_j} : s'_{i_j}, \top), \ldots|\!), \ldots], E, \oplus_{i=1\ldots n} M'_i\}\end{array}} \quad (58)$$

F Expansion et contraction d'un site

– Expansion d'un site :

$$\begin{array}{l}\{[sched = (\!|(Ag : s, state), (Ag' : s', state'), \ldots|\!), \ldots], E, M\} \\ \rightsquigarrow \{[sched = (\!|(Ag : s, state), \ldots\}, sched' = (\!|(Ag' : s', state')|\!), \ldots], E, M\}\end{array} \quad (59)$$

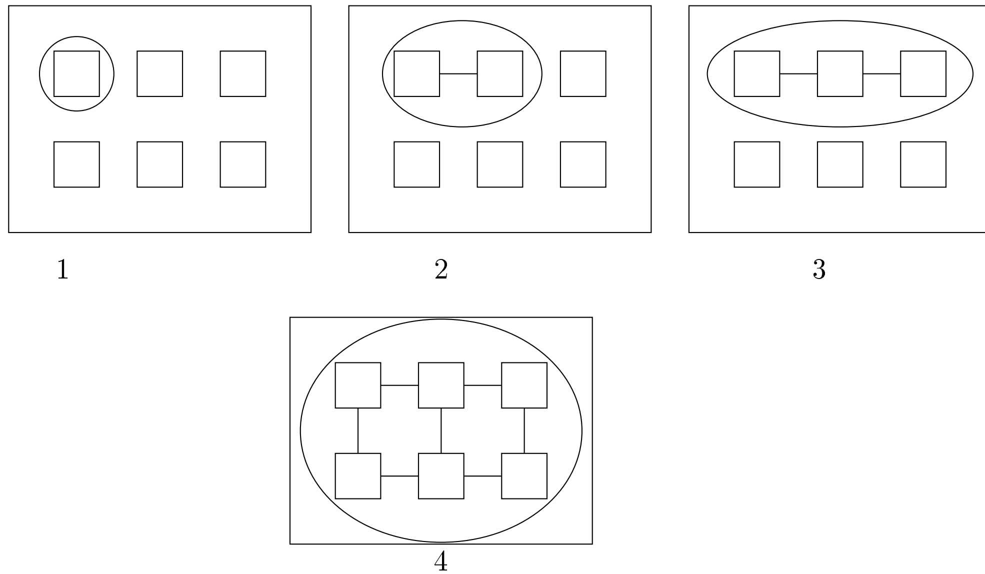
– Contraction d'un site :

$$\begin{array}{l}\{[sched = (\!|(Ag : s, state), \ldots|\!), sched' = (\!|(Ag' : s', state')|\!), \ldots], E, M\} \\ \rightsquigarrow \{, [sched = (\!|(Ag : s, state), (Ag' : s', state'), \ldots|\!), \ldots], E, M\}\end{array} \quad (60)$$

F.1 Migration automatique

$$\{[\ldots, sched = \llparenthesis (Ag : s, state), \ldots \rrparenthesis, sched' = \llparenthesis (Ag' : s', state'), \ldots \rrparenthesis, \ldots], E, M\}$$
$$\rightsquigarrow \{[\ldots, sched = \llparenthesis (Ag : s, state), (Ag' : s', state'), \ldots \rrparenthesis, sched' = \llparenthesis \ldots \rrparenthesis, \ldots], E, M\} \quad (61)$$

Pour mieux comprendre la fonctionnement d'expantion et contraction, on a decidé de les illusterer. vous trouvez l'images ci-desous.



Ici, on a un site et six cores où on a associé a chaque core un scheduler.

F.2 Contraction et extension en images

G Typage

– Valeurs

$$Ag, \Gamma \vdash v : \tau \tag{62}$$

– Creation de référence

$$\frac{Ag, \Gamma \vdash e : \tau_1 \quad Ag, \Gamma \cup x : \texttt{ref}_{Ag}\, \tau_1 \vdash s : \tau_2}{Ag, \Gamma \vdash \texttt{let}\, x \;=\; e\, \texttt{in}\, s\, \texttt{end} : \tau_2}\; l\ is\ fresh \tag{63}$$

– Accès à une référence

$$\frac{Ag, \Gamma \vdash e : \texttt{ref}_{Ag}\, \tau}{Ag, \Gamma \vdash !e : \tau} \tag{64}$$

– Affectation à une référence

$$\frac{Ag, \Gamma \vdash e_1 : \texttt{ref}_{Ag}\, \tau \quad Ag, \Gamma \vdash e_2 : \tau}{Ag, \Gamma \vdash e_1 := e_2 : \texttt{unit}} \tag{65}$$

– Appel de fonction

$$\frac{Ag, \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad f : \ \overrightarrow{\tau} \rightarrow \tau'}{Ag, \Gamma \vdash f(\overrightarrow{e}) : \tau'} \tag{66}$$

– Séquence

$$\frac{Ag, \Gamma \vdash s_1 : \tau_1 \quad Ag, \Gamma \vdash s_2 : \tau_2}{Ag, \Gamma \vdash s_1 ; s_2 : \tau_2} \tag{67}$$

– if

$$\frac{Ag, \Gamma \vdash e : bool \quad Ag, \Gamma \vdash s_1 : \tau \quad Ag, \Gamma \vdash s_2 : \tau}{Ag, \Gamma \vdash \texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2 : \tau} \tag{68}$$

– Repeat

$$\frac{Ag, \mathit{\Gamma} \vdash e : \texttt{int} \;\; Ag, \mathit{\Gamma} \vdash s : \tau}{Ag, \mathit{\Gamma} \vdash \texttt{repeat}\; e \;\texttt{do}\; s \;\texttt{end} : \texttt{unit}} \tag{69}$$

– While

$$\frac{Ag, \mathit{\Gamma} \vdash e : \tau \;\; Ag, \mathit{\Gamma} \vdash s : \tau}{Ag, \mathit{\Gamma} \vdash \texttt{while}\; e \;\texttt{do}\; s \;\texttt{end} : \texttt{unit}} \tag{70}$$

– Exécution parallèle

$$\frac{Ag, \mathit{\Gamma} \vdash s_1 : \tau \;\; Ag, \mathit{\Gamma} \vdash s_2 : \tau}{Ag, \mathit{\Gamma} \vdash s_1 \,||\, s_2 : \tau} \tag{71}$$

– Launch

$$\frac{Ag, \mathit{\Gamma} \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad e = (ev, e_1, ..., e_n) \;\; ev \in \mathit{Events} \;\; m : \;\; \overrightarrow{\tau} \rightarrow \tau}{Ag, \mathit{\Gamma} \vdash \texttt{launch}\; m(\overrightarrow{e}) : \texttt{unit}} \tag{72}$$

$$\frac{Ag, \mathit{\Gamma} \vdash ev : \mathit{event}_{Ag}}{Ag, \mathit{\Gamma} \vdash \texttt{generate}\; ev \;\texttt{with} : \texttt{unit}} \tag{73}$$

– Await

$$\frac{Ag, \mathit{\Gamma} \vdash ev : \mathit{event}_{Ag}}{Ag, \mathit{\Gamma} \vdash \texttt{await}\; ev : \texttt{unit}} \tag{74}$$

– Watching

$$\frac{Ag, \mathit{\Gamma} \vdash ev : \mathit{event}_{Ag} \;\; Ag, \mathit{\Gamma} \vdash s : \tau}{Ag, \mathit{\Gamma} \vdash \texttt{do}\; s \;\texttt{watching}\; ev : \texttt{unit}} \tag{75}$$

– Drop

$$\frac{Ag', \Gamma \vdash s : \tau}{Ag, \Gamma \vdash \texttt{drop}\ s\ \texttt{in}\ Ag' : :\texttt{unit}} \tag{76}$$

– Création d'agent

$$\frac{\Gamma' = \Gamma \cup x_i : \tau_i \;\; Ag', \Gamma' \vdash s : \tau'}{Ag, \Gamma \vdash \texttt{letagent}\ Ag'_{site}\{\ldots, x_i = e_i, \ldots\}\ \texttt{with}\ s\ \texttt{end} : \texttt{unit}} \tag{77}$$

H Exemple

Voici deux exemples simples écrits dans notre modèle. Le premier est un programme simple qui définie deux locations mémoire (x et y) et il essaie d'y accéder plus tard. La sémantique utilisée est correcte et il est correctement typable. On ne verra que l'évalution de sa sémantique.

Par contre, dans le deuxième exemple, même si le sémantique est correcte, on crée un data-race. Le typage on interdit la compilation du programme. Exemple 1 :

```
drop
  letagent Ag with
    let x = ref 1 in
       !x
    end
    ||
    let y = ref 2 in
       !y
    end
in site
```

Voici le typage de notre exemple :

$$\cfrac{\cfrac{\cfrac{site : Ag, \emptyset \vdash \mathit{1} : int \quad \cfrac{site : Ag, \{x\!: \texttt{ref}_{Ag}\ int\} \vdash x : \texttt{ref}_{Ag}\ int}{site : Ag, \{x\!: \texttt{ref}_{Ag}\ int\} \vdash !x : \texttt{unit}}}{site : Ag, \emptyset \vdash \texttt{let}\ x\ =\ ref\ \mathit{1}\ \texttt{in}\ !x\ \texttt{end} : \texttt{unit}} \quad \cfrac{(meme\ qu'autre\ branche)}{site : Ag, \emptyset \vdash \texttt{let}\ y\ =\ ref\ \mathit{2}\ \texttt{in}\ !y\ \texttt{end} : \texttt{unit}}}{site : Ag, \emptyset \vdash \texttt{let}\ x\ =\ ref\ \mathit{1}\ \texttt{in}\ !x\ \texttt{end} \,||\, \texttt{let}\ y\ =\ ref\ \mathit{2}\ \texttt{in}\ !y\ \texttt{end} : \texttt{unit}}}{\emptyset, \emptyset \vdash \texttt{drop}\ \texttt{letagent}\ Ag\ \texttt{with}\ \texttt{let}\ x\ =\ ref\ \mathit{1}\ \texttt{in}\ \texttt{print}\ !x\ \texttt{end} \,||\, \texttt{let}\ y\ =\ ref\ \mathit{2}\ \texttt{in}\ \texttt{print}\ !y\ \texttt{end}\ \texttt{end}\ \texttt{in}\ site : :\texttt{unit}}$$

Exemple 2 :

```
drop
  letagent Ag with
    let x = ref 1 in
      drop
        x := 2
      in site:Ag'
    end
in site
```

Pour être capable de typer le script on doit être capable de typer :

$$site : Ag, \emptyset \vdash \texttt{let}\, x \; = \; ref \;\; 1 \; \texttt{in}\, \texttt{drop}\, x := 2 \; \texttt{in}\, site : Ag' \, \texttt{end} :?$$

et pour pouvoir typer cela on a besoin du typage de :

$$site : Ag, \{x : ref_{Ag} int\} \vdash \texttt{drop}\, x := 2 \; \texttt{in}\, site : Ag' :?$$

et donc il faut pouvoir typer :

$$site : Ag', \{x : ref_{Ag} int\} \vdash x : \texttt{ref}_{Ag'} int$$

on est arrivé à une contradiction ($x : ref_{Ag}int \neq x : ref_{Ag'}int$). Donc le programme n'est pas correctement typable.

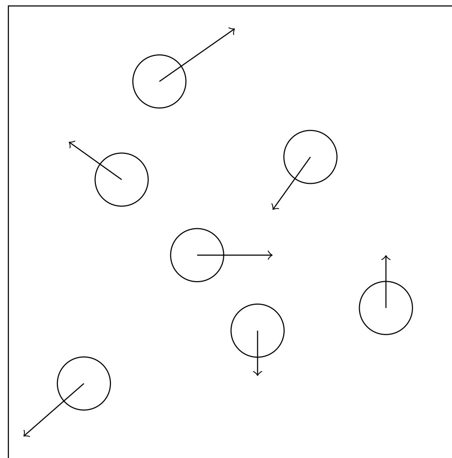
H.1 Collision des billes

L'exemple le plus parlant dans notre cas est la collision des billes. Dans cet exemple on peut voir tout les entités et avoir une idée générale de comment et où utiliser les differents types d'entités qu'on a présenté (script, agent, etc.).

Dans le cas des collision des billes chaque bille a besoin de savoir où elle se situe et dans quelle direction elle doit faire son mouvement et puis s'afficher. chaqu'un de ces comportement peut etre vu comme un script. Donc un bille sera compose de trois script :

– Affichage graphique
– Calcule la trajectoire
– Annoncé son emplacement

et une bille sera un agent composé de ces trois scripts paralléles. toutes les billes seront dans le même site et chaqu'une d'entre elle annonce son emplacement à l'aide d'un événement (*generate*). A la fin de l'instant tous les billes collectent ces informations (*getAll*) pour savoir s'il y aura une collision au début de l'instant suivant et changer la trajectoire dans le cas nécessaire :



I Preuve de correction de typage

Pour prouver que notre typage est correct il suffit de montrer que la proprieté recherché, l'absence de data-race, est vérifiée.

D'arbod on a besoin de faire quelque rappels :

Agent : un agent est un script (un ensemble de scripts parallèle) qui a sa propre mémoire et en cas de migration il ne change pas d'état.

Mémoire : la mémoire dans notre modèle est un ensemble disjoint de mémoires qui appartiennent à chaque agent :

$$M = M_{Ag_1} \oplus ... \oplus M_{Ag_n}$$

Data-race : un data-race se produit quand deux entités differentes essaient d'accèder à la même donnée. Dans notre modèle, un data-race ne doit pas se

produire. On a des mḿoire disjointes et donc les scripts n'appartenant pas à un agent ne doivent pas accéder à sa memoire.

Preuve : Pour prover que notre système de type est correct il suffit de montrer qu'aucun data-race ne peut se produire dans un programme correctement typé.

Les seules instruction qui accèdent à la mémoire sont : $!e$, $e_1 := e_2$ et $this.e$. Dans le cas de $this.e$ il n'y a aucun problème vu qu'on essaie de regarder les champs de notre agent et si la demande est incohérent alors on retourne *unit*.

Il nous reste à vérifier $e_1 := e_2$ et $!e$. Dans les deux cas, on vérifie que cette location existe dans l'environement de typage. S'il n'y est pas alors le programme est rejeté et il n'est pas correctement typé. Dans le cas contraire, grâce à notre environement de typage (Γ) on ne peut pas avoir des doublons donc la location trouvée ne peut être qu'unique. Soit cette location appartient à l'agent dans lequel notre script est en train de s'exécuter alors on a vérifié qu'il n'y a pas de data-race. S'il appartient à un autre agent alors il n'est pas typable.

J Preuve d'existence de point fixe au cours du même instant

Pour prover l'existence de point fixe au cours d'un instant, il suffit de prover qu'il n'existe pas de programme qui peut nous empêcher de passer à l'instant suivant. Pour qu'un programme puisse nous empêcher de passer à l'instant suivant il faut que son exécution dure à l'infini. Les seuls cas où l'exécution d'un programme peut durer sont : *while*, $launch\ m(\overrightarrow{e})$.

Dans le cas de $launch\ m(\overrightarrow{e})$, on est assuré qu'il nous rendra la main au bout d'un temps fini à l'aide de FunLoft. FunLoft vérifie qu'aucun de nos modules ne fait des boucles instantanée et donc ne peut continuer à s'exécuter à l'infini au cours du même instant.

Dans le cas de *while*, par la sémantique on est assueré qu'on ne peut pas avoir des boucles instantanée (le corps de while ne peut être exécuté qu'une fois au plus dans l'instant).

Donc on a réussi à montrer que notre modèle admet un point fixe au cours d'un instant.

Table des matières

1 contexte **3**
1.1 Introduction 3
1.2 Historique de controverse 4

2 FunLoft **5**
2.1 Principes 5
2.2 Fonctionnement 5
2.3 Contrôle des ressources et réactivité 6
2.4 Utilisation des ressources(schedulers) 6
2.5 Description informelle du modèle 6
2.6 Description informelle du langage 7
2.7 Les fonctions et les modules 8
2.8 Mémoire 8
2.9 Script, Agents, Sites 9
2.10 Atomicité 9
2.11 Evénements 9
2.12 Les vérifications statiques 10
2.13 Syntaxe 10
2.13.1 Valeurs 10
2.13.2 Expressions 11
2.13.3 Scripts 11
2.14 Sémantique 12
2.15 Exécution d'un site 13
2.15.1 Absorbtion d'un script ou agent 14
2.15.2 Exécution d'un site 14
2.16 Implémentation 16
2.16.1 Exécution d'un site avec des schedulers 16
2.16.2 Expansion et contraction d'un site 17
2.17 Typage 18
2.18 Preuve de correction du typage 19
2.19 Preuve d'existence d'un point fixe 19

3 Conclusion **20**
3.1 Bilan 20
3.2 Perspective 20

A Syntaxe 23
A.1 Syntaxe des expressions 23
A.2 Syntaxe des Scripts 23

B Sémantique 23
B.0.1 Expressions 23
B.0.2 Scripts 24

C Absorbtion d'agents et de scripts 28

D Exécution d'un site au point de vue de l'utilisateur 29
D.1 Fin d'instant 29

E Implementation 29
E.1 Exécution d'un site 30

F Expansion et contraction d'un site 30
F.1 Migration automatique 31
F.2 Contraction et extension en images 31

G Typage 31

H Exemple 34
H.1 Collision des billes 35

I Preuve de correction de typage 36

J Preuve d'existence de point fixe au cours du même instant 37