UNIVERSITÉ D'ORLÉANS

Mathématiques, Informatique, Physique Théorique et Ingénierie des Systèmes

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Jordan Ischard

soutenue le : **15 octobre 2025**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

Sémantique de langages de programmation fonctionnels réactifs avec effets

THÈSE DIRIGÉE PAR :

M. LOULERGUE Frédéric	Professeur, Université d'Orléans
M. CHOUQUET Jules	MCF, Université d'Orléans (Co-encadrant)
M. DABROWSKI Frédéric	MCF HDR, Université d'Orléans

RAPPORTEURS :

M. POUZET Marc	Professeur, ENS Paris
M. GAVA Frédéric	Professeur, Université de Paris Est

JURY :

M. LIMET Sébastien	(Président) - Professeur, Université d'Orléans
Mme PICARD Célia	Enseignante-Chercheuse, ENAC
M. BOURKE Timothy	Chargé de recherche, INRIA

Remerciements

Le travail présenté dans ce manuscrit est le résultat de 4 années de doctorat. Comme tout projet de longue durée, il y a eu des hauts et des bas d'un point de vue recherche et moral. J'aimerais donc remercier toutes les personnes qui m'ont soutenu.

Tout d'abord, merci à mon encadrement, Frédéric Loulergue, Frédéric Dabrowski et Jules Chouquet, de m'avoir proposé ce sujet de thèse et m'avoir soutenu scientifiquement et moralement durant l'entièreté de mon doctorat.

Plus précisément, merci à Frédéric Loulergue pour son aide dans les démarches administratives, son partage d'expérience pour la suite de mon parcours professionnel et de m'avoir proposé de travailler sur le projet SyDPaCC avec lui.

Merci à Jules Chouquet pour le temps et les connaissances qu'il m'a offerts. Il m'a été d'une grande aide pour me mettre à niveau sur les parties les plus théoriques, avec lesquelles je n'étais pas familier.

Merci à Frédéric Dabrowski pour ses suggestions et ses conseils avisés sur le travail que je propose dans cette thèse. Je tiens spécialement à le remercier, car c'est grâce à lui et Wadoud Bousdira que j'ai eu l'opportunité de rentrer dans le monde de la recherche.

Ensuite, je tiens à faire un remerciement général au LIFO et surtout à l'équipe LMV dans laquelle j'ai été chaleureusement accueilli durant mon stage de L3 et qui est ensuite devenue une part de ma vie très importante.

Merci à mes collègues et amis doctorants ou non-permanents (Aymeric Beauchamp, Térence Clastres, Maël Dumas, Joanne Dumont, Sofiane Elguendouze, Silvia Federzoni, Mathieu Guilbert, Florian Groult, Nicolas Hiot, Darine Rammal, Etienne Lehembre, Thibaut Martinet, et les autres) pour les discussions, les petits-déjeuners et les sorties en groupe.

Merci mes co-bureaux Florian Groult et Nicolas Hiot pour l'aide qu'ils m'ont apportée et de m'avoir écouté durant mes long monologues sur une potentielle solution miracle.

Merci à ma famille qui, bien que très éloignée du domaine de l'informatique, a toujours cru en ma réussite.

Merci à Mathieu Guilbert et Mathis Querault, qui sont mes amis depuis la première année de licence, ainsi qu'à Anthony Perez et à Anaïs Lefeuvre-Halftermeyer, pour le soutien émotionnel, les bières et les activités sportives. Merci à Wadoud Bousdira de m'avoir accepté en stage de L3 et pour toutes nos discussions. Et pour finir, merci à Joanne de partager ma vie et d'avoir relu cette section.

Sommaire

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

L'informatique est omniprésente dans notre quotidien, téléphone portable, ordinateur, climatisation, feu de circulation et bien d'autres encore. Chaque champ d'application a des besoins spécifiques qui mènent à la création de langages de programmation dit *dédiés* ou de bibliothèques dans des langages de programmation dit *généraux*. Dans le cadre de cette thèse, le champ d'application est celui des systèmes ou programmes *réactifs*. Un système ou programme dit *réactif* [52] est en perpétuelle interaction avec son environnement : il *réagit* aux stimulus extérieurs, souvent nommées des *signaux*, en modifiant son comportement interne ainsi qu'en renvoyant des informations à l'extérieur. Ce genre de système est assez commun : tout ce que nous nommons *système embarqué*, *système critique* ou encore *domotique* sont presque toujours des systèmes réactifs. Prenons l'exemple d'un thermostat qui contrôle un unique radiateur, voir la figure 1.1. Il adapte la température d'un logement en fonction de la température courante et d'un programme défini soit par défaut, soit par l'utilisateur. Le capteur thermique du thermostat échantillonne la température et la fournit au programme qui, à son tour, informe le radiateur du maintien ou de l'inversion de son état.

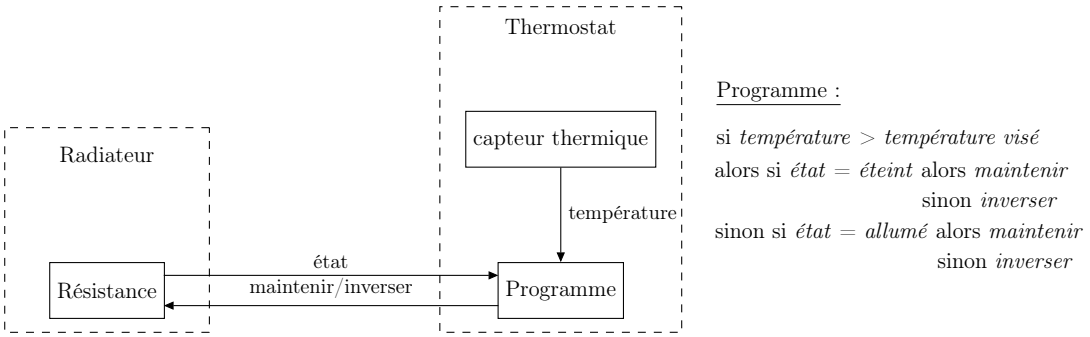


FIGURE 1.1 – Schéma d'un thermostat avec un radiateur. Le pseudo-code associé est fourni à droite du schéma.

Le développement de tels programmes dans un langage général est possible, mais demande au développeur de gérer lui-même toutes les spécificités du champ d'application, ce qui est difficile et entraine des dysfonctionnements du programme. Par exemple, dans un système réactif la gestion du temps, de l'échantillonnage (c'est-à-dire la récupération d'une information qui change dans le temps) des stimulus extérieurs et la propagation du changement sont des points sensibles qui peuvent mener à des inconsistances, d'où la création de langages dédiés qui prennent en charge ces points et propose des primitives (opérateurs et/ou structures de données prédéfinis) pour que le développeur se concentre uniquement sur le cas d'usage ou les spécificités fonctionnelles.

Les premiers langages dédiés à la programmation de systèmes réactifs sont des langages dit *synchrones* qui ont vu le jour dans les années 70. Les plus anciens communément cités sont ESTEREL [12], LUSTRE [22] et SIGNAL [50]. Conceptuellement, un programme synchrone s'abstrait du temps réel et découpe son cycle de vie en *instants logiques*. À chaque début d'instant, l'environnement est échantillonné puis le programme est exécuté ce qui produit finalement des résultats renvoyés à l'environnement et potentiellement une modification de l'état interne du programme. En résumé, un instant représente une réaction complète du programme aux entrées. L'hypothèse faite ici, nommée l'*hypothèse synchrone*, est que la réaction du programme sera assez courte pour être considérée *atomique* par l'environnement, c'est-à-dire qu'il n'est pas possible d'interférer avec son exécution.

Les langages dits synchrones [12, 18, 19, 20, 22, 23, 39, 50, 72, 91, 97] ont souvent misé sur une limitation de l'expressivité du langage dans le but de garantir des propriétés plus forte sur la sémantique du langage, c'est-à-dire son comportement, comme borner le temps d'exécution dans l'instant logique. Ce genre de propriétés sont cruciales dans le cas de système temps-réel, c'est-à-dire lorsque le temps de réponse doit être minimal. Cependant, les concepts synchrones se transportent naturellement dans d'autres cas d'utilisation (applications web, jeux vidéos, interfaces graphiques, etc) qui nécessitent moins de garanties, mais plus d'expressivité. Sur cette observation, C. Elliott et P. Hudak proposa en 1997 un langage dédié à la programmation d'interfaces graphiques nommé FRAN [42] signifiant « Functional Reactive Animation ».

Un programme FRAN a un *comportement* défini via des termes du langage, comportement qui peut évoluer via des *évènements*. Ce langage est implémenté dans HASKELL sous la forme d'une bibliothèque et définit ses concepts de *comportements* et d'*évènements* via des *monades*. Ce langage est souvent cité comme l'ancêtre du paradigme fonctionnel réactif (FRP). En effet, de nombreux langages ont découlé de celui-ci [4, 30, 32, 43, 56, 81, 99, 100, 101, 102, 104].

Bien que puissant, FRAn souffre d'un défaut de conception majeur. En effet, la liberté d'interaction avec l'historique proposée dans ce langage provoque des *fuites mémoires*. Il est possible de récupérer, dans FRAn, une information arbitrairement loin dans le passé pour calculer le comportement courant. Par conséquent, toutes les informations produites durant la période entre celle récupérée et l'actuelle sont stockées inutilement. Une cause avancée est le statut des comportements et évènements qui sont des citoyens de première classe, c'est-à-dire qu'ils peuvent être pris en arguments d'une fonction.

P. Hudak, A. Courtney, H. Nilsson et J. Peterson proposèrent en 2002 le langage Yampa [56]. Dans ce langage, lui aussi embarqué sous la forme d'une bibliothèque Haskell, les informations extérieures sont transportées via un unique signal qui ne peut être utilisé que via des *fonctions de signal*. Intuitivement, ces fonctions de signal sont des composants logiciels qui se combinent et s'appliquent au signal d'entrée pour en modifier ces valeurs point-à-point. L'historique d'un signal, c'est-à-dire ses valeurs depuis le début de l'exécution, n'est plus accessible, cependant il est toujours possible d'accumuler des informations dans le temps grâce à un combinateur prédéfini. (la section 3.3 définit plus précisément ce modèle de programmation).

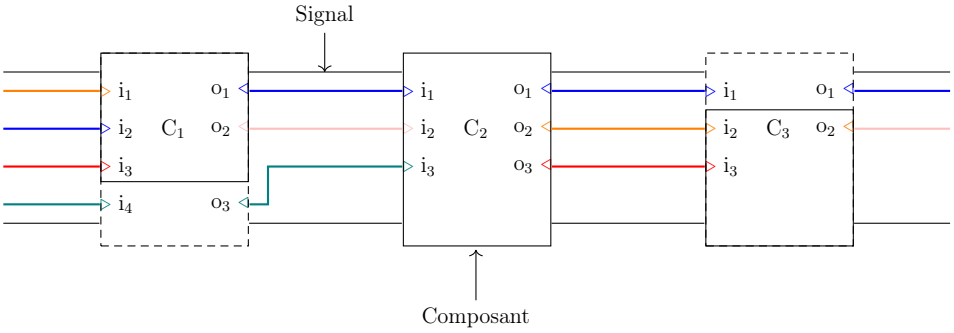


Figure 1.2 – Représentation simplifiée d'un programme Yampa à trois composants.

Dans la figure 1.2 ci-dessus, nous donnons un exemple simplifié d'un programme Yampa avec trois composants. Chaque information extérieure est représentée par un fil coloré qui, une fois regroupés, forment le signal. Chaque composant récupère l'*ensemble* du signal, calcul son potentiel état interne et retourne un nouveau signal pour le composant suivant. Lorsque des entrées/sorties sont dans la partie en pointillée du composant, cela veut dire que le composant ignore *consciemment* la valeur d'entrée (nous considérons ici qu'il transmet uniquement l'information). Dans notre exemple, C_1 ignore l'entrée i_4 et C_3 ignore l'entrée i_1. Ignorer une valeur d'entrée est nécessaire au bon fondement de Yampa et,

est donc possible via un combinateur prédéfini. En effet, cela permet d'appliquer une fonction de signal sur *une partie* des informations d'entrées tout en prenant l'*entièreté* du signal en entrée.

Malgré son applicabilité dans divers domaines [56, 80], la gestion des entrées/sorties ainsi la maintenance des programmes restent une faiblesse du langage. En effet, chaque nouvelle information extérieure ou modification d'une fonction de signal pertube la composition du programme. Dans la figure 1.2, si C_3 nécessite une information supplémentaire, donc un fil coloré, alors il faudra que C_1 et C_2 soient modifiés afin de transmettre l'information jusqu'à C_3. La figure 1.3 représente l'impact de la modification décrite précédemment sur la représentation simplifiée de la figure 1.2.

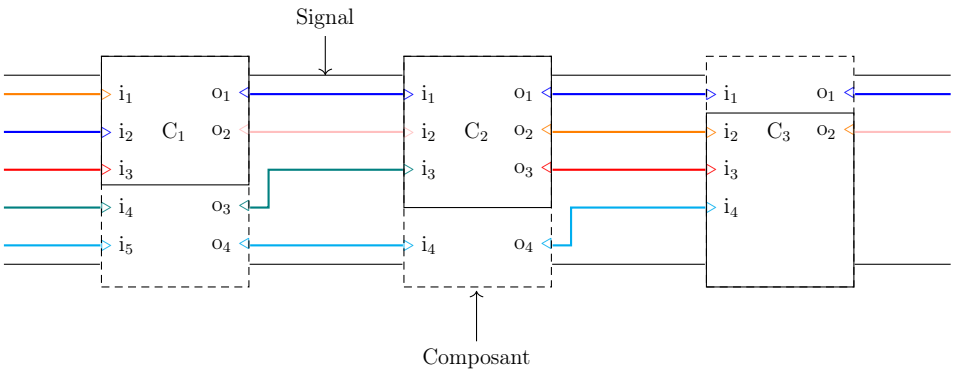


FIGURE 1.3 – Représentation simplifiée de la figure 1.2 après modification.

La problématique sur laquelle cette thèse se concentre est la construction d'un langage fonctionnel réactif dont la sémantique permet des ajouts et des modifications en minimisant l'effort de leur implémentation au programme.

L'intuition est d'utiliser des effets de bords, plus précisément des interactions avec la mémoire, afin de « cacher » les fils supplémentaires au reste du programme. Cependant, cet ajout n'est pas trivial dans un langage où il est nécessaire de garder la compositionnalité des fonctions de signal et l'hypothèse synchrone valide. Dans la littérature, la première proposition répondant à cette problématique est le langage WORMHOLES [102,103]. Proposé par D. Winograd-Cort et P. Hudak en 2012, ce langage de programmation a d'abord été implémenté en HASKELL [103] puis une formalisation papier a été faite [102]. Le concept est simple : en plus des combinateurs de YAMPA, le langage introduit la possibilité d'interagir avec l'environnement à travers un nouveau combinateur nommé *rsf* ainsi que la possibilité de créer des « références locales », via une paire de ressources, dans le programme grâce à un combinateur nommé *wormhole*. Une interaction avec l'environnement est définie comme une *ressource*. L'utilisation des ressources est restreinte à une

fois par instant logique. Cette contrainte est cohérente avec l'hypothèse synchrone qui suppose que l'environnement voit l'exécution du programme comme atomique. Une utilisation sans restriction des ressources pourrait donc briser cette hypothèse à travers la modification de la mémoire via l'utilisation du combinateur *rsf*. Cependant, elle restreint plus que nécessaire une partie des ressources. En effet, les ressources utilisées comme « références locales » ne sont pas visibles de l'extérieur et, par conséquent, n'impactent pas l'hypothèse synchrone. Nous avons vu ici une possibilité de relaxation du langage.

Avant toute modification du langage existant, WORMHOLES, nous souhaitons garantir que les propriétés attribuées au langage, dans la formalisation papier [102], sont effectivement respectées. Les démonstrations papier ont le défaut de l'erreur humaine, il est possible même pour un expert de faire une erreur d'inattention qui peut mener à une démonstration fausse. De plus, la relecture d'une démonstration qui peut faire plusieurs pages est ardue et peu aussi mener à des mégardes. Nous avons donc décidé de mécaniser, c'est-à-dire formaliser avec l'aide d'un assistant de preuve, le langage en utilisant ROCQ. Cette mécanisation nous a permis de détecter des problèmes dans la formalisation papier initiale tels que la perte de la progression ou l'échappement de la portée syntaxique dans le système de types. Par conséquent, nous avons proposé une variante de WORMHOLES dans le chapitre 4 qui résout les problèmes que nous avons trouvé et qui satisfait toutes les propriétés du langage énoncé dans l'article original.

Une fois la mécanisation faite, nous avons décidé de relâcher la contrainte d'utilisation unique des ressources locales dans l'instant. Pour ce faire, nous avons défini, dans le chapitre 5, un langage nommé MOLHOLES (qui est la contraction de taupe (mole) et trous (holes), là où WORMHOLES fait référence aux trous de ver (worm)). Ce langage divise les ressources en trois groupes : *entrée*, *sortie*, *interne* et introduit deux combinateurs inspirés de *rsf* : *get* et *set*. Nous avons démontré que l'utilisation répétée d'une ressource interne dans un unique instant ne provoquait pas de problème de préservation de typage ou de progression dans l'exécution. De plus, nous avons défini une équivalence entre un programme MOLHOLES et un programme YAMPA (plus précisément un sous ensemble approprié de YAMPA).

1.1 Contributions

Dans ce manuscrit, nous présentons trois contributions scientifiques :

1. Proposition d'une variante [59] de WORMHOLES qui corrige les problèmes de la formalisation originale avec une mécanisation en ROCQ [60] ;
2. Présentation d'un nouveau langage fonctionnel réactif avec effets nommé MOLHOLES [36] qui étend les concepts de WORMHOLES ;
3. Création d'une bibliothèque ROCQ nommée `DeBrLevel` [58] pour gérer les identifiants via les niveaux de De-Bruijn.

1.2 Structure du manuscrit

Dans un premier temps, nous introduisons l'ensemble des concepts et des notations utilisées dans la suite du manuscrit dans le chapitre 2. Puis, nous proposons un état de l'art de la programmation réactive dans le chapitre 3. Ce chapitre débute sur le paradigme synchrone suivi par le paradigme fonctionnel réactif et fini par le paradigme fonctionnel réactif avec flèches. Nous introduisons par la même occasion les langages YAMPA (plus précisément le sous-ensemble que nous utiliserons) et WORMHOLES. Ensuite, nous définissons, dans le chapitre 4, la variante de WORMHOLES qui corrige les erreurs trouvées lors de la mécanisation de la version originale. Enfin, nous présentons notre langage MOLHOLES dans le chapitre 5. Ce chapitre contient une définition formelle d'un sous ensemble minimal de YAMPA nommé YAMPACORE ainsi que la définition de MOLHOLES : la syntaxe, les sémantique statique et dynamique. Pour finir, nous démontrons à la fin du chapitre l'équivalence entre un programme MOLHOLES et un programme YAMPACORE. Finalement, nous concluons dans le chapitre 6 en rappelant les contributions et en discutant des perspectives de recherche. En annexe, dans le chapitre A, nous présentons la bibliothèque ROCQ nommée `DeBrLevel` que nous avons développé dans le cadre de la mécanisation de WORMHOLES. Elle permet de gérer les identifiants via les niveaux de De-Bruijn.

Chapitre 2

Préliminaires

Sommaire

Dans ce chapitre, nous fournissons les notions essentielles à la bonne compréhension des contributions de ce manuscrit. Premièrement, nous fournissons les notations utilisées dans la suite comme les fonctions, les relations, etc. Deuxièmement, nous introduisons le lambda-calcul historiquement, tout en donnant la syntaxe et une sémantique opérationnelle. Troisièmement, nous fournissons les définitions ou concepts des *catégories* requis pour le chapitre 5. Plus précisément, nous explicitons la définition de catégorie, catégorie cartésienne ainsi que le concept de foncteurs, monades et flèches. Finalement, nous présentons l'assistant de preuve Rocq, son fonctionnement, sa syntaxe et quelques fonctionnalités plus poussées (modules et interfaces).

2.1 Notations

2.1.1 Fonctions/Relations

Fonctions Une fonction établit pour tout élément d'un ensemble d'entrée, appelé *domaine*, un résultat appartenant à un ensemble de sortie, appelé *codomaine*. La fonction f qui prend des éléments du domaine A et produit des éléments du codomaine B est noté $f : A \to B$. Les deux fonctions `dom` et `cod` retournent respectivement le domaine et le codomaine d'une fonction prise en paramètre. En général lorsque nous définissons une fonction, nous définissons une fonction *totale*, c'est-à-dire qu'il existe un résultat pour toute entrée du domaine. Cependant, il existe aussi des fonctions *partielles* qui peuvent contenir des résultats indéfinis. Nous notons $f : A \rightharpoonup B$ une fonction partielle qui prend des éléments du domaine A et produit des éléments du codomaine B.

Relations En théorie des ensembles, une relation est un sous-ensemble d'un produit cartésien. Par exemple, « être strictement supérieur à », donc l'opérateur infixe $(>)$, est un sous-ensemble du produit cartésien $\mathbb{N} \times \mathbb{N}$. Une relation R sur un ensemble X peut être :

— *réflexive*, c'est-à-dire que pour tout $x \in X$ nous avons $x \, R \, x$;

— *symétrique*, c'est-à-dire que pour tout $x, y \in X$ si $x \, R \, y$ alors $y \, R \, x$;

— *transitive*, c'est-à-dire que pour tout $x, y, z \in X$ si $x \, R \, y$ et $y \, R \, z$ alors $x \, R \, z$.

Par exemple, une relation d'équivalence est réflexive, symétrique et transitive. Dans le chapitre 4, nous utilisons $R^\star$ la clôture réflexive transitive de la relation R comme suit :

$$\forall x, y \in X,\ x \, R \, y \implies x \, R^\star \, y \tag{2.1a}$$

$$\forall x, y, z \in X,\ x \, R \, y \text{ et } y \, R^\star \, z \implies x \, R^\star \, z \tag{2.1b}$$

2.1.2 Structures

Dictionnaires Un dictionnaire est une structure de donnée composée de *clés* et de *valeurs*. Une clé x est liée à une valeur e et tout ajout d'une nouvelle valeur remplace e pour x. Les contextes et environnements définis dans la suite du manuscrit se basent sur des dictionnaires [1]. Soit D un dictionnaire, nous notons $D[x \mapsto e]$ l'ajout de la clé x associée à e dans un dictionnaire D. Lorsque nous écrivons $D \, x = e$, cela signifie que x est associée à e dans D. Dans le cas où la clé

1. Nous pouvons aussi voir dictionnaire comme une fonction partielle ou comme un sous-ensemble d'un produit cartésien.

x n'est pas présente dans D, nous écrivons $D\,x = \bot$. Nous simplifions la notation $D[x \mapsto e]$ par $[x \mapsto e]$ lorsque D est vide. Les dictionnaires satisfont l'ensemble d'équations 2.2 où $D_1 \equiv D_2$, avec D_1, D_2 deux dictionnaires, signifie que pour toute clé x, $D_1\,x = D_2\,x$.

$$D[x \mapsto e'][x \mapsto e] \equiv D[x \mapsto e] \tag{2.2a}$$
$$D[x \mapsto e'][y \mapsto e] \equiv D[y \mapsto e][x \mapsto e'] \qquad \text{si } x \neq y \tag{2.2b}$$
$$D[x \mapsto v] \equiv D \qquad \text{si } D\,x = v \tag{2.2c}$$

L'équation (2.2a) affirme que l'association clé/valeur précédente est écrasée par la nouvelle. L'équation (2.2b) énonce qu'il est possible de permuter deux ajouts s'ils ne s'appliquent pas sur la même clé. Finalement, l'équation (2.2c) établit qu'un ajout avec la même valeur ne change pas l'état du dictionnaire.

Listes Une liste est une structure avec un ordre et une taille finie, mais non bornée. Tous les éléments contenus dans une liste ont le même type. Nous notons [] la liste vide, $[v]$ la liste contenant un seul élément et $[v_1, \ldots, v_n]$ la liste contenant n éléments.

De plus, nous notons (::) l'opérateur infixe d'ajout d'un élément en tête d'une liste et (++) l'opérateur infixe de concaténation. La valeur à la position n dans une liste l est notée l_n. $\mid l \mid$ retourne la taille de la liste.

Finalement, nous définissons la fonction `splitn` qui prend un entier n et une liste l et retourne une paire de listes (l_1, l_2). La liste l_1 contient les n premiers éléments de l et l_2 le reste. Si $n > \mid l \mid$ alors $l_1 = l$ et $l_2 = []$.

2.1.3 Calcul des séquents

Le calcul des séquents est un système de déduction créé par Gerhard Gentzen [46]. Il permet de formaliser une démonstration via des séquents et des règles. Soit $\mathcal{L}$ un langage de formules, un *séquent* est un couple de listes finies de formules appartenant à $\mathcal{L}$, séparé par le symbole $\vdash$. Par exemple,

$$A_1, \ldots, A_n \vdash B_1, \ldots, B_m$$

est un séquent. Les formules $A_1, \ldots, A_n$ représentent les hypothèses, et $B_1, \ldots, B_m$ les formules vraies sous ces hypothèses. Ce séquent signifie donc que « B_1 est vraie ou … ou B_m est vraie, car A_1 et … et A_n sont vraies ». Une *règle* dans le calcul

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \texttt{R-And} \qquad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}\ \texttt{R-Imp} \qquad \frac{}{A, \Gamma \vdash A}\ \texttt{R-Ax}$$

FIGURE 2.1 – Quelques règles de la logique propositionnelle.

des séquents a la forme suivante :

$$\frac{\text{prémisses}}{\text{conclusion}}\ \texttt{nom-de-la-regle}$$

En haut de la barre verticale se trouvent les prémisses, et en dessous la conclusion. Une règle peut se lire comme suit : « D'après `nom-de-la-regle`, si les prémisses sont valides alors la conclusion l'est aussi ». Cette notation est utilisée pour définir les règles d'un système de types ou encore d'une sémantique opérationnelle, comme dans le chapitre 4. S'il n'y a pas de prémisse alors c'est un axiome. Par exemple,

$$\frac{}{\vdash A}$$

est l'axiome qui stipule que « A est vraie sans condition ». Une démonstration, ou dérivation, est un unique séquent représentant la conclusion où nous appliquons des règles sur les hypothèses.

Exemple 2.1. *Supposons les règles de la figure 2.1 avec* Γ *un ensemble de formules et* A *et* B *sont deux formules logiques. Nous démontrons que* $A \to B \to A \wedge (B \wedge B)$ *est une formule vraie comme suit :*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B, A \vdash A}\ \texttt{R-Ax} \quad \dfrac{\dfrac{}{B, A \vdash B}\ \texttt{R-Ax} \quad \dfrac{}{B, A \vdash B}\ \texttt{R-Ax}}{B, A \vdash B \wedge B}\ \texttt{R-And}}{B, A \vdash A \wedge (B \wedge B)}\ \texttt{R-And}}{A \vdash B \to A \wedge (B \wedge B)}\ \texttt{R-Imp}}{\vdash A \to B \to A \wedge (B \wedge B)}\ \texttt{R-Imp}$$

2.2 Lambda-calcul

Le lambda-calcul [27] est un système formel introduit par Alonzo Church dans les années 30. Intuitivement, il permet de définir des fonctions et des calculs de fonctions dans un système simplifié composé de trois familles de termes : les *variables*, les *abstractions* et les *applications*. Les abstractions peuvent être vues comme des fonctions. Notée $\lambda x.t$ pour x une variable et t un terme, une abstraction *lie* une variable x à un terme t, x agit comme un argument de la fonction et peut apparaitre une multitude, ou zéro, fois dans t. L'application,

notée $t_1\, t_2$ pour deux termes t_1 et t_2, représente le calcul qui applique t_2 à t_1. Si $t_1 = \lambda x.t$ alors t_2 remplacera x dans le résultat du calcul, c'est ce que l'on appelle la *β-réduction.*

Il a été démontré que le lambda-calcul est Turing-complet, c'est-à-dire qu'il possède un pouvoir expressif équivalent à celui des machines de Turing, et il sert de base aux langages de programmation fonctionnels. Dans cette section, nous détaillons le lambda-calcul classique avec sa syntaxe, son évaluation, sa notion d'équivalence puis nous énonçons ce qu'est le lambda-calcul simplement typé, car il fonde les langages de programmation fonctionnels.

Lambda-calcul non typé Le lambda-calcul non typé est une théorie équationnelle qui étudie les fonctions, leur construction et leur comportement applicatif, c'est-à-dire comment le mécanisme d'évaluation fonctionne. Cette théorie se repose sur un ensemble de termes, que nous avons introduit informellement précédemment, dénoté Λ, défini inductivement comme suit :

— $x \in \Lambda$, pour x une variable de X un alphabet ;

— $\lambda x.t \in \Lambda$ si $t \in \Lambda$ et $x \in X$;

— $(t_1\, t_2)$ [2] $\in \Lambda$ si $t_1 \in \Lambda$ et $t_2 \in \Lambda$.

Exemple 2.2. *Les expressions ci-dessous sont des termes du lambda-calcul, généralement nommés lambda-termes.*

$$t_1 \stackrel{def}{=} \lambda x.x\, x$$
$$t_2 \stackrel{def}{=} \lambda x.((\lambda y.x\, y)\, x)$$
$$t_3 \stackrel{def}{=} z\, y$$

Dans un terme, il peut y avoir des variables dites *libres* et des variables dites *liées.* Une variable libre apparaît dans le terme sans être associée à une abstraction, alors qu'une variable liée l'est.

Exemple 2.3. *Soit $t \stackrel{def}{=} \lambda x.((\lambda y.x\, y)\, z)$ est un lambda-terme contenant deux variables liées, x et y, et une variable libre z.*

Pour aborder le calcul d'un lambda-terme, il faut d'abord décrire la fonction de *substitution.* Noté $[x := t]$, avec x une variable et t un terme, la substitution est une fonction qui prend un terme t' et remplace toutes les occurrences de x par t dans t'.

2. Nous omettons, quand cela est possible, les parenthèses dans les termes. Par convention, le parenthésage est associatif à gauche, c'est-à-dire que $x\, y\, z = ((x\, y)\, z)$.

$$[x := t]x = t \tag{2.3a}$$
$$[x := t]y = y \qquad \text{si } x \neq y \tag{2.3b}$$
$$[x := t_1]\lambda y.t_2 = \lambda y.([x := t_1]t_2) \qquad \text{si } x \neq y \tag{2.3c}$$
$$[x := t_3](t_1\, t_2) = ([x := t_3]t_1)\,([x := t_3]t_2) \tag{2.3d}$$

Il est important dans l'équation (2.3c) que la variable x diffère de celle liée à l'abstraction, car sinon il peut y avoir un problème de *capture de noms.*

Exemple 2.4. *Soit* $t_1 \stackrel{def}{=} \lambda x.((\lambda y.x\, y)\, x)$ *et* $t_2 \stackrel{def}{=} y$*. Dans le terme* t_1*,* x *et* y *sont deux variables liées et il n'y a pas de variables libres. Le terme* t_2 *est composé d'une unique variable libre* y*. Si nous tentons d'appliquer* t_2 *à* t_1 *alors nous remplacerons* x *par* y *dans* t_1*, ce qui donnerait le résultat suivant :*

$$t_1\, t_2 = ((\lambda y.y\, y)\, y)$$

La variable libre y *a été capturée par l'abstraction à cause de son nom. Le problème ici est que la substitution a modifié l'état de la variable* y *dans* t_2 *en la faisant passer de libre à liée.*

Pour éviter cela, il est fréquent que l'on raisonne sur des classes d'équivalence de termes et non sur les termes eux-même via l'α-conversion. Cette relation, notée $=_\alpha$, est une relation d'équivalence décrite comme suit :

$$t_1 =_\alpha t_1 \tag{2.4a}$$
$$t_2 =_\alpha t_1 \qquad \text{si } t_1 =_\alpha t_2 \tag{2.4b}$$
$$t_1 =_\alpha t_2 \qquad \text{si } t_1 =_\alpha t_3 \text{ et } t_3 =_\alpha t_2 \tag{2.4c}$$
$$t_1\, t_3 =_\alpha t_2\, t_4 \qquad \text{si } t_1 =_\alpha t_2 \text{ et } t_3 =_\alpha t_4 \tag{2.4d}$$
$$\lambda x.t =_\alpha \lambda y.([x := y]\, t) \qquad \text{si } y \text{ n'apparaît pas liée dans } t \tag{2.4e}$$

Les équations (2.4a) à (2.4c) énoncent que la relation est réflexive, symétrique et transitive, c'est-à-dire que c'est une équivalence. L'équation (2.4d) implique que la relation est une congruence. Finalement, l'équation (2.4e) implique qu'une variable liée peut être renommée.

Exemple 2.5. *Soit* $t \stackrel{def}{=} \lambda x.(x\, y)$ *avec* y *libre dans* t*, nous avons :*

$$\lambda x.(x\, y) =_\alpha \lambda z.(z\, y)$$
$$\lambda x.(x\, y) \neq_\alpha \lambda y.(y\, y)$$

Nous pouvons maintenant redéfinir la substitution sans capture via cette équivalence. Nous préservons la même notation, mais nous remplaçons l'équation (2.3c) par les deux équations ci-dessous :

$$[x := t_1]\,\lambda y.t_2 =_\alpha \lambda y.([x := t_1]\,t_2) \qquad \text{si } x \neq y \text{ et } y \text{ n'est pas libre dans } t_1 \quad (2.5)$$

$$[x := t_1]\,\lambda y.t_2 =_\alpha \lambda z.([x := t_1]\,[y := z]\,t_2) \qquad \text{avec } z \text{ une variable fraîche}^3 \quad (2.6)$$

Finalement, nous donnons, ci-dessous, l'équation (2.7), historiquement nommée la β-conversion, qui exprime l'égalité entre une application et son « évaluation ».

$$(\lambda x.t_1)\,t_2 =_\alpha [x := t_2]\,t_1 \qquad \text{si } x \text{ libre dans } t_2 \quad (2.7)$$

Dans une volonté d'effectivement calculer un lambda-terme, il faut « orienter » les équations proposées précédemment. Dans ce cas, la β-conversion va s'appliquer de la gauche vers la droite ce qui est capturé par la relation suivante :

$$\beta \stackrel{def}{=} \{((\lambda x.t_1)\,t_2, [x := t_1]t_2) \mid t_1, t_2 \in \Lambda \text{ et } x \in X\}$$

Via cette relation, il est possible d'en définir une nouvelle nommée la *réduction*. Cette réduction est une relation binaire, dénotée par l'opérateur infixe $\mapsto_\beta$, qui suit les règles suivantes :

$$t_1 \mapsto_\beta t_2 \qquad \text{si } (t_1, t_2) \in \beta \quad (2.8a)$$

$$t_1\,t_3 \mapsto_\beta t_2\,t_3 \qquad \text{si } t_1 \mapsto_\beta t_2 \quad (2.8b)$$

$$t_3\,t_1 \mapsto_\beta t_3\,t_2 \qquad \text{si } t_1 \mapsto_\beta t_2 \quad (2.8c)$$

$$\lambda x.t_1 \mapsto_\beta \lambda x.t_2 \qquad \text{si } t_1 \mapsto_\beta t_2 \quad (2.8d)$$

Cette relation peut être vue comme une *sémantique opérationnelle à petit-pas*, c'est-à-dire qu'elle exprime étape par étape comment un terme va se réduire. Maintenant qu'il est possible de réduire un terme, il faut définir un terme réduit au maximum. Ce concept est capturé par la *forme normale*. C'est un sous-ensemble de Λ composé de termes qui ne contiennent plus aucune application possible.

Exemple 2.6. *Le terme $\lambda x.x$ est en forme normale, car aucune application n'est possible. Par contre, le terme $\lambda x.(\lambda y.(x\,y))\,z$ n'est pas en forme normale, car nous pouvons appliquer l'équation* (2.8d) *qui produit le terme $\lambda y.(z\,y)$.*

Certains termes ne peuvent atteindre une forme normale. Par exemple, le terme $\Omega \stackrel{def}{=} (\lambda x.(x\,x))\,(\lambda x.(x\,x))$ va se réduire en lui-même à chaque application

3. Une variable est dite fraîche si elle n'est pas du tout utilisée dans le terme.

de l'équation (2.8a), ce qui permet d'introduire un point fixe au lambda-calcul, historiquement nommé Y, défini comme suit :

$$Y \stackrel{def}{=} \lambda f.\,(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

Exemple 2.7. *Soit g une fonction, $(Y\,g)$ a la réduction suivante :*

$$\begin{aligned} Y\,g &= \lambda f.((\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)))\,g \\ &= (\lambda x.g\,(x\,x))\,(\lambda x.g\,(x\,x)) && \textit{équation } (2.8a) \\ &= g\,((\lambda x.g\,(x\,x))\,(\lambda x.g\,(x\,x))) && \textit{équation } (2.8a) \\ &= g\,(Y\,g) \end{aligned}$$

Lambda-calcul simplement typé À cause des termes comme Ω et Y la logique du lambda-calcul permet de définir des paradoxes, comme le paradoxe de Russel [89]. Plus généralement, les termes qui se réduisent à l'infini, comme Ω, sont un problème lorsque nous souhaitons calculer un résultat. Pour pallier cela, Alonzo Church en 1940 proposa le lambda-calcul simplement typé [28]. À savoir qu'un autre paradoxe, nommé le paradoxe de Curry [53] fut trouvé après cela dans le lambda-calcul non typé. Dans la théorie du lambda-calcul simplement typé, à la Curry, l'ensemble de lambda-termes est conservé, mais il est contraint via des types. L'ensemble des types, dénoté $\mathcal{T}$, est défini inductivement comme suit :

— $A \in \mathcal{T}$ pour toute variable de type provenant d'un ensemble infini dénombrable Y ;
— $A \implies B \in \mathcal{T}$ si $A \in \mathcal{T}$ et $B \in \mathcal{T}$.

Le type $A \implies B$, pour $A, B \in \mathcal{T}$, représente le type d'une fonction. A est le type de la variable liée et B celle de son sous-terme. Pour vérifier le type d'un terme, il est nécessaire d'avoir un *contexte* qui associe les variables aux types. Ce contexte, souvent dénoté par Γ, est une fonction partielle de domaine X et codomaine $\mathcal{T}$. Avec cela, nous définissons ce qu'est un terme bien typé. Nous notons $\Gamma \vdash t : A$ la propriété de bon typage de t qui peut se lire « Sachant un contexte Γ, t est associé au type A par les règles de la figure 2.2 ».

$$\frac{\Gamma\,x = A}{\Gamma \vdash x : A}\ \texttt{WT-Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \implies B}\ \texttt{WT-Abs}$$

$$\frac{\Gamma \vdash t_1 : A \implies B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\,t_2 : B}\ \texttt{WT-App}$$

FIGURE 2.2 – Règle de typage du lambda-calcul simplement typé.

Exemple 2.8. *Le point fixe Y du lambda-calcul n'est pas typable dans le lambda-calcul simplement typé, car il contient le terme $(x\,x)$ qui n'est pas typable comme le démontre la dérivation de typage ci-dessous.*

$$\dfrac{\dfrac{[x \mapsto A \implies B] \vdash x : A \implies B \quad [x \mapsto A \implies B] \nvdash x : A}{[x \mapsto A \implies B] \nvdash x\,x : \times}\ \texttt{WT-App}}{\varnothing \nvdash \lambda x.(x\,x) : \times}\ \texttt{WT-Abs}$$

En effet, x ne peut pas être typé comme une fonction de A vers B et en même temps être de type A. Par contre, l'expression $\lambda f.\lambda x.f\,x\,x$ est de type $(A \implies A \implies B) \implies A \implies B$ comme le démontre la dérivation de typage donnée dans la figure 2.3.

Le typage des termes restreint l'expressivité du lambda-calcul en interdisant les termes dont les variables sont à la fois des fonctions et des arguments. Grâce à cela, toute réduction d'une expression bien typée converge vers une forme normale, c'est-à-dire qu'il y a un nombre d'étapes de réduction fini pour atteindre la forme normale.

Il existe un lien entre le lambda-calcul simplement typé et la logique intuitionniste appelé la correspondance de Curry-Howard [44]. Cette correspondance a été plus tard complétée par Lambek [68] qui a proposé une sémantique dénotationnelle du lambda-calcul simplement typé via des catégories cartésiennes closes.

2.3 Théorie des catégories

La théorie des catégories [40] a été mise en place par Samuel Eilenberg et Saunders MacLane en 1942-45, et a pour but d'étudier des structures mathématiques via leurs relations. Cette théorie généralise celle des ensembles. Une catégorie se compose de deux types d'éléments abstraits : des objets et des morphismes. Dans cette section, nous limitons nos explications au minimum nécessaire pour la bonne compréhension du chapitre 5. Nous expliquons ce qu'est une catégorie et certaines familles de catégories. De plus, nous définissons les foncteurs, les monades et les flèches. Cette section se repose sur le livre d'Andrea Asperti, Giuseppe Longo [2] et celui de Benjamin C. Pierce [86]. Certains parallèles sont faits avec le langage Haskell dans cette section. Nous supposons que le lecteur a un minimum de connaissance dans ce langage.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\, f = A \implies A \implies B}{\Gamma \vdash f : A \implies A \implies B}\,\texttt{WT-Var}}{[x : A][f \mapsto A \implies A \implies B] \vdash x : A}\,\texttt{WT-App} \quad \dfrac{[x : A][f \mapsto A \implies A \implies B]\, x = A}{[x : A][f \mapsto A \implies A \implies B] \vdash x : A}\,\texttt{WT-Var}}{([x : A][f \mapsto A \implies A \implies B]) \vdash f\, x : A \implies B}\,\texttt{WT-App} \quad \dfrac{[x : A][f \mapsto A \implies A \implies B]\, x = A}{[x : A][f \mapsto A \implies A \implies B] \vdash x : A}\,\texttt{WT-Var}}{\dfrac{\dfrac{[x : A][f \mapsto A \implies A \implies B] \vdash f\, x\, x : B}{[f \mapsto A \implies A \implies B] \vdash \lambda x.f\, x\, x : A \implies B}\,\texttt{WT-Abs}}{\varnothing \vdash \lambda f.\lambda x.f\, x\, x : (A \implies A \implies B) \implies A \implies B}\,\texttt{WT-Abs}}$$

FIGURE 2.3 – Dérivation de typage du lambda-terme $\lambda f.\lambda x.f\, x\, x$ avec $\Gamma = [x : A][f \mapsto A \implies A \implies B]$.

2.3.1 Catégories

Définition 2.1 (Catégorie)**.** *Une catégorie $\mathcal{C}$ est un regroupement d'objets $\mathcal{C}_{obj}$ et de morphismes $\mathcal{C}_{arr}$. Nous notons $\mathcal{C}(A, B)$ l'ensemble des morphismes*[4] *partant d'un objet A et allant vers un objet B. L'ensemble des morphismes inclut un morphisme identité id_A pour tout objet $A \in \mathcal{C}_{obj}$ et il existe un opérateur de composition $\circ_{ABC} \in \mathcal{C}(A, B) \times \mathcal{C}(B, C) \to \mathcal{C}(A, C)$ pour tout triplet d'objets A, B et C tel que :*

$$id_B \circ f = f = f \circ id_A \qquad \text{(neutre)}$$
$$(h \circ g) \circ f = h \circ (g \circ f) \qquad \text{(associativité)}$$

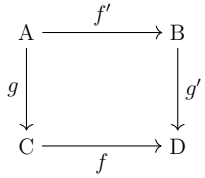
pour tous objets A, B, C, D, et morphismes $f \in \mathcal{C}(A, B)$, $g \in \mathcal{C}(B, C)$ et $h \in \mathcal{C}(C, D)$.

L'ensemble des morphismes identités d'une catégorie $\mathcal{C}$ est dénoté par $Id_{\mathcal{C}}$. Nous notons en majuscule les objets d'une catégorie et en minuscule les morphismes. En général, nous nous autorisons à écrire simplement $\mathcal{C}$ à la place de $\mathcal{C}_{obj}$ ou $\mathcal{C}_{arr}$, quant le contexte le permet afin de gagner en lisibilité.

Exemple 2.9. *La catégorie la plus simple définissable contient un objet et un morphisme, qui est l'identité de l'unique objet. Dans ce cas, la composition est triviale.*

La catégorie **Set** *a pour objets les ensembles et pour morphismes les fonctions. La composition est celles des fonctions.*

En théorie des catégories, les équations sont représentées par des *diagrammes de commutations*. Dans un tel diagramme les nœuds sont des objets et les flèches sont des morphismes (donc elles sont orientées). Un diagramme *commute* si, pour toute paire de nœuds x, y, tous chemins entre x et y sont égaux. Par exemple, dire que le diagramme suivant commute revient à dire que $g' \circ f' = f \circ g$.



Les lois d'identité et d'associativité peuvent être définies sous forme de diagramme comme présenté dans la figure 2.4. Si une flèche est annotée par un point d'exclamation alors cela signifie que ce morphisme est unique et si une flèche est en pointillé alors ça implique qu'elle doit exister si le reste des flèches existent.

4. Nous nous limitons aux catégories localement petites, c'est-à-dire que pour tout A, B de $\mathcal{C}_{obj}$, $\mathcal{C}_{arr}(A, B)$ est un ensemble.

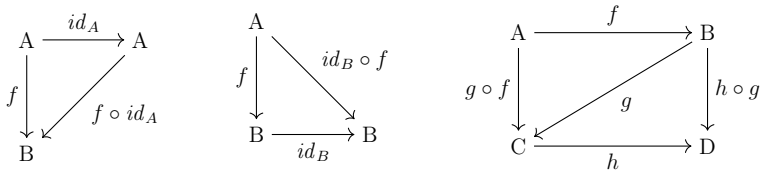


FIGURE 2.4 – Diagrammes de commutations des lois d'une catégorie.

Catégories cartésiennes Une catégorie cartésienne est une catégorie équipée d'un objet terminal et d'un produit cartésien. Intuitivement, un objet terminal, dénoté par **1**, est le point d'arrivée de tout objet de la catégorie.

Définition 2.2 (Objet terminal)**.** *Dans une catégorie $\mathcal{C}$, un objet A est dit* terminal *si, pour tout objet B, il existe un* unique morphisme *allant de B vers A, c'est-à-dire :*

$$B \xrightarrow{\quad ! \quad} A$$

Exemple 2.10. *Dans la catégorie contenant un unique objet A et un unique morphisme f, l'objet terminal est A. Dans la catégorie* **Set***, les objets terminaux sont les singletons.*

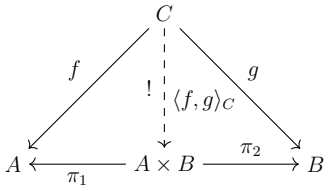
Un objet terminal est unique à *isomorphisme* près. D'un point de vue ensembliste, un isomorphisme est un morphisme qui préserve la structure des objets. D'un point de vue catégorique, c'est un morphisme qui admet un morphisme inverse.

Définition 2.3 (Isomorphisme)**.** *Soit $\mathcal{C}$ une catégorie, un isomorphisme est un morphisme $f \in \mathcal{C}(A, B)$, pour A, B deux objets de $\mathcal{C}$, qui a un morphisme inverse $g \in \mathcal{C}\, B\, A$, c'est-à-dire que :*

$$f \circ g = id_B \ \wedge \ g \circ f = id_A$$

Le produit cartésien provient des ensembles. D'un point de vue catégorique, cela revient à dire que les paires d'objets existent avec leur projections dans la catégorie.

Définition 2.4 (Produit)**.** *Dans une catégorie $\mathcal{C}$, un produit de deux objets A et B est un objet $A \times B$ accompagné de deux morphismes $\pi_1 \in \mathcal{C}(A \times B, A)$ et $\pi_2 \in \mathcal{C}(A \times B, B)$ tels que pour tout objet C et morphismes $f \in \mathcal{C}(C, A)$ et $g \in \mathcal{C}(C, B)$, il existe un unique morphisme $\langle f, g \rangle_C : \mathcal{C}(C, A \times B)$ faisant commuter le diagramme suivant :*



Exemple 2.11. *Un exemple direct est le produit cartésien dans la catégorie* **Set**.

Dans une catégorie $\mathcal{C}$, un produit entre deux objets A et B est unique à isomorphisme près. Comme pour les catégories, nous nous autorisons à ne pas spécifier C dans $\langle \cdot, \cdot \rangle_C$ quand le contexte le permet.

Définition 2.5 (Catégorie cartésienne)**.** *Une catégorie $\mathcal{C}$ est cartésienne si et seulement si elle a un objet terminal et que pour toute paire d'objets A, B de $\mathcal{C}$, le produit $A \times B$ existe dans $\mathcal{C}$.*

Nous profitons de cette définition pour introduire les catégories produits proches du concept de catégories cartésiennes. Intuitivement, une catégorie produit reprend le concept de produit de deux ensembles.

Définition 2.6 (Catégorie produit)**.** *Soient $\mathcal{C}$ et $\mathcal{D}$ deux catégories, la catégorie produit $\mathcal{C} \times \mathcal{D}$ a pour objets toutes paires $A \times B$, avec $A \in \mathcal{C}$ et $B \in \mathcal{D}$, et pour morphismes toutes paires $f \times g$, avec $A, C \in \mathcal{C}$, $B, D \in \mathcal{D}$, $f \in \mathcal{C}\,(A, C)$ et $g \in \mathcal{D}\,(B\,D)$.*

Soit $\mathcal{C}$ une catégorie cartésienne, nous définissons un ensemble de morphismes dans la figure 2.5. Pour ce faire, nous utilisons une notation issue du lambda-calcul simplement typé avec paires et projections, mais sans évaluation. Ces morphismes seront utilisés dans le reste du manuscrit, notamment dans cette section et le chapitre 5.

2.3.2 Foncteurs

Il est possible de mettre en relation des catégories via des *foncteurs*. D'après Eilenberg et MacLane dans leur article initial, cela est même le concept essentiel, avec les *transformations naturelles*, qui a motivé la création des catégories. Intuitivement un foncteur permet de projeter les objets d'une catégorie à une autre tout en préservant leurs morphismes.

Définition 2.7 (Foncteur)**.** *Un foncteur $\mathbf{F}$ entre deux catégories $\mathcal{C}$ et $\mathcal{D}$ est une paire d'opérateurs $\mathbf{F}_{obj} : \mathcal{C}_{obj} \to \mathcal{D}_{obj}$ et $\mathbf{F}_{arr} : \mathcal{C}_{arr} \to \mathcal{D}_{arr}$ telle que :*

— $\mathbf{F}_{arr}\, f \in \mathcal{D}\,(\mathbf{F}_{obj}\, A, \mathbf{F}_{obj}\, B)$

$$
\begin{aligned}
id_A &\in \mathcal{C}\,(A, A) \\
id_A &= \lambda x.x
\end{aligned}
$$

$$
\begin{aligned}
dup_A &\in \mathcal{C}\,(A, A \times A) \\
dup_A &= \lambda x.(x, x)
\end{aligned}
$$

$$
\begin{aligned}
sdup_{A,B} &\in \mathcal{C}\,(A \times B, (A \times B) \times B) \\
sdup_{A,B} &= \lambda(x, y).((x, y), y)
\end{aligned}
$$

$$
\begin{aligned}
swap_{A,B} &\in \mathcal{C}\,(A \times B, B \times A) \\
swap_{A,B} &= \lambda(x, y).\,(y, x)
\end{aligned}
$$

$$
\begin{aligned}
perm_{A,B,C} &\in \mathcal{C}\,((A \times B) \times C, (A \times C) \times B) \\
perm_{A,B,C} &= \lambda((x, y), z).\,((x, z), y)
\end{aligned}
$$

$$
\begin{aligned}
assoc_{A,B,C} &\in \mathcal{C}\,((A \times B) \times C, A \times (B \times C)) \\
assoc_{A,B,C} &= \lambda((x, y), z).\,(x, (y, z))
\end{aligned}
$$

$$
\begin{aligned}
unassoc_{A,B,C} &\in \mathcal{C}\,(A \times (B \times C), (A \times B) \times C) \\
unassoc_{A,B,C} &= \lambda(x, (y, z)).\,((x, y), z)
\end{aligned}
$$

$$
\begin{aligned}
(\times)_{A,B,C,D} &\in \mathcal{C}\,(A, B) \to \mathcal{C}\,(C, D) \to \mathcal{C}\,(A \times C)\,(B \times D) \\
(f \times g)_{A,B,C,D} &= \lambda(x, y).(f\,x, g\,y)
\end{aligned}
$$

FIGURE 2.5 – Morphismes et opérateurs pour une catégorie cartésienne $\mathcal{C}$.

— $\mathbf{F}_{arr}\ id_A = id_{\mathbf{F}_{obj}\ A}$

— $\mathbf{F}_{arr}\ (g \circ f) = (\mathbf{F}_{arr}\ g) \circ (\mathbf{F}_{arr}\ f)$

pour tous objets A, B, C de $\mathcal{C}$ et morphisme $f \in \mathcal{C}\,(A, B)$, $g \in \mathcal{C}\,(B, C)$.

Exemple 2.12. *Il est possible de définir un programme ayant un comportement non déterministe, c'est-à-dire que plusieurs moyens d'évaluer son application à une expression est possible. Un foncteur peut définir des exécutions non déterministes. Soient $\mathcal{C}$ une catégorie et $\mathcal{D}$ une catégorie dont les objets sont des ensembles finis d'objets de $\mathcal{C}$. Nous définissons le foncteur* **Nd** *qui plonge tout objet $A \in \mathcal{C}$ dans $\mathcal{D}$ tel que :*

$$\mathbf{F}_{obj}\ A = \{A\}$$
$$\mathbf{F}_{arr}\ f\ C = map\ f\ C$$

avec $C \in \mathcal{D}$, $f \in \mathcal{C}\,(C, B)$ et map une fonction qui prend une fonction et applique cette fonction à tous les éléments d'un ensemble. Un calcul prend donc un argument et retourne l'ensemble des résultats possibles. En Haskell, *il peut être défini comme suit :*

```haskell
newtype Nd a = Nd [a]

instance Functor Nd where
  fmap :: (a -> b) -> Nd a -> Nd b
  fmap f (Nd as) = Nd (map f as)
```

Le type `Nd` *est le foncteur et l'instance permet de définir la traduction du morphisme de la catégorie de départ vers celle d'arrivée.*

Si la catégorie source est la même que la catégorie cible, c'est-à-dire $\mathcal{C} = \mathcal{D}$ dans la définition, alors on parle d'*endofoncteur*. Si la catégorie source est une catégorie produit, c'est-à-dire la catégorie $\mathcal{C} \times \mathcal{D}$ pour deux catégories $\mathcal{C}$ et $\mathcal{D}$, alors on parle de *bifoncteur*. Comme pour les catégories, si le contexte permet de différencier $\mathbf{F}_{obj}$ de $\mathbf{F}_{arr}$ pour un certain foncteur $\mathbf{F}$ alors nous nous autorisons à ne pas le préciser.

2.3.3 Monades

Les monades ont été introduites par Roger Godement en 1958 [47] sous le nom de *constructions standards*. Initialement définie dans l'intention de généraliser le comportement des monoïdes dans une algèbre, la notion de monade a été utilisée dans différents domaines des mathématiques. En 1989, Eugenio Moggi montre

l'utilisation des monades dans un contexte de programmation [74]. En effet, il fait remarquer que le lambda-calcul vit dans un monde idéal ou toutes les fonctions sont totales ce qui ne correspond pas à la réalité de la programmation. Pour éviter cela, il utilise les monades pour définir différents comportements comme le non-déterminisme, la potentialité d'échouer ou encore les effets de bords. Nous commençons par définir la monade d'un point de vue catégorique, puis d'un point de vue programmation.

Définition 2.8 (Monade catégorique)**.** *Une monade M, sur une catégorie $\mathcal{C}$, est un triplet $(\mathbf{T}, \eta, \mu)$, avec $\mathbf{T}$ un endofoncteur, $\eta : Id_{\mathcal{C}} \mapsto \mathbf{T}$ et $\mu : \mathbf{T}\,\mathbf{T} \mapsto \mathbf{T}$ deux morphismes*[5] *tels que les diagrammes suivants commutent :*

$$\begin{array}{ccc} \mathbf{T}\,A & \xrightarrow{\mathbf{T}\,(\eta\,A)} & \mathbf{T}\,(\mathbf{T}\,A) \\ {\scriptstyle \eta\,(\mathbf{T}\,A)}\downarrow & & \downarrow{\scriptstyle \mu\,A} \\ \mathbf{T}\,(\mathbf{T}\,A) & \xrightarrow{\mu\,A} & \mathbf{T}\,A \end{array} \qquad \begin{array}{ccc} \mathbf{T}\,(\mathbf{T}\,(\mathbf{T}\,A)) & \xrightarrow{\mathbf{T}\,(\mu\,A)} & \mathbf{T}\,(\mathbf{T}\,A) \\ {\scriptstyle \mu\,(\mathbf{T}\,A)}\downarrow & & \downarrow{\scriptstyle \mu\,A} \\ \mathbf{T}\,(\mathbf{T}\,A) & \xrightarrow{\mu\,A} & \mathbf{T}\,A \end{array}$$

Intuitivement, pour une catégorie $\mathcal{C}$, une monade M est un monoïde sur la catégorie des foncteurs $\mathcal{C} \to \mathcal{C}$. Le morphisme η représente le neutre et μ l'opérateur du monoïde.

Exemple 2.13. *Soient S l'ensemble non vide de mémoires, $\mathcal{C}$ une catégorie cartésienne close et $\mathbf{T} \in \mathcal{C} \to (\mathcal{C} \times S)^S$ un endofoncteur. En simplifiant à l'extrême, une catégorie cartésienne close permet de représenter des fonctions comme des objets, nous utilisons la notation ensembliste pour les définir, et est équipée d'une fonction d'évaluation nommée eval. Nous définissons la monade d'état St comme un triplet $(\mathbf{T}, \eta, \mu)$ avec :*

- $\eta_A = \lambda a : A.\lambda s : S.(a, s)$
- $\mu_A = \lambda f : \mathbf{T}\,A.\lambda s : S.eval\,(f\ s)$

Intuitivement, elle modélise un calcul qui prend une mémoire et retourne le résultat avec la mémoire mise à jour.

D'un point de vue programmation, les monades sont un moyen de définir un comportement comme la potentialité du résultat ou encore l'interaction avec l'environnement. Elles rendent « opaques » les valeurs aux yeux du programme ce qui permet garder pur un programme qui utilise des effets de bords. Haskell est un partisan de la programmation fonctionnelle pur et contient un moyen

5. Plus précisément, η et μ sont des *transformations naturelles* que nous considérons en dehors de la portée de la thèse. La définition formelle de ces transformations se trouve dans la littérature proposée dans l'introduction de la section.

efficace de définir monades, foncteurs et catégories. Cependant, la définition de monade dans HASKELL est légèrement différente, car elle se base sur les monades fortes [66], qui ne seront pas développées ici.

Définition 2.9. *Dans* HASKELL, *une monade* M *sur une catégorie* $\mathcal{C}$ *est un triplet* $(\mathbf{T}, \texttt{return}, \texttt{bind})$, *avec* $\mathbf{T}$ *un endofoncteur et, pour tous objets* A, B *de* $\mathcal{C}$, $\texttt{return}_A : A \to \mathbf{T}\,A$, $\texttt{bind}_{A,B} : (\mathbf{T}\,A) \to (A \to \mathbf{T}\,B) \to (\mathbf{T}\,B)$ *deux opérateurs tels que :*

$$\texttt{bind}_{A,B}\,(\texttt{return}_A\,a)\,f = f\,a \tag{2.9a}$$

$$\texttt{bind}_{A,A}\,m\,\texttt{return}_A = m \tag{2.9b}$$

$$\texttt{bind}_{B,C}\,(\texttt{bind}_{A,B}\,m\,f)\,g = \texttt{bind}_{A,C}\,m(\lambda x.\texttt{bind}_{B,C}\,(f\,x)\,g) \tag{2.9c}$$

pour tous objets A, B *et* C *de* $\mathcal{C}$, $a : A$, $m : \mathbf{T}\,A$ *et* $f : A \to \mathbf{T}\,B$. *Les équations* (2.9a) *et* (2.9b) *représentent les propriétés de neutre gauche et droit et l'équation* (2.9c) *représente l'associativité.*

Exemple 2.14. *En* HASKELL, *la monade d'état définie dans l'exemple précédent s'instancie comme suit :*

```
newtype St a = St (Mem -> (a,Mem))

instance Monad St where
  (>>=) :: St a -> (a -> St b) -> St b
  (>>=) (St a) f = St (\m -> let (a',m') = a m in
                             let St b = f a' in b m')
  return :: a -> St a
  return a = St (\m -> (a,m))
```

L'opérateur `bind` *est noté* `(>>=)`. `Mem` *est le type représentant la mémoire et* `St` *est la monade d'état.*

L'opérateur `return` agit comme η et l'opérateur `bind` est une utilisation spécifique de μ. En effet, `bind` peut se définir en fonction du foncteur $\mathbf{T}$ et μ comme suit :

$$\texttt{bind}\,(\mathbf{T}\,A)\,f = \mu \circ (\mathbf{T}\,(f\,A))$$

Dans le chapitre 5, lorsque nous parlons de monade, nous faisons référence aux monades HASKELL. Une monade M sur une catégorie $\mathcal{C}$ définit une nouvelle catégorie appelée catégorie de *Kleisli*. Elle a pour objets ceux de $\mathcal{C}$ et ses morphismes appliquent la monade au résultat, c'est-à-dire que l'objet d'arrivée du morphisme est encapsulé par la monade M.

Cette catégorie est utile pour définir des variantes d'un langage dans un contexte d'informatique fondamentale. En effet, le lambda-calcul peut être interprété dans une catégorie cartésienne close. Maintenant, il est possible de définir, avec une monade qui représente le non déterminisme, une nouvelle catégorie qui est une variante du lambda-calcul qui prend en compte cela.

Définition 2.10 (Kleisli)**.** *Soit M une monade $(\mathbf{T}, \eta, \mu)$ sur une catégorie $\mathcal{C}$. La catégorie de Kleisli pour M, noté $\mathcal{C}_\mathbf{T}$, est la catégorie où :*

- *les objets sont ceux de $\mathcal{C}$;*
- *tout morphisme $f \in \mathcal{C}_\mathbf{T}(A, B)$, pour A,B dans $\mathcal{C}$, il existe un morphisme correspondant $g \in \mathcal{C}(A, \mathbf{T}\,B)$.*

Pour tout objet $A \in \mathcal{C}_\mathbf{T}$, le morphisme identité est η_A et l'opérateur de composition est défini comme suit :

$$g \circ_{\mathcal{C}_\mathbf{T}} f = \mu \circ_\mathcal{C} \mathbf{T}\, g \circ_\mathcal{C} f$$

avec A, B, C des objets de $\mathcal{C}_\mathbf{T}$ et $f \in \mathcal{C}_\mathbf{T}(A, B)$, $g \in \mathcal{C}_\mathbf{T}(B, C)$ deux morphismes.

2.3.4 Flèches

Introduite dans un article de J. Hughes en 2000 [57], la notion de flèche est une généralisation de la notion de monade Haskell. Dans cet article, J. Hughes fait mention de l'algorithme d'analyse syntaxique de D. Swierstra et L. Duponcheel [94] qui n'utilise pas de monades à cause de l'opérateur `bind` trop restrictif dans ce cas. Intuitivement, une donnée statique dans une monade peut être inutilisé durant le calcul. Par conséquent, c'est une utilisation de la mémoire non-nécessaire, qu'il serait préférable de retirer dès que possible. Cependant, il n'est pas possible de faire cela dans une monade, car l'objet est protégé et le seul moyen de l'« ouvrir » est via l'opérateur `bind`. Dans une composition de fonction cela reviendrait à réellement calculer une partie du programme non souhaitée. Pour éviter cela, la flèche n'encapsule pas l'objet, mais la fonction d'analyse.

Différents articles [54, 61] ont démontré que les flèches sont effectivement une généralisation des monades et en ont fourni une définition catégorique. Là où une monade correspond à un monoïde sur un foncteur $\mathbf{F} \in \mathcal{C} \to \mathcal{C}$, pour une catégorie $\mathcal{C}$, une flèche correspond à un monoïde sur un bifoncteur $\mathbf{G} \in \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$. Nous fournissons, ci-dessous, une version simplifiée de la définition.

Définition 2.11 (Flèche)**.** *Soit $\mathcal{C}$ une catégorie cartésienne localement petite, une flèche F est un quadruplet $(\mathbf{F}, arr, first, \ggg)$ avec :*

- *$\mathbf{F} \in \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ un bifoncteur ;*

— $arr_{A,B} \in \mathcal{C}\,(A,B) \to \mathbf{F}\,(A,B)$, *pour tout* $A, B \in \mathcal{C}$;

— $(\ggg)_{A,B,C} : \mathbf{F}\,(A,B) \to \mathbf{F}\,(B,C) \to \mathbf{F}\,(A,C)$, *pour* $A, B, C \in \mathcal{C}$;

— $first_{A,B,C} : \mathbf{F}\,(A,B) \to \mathbf{F}\,(A \times C, B \times C)$, *pour tout* $A, B, C \in \mathcal{C}$

tel que les lois suivantes sont satisfaites :

$$arr\ id \ggg a = a \tag{2.10a}$$
$$a \ggg arr\ id = a \tag{2.10b}$$
$$(a \ggg b) \ggg c = a \ggg (b \ggg c) \tag{2.10c}$$
$$arr\,(g \circ f) = arr\ f \ggg arr\ g \tag{2.10d}$$
$$first\ a \ggg arr\ \pi_1 = arr\ \pi_1 \ggg a \tag{2.10e}$$
$$first\ a \ggg arr\,(id \times f) = arr\,(id \times f) \ggg first\ a \tag{2.10f}$$
$$first(first\ a) \ggg arr\ assoc = arr\ assoc \ggg first\ a \tag{2.10g}$$
$$first\,(arr\ f) = arr\,(f \times id) \tag{2.10h}$$
$$first(a \ggg b) = first\ a \ggg first\ b \tag{2.10i}$$

Pour une catégorie cartésienne localement petite $\mathcal{C}$, une flèche F pour un bifoncteur $\mathbf{F}$ donne un objet $\mathbf{F}(A,B)$ pour tous objets A et B de $\mathcal{C}$. Dans ce contexte, *arr* relie un morphisme $g \in \mathcal{C}\,(A,B)$ à cet objet $\mathbf{F}(A,B)$. Le premier argument est $\mathcal{C}^{op}$, la catégorie opposée à $\mathcal{C}$, pour que les lois soient cohérentes avec la composition et les morphismes des flèches.

Remarque 2.1. *Une flèche est définissable, a minima, via une catégorie cartésienne à cause de la famille d'opérateurs first qui nécessite le produit.*

Exemple 2.15. *Une utilisation simple des flèches est l'application aux fonctions ordinaires. Soient $\mathcal{L}$ une catégorie cartésienne close qui représente le lambda-calcul typé (avec paires et projections) et Func la flèche $(\mathbf{F}, arr, first, \ggg)$ qui encapsule les fonctions ordinaires telles que :*

— $\mathbf{F} : \mathcal{L}^{op} \times \mathcal{L} \to \mathcal{L}$ *est un bifoncteur avec*

  — $\mathbf{F}(X,Y) = X \times Y$;

  — $\mathbf{F}f = f$;

— $arr\ f = f$;

— $first\ g = g \times id$;

— $f \ggg g = g \circ f$.

Dans Haskell*, il est possible de l'instancier comme suit :*

```haskell
newtype Func a b = F (a -> b)

instance Category Func where
  id :: Func a a
  id = F Prelude.id
  (.) :: Func b c -> Func a b -> Func a c
  (.) (F f) (F g) = F (f Prelude.. g)

instance Arrow Func where
  arr :: (b -> c) -> Func b c
  arr = F
  first :: Func b c -> Func (b, d) (c, d)
  first (F f) = F (\(x,y) -> (f x,y))
```

L'opérateur infixe `Prelude..` *est la composition classique mathématique, nous spécifions la source, c'est-à-dire* `Prelude`*, car il y a une surcharge de l'opérateur.*

Remarque 2.2. *En* HASKELL*, il n'est pas nécessaire de définir* ($\ggg$)*, car il récupère directement la composition définit dans l'instance de la catégorie associée. Cependant, si nous souhaitons vraiment le définir, nous ajouterions le code suivant :*

```haskell
(>>>) (F f) (F g) = F (g Prelude.. f)
```

Soient $\mathcal{C}$ une catégorie, F une flèche sur un bifoncteur $\mathbf{F}$. D'autres familles d'opérateurs dérivent des trois familles d'opérateurs primaires comme *second* et *split* qui sont définies ci-dessous avec $A, B, C \in \mathcal{C}$, $a \in \mathbf{F}\,A\,B$ et $b \in \mathbf{F}\,B\,C$.

$$
\begin{aligned}
second_{A,B,C} &: \mathbf{F}\,A\,B \to \mathbf{F}\,(C \times A)\,(C \times B) \\
second_{A,B,C}\,a &= arr\ swap \ggg a \ggg arr\ swap \\
split_{A,B,C,D} &: \mathbf{F}\,A\,C \to \mathbf{F}\,B\,D \to \mathbf{F}\,(A \times C)\,(B \times D) \\
split_{A,B,C,D}\,a\,b &= first\ a \ggg second\ b
\end{aligned}
$$

Il est aussi possible de restreindre une flèche F, comme précédemment définie, en rajoutant une nouvelle famille d'opérateurs, non constructible par les opérateurs primaires, qui doit satisfaire de nouvelles lois. C'est le cas de la famille d'opérateurs $loop_{A,B,C} : C \to \mathbf{F}\,(A \times C)\,(B \times C) \to \mathbf{F}\,A\,B$ qui doit satisfaire

l'ensemble de ci-dessous.

$$loop\, c\, (first\, a \ggg b) = a \ggg loop\, c\, b \tag{2.12a}$$
$$loop\, c\, (a \ggg first\, b) = loop\, c\, a \ggg b \tag{2.12b}$$
$$loop\, c\, (loop\, d\, a) = loop\, (c, d)\, (arr\ unassoc \ggg a \ggg arr\ unassoc) \tag{2.12c}$$

Exemple 2.16. *Dans le cadre de notre flèche Func, la valeur c de l'opérateur loop agit comme une constante dans le calcul.*

```
loop :: c ->  Func (b, d) (c, d) -> Func b c
loop i (F f) = F (\x -> let (y,i') = f (x,i) in y)
```

2.4 L'assistant de preuve Coq/Rocq

Un assistant de preuve est un système de gestion de démonstrations formelles automatique ou semi-automatique. L'utilisateur à un langage de spécification qui lui permet de définir l'objet de son raisonnement ainsi que les lemmes ou théorèmes qu'il aimerait démontrer. Ensuite, dans le cas d'un assistant automatique, le programme, qui compile toute la spécification, va tenter de satisfaire les différentes propriétés sans l'aide d'un tiers (ce qui peut mener à un échec de la démonstration). Dans le cas d'un assistant semi-automatique, en plus du langage de spécification, il y a un langage de tactique qui permet à l'utilisateur de guider l'assistant de preuve dans le raisonnement.

Coq est un assistant de preuve semi-automatique basé sur le calcul inductif des constructions [31]. Il a vu le jour en 1984 grâce Thierry Coquand et Gérard Huet sous la forme d'une implémentation du calcul des constructions. Il a ensuite été enrichi pour implémenter le calcul inductif des constructions en 1990 via le travail de Christine Paulin [85]. Finalement, après plus de 40 ans de développement, le logiciel est à sa version 9 et est mondialement connu. En effet, à partir de la moitié des années 2000, cet assistant de preuve a permis de formaliser le compilateur C CompCert [70] ainsi que de démontrer des théorèmes mathématiques comme le théorème de Feit-Thompson en 2012 ou encore le théorème des quatre couleurs en 2008 [48]. Récemment, plus précisément pour sa version 9, Coq a été renommé Rocq, par conséquent nous utilisons le terme Rocq dans la suite. Cependant, les développements Rocq dont nous parlons dans les contributions présentées dans ce manuscrit sont encore dans la version 8.20.1, donc techniquement sous Coq.

Dans cette section, nous expliquons par l'exemple le fonctionnement basique de Rocq. Puis, nous fournissons des informations complémentaires sur la notion

de module. Si le lecteur souhaite s'essayer à la preuve formelle en Rocq, il y a le site internet de l'assistant [6] ainsi que les excellents livres « Software Foundations » disponibles en ligne [7].

Concepts basiques Rocq est constitué de deux langages : un langage de spécification de programmes de haut niveau appelé Gallina et un langage de tactiques appelé Ltac. Le premier langage est un langage fonctionnel qui permet d'établir des définitions, des fonctions, des structures inductives, des relations ainsi que des lemmes, des corollaires, des théorèmes, etc.

Exemple 2.17. *Nous pouvons décrire les entiers naturels, en suivant la modélisation de Peano, comme suit :*

```
Inductive entier : Set :=
  | zero : entier
  | succ (n: entier) : entier.

Definition incr (n: entier) := succ n.

Fixpoint plus (n m: entier) :=
  match n with
    | zero   => m
    | succ p => succ (plus p m)
  end.
```

Le mot-clé `Inductive` *permet de définir une structure inductive,* `Definition` *permet d'établir une définition et* `Fixpoint` *permet de définir une fonction récursive.*

Une fonction récursive dans Rocq termine toujours. En effet, lors de la compilation du programme, il y a une vérification d'une décroissance d'un argument de la fonction dans les appels récursifs pour garantir la terminaison. Dans le cas de l'exemple précédent, `n` décroit à chaque nouvel appel. Il est possible de définir des notations dans le programme, via les mots-clés `Notation` et `Infix`, afin de simplifier la lecture.

Exemple 2.18. *Soit* + *l'opérateur infixe représentant la fonction* `plus` *via le mot-clé* `Infix`*. Il est possible de spécifier le niveau de priorité ainsi que l'associativité de l'opérateur.*

```
Infix "+" := plus (at level 50, left associativity).
```

6. https://rocq-prover.org/
7. https://softwarefoundations.cis.upenn.edu/

Le langage GALLINA contient le quantificateur universel et existentiel ainsi que les connecteurs logiques classiques. Grâce à cela, et aux mots-clés `Lemma`, `Theorem` ou encore `Corollary`, il est possible de définir des propriétés.

Exemple 2.19. *La commutativité de l'addition peut s'énoncer comme suit :*

```
Lemma plus_comm: forall (n m: entier), n + m = m + n.
```

La démonstration d'une propriété est rédigée via le langage de tactique LTAC, entre les mots-clés `Proof` et `Qed`, et elle est faite interactivement avec l'assistant de preuve. En effet, généralement le développement de projet ROCQ se fait de manière interactive. Cela permet de faire une démonstration au fur et à mesure. Si l'IDE utilisée le tolère, voir la figure 2.6, alors la fenêtre de l'IDE va se scinder en trois sous-fenêtres :

— une fenêtre contenant le fichier en cours d'interprétation ;

— une fenêtre contenant les hypothèses courantes de la démonstration au-dessus d'une barre horizontale et le but de la démonstration, c'est-à-dire la propriété à démontrer, en dessous de celle-ci ;

— une fenêtre contenant les potentiels messages d'avertissement ou d'erreurs.

La démonstration se fait de manière interactive en interprétant tactique par tactique tout en mettant à jour le but et les hypothèses. Les tirets permettent de focaliser le but et les hypothèses à un sous-but, car lors d'une induction plusieurs sous-buts peuvent apparaitre. Dans la figure 2.6, nous interprétons le second cas de l'induction sur m avec comme but

```
succ m = succ m + zero
```

et comme hypothèses

```
m : entier
IHm : m = m + zero
```

Nous fournissons dans le tableau 2.1 une liste non exhaustive de tactiques à appliquer selon le connecteur logique et la position d'une proposition, soit dans les hypothèses, soit dans le but. Nous rajoutons `simpl`, la tactique qui ouvre les définitions pour simplifier le but et `induction` qui permet de faire une preuve par induction sur des termes inductifs. Il est intéressant de noter que le schéma d'induction pour toute définition inductive faite via le mot-clé `Inductive` est déjà généré à l'interprétation de la définition.

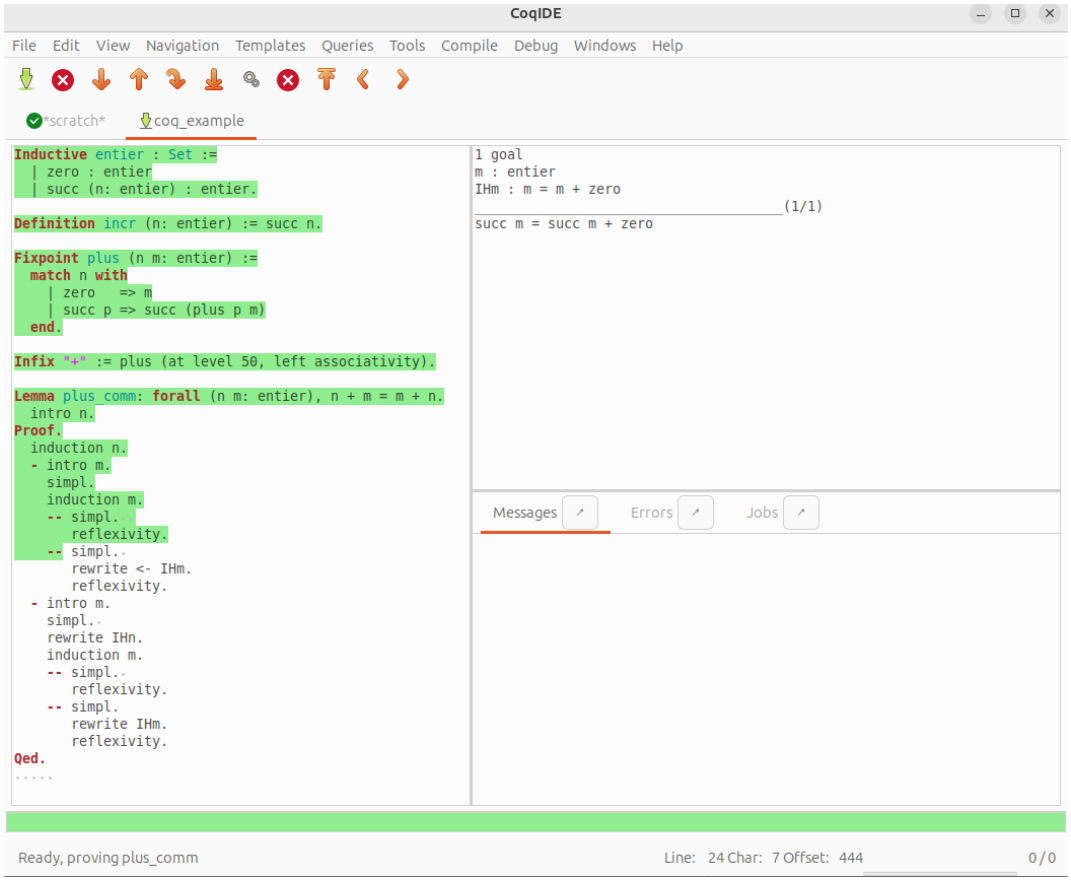


FIGURE 2.6 – Exemple d'utilisation interactive de ROCQ via COQIDE.

Concepts avancés Dans le chapitre A, nous définissons des *modules* et des *interfaces*. Comparativement, ce sont les mêmes que ceux en OCAML. Pour les utilisateurs de C, nous pouvons faire le lien avec l'extension « *.c* » et « *.h* ». Un module est une structure qui porte un nom et contient un ensemble de définitions, fonctions et lemmes/théorèmes. Pour accéder à un élément d'un module en dehors de celui-ci, il faut le préfixer par son nom.

Exemple 2.20. *Voici un module* `N` *qui regroupe la définition des entiers précédemment donnée et son utilisation.*

```
Module N.
  Inductive entier : Set :=
    | zero : entier
    | succ (n: entier) : entier.

  Definition incr (n: entier) := succ n.
```

	$\Longrightarrow$	$\forall$	$\exists$	$\wedge$
Hypothèse	`apply`	`apply`	`elim`	`elim`
But	`intros`	`intros`	`exists`	`split`
	$\vee$	$\neg$	$=$	
Hypothèse	`elim`	`elim`	`rewrite`	
But	`left` or `right`	`intro`	`reflexivity`	

TABLE 2.1 – Liste non-exhaustive des tactiques en ROCQ.

```
  Fixpoint plus (n m: entier) :=
    match n with
      | zero   => m
      | succ p => succ (plus p m)
    end.
End N.

Definition un : N.entier := N.incr N.zero.
```

Il est aussi possible de paramétrer un module et d'inclure d'autres modules. Inclure un module permet d'accéder directement aux éléments qu'il contient.

Exemple 2.21. *Nous définissons un nouveau module* `Nsub` *qui étend* `N` *avec une nouvelle fonction* `sub`.

```
Module Nsub.
  Include N.

  Fixpoint sub (n m: entier) :=
    match n with
      | zero => zero
      | succ n' => match m with
                     | zero => n
                     | succ m' => sub n' m'
                   end
    end.
End Nsub.

Check Nsub.sub (Nsub.succ Nsub.zero) un = Nsub.zero.
```

Une interface, ou plus précisément un « module type », est la signature du module, c'est-à-dire le type de l'ensemble de ses éléments. Elle porte un nom

comme un module, mais elle est composée de paramètres définis par un nom et un type.

Exemple 2.22. *Une interface correspondante au module* `N` *est la suivante :*

```
Module Type IN.
  Parameter entier : Set.
  Parameter incr : entier -> entier.
  Parameter plus : entier -> entier -> entier.
End IN.
```

Un module peut implémenter une interface via l'opérateur `<:`. Dès lors, tous les paramètres de l'interface doivent être implémentés dans le module.

Exemple 2.23. *Le module* `N` *implémente l'interface* `IN`.

```
Module N <: IN.
...
```

Finalement, un module peut être paramétré comme une fonction, où les types des paramètres sont des noms d'interfaces.

Exemple 2.24. *Il est possible de définir un module* `N'` *qui prend en paramètre un élément* `E` *de type* `IN`*, c'est-à-dire que* `E` *contient les définitions de* `IN`.

```
Module N' (E : IN).
  Definition plus3 (n : E.entier) := E.incr (E.incr (E.incr n)).
End N'.

Module N'N := N' N.
Check N'N.plus3 N.zero = N.succ (N.succ (N.succ N.zero)).
```

Pour utiliser `N'`*, il faut lui fournir comme argument un module qui implémente l'interface* `IN`*. Dans notre cas, ce module est* `N`.

Remarque 2.3. *Le module peut contenir plus d'éléments que l'interface qu'il implémente, cependant si le module est utilisé comme argument pour un autre module alors seuls les éléments de l'interface seront visibles.*

Chapitre 3

Programmation Réactive Fonctionnelle

Sommaire

La programmation réactive fonctionnelle (FRP) est un paradigme de programmation qui trouve ses racines dans la programmation réactive synchrone. Le paradigme synchrone a été défini pour programmer des systèmes *réactifs* ou *temps-réel*. Un système, ou programme, est dit *réactif* s'il maintient une interaction permanente avec son environnement et il est dit *temps-réel* s'il a, en plus, un temps de réponse limite fixé par l'environnement exterieur. Une application réactive peut ne pas être temps-réel, comme le protocole de communication *UDP* qui interagit constamment avec son environnement pour recevoir ou envoyer, cependant l'expéditeur n'a pas d'obligation de répondre vite, c'est le destinataire qui choisira d'attendre un certains temps avant de supposer une coupure de communication. Au contraire, un programme critique sur une voiture, comme le freinage

automatique d'urgence (AEBS) qui active un freinage s'il détecte une collision imminente, est temps-réel, car il a une contrainte sur le temps d'exécution.

Les langages de programmation réactifs synchrones sont majoritairement des langages bas-niveaux [13, 22, 29, 50, 97], même si ce n'est pas toujours le cas [23, 72, 100]. En effet, la plupart de ces langages ont une portée plus ou moins directe à l'industrie qui ne nécessite pas de langages très expressifs, mais plutôt des langages simples, concis et avec de fortes garanties sur les programmes qui en découlent. De son côté, la programmation FRP [34, 42, 49] apporte les concepts synchrones dans des langages plus expressifs, étend les domaines d'application, comme le web, les interfaces graphiques ou encore le jeu vidéo, et rend plus flexible l'utilisation des concepts synchrones, majoritairement via des bibliothèques dans des langages hôtes comme Haskell [42], Java [32] ou encore Racket [30]. En effet, la plupart des langages FRP sont des langages embarqués dans un langage hôte ce qui permet à « l'extension synchrone » de profiter de la puissance de ce langage (expressivité, typage, structures de donnée) tout en laissant la compilation, ou l'interprétation, au compilateur, ou l'interpréteur, de l'hôte. Cependant, ces améliorations et simplifications viennent avec une perte totale de la partie compilation et de garanties fortes, selon le langage hôte. Par exemple, la borne en espace et en temps d'un programme FRAn [42] n'est pas garantie statiquement.

Les pionniers du paradigme FRP sont Conal Elliott et Paul Hudak qui ont défini, en 1997, la bibliothèque Haskell nommée FRAn [42]. De nombreux langages ont fait leur apparition après celui-ci dans le but d'optimiser le modèle d'exécution [43], d'enrichir le langage [30, 99, 100, 104], de corriger certaines limitations [4, 56, 81, 102, 103] ou encore de porter ses concepts dans d'autres langages hôtes [21, 30, 32, 73]. En 2001, A. Courtney et C. Elliott propose Fruit [33] une bibliothèque Haskell qui utilise les flèches pour définir des transformateurs de flux : c'est la naissance de la programmation réactive fonctionnelle *avec flèches* (AFRP) [75]. Le concept de flèche est décrit dans la section 2.3.4.

Dans le but de synthétiser l'apparition des langages de chaque paradigme, nous fournissons dans la figure 3.1 une chronologie contenant un ensemble non-exhaustif de langages divisé en trois groupes : synchrone (en orange), FRP (en bleu) et AFRP (en vert). Les langages AFRP sont inclus dans les langages FRP qui sont eux même inclus dans les langages réactifs synchrones. La partie purement réactive synchrone, quant à elle, regroupe surtout les langages pionniers et leurs descendants directs.

Nous commençons par la section 3.1 qui porte principalement sur le début de la programmation réactive synchrone et son développement dans l'industrie. Ensuite, nous expliquons et discutons du modèle réactif fonctionnel dans la section 3.2. Finalement, nous définissons et discutons du paradigme réactif fonction-

nel avec flèches dans la section 3.3.

3.1 Paradigme synchrone

Le paradigme réactif synchrone [9, 15] est un style de programmation établi au milieu des années 80 qui répond à un besoin de méthodes simples et sûres pour modéliser des systèmes *réactifs* ou encore *temps-réel* dans une époque où seul des langages généralistes comme C, ADA ou encore l'assembleur permettaient de développer pour ce genre de systèmes. Le but était d'avoir des langages « user-friendly » basés sur un paradigme mathématiquement fiable. Les langages réactifs synchrones pionniers couramment cités sont ESTEREL [13] un langage synchrone impératif proposé par Gérard Berry en 1984, LUSTRE [51] un langage synchrone déclaratif proposé par N. Halbwachs *et al.* en 1986 et SIGNAL [50] un langage synchrone orienté description graphique proposé par P. Le Guernic *et al.* en 1991. Dans cette section, nous commençons par expliquer le concept du modèle synchrone, illustrer par des exemples en ESTEREL et en LUSTRE, puis nous explorons les travaux effectués avec ce modèle.

Un programme, défini dans un langage réactif synchrone, s'exécute en plusieurs *instants* successifs. Chaque instant représente la « réaction » du système à un stimulus extérieur (« event-driven ») ou à une horloge (« sample-driven »). L'*hypothèse synchrone* stipule que cette réaction ne prend pas de temps, c'est-à-dire que l'entrée est synchrone avec la sortie. Par conséquent, les communications internes sont aussi *instantanées*. Il existe deux types de langages réactifs synchrones qui sont complémentaires : les langages orientés *flot de contrôle* et les langages orientés *flot de données*.

3.1.1 Flot de contrôle

Dans un langage réactif synchrone orienté *flot de contrôle*, par exemple ESTEREL, un programme consiste en un ensemble de processus s'exécutant simultanément dont l'exécution est synchronisée par une unique horloge. Au début de chaque instant, chaque processus reprend son exécution là où elle s'est arrêtée lors de l'instant précédent et exécute la suite de son programme. Un processus s'arrête soit à cause d'une instruction spécifique, soit parce qu'il a fini d'exécuter son programme. La communication entre processus peut se faire via des *signaux* qui sont soit présents, c'est-à-dire émis au début ou « durant » l'instant, soit absents. La décision de quand est-ce que l'absence est prise en compte dépend du langage et peut provoquer un problème de *causalité* dont nous discuterons après. Un signal peut être *pur*, c'est-à-dire qu'il se réduit à son information de présence,

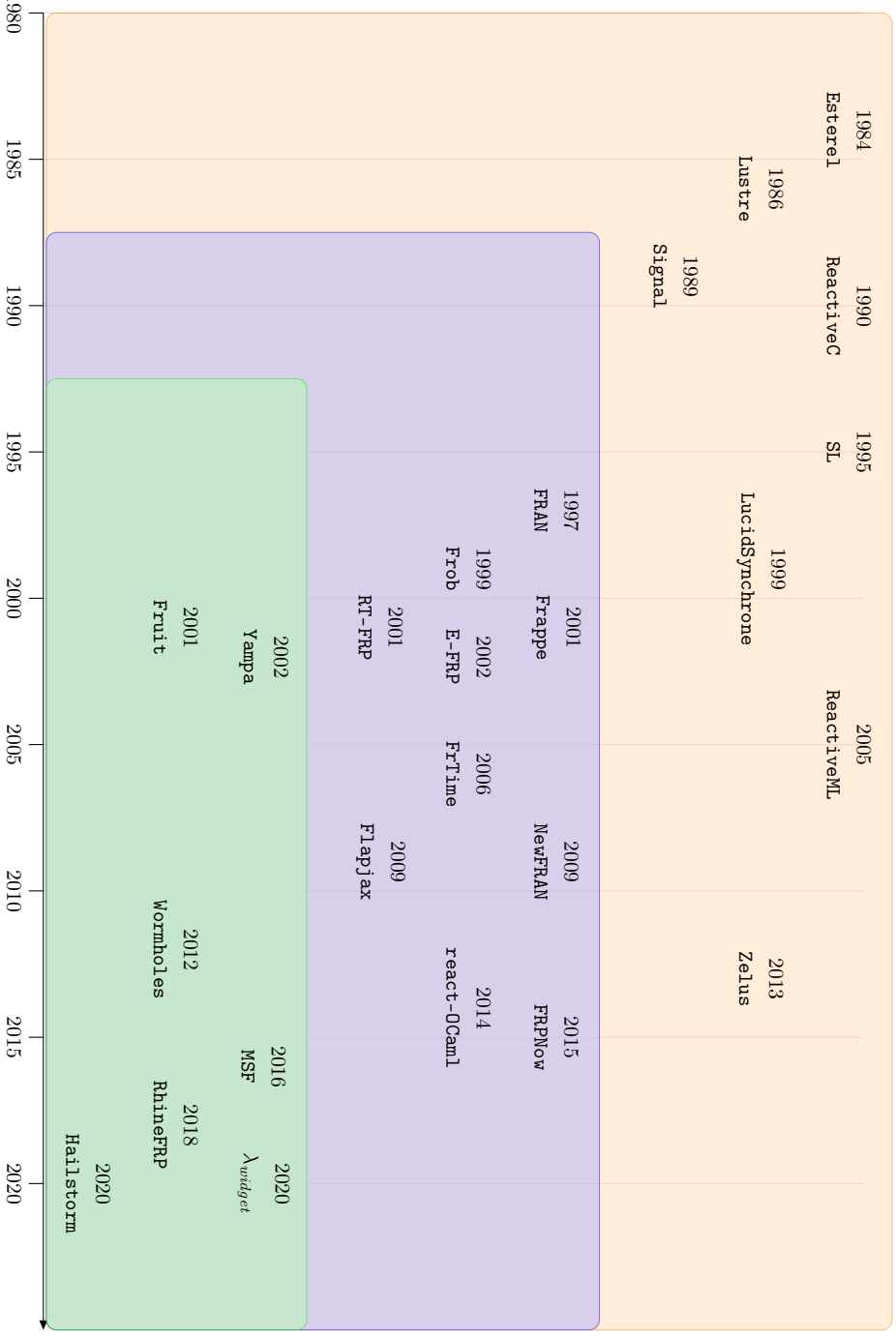


FIGURE 3.1 – Chronologie non exhaustive des langages de programmation fonctionnels réactifs.

ou *impur*, c'est-à-dire qu'il porte une donnée quand il est présent. Nous donnons ci-dessous une liste d'instructions primaires tirée de ESTEREL.

— *nothing* est l'instruction qui ne fait rien.

— *emit s* est une instruction qui émet un signal s, le rendant présent dans l'instant courant.

— *pause* est une instruction qui attend l'instant suivant.

— *present s then* i_1 *then* i_2 est une instruction qui effectue l'instruction i_1 si le signal s est présent et i_2 sinon.

— *await s* est une instruction qui met en pause le thread tant que s n'est pas présent.

— $\cdot \,||\, \cdot$ est un opérateur qui exécute deux programmes en parallèle.

— *loop i end* est une instruction qui répète son corps i dès qu'il se termine.

— ? est un opérateur unaire qui retourne la valeur d'un signal impur.

— *trap s in i end* est une instruction qui agit comme l'instruction i tant que le signal s n'est pas émis. Une fois le signal émis le « piège » se referme et l'expression est remplacée par *nothing*.

— *do i watching s* est une instruction qui exécute l'instruction i quand le signal s est présent.

Exemple 3.1. *Nous reprenons l'exemple du thermostat avec 20 degrés comme température idéale. Voici, ci-dessous, un programme qui effectue cette tâche avec* `T` *le signal portant la température courante.*

```
signal ON, OFF
extern signal T

loop await ON; start_res; pause end
||
loop await OFF; stop_res; pause end
||
loop
  present T then
    tmp = ?T
    if tmp > 20 then emit OFF; pause
    else
      if tmp < 18 then emit ON; pause
                  else pause
```

```
  else
    pause
end
```

À chaque instant, le programme est évalué. La `loop`, et son corps, sont conservés. Au départ, trois processus sont lancés en parallèle. Le premier attend que le signal `ON` soit émis et allume la résistance si c'est le cas, le deuxième attend que le signal `OFF` soit émis et éteint la résistance si c'est le cas et le troisième teste la présence du signal `T`, l'échantillonne en cas de présence et émet le bon signal en fonction de la température courante.

Lorsque nous avons un concept de temps qui passe, la propriété de causalité intervient et est même une propriété obligatoire. La causalité est une propriété qui donne un ordre aux actions : l'action A se déroule avant l'action B, etc. Par exemple, en ESTEREL, le programme ci-dessous émet s' seulement si s est présent. L'émission de s est la *cause* de l'émission de s'.

```
present s then emit s' else pause
```

Un problème de causalité arrive lorsque l'action B invalide la véracité de l'action A. Toujours en ESTEREL, durant l'instant si un test de présence est fait et que, au moment de l'exécution de l'instruction, le signal est absent alors le signal sera considéré absent pour le « reste » de l'instant. Cette *hypothèse d'absence* est nécessaire pour définir des programmes simulant des circuits, cependant il est possible d'écrire le programme suivant :

```
present s then pause else emit s
```

Intuitivement, ce programme teste si s est présent et émet s si ce n'est pas le cas. Nous avons là un problème de causalité. Pour pallier cela, le compilateur de ESTEREL fait une vérification statique pour rejeter ce genre de programme.

3.1.2 Flot de données

Dans un langage orienté *flot de données*, par exemple LUSTRE, un programme est un ensemble d'équations qui dérivent la sortie en fonction des entrées et des informations des instants précédents. L'exécution d'un tel programme est donc une résolution d'équations. Prenons l'exemple de LUSTRE. Dans ce langage, il est possible de définir des nœuds qui sont des fonctions sur les flux. Un nœud prend des paramètres d'entrées et de sorties, qui sont tous des flux, et contient l'ensemble d'équations permettant de calculer les sorties en fonctions des entrées.

Exemple 3.2. *Cet exemple est repris de l'article de A. Beveniste [10] de 2003. Nous voulons compter tous les évènements présents sur un flux* `event`*. De plus ce compteur doit être réinitialisable. En* Lustre*, cela peut s'écrire comme suit :*

```
node COUNT (event,reset: bool)
returns (count: int)
let
  count = if (true->reset) then 0
                           else if event then pre(count) + 1
                                         else pre(count)
tel
```

Le nœud se nomme `COUNT`*, ses entrées sont* `event` *et* `reset` *et sa sortie est* `count`*. Les opérateurs utilisés sont décrits dans la suite. L'exemple est accompagné d'un graphique, dans la figure 3.2, qui représente la valeur courante de flux à chaque instant.*

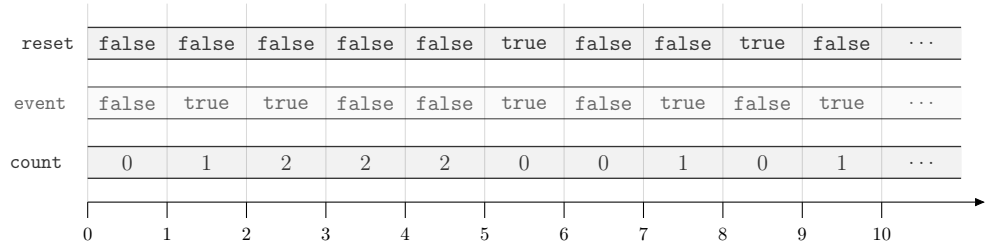


Figure 3.2 – Exécution des expressions de l'exemple 3.2.

Ces nœuds ont été introduits afin de décomposer un programme Lustre en plusieurs sous-programmes qui se composent entre eux. Toute variable représente un flux. Par exemple, l'expression *if* $x > 0$ *then* y *else* z est comprise comme suit : au temps logique n, si la valeur x_n, provenant de l'échantillonnage de x au temps n, est supérieur à 0 alors on retourne y_n sinon on retourne z_n. Des opérateurs dit *temporels* permettent de travailler sur ces flux, nous en donnons quatre ci-dessous.

— $pre(x)$ produit le flux x décalé d'un instant logique en commençant par *nil*.

— $\cdot \rightarrow \cdot$ est un opérateur qui prend deux flux x et y et produit le flux y en remplaçant y_1 par x_1.

— $\cdot\, when \,\cdot$ est un opérateur qui prend un flux x et un flux de booléen b et qui produit un flux ayant les valeurs de x quand la valeur de b est vraie.

— $current\, x$ produit le flux x cadencé sur l'horloge générale. Si x est plus lent, la dernière valeur récupérée est maintenue.

Lustre permet de définir des programmes multi-horloges, c'est-à-dire que certains flux vont avoir une fréquence plus lente ou plus rapide que d'autres. Par conséquent, le système calcul une horloge générale cadencée à l'horloge la plus rapide qui produit donc à chaque instant, laissant uniquement des horloges plus lentes.

Comme pour Esterel, le compilateur Lustre vérifie qu'aucun programme qui brise la causalité ne compile. Par exemple, le programme suivant ne compile pas :

```
x = x + 1
```

car le flux x ne peut pas dépendre de sa valeur courante pour définir sa valeur courante. Les définitions récursives doivent utiliser l'opérateur *pre*.

```
x = 0 -> pre(x) + 1
```

La programmation réactive synchrone a été adoptée avec succès par l'industrie de l'aéronautique chez Airbus [14] et Dassault [16], de l'énergie via Schneider Electric, de la télécommunication avec Snecma, etc [1]. Esterel a été mis sous licence privée dès 1998 par l'entreprise Simulog puis en 1999 dans l'entreprise Esterel Technologies jusqu'en 2018 et appartient maintenant à Ansys [1]. Lustre a été utilisé à des fins industrielles qui ont mené à la création de SAGA un outil développé en 1986 initialement pour Schneider Electric qui fut enrichi pour devenir Scade [39] un langage de haut niveau et un environnement pour le développement de logiciels de systèmes critiques pour la sécurité détenu par Ansys.

Il existe plusieurs descendants de Esterel [19,20,72], comme ReactiveC [19] un portage des concepts synchrones impératifs dans le langage C fait par F. Boussinot en 1991 ou encore ReactiveML [72] dans le langage OCaml fait par L. Mandel et M. Pouzet en 2005. Ces deux langages gagnent en expressivité, mais perdent en garantie par rapport à Esterel. Le langage SL [20], proposé par F. Boussinot et R. de Simone en 1996, est une réponse au problème de causalité présent dans Esterel. En effet, au lieu de laisser le compilateur vérifier que le programme est correct, ils décident de modifier l'interprétation de l'absence d'un signal dans un instant logique. En Esterel, lors d'un test de présence, si le signal n'est pas émis alors le système suppose qu'il ne le sera pas durant le reste de l'instant. En SL, la conséquence de l'absence d'un signal ne s'applique qu'à l'instant suivant, ce qui empêche ce problème de causalité. En effet, si nous reprenons le l'exemple :

```
present s then nothing else emit s
```

1. https://www.ansys.com/

La décision de la présence de `s` au temps t ne sera faite qu'au temps $t + 1$. Par conséquent, si `s` n'est pas présent au temps t alors `s` sera émis au temps $t+1$, ce qui ne brise pas la propriété de causalité. Une conséquence à cela est que SL est moins expressif que ESTEREL, car il interdit tout programme qui réagit instantanément à l'absence, même ceux qui sont causaux. Malgré cela, REACTIVEML suit le choix de SL.

LUSTRE a eu quelques descendants [18, 23, 97] dont LUCIDSYNCHRONE [23] une extension fonctionnelle de LUSTRE ou encore ZELUS [18] un langage déclaratif synchrone avec une extension gérant les équations différentielles ordinaires dans le cadre de simulation. De plus, le compilateur de LUSTRE est certifié [17] en ROCQ.

Une grande réussite industrielle du paradigme réactif synchrone est SCADE [39]. SCADE, qui signifie *Safety Critical Application Development Environment*, est un environnement de développement graphique et un langage réactif synchrone orienté flot de données dérivé de LUSTRE. Il appartient à l'entreprise Ansys qui développe toujours plus d'outils autour de cet environnement. Par exemple, il y a SCADE SUITE qui fournit un générateur de code automatique à partir d'un programme SCADE vers un programme équivalent en C. De plus, ce générateur de code est certifié conformément à plusieurs normes industrielles, notamment DO-178B ou C (aéronautique), EN 50128 :2011 (chemins de fer), ISO 26262 (automobile) et IEC 60880 (nucléaire).

3.2 Paradigme FRP classique

La programmation réactive fonctionnelle classique a été introduite par Conal Elliott et Paul Hudak en 1997 avec la bibliothèque HASKELL nommée FRAN [41, 42]. Leur but était de simplifier la programmation d'interface graphique en HASKELL. Ce paradigme de programmation a donné lieu à diverses applications [83, 84, 88, 101], par exemple du prototypage de systèmes visuels temps-réel, qui ont fourni une exposition conséquente au paradigme. Par conséquent, une communauté s'est crée autour de FRAN et a tenté de l'améliorer [43, 67, 96], de vérifier certaines propriétés [62, 64, 100, 104] ou de le porter à d'autres langages hôtes [21, 30, 32, 73]. Dans cette section, nous faisons un état de l'art de FRP classique en débutant par une explication des concepts du paradigme via le langage FRAN. Puis nous proposons d'étayer un peu plus chaque groupe d'articles mentionnés précédemment.

3.2.1 FRAN : un langage dédié à l'animation

FRAN [42], et plus généralement les langages FRP dit classique, se base sur deux types de signaux : les *comportements* et les *événements* :

— Un *comportement* est une information continue qui évolue dans le temps, par exemple la température ou encore le temps lui-même. Il est modélisé comme une fonction ayant pour domaine le temps :

$$Behavior\ A : Time \to A$$

où A est un type et *Time* le type représentant le temps.

— Un *événement* est une information intermittente dans le temps, par exemple les entrées clavier. Il est modélisé comme une paire composée de la valeur de l'évènement et du moment où il se produit :

$$Event\ A : Time \times A$$

où A est un type.

Le temps est défini comme un réel avec une borne inférieur $-\infty$ et une borne supérieur ∞ dans l'article de 1997. Dans l'implémentation, le type *Time* est un alias du type *double*. Pour construire et manipuler ces signaux, le modèle est équipé des constructeurs/opérateurs primaires *time*, $lift_n$ et $(\cdot\ untilB\ \cdot)$ pour les comportements et *constEv*, $(\cdot \Rightarrow \cdot)$ et *snapshot* pour les évènements.

— *time* est de type *Behavior Time*. Il retourne le temps passé depuis le début de l'exécution.

— $lift_n$ est de type $(A_1 \to \ldots \to A_n \to B) \to Behavior\ A_1 \to \ldots \to Behavior\ A_n \to Behavior\ B$, pour $n > 0$. Il transforme une fonction d'arité n en une fonction de la même arité sur les comportements.

— $(\cdot\ untilB\ \cdot)$ est de type $Behavior\ A \to Event\ (Behavior\ A) \to Behavior\ A$. C'est un opérateur binaire qui prend un comportement et un évènement et retourne un comportement. Il remplace le premier paramètre par celui contenu dans l'évènement lorsque celui-ci se manifeste. Dans d'autres variantes de FRP, cet opérateur s'écrit aussi $(\cdot\ switch\ \cdot)$.

— *constEv* est de type $Time \to A \to Event\ A$. C'est un constructeur d'évènement qui prend un point dans le temps t et une valeur a et qui produit un évènement qui s'activera au temps t avec la valeur a.

— *snapshot* est de type $Behavior\ A \to Event\ Unit \to Event\ A$. C'est une fonction qui prend un comportement b et un évènement e et retourne un évène-

ment e' qui se manifeste en même temps que e avec la valeur échantillonnée de b à cet instant.

— $(\cdot \mathbin{+\!\!=\!\!>} \cdot)$ est de type $\mathit{Event}\ A \to (\mathit{Time} \to A \to B) \to \mathit{Event}\ B$. Cet opérateur applique une fonction sur la valeur de l'évènement pris en paramètre lorsque celui-ci se manifeste. Par conséquent, l'évènement de sortie s'activera au même moment que l'évènement d'entrée.

Exemple 3.3. *Voici, ci-dessous, un ensemble d'expressions* FRAN *rédigé en* HASKELL. *Il est accompagné d'un graphique, dans la figure 3.3, qui représente le changement dans le temps de chaque expression.*

```
estPair :: Behavior Bool
estPair = lift (\t -> mod t 2 == 0) time

ralenti :: Behavior Time
ralenti = lift (\t -> t / 2) time

maj :: Behavior Time
maj = time untilB (constEv 4 ralenti)

echantillon :: Event Time
echantillon = snapshot maj (constEv 7 ())

modifEv :: Event Time
modifEv = (constEv 4 ()) +=> (\t () -> t)
```

Le comportement `estPair` *produit à chaque instant logique la valeur* `true` *si le temps courant arrondi est pair et* `false` *sinon. Le comportement* `ralenti` *divise par deux le temps. Le comportement* `maj` *est le même que* `time` *jusqu'au temps logique* 4 *exclu puis agit comme le comportement* `ralenti`. *Finalement, l'évènement* `modifEv` *retourne le temps courant à l'instant logique* 4.

3.2.2 Applications et Évolution du paradigme

La programmation réactive fonctionnelle classique a démontré au travers des années son applicabilité dans divers domaines. Tout d'abord avec FRAN [42] en 1997, une bibliothèque dédiée à la programmation d'interfaces graphiques. Puis avec FVISION [88] en 1999, une bibliothèque d'orchestration au-dessus de XVISION, un framework écrit en C++, pour du traitement visuel. Enfin, le domaine de la robotique s'est aussi accaparé ce paradigme via les bibliothèques

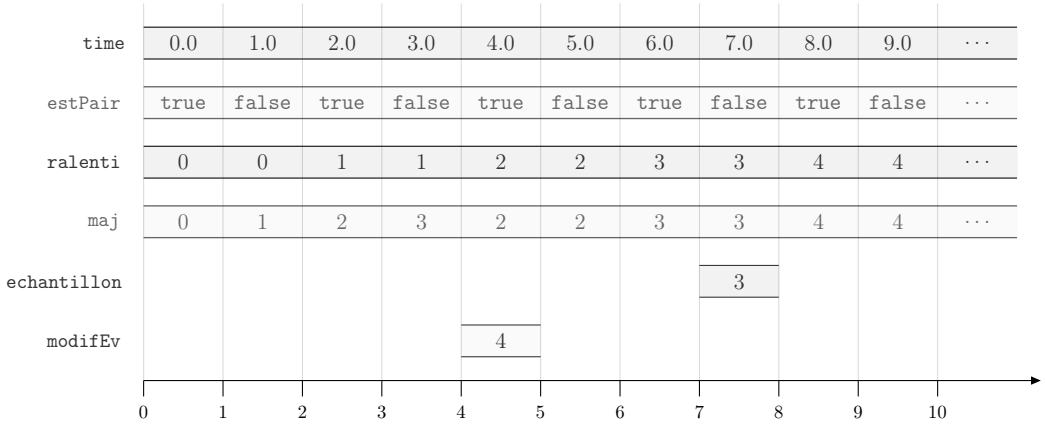


FIGURE 3.3 – Exécution des expressions de l'exemple 3.3. L'abscisse représente le temps logique et chaque ligne correspond à une expression.

FROB [83, 84] en 1999 et E-FRP [101] en 2002. Il y a aussi eu plusieurs propositions de portage des concepts dans un autre langage hôte que HASKELL comme FRAPPÉ [32] en 2001 dans le langage JAVA, FRTIME [30] en 2006 dans le langage RACKET, REACT [21] en 2009, FROC en 2010 dans le langage OCAML, ou encore FLAPJAX [73] en 2009 pour les applications AJAX.

Pour ce qui est de la fondation théorique de la programmation réactive fonctionnelle, il y a eu plusieurs travaux pour garantir des propriétés de vivacité [24, 63] de programmes FRP grâce à un typage utilisant logique temporelle linéaire (LTL) [87].

Finalement, un gros effort de recherche sur les lacunes de FRAN, c'est-à-dire les fuites mémoires et l'inclusion des entrées sorties dans le modèle, a produit de nombreux langages fils dont la sous-branche appelée FRP avec flèches que nous traitons dans la prochaine section.

Nous appelons une fuite mémoire (« space-leak » en anglais) une zone mémoire non libérée alors qu'elle n'est et ne sera plus utilisée.

Exemple 3.4. *Cet exemple est repris de l'article [96] de A. Ploeg et K. Claessen sur* FRPNOW. *La fonction snapshot peut provoquer une fuite mémoire. En effet, cette fonction prend un comportement b et un évènement e et échantillonne b à chaque occurrence de e. Si nous avons le terme*

$$e = \mathit{snapshot\ mousepos\ january}$$

qui échantillonne la position de la souris uniquement le 1er janvier et que e est contenu dans un autre évènement e' qui lui n'apparait que le 1er décembre alors, par sa stratégie d'évaluation paresseuse, le système va devoir stocker les informations de mousepos entre le 1er janvier et le 1er décembre et attendre l'évaluation

de e' pour échantillonner mousepos, ce qui provoque un stockage potentiellement inutile.

Certaines fuites mémoires sont inhérentes aux modèles, que ce soit par la stratégie d'évaluation ou la définition de certains constructeurs alors que d'autres proviennent d'un mauvais développement fait par l'utilisateur de la bibliothèque et plus généralement par la trop grande liberté que le langage hôte donne à l'utilisateur. En effet, FRAN se base sur une évaluation « pull-based » (ou paresseuse), c'est-à-dire que l'échantillonnage d'un comportement ne se fait que lorsque le calcul dans lequel il est impliqué est effectué. Cette stratégie est bien adaptée aux comportements réellement continus dans le temps, comme le temps lui-même, cependant ce n'est pas optimal pour un comportement plus stable, comme la température, où une évaluation « push-based », c'est-à-dire échantillonner dès que la valeur change, serait plus profitable et aurait moins de chance de provoquer de la latence. En 2009, Conal Elliott proposa une révision de la stratégie d'évaluation des signaux dans une nouvelle version de FRAN nommée NEWFRAN [43]. Le modèle d'évaluation de NEWFRAN, nommé « push-pull based », est une composition des deux. D'autres propositions existent [5,67,96] comme FRPNOW [96] défini par A. Ploeg et K. Claessen en 2015. Dans cette variante de FRAN, ils restreignent l'utilisation des signaux, principalement lors de l'utilisation de l'opérateur $\cdot$ *untilB* $\cdot$, ce qui autorise le système à effacer certains historiques de valeurs.

Dans FRAN, il existe quelques primitives qui permettent d'interagir avec l'environnement, par exemple l'opérateur *lbp* est une primitive qui retourne un évènement lorsque qu'il y a un clic gauche de souris. Cependant, il n'y a pas de primitive générale pour les entrées/sorties. Par conséquent, il faut utiliser une monade I/O comme surcouche pour nos comportements et évènements, ce qui complique le développement. Il existe des propositions [96, 102] dont FRPNOW qui, en plus de proposer de corriger les fuites mémoires, fournit un opérateur *async* qui transforme une entrée ou sortie, c'est-à-dire une monade I/O, en évènement.

Un langage FRP qui dénote un peu du reste de ceux présentés précédemment est RT-FRP [100, 104]. Défini par Z. Wan *et al.* en 2001, RT-FRP est langage statiquement typé dans lequel le coût en temps et en espace de chaque étape d'exécution est statiquement connu.

3.3 Paradigme FRP avec flèches

Une des causes de la fuite mémoire de FRAN provient de l'accessibilité des comportements et évènements par l'utilisateur. Il faut, en effet, pouvoir agir dessus, cependant une manipulation directe des signaux n'est pas nécessaire. En

2001, A. Courtney et C. Elliott propose une bibliothèque HASKELL, nommée FRUIT [33], qui définit les fondations du paradigme de programmation réactif fonctionnelle *avec flèches* (AFRP). Ce paradigme utilise les flèches, voir la section 2.3.4, pour définir des transformateurs de flux appelés *fonctions de signal*. Un programme dans ce paradigme se résume donc à une composition de fonctions de signal. Il fut ensuite popularisé par YAMPA [56] en 2002 et est devenu une nouvelle branche du paradigme FRP. Nous commençons par expliquer plus longuement le concept via YAMPA, puis nous explorons les travaux faits autour de ce paradigme et enfin nous nous focalisons sur le langage WORMHOLES et ses concepts.

3.3.1 YAMPA : un langage fonctionnel réactif avec flèches

Généralement, le type des fonctions de signal est noté $sf\ A\ B$ avec A et B des types représentant respectivement le type de la valeur de l'entrée et le type de la valeur de sortie. Une fonction de signal de type $sf\ A\ B$ prend une entrée de type A et produit à la fois une sortie de type B et une version actualisée d'elle-même. Plus précisément, cette exécution de l'instant logique est effectuée par une fonction `step` de type $sf\ A\ B \to A \to B \times sf\ A\ B$. Lors du traitement d'un flux d'entrée, la fonction de signal est appliquée à la tête du flux pour produire la sortie courante. La fonction de signal mise à jour est ensuite appliquée à la queue du flux, ce qui permet à l'état d'évoluer étape par étape. Le temps n'est pas un paramètre des fonctions de signal, c'est le système derrière qui s'en charge. Les constructeurs/opérateurs primaires sont *arr*, *first*, (⋙) et *loop*. Nous définissons leurs comportements ci-dessous et une représentation graphique est fournie dans la figure 3.4.

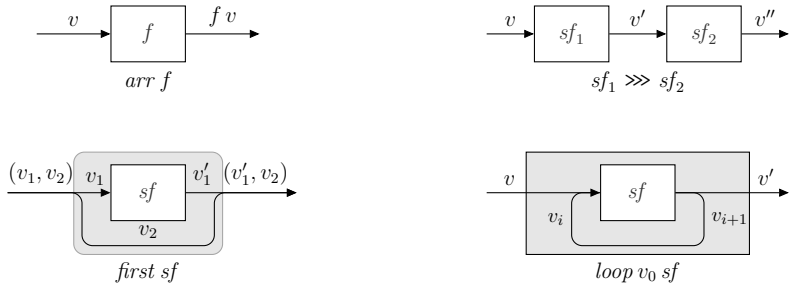


FIGURE 3.4 – Représentation graphique des constructeurs primaires. v, v', v'', ... sont des valeurs, f une fonction et sf, sf_1 et sf_2 des fonctions de signal.

— *arr* est de type $(A \to B) \to sf\ A\ B$. Il transforme une fonction HASKELL en une fonction de signal. Par exemple, l'expression $arr\ (\lambda x.x * 2)$ représente une fonction de signal qui double la valeur numérique d'entrée.

— *first* est de type $sf\ A\ B \to sf\ (A \times C)\ (B \times C)$. Il prend une fonction de signal sf et l'applique sur le premier élément de la paire de valeur d'entrée, laissant donc le second élément inchangé.

— ($\ggg$) est de type $sf\ A\ B \to sf\ B\ C \to sf\ A\ C$. Il compose deux fonctions de signal en séquence.

— *loop* est de type $C \to sf\ (A \times C)\ (B \times C) \to sf\ A\ B$. Étant donné une valeur initiale v et une fonction de signal sf, la fonction de signal $loop\ v\ sf$ contient une valeur de retour, persistante entre instants logiques. La valeur v représente l'état initial du calcul. Dans la figure 3.4, la valeur v_i est donnée en entrée avec v et la valeur v_{i+1} est la nouvelle valeur persistante pour l'instant suivant.

Remarque 3.1. *Ce modèle ressemble énormément à celui proposé dans* Lustre. *En effet, dans* Lustre *les flux ne sont pas des citoyens de première classe non plus.*

Exemple 3.5. *Voici, ci-dessous, un ensemble d'expressions* Yampa *rédigé en* Haskell. *Il est accompagné d'un graphique, dans la figure 3.5, qui représente la valeur courante de chaque expression pour dix instants. La fonction de signal* `time` *est une fonction prédéfinie dans* Yampa *qui retourne le temps en seconde passé depuis le lancement de l'exécution du programme.*

```
estPair :: SF () Bool
estPair = time >>> arr (\t -> mod t 2 == 0)

retard :: SF Time Time
retard = loop 0.0 (arr (\(x,y) -> (y,x)))

entier :: SF () Integer
entier = loop 0 (arr (\((),x) -> (x,x+1)))
```

Le type `Time` *est un double qui représente le temps. La fonction de signal* `estPair` *produit le flux de booléen vrai quand le temps courant est pair et faux sinon. La fonction de signal* `retard` *prend le temps courant et le retourne, mais retardé d'un instant. La première valeur de cette fonction est* 0.0. *Finalement, la fonction de signal* `entier` *produit le flux des entiers naturels.*

3.3.2 Applications et Évolution du paradigme

Les cas d'application de la programmation réactive fonctionnelle avec flèches englobent celles de la programmation fonctionnelle réactive classique. En effet,

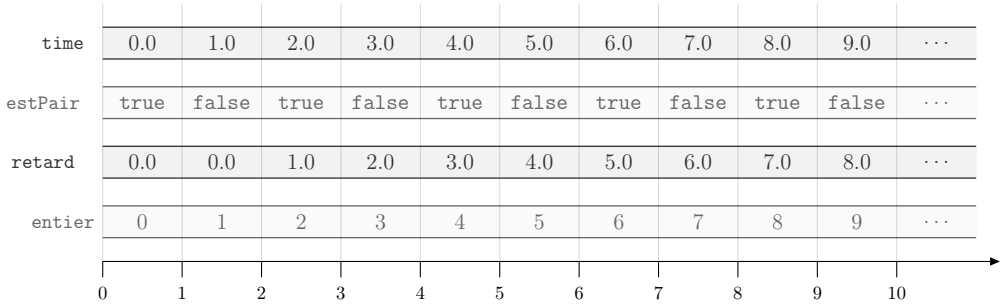


FIGURE 3.5 – Exécution des expressions de l'exemple 3.5. L'abscisse représente le temps logique et chaque ligne correspond à une expression.

il existe une Interface de Programmation d'Applications (API) pour la programmation d'interfaces graphiques nommée λ_{widget} [49] qui a, en plus de cela, de fortes garanties grâce à son système de types utilisant LTL. Nous avons aussi des démonstrations d'application à la robotique via YAMPA [56] proposées par P. Hudak *et al.* en 2002. En plus de cela, il a été montré par A. Courtney *et al.* que YAMPA peut aussi servir au développement de jeux vidéos dans un article [34] de 2003 et que le paradigme AFRP peut être utilisé dans le cadre de l'Internet des Objets (IoT) via HAILSTORM [92], un langage de programmation qui reprend les concepts AFRP et une partie des concepts de WORMHOLES [102] un autre langage AFRP dont nous parlerons dans la suite. Pour finir, un papier de 2023 de I. Perez [80], montre l'utilisation de YAMPA combinée à la bibliothèque HASKELL GLOSS pour de l'animation.

En 2012 A. Jeffrey et W. Jeltsch démontrent indépendamment l'un de l'autre qu'il existe un lien entre LTL et le paradigme FRP classique dans deux articles capitaux [62, 64]. Intuitivement, le système de type proposé mets en relation une formule LTL avec un programme FRP analogue à la correspondance de Curry-Howard [44] qui est établie pour le lambda-calcul simplement typé et la logique intuitionniste. En particulier, A. Jeffrey décompose les fonctions en trois catégories :

1. la catégorie *sans-état* qui est composée de fonctions « simples ». Une fonction *sans-état* est de type $A \to B$, pour A, B deux types, et ne varie pas dans le temps.

2. la catégorie *causale* qui contient une grande partie des fonctions de signal. Une fonction *causale* est de type $A \trianglerighteq B$, pour A, B deux types. Sa sortie dépend de l'entrée courante et de son historique. Cachée derrière $A \trianglerighteq B$ se trouve la formule LTL $\neg\,(A \underline{\mathcal{U}} \neg\, B)$ qui signifie intuitivement qu'il n'existe pas de possibilité d'atteindre B tant que A n'est pas vraie. Par exemple,

l'opérateur *arr* et la composition sont typés comme suit :

$$arr : \forall\ A\ B, \Box(A \to B) \to A \trianglerighteq B$$
$$\ggg\ : \forall\ A\ B\ C, A \trianglerighteq B \to B \trianglerighteq C \to A \trianglerighteq C$$

3. la catégorie *dissociée* qui contient des fonctions de signal. Une fonction *dissociée* est de type $A \triangleright B$, pour A, B deux types. Sa sortie dépend uniquement de l'historique. Cachée derrière $A \triangleright B$ se trouve la formule LTL $\neg\,(A\,\mathcal{U}\,\neg\,B)$ qui signifie intuitivement qu'il n'existe pas de possibilité d'atteindre B tant que A n'a pas été vraie précédemment. Par exemple, la fonction de signal *constant* qui retourne une même valeur indépendamment de l'entrée est une fonction *dissociée* et est typée comme suit :

$$constant : \forall\ A\ B, \Box B \to A \triangleright B$$

L'entièreté de la formalisation proposée par A. Jeffrey a été mécanisée en Agda, un assistant de preuve, via des types dépendants et un lien avec les catégories ventilateurs (fan categories) a été démontré par W. Jeltsch.

Yampa est le langage AFRP le plus populaire de sa branche. Toujours activement en développement, son implémentation via la « typeclass » `Arrow` de Haskell, les notations de Paterson [77] qui simplifie l'écriture de composition de flèches, ainsi que le nombre de fonctions de signal prédéfinies et d'extensions accumulées depuis 2002 permettent un développement simple et efficace dans ce langage, il existe même une formalisation en Agda du coeur de Yampa [93]. Cependant, Yampa a, lui aussi, un problème d'inclusion des entrées/sorties, et plus généralement des interactions avec l'environnement, comme FRAn. Dans ce paradigme, il n'est pas possible d'intégrer simplement une interaction mémoire « au milieu » du programme, car un programme est une composition de fonctions de signal et toute interaction d'entrée ou de sorties est faite avant ou après celui-ci. Cette problématique est la base du travail de Wormholes [102, 103] et est, en partie, résolu dans le langage MSF [81].

Wormholes [103] est une bibliothèque Haskell implémentée en 2012 par D. Winograd-Cort et P. Hudak qui a été formalisée [102] durant la même année. Dans ce langage, l'utilisateur peut interagir avec l'environnement via des *ressources*. Ces ressources sont accessibles via un nouvel opérateur *rsf*. Il est aussi possible de simuler une référence locale grâce à l'opérateur *wormhole*, ce qui permet de définir des programmes avec une mémoire entre instants. Dans Yampa, le *loop* a déjà ce rôle, cependant, dans Wormholes il n'est pas présent. En effet, *wormhole* et *rsf* permettent de définir la boucle réactive, ce qui donne à

Wormholes un langage plus fin que le coeur de Yampa. Ces interactions sont limitées à une par ressource et instant logique afin de garantir l'*hypothèse synchrone*. Intuitivement, Wormholes autorise à localiser les interactions dans le programme sans perdre les garanties d'un programme Yampa. Cette variante offre une gestion d'effets de bords dans un style impératif tout en préservant la compositionnalité du programme et sa pureté. Un descendant direct de Wormholes est Hailstorm [92] un langage défini pour la programmation d'application IoT (Internet des Objets). Il récupère le concept de ressources, mais il se sépare du lieur syntaxique *wormhole* et par conséquent des ressources locales. Le langage est divisé en deux niveaux : la partie fonction de signal et la partie expressions, ce qui diffère aussi de Wormholes. Un point intéressant d'Hailstorm est que ce langage dédié n'est pas embarqué en Haskell comme ses prédécesseurs. En effet, il est compilé vers du C.

MSF [81] (Monadic Stream Function) est un langage AFRP proposé par I. Perez *et al.* en 2016 et implémenté sous forme d'une bibliothèque nommée Dunai en Haskell. Intuitivement, cette bibliothèque propose d'encapsuler le résultat de la fonction de signal dans une monade pour lui appliquer divers effets. Le type des fonctions de signal est $msf\ m\ A\ B$ avec A le type du flux d'entrée, B le type du flux de sortie et m la monade appliquée sur le résultat de l'évaluation d'un instant logique. L'évolution étape par étape ne change pas et l'application sur un flux est analogue, cependant la fonction `step` est de type $msf\ m\ A\ B \rightarrow A \rightarrow m\ (B \times msf\ m\ A\ B)$. MSF propose un modèle plus général que Yampa qui permet d'appliquer différentes monades tout en préservant l'esprit d'Haskell et de la programmation fonctionnelle pure. À noter que l'ajout de la monade permet d'interagir avec l'environnement « au milieu » du programme mais cela ne rend pas plus simple l'ajout ou la modification dans celui-ci. MSF a été utilisé dans divers cas depuis 2016 [7,78,79,82], par exemple, le langage RhineFRP [7] utilise MSF pour définir des fonctions de signal ayant une horloge qui leur est propre, ce qui permet de définir un programme avec plusieurs horloges.

3.3.3 Wormholes : un langage fonctionnel réactif avec effets

Wormholes introduit deux nouveaux opérateurs, en plus de ceux présentés dans la section 3.3.1, ainsi que le concept de *ressource*. Dans cette section, nous expliquons informellement ces ajouts via de l'intuition, des exemples et des supports visuels. Nous nous autorisons à utiliser des expressions contenant des paires, des entiers naturels et des opérateurs arithmétiques dans nos exemples.

Ressource

Une *ressource* représente une interaction avec l'environnement et se définit par un identifiant, principalement noté r, et une *cellule mémoire locale*. Une ressource peut être la lecture dans une zone mémoire, l'écriture dans un fichier, l'envoie d'un dossier à imprimer ou encore la mise à jour de l'état d'un système tiers. Par conséquent, la simulation d'une référence se fait avec deux ressources : une pour la lecture et une pour l'écriture.

La cellule mémoire locale est stockée dans une mémoire tampon qui s'initialise au début de l'instant logique avec les valeurs effectives de l'environnement. Elle est donc locale à l'instant et au programme. Dans WORMHOLES, les interactions sont uniques durant un instant logique. Cette contrainte se vérifie statiquement et dynamiquement dans la sémantique. Afin de représenter cette contrainte dans la sémantique dynamique, une cellule mémoire locale se voit attribuer un état. Il y a deux états possibles : *utilisée* ou *non-utilisée*. Dans le fonctionnement, cela est symbolisé par un booléen : vrai si non utilisée et faux sinon. Graphiquement, nous représentons les cellules comme un boîte au coin arrondi avec un fond qui diffère en fonction de l'état.

L'état « non-utilisé », aussi nommé « initial », a un fond blanc et l'état « utilisé » a un fond gris. Dans le reste du chapitre, nous utiliserons le mot « ressource » pour « identifiant de ressource » et « cellule » au lieu de « cellule mémoire locale » afin d'alléger la lecture.

Fonction de signal avec ressource

L'interaction avec l'environnement ne peut se faire qu'à travers une fonction de signal $rsf[r]$, où r est une ressource. Elle procède à un échange : la valeur stockée par la ressource est retournée en échange de la valeur du signal d'entrée.

Exemple 3.6. *Dans la figure 3.6, la ressource r est associée à une cellule nommée c, qui contient la valeur* `tt`*. c récupère la valeur du signal d'entrée, ici* 2*, et la remplace par* `tt`*. La valeur* 2 *est donc stockée dans la cellule mémoire c jusqu'à la fin de l'instant.*

Comme le montre l'exemple ci-dessus, l'opérateur *rsf* ne peut s'appliquer que sur une ressource dont la mémoire est étiquetée « non-utilisée ». De plus, l'état de la mémoire change après application de la fonction de signal et devient « utilisée », bloquant toutes possibilités de réutilisation. Intuitivement, au début d'un instant logique le programme va initialiser ces cellules avec des valeurs provenant de

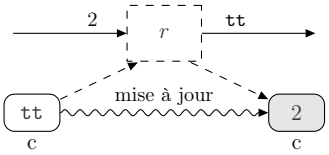


FIGURE 3.6 – Représentation graphique de la fonction de signal $rsf[r]$.

l'environnement avec l'état « non-utilisée », puis il va effectuer le calcul et enfin récupérer les valeurs des cellules « utilisée » pour les renvoyer à l'environnement. Pour être plus précis, c'est le cycle de vie d'une cellule associée à une ressource *libre*, le cas des ressources liées est détaillé dans la section 3.3.4.

Exemple 3.7. *Soit P un programme, défini ci-dessous, qui prend en entrée une paire (t_1, t_2) et retourne un résultat t. Son calcul de t se décompose en deux parties : 1. il applique d'abord un nombre $n-1$ de fonctions sur t_1, ce qui donne t'_1 ; 2. puis il applique une fonction f_n sur (t'_1, t_2). Il est représenté dans la figure 3.7.*

$$first\,(arr\,f_1 \ggg \ldots \ggg arr\,f_{n-1}) \ggg arr\,f_n$$

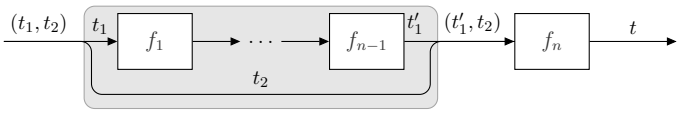


FIGURE 3.7 – Représentation graphique du programme P.

Dans WORMHOLES, *et comme nous le verrons plus tard, un programme ne prend qu'une valeur de type Unit et toutes les entrées proviennent de ressources. Par conséquent, nous définissons un programme P', détaillé ci-dessous, qui effectue la même tâche en utilisant une ressource r_{t_1}, qui échange* `tt` *contre t_1, une ressource r_{t_2}, qui agit comme r_{t_1} mais pour t_2 et une ressource r_t qui échange t contre un* `tt`*. Il est représenté dans la figure 3.8. Pour rappel, le combinateur second est le dual du combinateur first.*

$$\begin{aligned} &rsf[r_{t_1}] \ggg arr\,f_1 \ggg \ldots \ggg arr\,f_{n-1} \ggg \\ &\qquad arr\,(\lambda x.(x, \texttt{tt})) \ggg second\,(rsf[r_{t_2}]) \ggg arr\,f_n \ggg rsf[r_t] \end{aligned}$$

Là où les expressions t_1 et t_2 sont introduits au début par le système dans le programme P, un programme WORMHOLES, *comme P' va demander d'introduire manuellement les expressions via des ressources.*

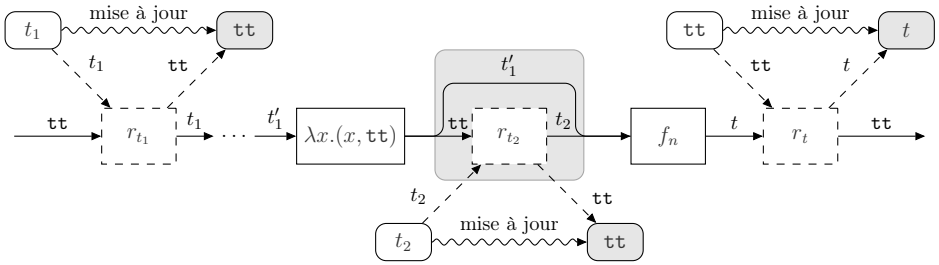


FIGURE 3.8 – Représentation graphique du programme P'.

3.3.4 Ressources locales et lieurs

L'opérateur *wormhole* est un *lieur* réactif. Le terme $\mathit{wormhole}[r_r, r_w](t_1; t_2)$ rend disponible les ressources r_r et r_w dans t_2 sachant que r_r est initialisée avec t_1 et r_w avec `tt`, le terme de type *Unit* (voir figure 3.9). Les ressources représentent respectivement la lecture unique et l'écriture unique. Mises ensemble, ces deux ressources composent une *référence locale* que nous pouvons lire et modifier une fois par instant. Les ressources liées ainsi que leur valeur courante, dans le cas initial la valeur est t_1, sont sauvegardées par le système pour un traitement différent des ressources libres. En effet, à la place de récupérer au début de l'instant une valeur dans l'environnement, nous récupérons celle sauvegardée, ensuite le calcul est effectué, puis finalement si la ressource r_w est dans l'état « utilisée », c'est-à-dire que nous avons une nouvelle valeur pour notre référence, alors nous remplaçons la valeur sauvegardée par la nouvelle. Cela permet de simuler une référence locale dont l'écriture est prise en compte à l'instant suivant. Cet opérateur permet donc de définir des fonctions de signal avec mémoire tampon comme le propose la boucle réactive *loop* (voir la section 2.3.4).

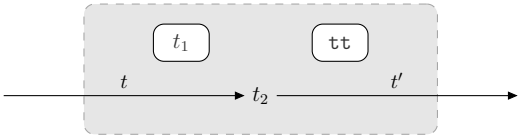


FIGURE 3.9 – Représentation graphique de la fonction de signal $\mathit{wormhole}[r_r, r_w](t_1; t_2)$ où r_r est associée à la cellule de gauche et r_w à celle de droite.

Remarque 3.2. *Dès que le système sauvegarde ce triplet d'information, c'est-à-dire les deux ressources et la valeur initiale, il n'est plus nécessaire de maintenir le lieur réactif, car le système a maintenant connaissance des ressources à traiter différemment. Seule la fonction de signal t_2 devient indispensable.*

Exemple 3.8. *La fonction delay prend une valeur initiale v_0 et retourne une fonction de signal qui décale d'un instant logique la valeur d'entrée du signal en*

commençant par v_0. Dans le langage WORMHOLES*, cette fonction est définie ci-dessous. Une version graphique, avec v_0 fixée, est disponible dans la figure 3.10.*

$$delay = \lambda v_0.(wormhole[0,1](v_0; rsf[1] \ggg rsf[0]))$$

Au début du premier instant logique, les ressources 0 *et* 1 *sont initialisées, avec l'état « non-utilisée », respectivement par v_0 et* `tt`*, et le système sauvegarde le triplet $(0, 1, v_0)$. Puis le calcul s'effectue, supposons avec v comme valeur du signal d'entrée. $rsf[1]$ récupère v, par conséquent la cellule de* 1 *contient v et devient « utilisée », et retourne* `tt`*. $rsf[0]$ récupère* `tt`*, par conséquent la cellule de* 0 *devient « utilisée », et retourne la valeur v_0. À la fin de cet instant, la ressource d'écriture, c'est-à-dire* 1*, a été utilisée donc le système récupère sa valeur, c'est-à-dire v, et remplace v_0 par v dans le triplet sauvegardé. Un nouvel instant peut débuter.*

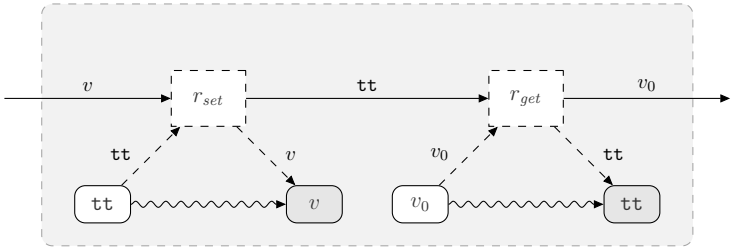


FIGURE 3.10 – Représentation graphique de la fonction de signal $delay_{v_0}$.

Chapitre 4

Sémantique formalisée de Wormholes

Sommaire

Dans Yampa, l'interaction avec l'environnement passe par un unique signal d'entrée et un unique signal de sortie. En effet, par son côté synchrone, il échantillonne les informations de l'environnement au début de son instant logique avant d'effectuer une étape de calcul qui produit une valeur de sortie transmise à l'environnement. De cette manière, l'environnement peut voir le pas d'un programme Yampa comme immédiat, ou encore comme prenant un temps nul. De plus, cela permet à l'utilisateur de construire son programme sans se soucier des effets de bords, car d'un point de vue interne, il n'y en pas. Cependant, toutes ces garanties et simplifications apportent un challenge quand il s'agit de maintenir ou modifier un programme existant.

Comparons notre programme à une chaîne de montage dans une usine : le signal est le tapis roulant, une fonction de signal *arr f* est un poste de travail

effectuant une tâche f, une fonction de signal *first sf* est un poste de travail dont le tapis s'est séparé en deux pour ne donner que le premier élément au poste *sf* et une fonction *loop v sf* un poste de travail qui garde un élément en stock. Supposons maintenant que nous voulons fabriquer des étagères sachant que le tapis roulant fournit une bûche par une bûche. Nous définissons quatre postes de travail :

1. `pl` un poste de travail qui prend une bûche et produit quatre planches ;
2. `demi` un poste de travail qui prend une planche et produit deux demi-planches ;
3. `base` un poste de travail qui prend deux planches et deux demi-planches et produit une étagère sans tablettes ;
4. et `finition` un poste de travail qui prend une étagère sans tablettes et deux demi-planches et produit une étagère avec deux tablettes.

Graphiquement, la chaine de montage, et donc le programme, est défini dans la figure 4.1. L'expression t associée à cet énoncé et cette représentation graphique est la suivante :

$$
\begin{aligned}
t = \; & \mathit{arr}\,\texttt{pl} \ggg \\
& \mathit{first}\,(\mathit{arr}\,\texttt{demi} \;***\; \mathit{arr}\,\texttt{demi}) \ggg \\
& \mathit{second}\,(\mathit{arr}\,\texttt{base}) \ggg \\
& \mathit{arr}\,\texttt{finition}
\end{aligned}
$$

Maintenant, nous rajoutons la condition que le poste `base` requiert un sac de vis pour produire la base de l'étagère. La seule solution possible dans ce cas est de prendre en plus de la bûche un sac de vis qu'il faut transporter sur le côté en attendant d'arriver au poste voulu. La nouvelle représentation graphique est donnée dans la figure 4.2 et l'expression t est légèrement modifiée comme suit :

$$
\begin{aligned}
t = \mathit{first}\,(& \\
& \mathit{arr}\,\texttt{pl} \ggg \\
& \mathit{first}\,(\mathit{arr}\,\texttt{demi} \;***\; \mathit{arr}\,\texttt{demi})) \ggg \\
& \mathit{second}\,(\mathit{arr}\,\texttt{base}) \ggg \\
& \mathit{arr}\,\texttt{finition}
\end{aligned}
$$

Ce genre de modification au cours du temps est courant pour un programme, cependant cela soulève deux problèmes. Le premier est qu'un programme YAMPA s'alourdi avec les modifications et perd en lisibilité même dans des parties de

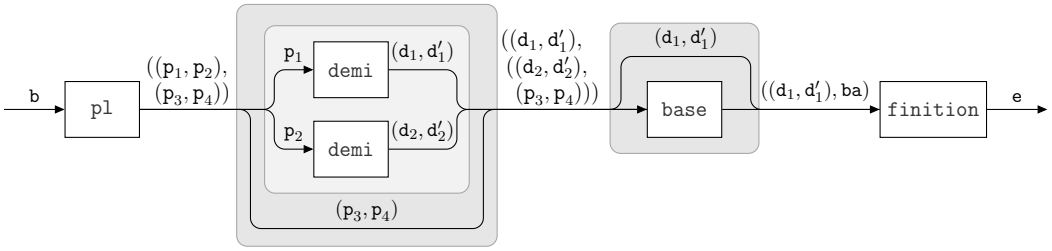


FIGURE 4.1 – Représentation graphique de la création d'étagère.

programme qui ne devrait pas être affectées. En effet, pour amener le sac de vis, tout le programme jusqu'à `base` a été modifié. Le deuxième problème est associé à la conception. Il est étrange de faire passer le sac de vis au même endroit que les bûches pour l'emmener à un seul poste de travail. En effet, il serait plus naturel de le donner uniquement au poste `base` lorsqu'il en a besoin.

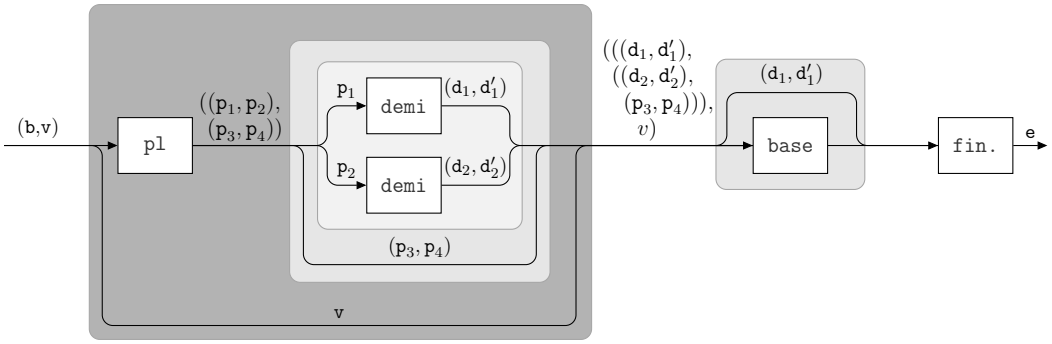


FIGURE 4.2 – Représentation graphique de la création d'étagère.

La gestion des entrées/sorties en YAMPA est connue pour être difficile. WORMHOLES [102] donne une première solution en 2012, suivie par d'autres [81,92] dont MSF. WORMHOLES propose d'interagir avec l'environnement dans le programme via deux nouveaux opérateurs : *rsf* et *wormhole* qui se basent sur un concept de ressource (voir la section 3.3.3). Le chapitre 5 est une propose une relaxation de la contrainte d'utilisation dans certains cas. Mais avant de proposer une extension de Wormholes, il convient de s'assurer de la solidité des bases de ce langage. Ce chapitre décrit donc le travail de correction et de formalisation, dans un assistant de preuve, du langage WORMHOLES. Dans ce chapitre, nous définissons principalement la variante du langage que nous proposons, puis nous passons rapidement sur la mécanisation (taille et structure du projet) afin que le lecteur curieux puisse aller voir comment l'implémentation en ROCQ a été faite.

4.1 Sémantique formelle d'une variante de WORMHOLES

Lors du travail de formalisation, nous avons remarqué des problèmes dans la description formelle originale du langage. Certaines propriétés du langage ne sont pas satisfaites par les définitions proposées dans l'article original. Par exemple, la progression n'est pas maintenue par l'entièreté de la sémantique dynamique à cause de blocages possibles lors de l'évaluation de l'instant logique (voir la section 4.1.4). Malgré ces problèmes dans les définitions, le langage WORMHOLES introduit des concepts et des propriétés intéressants qui nous semblaient plausibles. Par conséquent, nous avons décidé de suggérer de nouvelles définitions, proche de la version originale, qui satisferont les propriétés établies pour le langage.

Cette section définit formellement le langage avec les corrections que nous avons apportées et les lemmes et théorèmes qui lui sont associés. Elle se découpe en cinq parties. La première définit la syntaxe du langage, la deuxième définit la sémantique statique et les trois dernières sont réservées à la sémantique dynamique. La sémantique dynamique, c'est-à-dire le comportement d'une expression durant son évaluation est décomposée en trois transitions :

1. la *transition d'évaluation* qui rassemble l'ensemble des règles de réduction d'une expression vers une forme normale ;
2. la *transition fonctionnelle* qui effectue un instant logique en évaluant une entrée via le programme, qui est une fonction de signal, et en retournant un résultat ;
3. la *transition temporelle* qui simule la boucle d'évaluation et gère la récupération des ressources libres, l'initialisation des ressources libres et liées, le renvoie des valeurs associées aux ressources libres dans l'environnement et la mise à jour des valeurs associées aux ressources liées.

Une sous-section est donc réservée pour chaque transition. Les modifications ainsi que les ajouts complets de notre part dans les définitions seront encadrés dans la suite.

4.1.1 Syntaxe

La syntaxe de WORMHOLES se compose de la syntaxe d'un lambda-calcul étendu avec un terme de base `tt`, des paires, des projections et un point-fixe ainsi que les opérateurs de YAMPA *arr*, *first* et (⋙). Pour finir, elle contient aussi les deux nouveaux opérateurs précédemment définis informellement. La syntaxe

est définie ci-dessous.

$$\begin{aligned} & \boxed{Res : \mathbb{N}} \qquad X : \mathbb{N} \\ t = \mathtt{tt} \mid X \mid \lambda X.t \mid t\ t \mid (t,t) \mid \mathit{fst}\, t \mid \mathit{snd}\, t \mid & \boxed{\mathit{fix}\, t} \\ \mid \mathit{arr}\, t \mid \mathit{first}\, t \mid t \ggg t \mid & \mathit{rsf}[Res] \mid \mathit{wormhole}[Res, Res](t;t) \end{aligned}$$

La partie lambda-calcul est classique donc non expliquée ici, les opérateurs communs à YAMPA fonctionnent de la même manière et les deux nouveaux opérateurs ont été expliqués dans la section précédente. L'ensemble des variables X se doit d'être infini dénombrable d'où notre choix de les représenter par les entiers naturels. L'argument est quelque peu différent pour *Res*, l'ensemble des ressources, car nous utilisons les entiers naturels dans le cadre des niveaux de De-Bruijn. Cette représentation permet d'éviter les problèmes de capture de variables. Ce choix de représentation est expliquée plus en détail dans la section A.3.

Convention 4.1. *Dans cette section, nous supposons que tous les résultats énoncés sont vrais par l'intermédiaire d'une équivalence et d'une fonction de renommage des identifiants analogue à l'*α*-équivalence pour les variables.*

Exemple 4.1. *La boucle réactive de* YAMPA *n'est pas présente dans la syntaxe de* WORMHOLES*, car cela n'est pas nécessaire. En effet, les nouveaux opérateurs sont plus bas niveaux que l'opérateur loop, par conséquent il est possible de définir celui-ci dans le langage* WORMHOLES *comme suit (voir figure 4.3) :*

$$\begin{aligned} \mathit{loop}\, t_1\, t_2 = \mathit{wormhole}[r_{get}, r_{set}](t_1;\ & \mathit{arr}\, (\lambda x.(x, \mathtt{tt})) \ggg \\ & \mathit{second}\, (\mathit{rsf}[r_{get}]) \ggg \\ & \mathit{arr}\ \mathit{swap} \ggg \\ & t_2 \ggg \\ & \mathit{arr}\ \mathit{swap} \ggg \\ & \mathit{second}\, (\mathit{rsf}[r_{set}]) \ggg \\ & \mathit{arr}\ \mathit{snd}) \end{aligned}$$

Un problème rencontré dans l'article original est la mention d'une récursion sans sa présence dans la syntaxe. Nous avons donc rajouté un point-fixe. Cet ajout n'est pas anodin, car il brise la propriété de normalisation du langage, c'est-à-dire que tout programme bien typé ne peut pas forcément terminer. Cette modification a donc une influence sur l'ensemble des propriétés de progression.

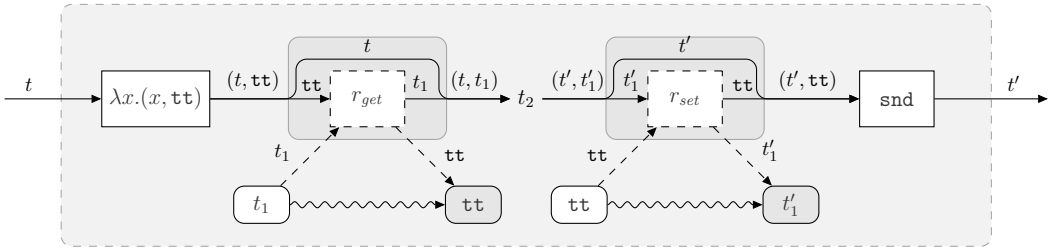


FIGURE 4.3 – Boucle réactive définie à partir du langage WORMHOLES.

4.1.2 Sémantique statique

WORMHOLES est un langage statiquement typé et ce typage garanti une utilisation unique de toute ressource présente dans l'expression. L'ensemble des types est défini ci-dessous :

$$\tau ::= \mathit{Unit} \mid \tau \times \tau \mid \tau \to \tau \mid \tau \overset{R}{\rightsquigarrow} \tau$$

Nous notons τ, τ_1, τ_2, ... des types du langage. Le dernier type est celui des fonctions de signal. Il reprend la notation du arrow-calcul [90] avec une légère variante. En effet, en plus du type d'entrée et de sortie de la fonction de signal, le type porte un ensemble de ressources R. Cet ensemble représente l'ensemble des interactions effectives de l'expression. Le système de type utilise cet ensemble pour garantir une utilisation conforme des ressources.

Définition de la sémantique statique

Nous notons $\Gamma \cdot \Re \vdash t : \tau$ le fait que l'expression t est de type τ selon les contextes Γ et $\Re$ et qu'elle respecte les règles des figures 4.4 et 4.5. Γ est un contexte qui associe une variable à un type et $\Re$ un contexte qui associe une ressource à une paire de types. Le premier type représente le type de la cellule associée avant utilisation et le second après utilisation. Les deux contextes sont des dictionnaires (voir la section 2.1.2). Les règles classiques du lambda-calcul étendu ont été rajoutées par rapport à l'article original, mais ne sont pas expliquées ici. Nous préférons nous focaliser sur la partie fonctions de signal (voir la figure 4.5).

La fonction de signal $(\mathit{arr}\, t)$ élève une fonction simple dans le monde réactif, elle a donc, comme la règle `Ty-Arr` l'indique, les mêmes types d'entrée et de sortie que la fonction t. De plus, la fonction t est simple et ne peut donc pas utiliser de ressources, par conséquent son ensemble de ressources est vide.

Exemple 4.2. *Soit* $f = \lambda x.(1, x)$ *une fonction, la fonction de signal* $(\mathit{arr}\, f)$ *est*

$$\frac{}{\Gamma \cdot \Re \vdash \texttt{tt} : \mathit{Unit}}\ \texttt{Ty-Unit} \qquad \frac{\Gamma\, x = \tau}{\Gamma \cdot \Re \vdash x : \tau}\ \texttt{Ty-Var} \qquad \frac{\Gamma[x \mapsto \tau_1] \cdot \Re \vdash t : \tau_2}{\Gamma \cdot \Re \vdash \lambda x.t : \tau_1 \to \tau_2}\ \texttt{Ty-Lam}$$

$$\frac{\Gamma \cdot \Re \vdash t : \tau_1 \times \tau_2}{\Gamma \cdot \Re \vdash \mathit{fst}\, t : \tau_1}\ \texttt{Ty-Fst} \qquad \frac{\Gamma \cdot \Re \vdash t : \tau \to \tau}{\Gamma \cdot \Re \vdash \mathit{fix}\, t : \tau}\ \texttt{Ty-Fix}$$

$$\frac{\Gamma \cdot \Re \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \cdot \Re \vdash t_2 : \tau_1}{\Gamma \cdot \Re \vdash t_1\ t_2 : \tau_2}\ \texttt{Ty-App}$$

$$\frac{\Gamma \cdot \Re \vdash t_1 : \tau_1 \quad \Gamma \cdot \Re \vdash t_2 : \tau_2}{\Gamma \cdot \Re \vdash (t_1, t_2) : \tau_1 \times \tau_2}\ \texttt{Ty-Pair} \qquad \frac{\Gamma \cdot \Re \vdash t : \tau_1 \times \tau_2}{\Gamma \cdot \Re \vdash \mathit{snd}\, t : \tau_2}\ \texttt{Ty-Snd}$$

FIGURE 4.4 – Règles de bon typage pour la partie lambda-calcul.

de type $(\tau \overset{\varnothing}{\rightsquigarrow} \tau \times \mathbb{N})$ *et satisfait la dérivation suivante :*

$$\cfrac{\cfrac{\cfrac{\cfrac{}{[x \mapsto \tau] \cdot \varnothing \vdash 1 : \mathbb{N}} \quad \cfrac{[x \mapsto \tau]\, x = \tau}{[x \mapsto \tau] \cdot \varnothing \vdash x : \tau}\ \texttt{Ty-Var}}{[x \mapsto \tau] \cdot \varnothing \vdash (1, x) : \tau \times \mathbb{N}}\ \texttt{Ty-Pair}}{\varnothing \cdot \varnothing \vdash \lambda x.(1, x) : \tau \to (\tau \times \mathbb{N})}\ \texttt{Ty-Lam}}{\varnothing \cdot \varnothing \vdash \mathit{arr}\, f : \tau \overset{\varnothing}{\rightsquigarrow} \tau \times \mathbb{N}}\ \texttt{Ty-Arr}$$

La fonction de signal (*first* t) quant à elle applique sa fonction de signal interne au premier élément du signal d'entrée (voir la règle `Ty-First`). Son ensemble de ressource est celui de sa fonction de signal interne.

Exemple 4.3. *Soit* (*arr* f) *la fonction de signal de l'exemple précédent. La fonction de signal* (*first* (*arr* f)) *est de type* $(\tau_1 \times \tau_2 \overset{\varnothing}{\rightsquigarrow} (\tau_1 \times \mathbb{N}) \times \tau_2)$ *comme l'indique la dérivation suivante :*

$$\cfrac{\cfrac{\textit{voir exemple 4.2}}{\varnothing \cdot \varnothing \vdash \mathit{arr}\, f : \tau_1 \overset{\varnothing}{\rightsquigarrow} \tau_1 \times \mathbb{N}}\ \texttt{Ty-Arr}}{\varnothing \cdot \varnothing \vdash \mathit{first}(\mathit{arr}\, f) : \tau_1 \times \tau_2 \overset{\varnothing}{\rightsquigarrow} (\tau_1 \times \mathbb{N}) \times \tau_2}\ \texttt{Ty-First}$$

Comme expliqué dans la section 3.3.3, la fonction de signal ($rsf[r]$), pour une ressource r, procède à un échange. Dans le contexte de ressources $\Re$, toutes ressources est associée à une paire de type représentant respectivement le type de la valeur que nous lui échangeons et le type de la valeur qu'elle contient au moment de l'échange. Le type de la fonction de signal est donc directement tiré du type de sa ressource, comme l'indique la règle `Ty-Rsf`. Cette règle est la seule qui permet d'ajouter directement une ressource à l'ensemble de ressources porté

$$\frac{\Gamma \cdot \Re \vdash t : \tau_1 \to \tau_2}{\Gamma \cdot \Re \vdash \mathit{arr}\, t : \tau_1 \xrightarrow{\varnothing} \tau_2}\ \texttt{Ty-Arr}$$

$$\frac{\Gamma \cdot \Re \vdash t : \tau_1 \xrightarrow{R} \tau_2}{\Gamma \cdot \Re \vdash \mathit{first}\, t : \tau_1 \times \tau \xrightarrow{R} \tau_2 \times \tau}\ \texttt{Ty-First} \qquad \frac{\Re\, r = \langle \tau_1, \tau_2 \rangle}{\Gamma \cdot \Re \vdash \mathit{rsf}[r] : \tau_1 \xrightarrow{\{r\}} \tau_2}\ \texttt{Ty-Rsf}$$

$$\frac{\Gamma \cdot \Re \vdash t_1 : \tau_1 \xrightarrow{R_1} \tau \quad \Gamma \cdot \Re \vdash t_2 : \tau \xrightarrow{R_2} \tau_2 \quad R_1 \cap R_2 = \varnothing}{\Gamma \cdot \Re \vdash t_1 \ggg t_2 : \tau_1 \xrightarrow{R_1 \cup R_2} \tau_2}\ \texttt{Ty-Comp}$$

$$\frac{\boxed{\{r_{set}; r_{get}\} \cap (\mathit{refs}\ \tau_1 \cup \mathit{refs}\ \tau_2) = \varnothing} \quad \boxed{r_{set} \neq r_{get}} \quad R = R' \backslash \{r_{get}; r_{set}\} \quad \Gamma \cdot \Re \vdash t_1 : \tau \quad \Gamma \cdot \Re[r_{get} \mapsto \langle \mathit{Unit}, \tau \rangle][r_{set} \mapsto \langle \tau, \mathit{Unit} \rangle] \vdash t_2 : \tau_1 \xrightarrow{R'} \tau_2}{\Gamma \cdot \Re \vdash \mathit{wormhole}[r_{get}, r_{set}](t_1; t_2) : \tau_1 \xrightarrow{R} \tau_2}\ \texttt{Ty-Wh}$$

FIGURE 4.5 – Règles de bon typage pour les fonctions de signal.

par le type de fonction de signal.

Exemple 4.4. *Soient* 0 *une ressource et* $[0 \mapsto \langle \mathbb{N}, \mathit{Unit} \rangle]$ *un contexte de ressources, la fonction de signal* $(\mathit{rsf}[0])$ *est de type* $(\mathbb{N} \xrightarrow{\{0\}} \mathit{Unit})$ *et satisfait la dérivation suivante :*

$$\frac{[0 \mapsto \langle \mathbb{N}, \mathit{Unit} \rangle]\, 0 = \langle \mathbb{N}, \mathit{Unit} \rangle}{\varnothing \cdot [0 \mapsto \langle \mathbb{N}, \mathit{Unit} \rangle] \vdash \mathit{rsf}[0] : \mathbb{N} \xrightarrow{\{0\}} \mathit{Unit}}\ \texttt{Ty-Rsf}$$

La règle `Ty-Comp` définit ce qu'est une composition de deux fonctions de signal correcte et porte la contrainte clé pour garantir l'utilisation unique de chaque ressource. En effet, l'ensemble de ressource utilisé par une composition est l'union des ensembles de ses sous-expressions. De plus, il est nécessaire que ces ensembles soient *disjoints*. Cette simple contrainte permet d'éviter une utilisation multiple de la même ressource.

Exemple 4.5. *Soient* 0 *et* 1 *deux ressources et* $\Re = [0 \mapsto \langle \mathbb{N}, \mathbb{N} \rangle][1 \mapsto \langle \mathit{Unit}, \mathbb{N} \rangle][2 \mapsto \langle \mathbb{N}, \mathit{Unit} \rangle]$ *un contexte. La fonction de signal* $\mathit{rsf}[1] \ggg \mathit{rsf}[0]$ *est bien typée comme le montre la dérivation de typage suivante :*

$$\frac{\dfrac{\Re\, 1 = \langle \mathit{Unit}, \mathbb{N} \rangle}{\varnothing \cdot \Re \vdash \mathit{rsf}[1] : \mathit{Unit} \xrightarrow{\{1\}} \mathbb{N}}\ \texttt{Ty-Rsf} \qquad \dfrac{\Re\, 0 = \langle \mathbb{N}, \mathbb{N} \rangle}{\varnothing \cdot \Re \vdash \mathit{rsf}[0] : \mathbb{N} \xrightarrow{\{0\}} \mathbb{N}}\ \texttt{Ty-Rsf} \qquad \{1\} \cap \{0\} = \varnothing}{\varnothing \cdot \Re \vdash \mathit{rsf}[1] \ggg \mathit{rsf}[0] : \mathit{Unit} \xrightarrow{\{0;1\}} \mathbb{N}}\ \texttt{Ty-Comp}$$

Une mauvaise composition de fonctions de signal utilisant des ressources serait $rsf[0] \ggg rsf[0]$. *En effet, si nous tentons de faire la dérivation du typage, comme ci-dessus, nous aurons* $\{0\} \cap \{0\} \neq \varnothing$.

Une fonction de signal $wormhole[r, r'](t_1, t_2)$ agit comme un lieur. Elle initialise r et r' respectivement avec t_1 et `tt` et les restreint à une portée locale, ce qui fait que seul l'expression t_2 y a accès. La règle `Ty-Wh` représente cette contrainte par l'ajout des deux ressources dans $\Re$ pour le typage de t_2 et leurs suppressions de l'ensemble porté par le type de t_2.

Exemple 4.6. *Soient* 0, 1 *et* 2 *trois ressources et* $\Re = [0 \mapsto \langle \mathbb{N}, \mathbb{N} \rangle][1 \mapsto \langle Unit, \mathbb{N} \rangle][2 \mapsto \langle \mathbb{N}, Unit \rangle]$, *et* t *une fonction de signal de type* $(Unit \overset{\{0\}}{\rightsquigarrow} \mathbb{N})$ *telle que* $t = (wormhole[1, 2](12; rsf[1] \ggg rsf[0]))$. *La dérivation suivante le démontre :*

$$\frac{\textit{voir exemple 4.5} \quad \{1;2\} \cap \{\} = \varnothing \quad 1 \neq 0 \quad \{0\} = \{0;1\} \backslash \{1;2\}}{\varnothing \cdot [0 \mapsto \langle \mathbb{N}, \mathbb{N} \rangle] \vdash wormhole[1, 2](12; rsf[1] \ggg rsf[0]) : Unit \overset{\{0\}}{\rightsquigarrow} \mathbb{N}} \texttt{Ty-Wh}$$

sachant que pour tout Γ, $\Re$, *nous avons* $\Gamma \cdot \Re \vdash 12 : \mathbb{N}$.

La règle `Ty-Wh` est la seule ayant eu des ajouts de notre part. Tout d'abord nous précisons que les deux identifiants de ressources liées ne sont pas égaux afin d'éviter le masquage de l'un par l'autre dans le contexte de ressources $\Re$. De plus, nous ajoutons une contrainte sur les ressources présentes dans les types τ_1 et τ_2. Plus précisément, nous interdisons à ces types de porter les ressources liées dans un de leur ensemble de ressources s'ils en contiennent. La fonction *refs* fait l'union de tous les ensembles de ressources présents dans un type. Cet ajout est nécessaire pour éviter l'échappement des ressources en dehors de leur portée. Il est en effet possible de récupérer dans le type de sortie d'une fonction de signal une ressource liée, voir l'exemple suivant. Cependant, cette ressource n'est plus connue dès lors que nous sortons de sa portée, car le contexte de ressource qui les connait est seulement celui utilisé pour typer le sous-terme. Si, en plus de cette faille, nous autorisons l'opérateur *app* qui permet d'effectuer un calcul avec une paire composée d'une expression et d'une fonction de signal alors nous pouvons définir un programme bien typé qui tente d'accéder à une ressource indisponible.

Exemple 4.7. *Nous notons* `Ty-Wh-Esc` *la règle original du système de type pour le terme wormhole qui est définie comme suit :*

$$\frac{\begin{array}{c} R = R' \backslash \{r_{get}; r_{set}\} \\ \Gamma \cdot \Re \vdash t_1 : \tau \quad \Gamma \cdot \Re[r_{get} \mapsto \langle Unit, \tau \rangle][r_{set} \mapsto \langle \tau, Unit \rangle] \vdash t_2 : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma \cdot \Re \vdash wormhole[r_{get}, r_{set}](t_1; t_2) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2} \texttt{Ty-Wh-Esc}$$

Avec cette règle, il est possible de typer l'expression ci-dessous par $\mathbb{N} \xrightarrow{\varnothing} (Unit \xrightarrow{\{r_{r_1}\}} \mathbb{N})$*. Le problème avec ce type est la présence de* r_{r_1} *dans le type de retour de la fonction de signal. En effet,* r_{r_1} *n'est pas connue du contexte et ne devrait pas figurer dans le type.*

$$wormhole[r_{r_1}, r_{w_1}](1; rsf[r_{w_1}] \ggg wormhole[r_{r_2}, r_{w_2}](rsf[r_{r_1}]; rsf[r_{r_2}]))$$

La dérivation du typage est donnée dans les figures 4.6 et 4.7.

Propriétés de la sémantique statique

Le système de type doit satisfaire certaines propriétés nécessaires au bon fonctionnement des théorèmes et lemmes de la sémantique dynamique. En effet, nous supposerons quasi-systématiquement que les expressions dont nous parlons sont bien typées. La plus importante est l'affaiblissement du contexte, et dans ce cas des contextes. Le théorème 4.1 l'énonce et les théorèmes 4.2 et 4.3 l'applique pour chaque contexte indépendamment. Cette contrainte est commune à tous systèmes de type avec lieurs.

Théorème 4.1 (Affaiblissement des contextes)**.** *Soient* Γ*,* Γ' *deux contextes de variables,* $\Re$*,* $\Re'$ *deux contextes de ressources, t une expression et* τ *un type. Si* $\Gamma \subseteq \Gamma'$ *et* $\Re \subseteq \Re'$ *alors* $\Gamma \cdot \Re \vdash t : \tau$ *implique* $\Gamma' \cdot \Re' \vdash t : \tau$*.*

Démonstration. La démonstration se fait par induction structurelle sur la propriété de bon typage $\Gamma \cdot \Re \vdash t : \tau$. □

Corollaire 4.2 (Affaiblissement du contexte de variables)**.** *Soient* Γ*,* Γ' *deux contextes de variables,* $\Re$ *un contexte de ressources, t une expression et* τ *un type. Si* $\Gamma \subseteq \Gamma'$ *alors* $\Gamma \cdot \Re \vdash t : \tau$ *implique* $\Gamma' \cdot \Re \vdash t : \tau$*.*

Démonstration. La démonstration est directe par l'application du théorème 4.1 et la réflexivité de l'inclusion. □

Corollaire 4.3 (Affaiblissement du contexte de ressources)**.** *Soient* Γ *un contexte de variables,* $\Re$*,* $\Re'$ *deux contextes de ressources, t une expression et* τ *un type. Si* $\Re \subseteq \Re'$ *alors* $\Gamma \cdot \Re \vdash t : \tau$ *implique* $\Gamma \cdot \Re' \vdash t : \tau$*.*

Démonstration. La démonstration est directe par l'application du théorème 4.1 et la réflexivité de l'inclusion. □

Une autre propriété, cette fois-ci inhérente au langage, est le lien entre les ressources présentes dans l'ensemble porté par le type des fonctions de signal et le contexte des ressources. En effet, il faut que cet ensemble soit inclus dans le domaine du contexte des ressources, comme énoncé dans le théorème 4.4.

$$\dfrac{\dfrac{\Re\, r_{r_1} = \langle \mathit{Unit}, \mathbb{N}\rangle}{\varnothing \cdot \Re \vdash \mathit{rsf}[r_{r_1}] : \mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N}}\,\texttt{Ty-Rsf} \qquad \{r_{r_2}\}\backslash\{r_{r_2}; r_{w_2}\} = \varnothing \qquad \dfrac{(\Re[r_{r_2} \mapsto \langle \mathit{Unit}, \mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N}\rangle][r_{w_2} \mapsto \langle \mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N}, \mathit{Unit}\rangle])\, r_{r_2} = \langle \mathit{Unit}, \mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N}\rangle}{\varnothing \cdot \Re[r_{r_2} \mapsto \langle \mathit{Unit}, \mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N}\rangle][r_{w_2} \mapsto \langle \mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N}, \mathit{Unit}\rangle] \vdash \mathit{rsf}[r_{r_2}] : \mathit{Unit} \overset{\{r_{r_2}\}}{\rightsquigarrow} (\mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N})}\,\texttt{Ty-Rsf}}{\varnothing \cdot \Re \vdash \mathit{wormhole}[r_{r_2}, r_{w_2}](\mathit{rsf}[r_{r_1}]; \mathit{rsf}[r_{r_2}]) : \mathit{Unit} \overset{\varnothing}{\rightsquigarrow} (\mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N})}\,\texttt{Ty-Wh-Esc}$$

FIGURE 4.6 – Dérivation de typage démontrant un échappement d'une portée locale (partie 2). On note $\Re$ un contexte de ressources égal à $\{r_{r_1} : \langle \mathit{Unit}, \mathbb{N}\rangle; r_{w_1} : \langle \mathbb{N}, \mathit{Unit}\rangle\}$.

$$\dfrac{\dfrac{}{\varnothing \cdot \varnothing \vdash 1 : \mathbb{N}}\,\texttt{Ty-Int} \qquad \{r_{w_1}\}\backslash\{r_{r_1}; r_{w_1}\} = \varnothing \qquad \dfrac{\{r_{w_1}\} \cap \varnothing = \varnothing \qquad \overset{\text{démonstration dans la figure 4.6}}{\vdots}}{\varnothing \cdot \Re \vdash \mathit{wormhole}[r_{r_2}, r_{w_2}](\mathit{rsf}[r_{r_1}]; \mathit{rsf}[r_{r_2}]) : \mathit{Unit} \overset{\{r_{w_1}\}}{\rightsquigarrow} (\mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N})}\,\texttt{Ty-Wh-Esc}}{\varnothing \cdot \varnothing \vdash \mathit{wormhole}[r_{r_1}, r_{w_1}](1; \mathit{wormhole}[r_{r_2}, r_{w_2}](\mathit{rsf}[r_{r_1}]; \mathit{rsf}[r_{r_2}])) : \mathit{Unit} \overset{\varnothing}{\rightsquigarrow} (\mathit{Unit} \overset{\{r_{r_1}\}}{\rightsquigarrow} \mathbb{N})}\,\texttt{Ty-Wh-Esc}$$

FIGURE 4.7 – Dérivation de typage démontrant un échappement d'une portée locale (partie 1).On note $\Re$ un contexte de ressources égal à $\{r_{r_1} : \langle \mathit{Unit}, \mathbb{N}\rangle; r_{w_1} : \langle \mathbb{N}, \mathit{Unit}\rangle\}$.

Lemme 4.4 (Inclusion)**.** *Soient* Γ *un contexte de variables,* $\Re$ *un contexte de ressources,* v *une valeur (définie dans la section suivante),* τ_1*,* τ_2 *deux types et* R *un ensemble de ressources. Si* $\Gamma \cdot \Re \vdash v : \tau_1 \overset{R}{\rightsquigarrow} \tau_2$ *alors* $R \subseteq \mathtt{dom}(\Re)$*.*

Démonstration. La démonstration se fait par induction structurelle sur v ainsi que par filtrage des motifs de v via $\Gamma \cdot \Re \vdash v : \tau_1 \overset{R}{\rightsquigarrow} \tau_2$ qui ne laisse passer que les fonctions de signal. □

4.1.3 Transition d'évaluation

La transition d'évaluation permet de réduire une expression en une valeur. Dans l'article original, il est indiqué que la stratégie d'évaluation est paresseuse, mais aucun ensemble de règles n'est fourni. De notre côté, nous avons opté pour une sémantique opérationnelle à petit pas avec une stratégie d'appel par valeur faible, c'est-à-dire que l'évaluation ne passe pas sous les abstractions. Nous motivons ce choix après les définitions.

Définition de la transition d'évaluation

Nous notons $t \mapsto t'$ la relation entre t et t' qui satisfait au moins une règle de l'ensemble défini dans la figure 4.8. Ces règles se base sur un ensemble de valeurs et une fonction de substitution. Pour cette dernière, rien ne change d'une fonction de substitution classique. L'ensemble de valeurs est un sous-ensemble des termes et est défini comme suit :

$$\begin{aligned} v = {} & \mathtt{tt} \mid \lambda X.t \mid (v, v) \mid \mathit{rsf}[\mathit{Res}] \mid \mathit{arr}\, t \\ & \mid \boxed{\mathit{first}\, v \mid v \ggg v \mid \mathit{wormhole}[\mathit{Res}, \mathit{Res}](v; v)} \end{aligned}$$

La proposition originale est moins restreinte, car toute fonction de signal est une valeur, qu'importe ses sous-termes. Notre choix de contraindre aussi les sous-termes provient des modifications apportées à la transition fonctionnelle (voir section 4.1.4).

Avant de passer aux propriétés de cette transition, nous motivons le choix d'une stratégie appel-par-valeur faible. Tout d'abord, notre but premier était de voir s'il est possible d'intégrer un tel langage dans un autre langage hôte que Haskell et plus spécifiquement un langage fonctionnel à traits impératifs comme OCaml. Par conséquent, un changement de stratégie est profitable. Ensuite, nous avons remarqué, et nous en reparlerons dans la suite, que lors de la transition fonctionnelle le programme allait entièrement se réduire, excepté sous les opérateurs *arr*, ce qui rend moins pertinent une stratégie paresseuse. Pour

$$C = [] \mid C\,t \mid v\,C \mid (C, t) \mid (v, C) \mid \mathit{fst}\,C \mid \mathit{snd}\,C \mid \mathit{fix}\,C \mid \mathit{first}\,C \mid C \ggg t \mid v \ggg C \mid \mathit{wormhole}[r_{get}, r_{set}](C, t) \mid \mathit{wormhole}[r_{get}, r_{set}](v, C)$$

$$\frac{}{\mathit{fix}\,(\lambda f.t) \mapsto [f := \mathit{fix}\,(\lambda f.t)]\,t}\ \texttt{ET-Fix}_v \qquad \frac{}{(\lambda x.t)\ v \mapsto [x := v]\,t}\ \texttt{ET-App}_v$$

$$\frac{}{\mathit{fst}\,(v_1, v_2) \mapsto v_1}\ \texttt{ET-Fst}_v \qquad \frac{}{\mathit{snd}\,(v_1, v_2) \mapsto v_2}\ \texttt{ET-Snd}_v \qquad \frac{t \mapsto t'}{C[t] \mapsto C[t']}\ \texttt{ET-Ctx}$$

FIGURE 4.8 – Règles de la transition d'évaluation de WORMHOLES.

finir, lors de la mécanisation en ROCQ, il est plus profitable d'avoir une stratégie d'appel-par-valeur faible, car cela rend la gestion des variables beaucoup plus simple.

Propriétés de la transition d'évaluation

Maintenant que tout est défini, nous énonçons les lemmes et théorèmes principaux avec une esquisse de démonstration pour chacun. Tout d'abord, il est intéressant de pointer le déterminisme de la transition d'évaluation (théorème 4.6), ainsi que le fait qu'une valeur est une forme normale (théorème 4.5).

Lemme 4.5 (Forme Normale)**.** *Soient t et t' deux expressions, si $t \mapsto t'$ alors t ne peut pas être une valeur.*

Démonstration. La démonstration se fait par induction structurelle sur $t \mapsto t'$. En supposant que t est une valeur, nous arrivons à une contradiction. □

Lemme 4.6 (Déterminisme)**.** *Soient trois expressions t, t' et t'', si $t \mapsto t'$ et $t \mapsto t''$ alors $t' = t''$.*

Démonstration. La démonstration se fait par induction structurelle sur $t \mapsto t'$. □

Ensuite, le théorème 4.8 énonce la préservation du typage à travers la transition d'évaluation. Il se base sur la préservation du typage à travers la substitution, énoncée dans le théorème 4.7, et permet de démontrer directement la préservation du typage pour sa clôture réflexive et transitive, énoncée dans le théorème 4.9.

Lemme 4.7 (Préservation à travers la substitution)**.** *Soient Γ un contexte de variables, $\Re$ un contexte de ressources, x une variable, v une valeur, t une expression et τ_1, τ_2 deux types. Si $\Gamma[x \mapsto \tau_2] \cdot \Re \vdash t : \tau_1$ et $\varnothing \cdot \Re \vdash v : \tau_2$ alors $\Gamma \cdot \Re \vdash [x := v]\,t : \tau_1$.*

Démonstration. La démonstration se fait par induction structurelle sur l'expression t. Elle nécessite l'affaiblissement des deux contextes, voir le théorème 4.1, ainsi que les propriétés de masquage et de permutation, voir les équations (2.2a) et (2.2b). □

Théorème 4.8 (Préservation à travers ($\mapsto$))**.** *Soient $\Re$ un contexte de ressources, t, t' deux expressions et τ un type, si $\varnothing \cdot \Re \vdash t : \tau$ et $t \mapsto t'$ alors $\varnothing \cdot \Re \vdash t' : \tau$.*

Démonstration. La démonstration se fait par induction structurelle sur $t \mapsto t'$. Elle nécessite le théorème 4.7 pour le cas de l'abstraction et du point fixe. □

Corollaire 4.9 (Préservation à travers ($\mapsto^\star$))**.** *Soient $\Re$ un contexte de ressources, t, t' deux expressions et τ un type, si $\varnothing \cdot \Re \vdash t : \tau$ et $t \mapsto^\star t'$ alors $\varnothing \cdot \Re \vdash t' : \tau$.*

Démonstration. La démonstration se fait par induction structurelle sur $t \mapsto^\star t'$ et applique directement le théorème 4.8. □

Finalement, la progression de l'évaluation est vraie et est énoncée dans le théorème 4.10. Nous en déduisons le théorème 4.11. Il établit qu'une expression bien typée peut toujours satisfaire la relation ($\mapsto^\star$).

Théorème 4.10 (Progression de ($\mapsto$))**.** *Soient $\Re$ un contexte de ressources, t une expression et τ un type, si $\varnothing \cdot \Re \vdash t : \tau$ alors soit t est une valeur, soit il existe une étape d'évaluation qui amène vers une nouvelle expression t', c'est-à-dire $t \mapsto t'$.*

Démonstration. La démonstration se fait par induction structurelle sur l'expression t. Soit l'expression t fait partie du sous-ensemble des valeurs, soit nous construisons le terme t' via les hypothèses d'induction. □

Corollaire 4.11. *Soient $\Re$ un contexte de ressources, t une expression et τ un type, si $\varnothing \cdot \Re \vdash t : \tau$ alors il existe zéro ou une multitude d'étapes d'évaluation qui amène vers une nouvelle expression t', c'est-à-dire $t \mapsto^\star t'$.*

Démonstration. Directe dérivation du résultat du théorème 4.10. □

4.1.4 Transition fonctionnelle

La transition fonctionnelle effectue le calcul d'un instant logique, c'est-à-dire qu'il prend une valeur d'entrée et produit une valeur de sortie. Au contraire d'un programme Yampa qui produit une valeur de sortie et une nouvelle fonction de signal représentant le programme mis à jour, un programme Wormholes va prendre, en plus de la valeur d'entrée, un environnement et produire, en plus de

sa valeur de sortie, une version réduite du programme d'entrée, un nouvel environnement et un ensemble dont nous discuterons plus tard. Par conséquent, si le programme d'entrée est déjà une valeur alors la transition fonctionnelle produira une valeur de sortie, exactement le même programme et un environnement modifié. Le changement d'état d'un programme YAMPA est donc une mise à jour de l'environnement dans un programme WORMHOLES.

Définition de la transition fonctionnelle

Nous notons $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$ la relation représentant la transition fonctionnelle. Elle est définie comme une sémantique opérationnelle à grand-pas dans la figure 4.9 et met en relation $\langle V, t_v, t\rangle$ avec $\langle V', t'_v, t', W\rangle$ qui sont, respectivement, le triplet d'entrée et le quadruplet de sortie de la transition. t_v et t'_v sont respectivement l'expression portée par le signal d'entrée et celle du signal de sortie. t et t' sont respectivement le programme et sa version réduite. V et V' sont deux environnements de ressources respectivement l'environnement initial et la version mise à jour. Enfin W est un ensemble de triplets chacun contenant deux ressources et une expression. Nous avons modifié, légèrement, des règles de l'article original et ajouté deux nouvelles règles : `FT-ET-Tv` et `FT-ET-Sf`. Avant de passer à l'explication des modifications, nous définissons avec plus de précision ce qu'est un environnement de ressources, puis nous expliquons brièvement les règles régissant cette relation et nous finissons par motiver nos corrections.

Un environnement de ressources, dénoté par la lettre V et ses variations, associe des identifiants de ressources à ce que nous avons précédemment nommé des cellules dans la section 3.3.3. Pour rappel, une cellule c est soit « non-utilisée », nous pouvons aussi la définir comme « initiale », soit « utilisée », sachant que dans les deux cas c contient une expression t. Nous notons alors $\boxed{\mathtt{t}}$ une cellule « non-utilisée » et $\boxed{\mathtt{t}}$ une cellule « utilisée ». Un environnement de ressources est, tout comme un contexte, modélisé par un dictionnaire.

Exemple 4.8. *Dans la figure 3.6, la mémoire c est d'abord égal à* $\boxed{\mathtt{tt}}$ *puis devient égal à* $\boxed{2}$.

La fonction de signal $(arr\, t)$, pour une certaine fonction t, se comporte de la même manière que son équivalent dans YAMPA en appliquant sa fonction à l'expression du signal d'entrée. Il n'interagit nullement avec l'environnement de ressources V et ne génère aucun ensemble W comme l'indique la règle `FT-Arr`.

Exemple 4.9. *Soient* V *un environnement de ressources et* $t = arr\,(\lambda x.(x, \mathtt{tt}))$ *une fonction de signal. Nous avons la relation suivante :*

$$\frac{}{\langle V, 87, arr\,(\lambda x.(x, \mathtt{tt}))\rangle \Rrightarrow \langle V, (\lambda x.(x, \mathtt{tt})), t, \varnothing\rangle}\ \texttt{FT-Arr}$$

$$\frac{}{\langle V, t_v, arr\ t\rangle \Rrightarrow \langle V, (t\ t_v), arr\ t, \varnothing\rangle}\ \texttt{FT-Arr}$$

$$\frac{\langle V, t_{v_1}, t\rangle \Rrightarrow \langle V', t'_{v_1}, t', W\rangle}{\langle V, (t_{v_1}, t_{v_2}), first\ t\rangle \Rrightarrow \langle V', (t'_{v_1}, t_{v_2}), first\ t', W\rangle}\ \texttt{FT-First}$$

$$\frac{\langle V, t_v, t_1\rangle \Rrightarrow \langle V', t'_v, t'_1, W_1\rangle \quad \langle V', t'_v, t_2\rangle \Rrightarrow \langle V'', t''_v, t'_2, W_2\rangle}{\langle V, t_v, t_1 \ggg t_2\rangle \Rrightarrow \langle V'', t''_v, t'_1 \ggg t'_2, W_1 \cup W_2\rangle}\ \texttt{FT-Comp}$$

$$\frac{V\ r = \boxed{t'_v}}{\langle V, t_v, rsf[r]\rangle \Rrightarrow \langle V[r \mapsto \boxed{t_v}], t'_v, rsf[r], \varnothing\rangle}\ \texttt{FT-Rsf}$$

$$\frac{\langle V[r_{get} \mapsto \boxed{t_1}][r_{set} \mapsto \boxed{\texttt{tt}}], t_v, t_2\rangle \Rrightarrow \langle V', t'_v, t'_2, W\rangle}{\langle V, t_v, wormhole[r_{get}, r_{set}](t_1, t_2)\rangle \Rrightarrow \langle V', t'_v, t'_2, W \cup (r_{get}, r_{set}, t_1)\rangle}\ \texttt{FT-Wh}$$

$$\frac{t \mapsto t' \quad \langle V, t_v, t'\rangle \Rrightarrow \langle V', t'_v, t'', W\rangle}{\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t'', W\rangle}\ \texttt{FT-ET-Sf}$$

$$\frac{t_v \mapsto t'_v \quad \langle V, t'_v, t\rangle \Rrightarrow \langle V', t''_v, t'', W\rangle}{\langle V, t_v, t\rangle \Rrightarrow \langle V', t''_v, t', W\rangle}\ \texttt{FT-ET-Tv}$$

FIGURE 4.9 – Règles de la transition fonctionnelle de WORMHOLES.

Remarque 4.1. *L'expression de retour n'est pas évaluée. Cependant, si la fonction de signal est composée avec une autre alors il sera possible de réduire cette expression via la règle* `FT-ET-Tv`*, car l'expression de retour de la première fonction de signal devient l'expression d'entrée de la seconde fonction de signal lors d'une composition.*

La fonction de signal ($first\ t$), pour une certaine expression t, utilise son expression interne sur le premier composant de l'expression du signal d'entrée. Son interaction avec V ainsi que la génération d'un ensemble W dépend directement de t (voir règle `FT-First`).

Exemple 4.10. *Soient V un environnement de ressources et t une fonction de signal telle que $t = first\ (arr\ (\lambda x.x - 4))$. Nous avons la relation suivante :*

$$\frac{\dfrac{}{\langle V, 87, arr\ (\lambda x.x - 4)\rangle \Rrightarrow \langle V, (\lambda x.x - 4)\ 87, arr\ (\lambda x.x - 4), \varnothing\rangle}\ \texttt{FT-Arr}}{\langle V, (87, \texttt{tt}), first\ (arr\ (\lambda x.x - 4))\rangle \Rrightarrow \langle V, ((\lambda x.x - 4)\ 87, \texttt{tt}), t, \varnothing\rangle}\ \texttt{FT-First}$$

Remarque 4.2. *L'expression du signal d'entrée doit être une paire sinon la règle ne pourra s'appliquer. La règle* `FT-ET-Tv` *est là pour éviter un blocage.*

La règle `FT-Comp` définit le comportement de la composition. Tout comme une composition mathématique classique, la seconde fonction de signal prend en entrée le résultat produit par la première. L'ensemble W_1 est concaténé à W_2 dans le but de devenir l'ensemble produit par la composition.

Exemple 4.11. *Soient V un environnement et $t_1 = first\,(arr\,(\lambda x.x - 4))$, $t_2 = arr\,fst$ deux fonctions de signal. Nous avons la relation suivante :*

$$\dfrac{\textit{voir exemple 4.10} \quad \dfrac{}{\langle V, ((\lambda x.x - 4)\ 87, \mathtt{tt}), arr\,fst\rangle \Rrightarrow \langle V, fst\,((\lambda x.x - 4)\ 87, \mathtt{tt}), t_2, \varnothing\rangle}\ \texttt{FT-Arr}}{\langle V, (87, \mathtt{tt}), first\,(arr\,(\lambda x.x - 4)) \ggg arr\,fst\rangle \Rrightarrow \langle V, fst\,((\lambda x.x - 4)\ 87, \mathtt{tt}), t_1 \ggg t_2, \varnothing\rangle}\ \texttt{FT-Comp}$$

La fonction de signal $rsf[r]$ interagit avec V afin d'échanger l'expression du signal d'entrée par l'expression stockée dans la cellule associée à r dans V. De plus, comme précisé dans la règle `FT-Rsf`, la cellule $V\,r$ doit exister et être « non-utilisée » avant évaluation et devient « utilisée » après évaluation (car nous ajoutons à V la cellule mise à jour qui masquera l'ancienne version). Cette fonction de signal ne produit pas d'ensemble W.

Exemple 4.12. *Soient 0, 1 deux ressources respectivement de type $\langle Unit, \mathbb{N} \times Unit\rangle$ et $\langle \mathbb{N} \times Unit, Unit\rangle$, $V = [0 \mapsto \boxed{(87, \mathtt{tt})}][1 \mapsto \boxed{\mathtt{tt}}]$ un environnement de ressources et $t = (arr\,(\lambda x.x - 4)) \ggg arr\,fst$ une fonction de signal. Nous avons la relation suivante :*

$$\dfrac{\dfrac{V\,0 = \boxed{(87, \mathtt{tt})}}{\langle V, \mathtt{tt}, rsf[0] \ggg t\rangle \Rrightarrow \langle V[0 \mapsto \boxed{\mathtt{tt}}], (87, \mathtt{tt}), rsf[0], \varnothing\rangle}\ \texttt{FT-Rsf} \qquad \textit{voir exemple 4.11}}{\langle V, \mathtt{tt}, rsf[0] \ggg t\rangle \Rrightarrow \langle V[0 \mapsto \boxed{\mathtt{tt}}], fst\,((\lambda x.x - 4)\ 87, \mathtt{tt}), rsf[0] \ggg t, \varnothing\rangle}\ \texttt{FT-Comp}$$

La dernière règle définissant le comportement d'une fonction de signal est `FT-Wh`. Une fonction de signal $t = wormhole[r_{get}, r_{set}](t_1; t_2)$ ajoute à l'environnement d'entrée les paires $(r_{get}, \boxed{t_1})$ et $(r_{set}, \boxed{\mathtt{tt}})$ avant de continuer l'évaluation de l'instant en considérant t_2 comme la suite du programme. Les deux paires sont les ressources associées à la portée de t et forment la référence locale. Le résultat produit est un quadruplet formé d'un environnement mise à jour contenant toujours les ressources r_{get} et r_{set}, une expression de sortie, uniquement l'expression réduite de t_2 et un ensemble W. En effet, lors de l'évaluation d'un instant, les lieurs réactifs sont retirées du programme laissant uniquement leurs expressions internes. La portée de la référence locale n'est plus assurée mais ses ressources qui le forme ainsi que l'expression initiale sont stockées dans un triplet qui est ajouté à l'ensemble W généré par l'évaluation de t_2.

Un ensemble W porte donc l'information de toutes les références locales précédemment contenues dans le programme d'entrée ainsi que leurs expressions initiales. Il est utilisé par la transition temporelle pour différencier les resources libres et liées dans l'environnement de sortie et gérer leur cycle de vie respectifs (voir la section 4.1.5).

Exemple 4.13. *Soit* $t = rsf[0] \ggg first\,(arr\,(\lambda x.x - 4)) \ggg arr\,fst$ *une fonction de signal. Nous avons la relation ci-dessous. Nous avons remplacé wormhole par wh par manque de place.*

$$\dfrac{\dfrac{\textit{voir exemple 4.12}}{\langle\{0 : \boxed{(87, \texttt{tt})}; 1 : \boxed{\texttt{tt}}\}, \texttt{tt}, t\rangle \Rrightarrow \langle\{0 : \boxed{\texttt{tt}}; 1 : \boxed{\texttt{tt}}\}, fst\,((\lambda x.x - 4)\ 87, \texttt{tt}), t, \varnothing\rangle}\ \texttt{FT-Comp}}{\langle\varnothing, \texttt{tt}, wh[0,1]((87, \texttt{tt}), t)\rangle \Rrightarrow \langle\{0 : \boxed{\texttt{tt}}; 1 : \boxed{\texttt{tt}}\}, fst\,((\lambda x.x - 4)\ 87, \texttt{tt}), t, \{(0, 1, (87, \texttt{tt}))\}\rangle}\ \texttt{FT-Wh}$$

Les deux règles `FT-ET-Tv` et `FT-ET-Sf` élève la transition d'évaluation dans la transition fonctionnelle. En effet, elles permettent respectivement d'appliquer une étape de réduction sur l'expression du signal d'entrée ou le programme d'entrée avant de continuer l'évaluation de l'instant.

Exemple 4.14. *Soient* V *un environnement et* $t = (\lambda f.first\ f)\ (arr\,(\lambda x.x - 4))$ *et* $t' = first\,(arr\,(\lambda x.x - 4))$ *deux fonctions de signal. Nous avons la relation suivante :*

$$\dfrac{t \mapsto t' \qquad \dfrac{fst\,((87, \texttt{tt}), \texttt{tt}) \mapsto (87, \texttt{tt}) \qquad \dfrac{\textit{voir exemple 4.10}}{\langle V, (87, \texttt{tt}), t'\rangle \Rrightarrow \langle V, (\lambda x.x - 4)\ 87, t', \varnothing\rangle}\ \texttt{FT-First}}{\langle V, fst\,((87, \texttt{tt}), \texttt{tt}), t'\rangle \Rrightarrow \langle V, (\lambda x.x - 4)\ 87, t', \varnothing\rangle}\ \texttt{FT-ET-Tv}}{\langle V, fst\,((87, \texttt{tt}), \texttt{tt}), t\rangle \Rrightarrow \langle V, ((\lambda x.x - 4)\ 87, \texttt{tt}), first\,(arr\,(\lambda x.x - 4)), \varnothing\rangle}\ \texttt{FT-ET-Sf}$$

La nouvelle transition fonctionnelle que nous avons proposé corrige un problème de blocage de la proposition originale. En effet, dans sa version originale, les règles `FT-ET-Tv` et `FT-ET-Sf` n'existent pas et toutes les autres règles comportent une réduction multiple sur le ou les sous-termes. Par exemple, la règle `FT-First` est originalement définie comme suit :

$$\dfrac{t \mapsto^\star t' \quad \langle V, t_{v_1}, t'\rangle \Rrightarrow \langle V', t'_{v_1}, t'', W\rangle}{\langle V, (t_{v_1}, t_{v_2}), first\ t\rangle \Rrightarrow \langle V', (t'_{v_1}, t_{v_2}), first\ t'', W\rangle}\ \texttt{FT-First-OG}$$

Il y a deux problèmes avec cette version. Premièrement, il est impossible de réduire l'expression du signal d'entrée, cependant, si ce n'est pas une paire la règle n'est pas applicable. De plus, nous ne pouvons pas supposer que l'expression est en forme normale, car la règle `FT-Arr` applique une fonction sur l'expression d'entrée, mais ne l'a réduit pas. Deuxièmement, un programme bien typé ne suffit pas pour

garantir son exécution, car $(\lambda f.\mathit{first}\ f)\ (\mathit{arr}\ (\lambda x.x-4))$ est une fonction de signal bien typée qui, n'étant pas réduite, ne peut utiliser la règle `FT-First-OG`.

Ces deux problèmes ont motivé la création des règles `FT-ET-Tv` et `FT-ET-Sf`. Les réductions multiples deviennent donc obsolètes d'où leur suppression dans la version corrigée. L'exemple 4.14 montre l'utilisation des deux règles ajoutées. Ces corrections nous permettent aussi de simplifier la démonstration de la réactivité.

Propriétés de la transition fonctionnelle

La transition fonctionnelle est le cœur de l'évaluation d'un programme WORMHOLES, elle porte donc les propriétés les plus importantes qui sont la préservation du typage, la réactivité et la sûreté du point de vue utilisation des ressources. Ces propriétés nécessitent des résultats intermédiaires ainsi que certaines hypothèses.

Tout d'abord, la transition fonctionnelle ne peut qu'ajouter de nouvelles ressources à l'environnement d'entrée V. Ces ressources proviennent toutes de lieurs réactifs et sont donc stockées dans l'ensemble W. Le théorème 4.12 décrit ce comportement.

Lemme 4.12 (Préservation des clés et correspondance)**.** *Soient V, V' deux environnements de ressources, W un ensemble de triplet et t_v, t'_v, t et t' quatre expressions. Si nous avons la relation $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$ alors nous savons que $\mathtt{dom}(V) \subseteq \mathtt{dom}(V')$ et $rs(W) = \mathtt{dom}(V')\backslash\mathtt{dom}(V)$, où rs est la fonction qui retourne l'ensemble des ressources présent dans un ensemble de triplet W.*

Démonstration. La démonstration se fait par induction structurelle sur la relation définissant la transition fonctionnelle, c'est-à-dire $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$. □

Ensuite, les propriétés de préservation et de progression ont besoin de deux hypothèses : 1. un lien entre le contexte de ressources et l'environnement de ressources ; 2. des hypothèses de terminaison. Pour la première, nous avons défini un concept de *cohérence* dans la définition 4.1. Elle énonce qu'un environnement et un contexte doivent partager les mêmes ressources et que, point-à-point, les types correspondent.

Définition 4.1 (Modélisation ctxt/env)**.** *Un contexte de ressources $\Re$ et un environnement de ressources V sont* cohérents *si et seulement si :*

1. *leurs domaines correspondent, c'est-à-dire que $\mathtt{dom}(\Re) = \mathtt{dom}(V)$;*
2. *pour toutes ressources r de type $\langle \tau_1, \tau_2\rangle$ et cellule c, si $\Re\, r = \langle \tau_1, \tau_2\rangle$ et $V\, r = c$ alors :*
   - *soit $c = \boxed{t}$, pour t une expression, et $\varnothing \cdot \Re \vdash t : \tau_2$;*

— *soit* $c = \boxed{t}$, *pour* t *une expression, et* $\varnothing \cdot \Re \vdash t : \tau_1$.

Pour la seconde, nous devons d'abord déterminer quelles expressions doivent terminer. Cette contrainte vient de l'ajout du point fixe qui casse la normalisation du langage. Les expressions directement impliquées dans la relation, c'est-à-dire t_v, t'_v, t et t', doivent terminer sinon, il serait possible de boucler via les règles `FT-ET-Sf` et `FT-ET-Tv` en essayant de réduire l'expression. De plus, la règle `FT-Rsf` implique l'expression associée à r dans V. Par conséquent, il faut aussi que toutes les expressions contenues dans l'environnement, d'entrée et de sortie, terminent. Finalement, dans la règle `FT-Arr`, la valeur t_v est appliquée à la fonction t contenue dans $arr\, t$. Par conséquent, il faut s'assurer de la bonne terminaison de ce calcul. Nous définissons ce que veut dire « terminer » pour une expression dans la définition 4.2 ainsi que pour des tuples dans les définitions 4.3 et 4.4 qui seront utilisées dans le reste des lemmes et théorèmes énoncés.

Définition 4.2 (Terminaison)**.** *Soit* t *une expression, nous disons que* t *termine s'il existe une valeur* v *telle que* $t \mapsto^\star v$. *Nous notons cette propriété* $halts(t)$.

Définition 4.3 (Terminaison d'un triplet)**.** *Soient* V *un environnement de ressources et* t, t' *deux expressions. Si* t *et* t' *ainsi que chaque expression* t'' *de* V *terminent alors le triplet termine, c'est-à-dire :*

$$halts_{triple}(V, t, t') = halts(t) \wedge halts(t') \wedge (\forall r \in V, halts(extract_{env}(V\, r)))$$

où la fonction $extract_{env}$ *prend une cellule et retourne son expression.*

Définition 4.4 (Terminaison d'un quadruplet)**.** *Soient* V *un environnement de ressources,* t, t' *deux expressions et* W *un ensemble de triplet. Si* t *et* t' *terminent ainsi que chaque expression* t'' *de* V *ou de* W *terminent alors le quadruplet de sortie termine, c'est-à-dire :*

$$halts_{quad}(V, t, t', W) = halts_{triple}(V, t, t') \wedge (\forall r \in W, (extract_{set}(W\, r)))$$

où la fonction $extract_{set}$ *prend un triplet et retourne son expression.*

Nous nous autorisons à ne pas noter les indices pour la propriété *halts*, car l'arité nous permet de savoir à quelle définition nous faisons référence. Nous finissons par définir la définition 4.5, utilisée dans les théorèmes principaux, qui exprime que pour un terme t et un contexte de ressources $\Re$ donnée, si t termine alors chaque fonction de signal $arr\, t'$ contenue dans t termine lorsque nous lui appliquons un paramètre adéquat. La récolte des fonctions de signal $arr\, t'$ se fait

$$\forall\ \Re\ t\ t', t \longmapsto t' \implies halts_{arr}(\Re, t) \implies halts_{arr}(\Re, t') \tag{4.1a}$$
$$\forall\ \Re\ t, halts_{arr}(\Re, first(t)) \iff halts_{arr}(\Re, t) \tag{4.1b}$$
$$\forall\ \Re\ t_1\ t_2, halts_{arr}(\Re, t_1) \text{ et } halts_{arr}(\Re, t_2) \implies halts_{arr}(\Re, t_1 \ggg t_2) \tag{4.1c}$$
$$\forall\ \Re\ t_1\ t_2, halts(\Re, t_1) \text{ et } halts(\Re, t_2) \implies halts_{arr}(\Re, t_1 \ggg t_2) \implies halts_{arr}(\Re, t_1) \text{ et } halts_{arr}(\Re, t_2) \tag{4.1d}$$
$$\forall\ \Re\ t_1\ t_2\ r_g\ r_s\ \tau, halts(\Re, t_1) \implies halts_{arr}(\Re, wormhole[r_g, r_s](t_1; t_2)) \implies halts_{arr}(\Re[r_g \mapsto \langle Unit, \tau\rangle][r_s \mapsto \langle \tau, Unit\rangle], t) \tag{4.1e}$$
$$\forall\ \Re\ \Re'\ t, \Re \subseteq \Re' \implies halts(\Re, t) \implies halts_{arr}(\Re', t) \tag{4.1f}$$

FIGURE 4.10 – Propriétés associés à $halts_{arr}$.

grâce à la fonction get_{arr} qui prend une valeur v et retourne une liste de termes. Elle est définie comme suit :

$$\begin{aligned}
get_{arr}\ \texttt{tt} &= [] & get_{arr}\ first\ t &= get_{arr}\ t \\
get_{arr}\ \lambda x.t &= [] & get_{arr}\ (t_1, t_2) &= (get_{arr}\ t_1) +\!\!+ (get_{arr}\ t_2) \\
get_{arr}\ rsf[r] &= [] & get_{arr}\ (t_1 \ggg t_2) &= (get_{arr}\ t_1) +\!\!+ (get_{arr}\ t_2) \\
get_{arr}\ arr\ t &= [t] & get_{arr}\ wormhole[r, r'](t_1; t_2) &= (get_{arr}\ t_1) +\!\!+ (get_{arr}\ t_2)
\end{aligned}$$

Définition 4.5 (Terminaison des fonctions élevées)**.** *Soient $\Re$ un contexte de ressources et t un terme du langage.*

$$\begin{aligned}
&\forall\, v, t \mapsto^\star v \text{ et } v \text{ est une valeur} \implies \\
&\qquad \forall\, \Re'\ \tau_1\ \tau_2\ t'\ t_v, t' \in (get_{arr}\ v) \text{ et} \\
&\qquad\qquad \Re \subseteq \Re' \text{ et } halts(t_v) \text{ et} \\
&\qquad\qquad \varnothing \cdot \Re' \vdash t' : \tau_1 \to \tau_2 \text{ et } \varnothing \cdot \Re' \vdash t_v : \tau_1 \implies halts(t'\ t_v)
\end{aligned}$$

Nous notons $halts_{arr}$ cette propriété et nous établissons l'ensemble d'équations dans la figure 4.10 qui fourni un ensemble de résultats sur $halts_{arr}$. Intuitivement, chaque résultat nous permet de propager la propriété dans un ou plusieurs sous-termes, excepté pour la propriété (4.1a) qui établit que la propriété est préservée à travers l'application de la transition d'évaluation et la propriété (4.1f) qui établit que la propriété est préservée en affaiblissant le contexte de ressources. Nous regroupons les démonstrations des équations présentées dans la figure 4.10 dans le théorème 4.13.

Lemme 4.13. *Les propriétés (4.1a) à (4.1f) sont vraies pour la définition 4.5.*

Démonstration. Nous démontrons la propriété (4.1a). Nous posons v une valeur ainsi que $\Re'$, τ_1, τ_2, t'', t_v et, en plus de $t \mapsto t'$ et $halts_{arr}(\Re, t)$ nous avons les hypothèses ci-dessous et nous devons démontrer que $halts(t''\, t_v)$.

$$t' \mapsto^\star v \qquad t'' \in (get_{arr}\, v)$$
$$\Re \subseteq \Re' \qquad \varnothing \cdot \Re' \vdash t'' : \tau_1 \to \tau_2 \qquad \varnothing \cdot \Re' \vdash t_v : \tau_1$$

Nous appliquons $halts_{arr}(\Re, t)$ à notre but, ce qui nous laisse $t \mapsto^\star v$ à démontrer (les autres buts étant directement présent dans les hypothèses). Sachant que $t \mapsto t'$ et que $t' \mapsto^\star v$, nous utilisons l'équation (2.1b) ce qui clôture notre démonstration. □

Il est maintenant possible de définir le théorème de préservation du typage à travers la transition fonctionnelle. Celui-ci est énoncé dans le théorème 4.14.

Théorème 4.14 (Préservation à travers ($\Rrightarrow$))**.** *Soient $\Re$ un contexte de ressources, V, V' deux environnements de ressources, W un ensemble de triplet, t_v, t'_v, t et t' quatre expressions, τ_1, τ_2 deux types et R un ensemble de ressources, si :*

1. *le triplet d'entrée termine, c'est-à-dire que $halts(V, t_v, t)$;*
2. *$halts_{arr}(\Re, t)$;*
3. *$\Re$ et V sont cohérents ;*
4. *t est de type $\tau_1 \overset{R}{\rightsquigarrow} \tau_2$ et satisfait la relation $\varnothing \cdot \Re \vdash t : \tau_1 \overset{R}{\rightsquigarrow} \tau_2$;*
5. *t_v est de type τ_1 et satisfait la relation $\varnothing \cdot \Re \vdash t_v : \tau_1$;*
6. *nous avons la transition fonctionnelle suivante : $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$*

alors

1. *chaque cellule associée à une ressource présente dans R est « non-utilisée » dans V ;*
2. *seules les cellules associées à une ressource présente dans R sont mises à jour durant le calcul de l'instant, c'est-à-dire que $\forall r \in \mathtt{dom}(V) \backslash R, V\, r = V'\, r$;*
3. *le quadruplet de sortie termine, c'est-à-dire que $halts(V', t'_v, t')$;*
4. *il existe un contexte de ressources $\Re'$ et un ensemble de ressources R' tels que :*
   (a) *le contexte $\Re$ est inclus dans $\Re'$ ainsi que R pour R' ;*

(b) $halts_{arr}(\Re', t')$ *est vraie ;*

(c) $\Re'$ *et* V' *sont cohérents ;*

(d) t' *est de type* $\tau_1 \overset{R'}{\rightsquigarrow} \tau_2$ *et satisfait la relation* $\varnothing \cdot \Re' \vdash t' : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2$ *;*

(e) t'_v *est de type* τ_2 *et satisfait la relation* $\varnothing \cdot \Re' \vdash t'_v : \tau_2$ *;*

(f) chaque ressource r *dans* $R' \backslash R$ *est dans* W*, mais pas dans l'environnement d'entrée* V *;*

(g) chaque cellule, dans V'*, associée à une ressource présente dans* R' *est « utilisée ».*

Démonstration. La démonstration se fait par induction structurelle sur la transition fonctionnelle, c'est-à-dire, la relation $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$. Nous développons le cas de la règle `FT-Comp`. Avant de s'attaquer aux buts de ce sous-cas, nous détaillons les hypothèses et l'application des hypothèses d'induction. La transition fonctionnelle pour la composition $(t_1 \ggg t_2)$ est la séquence de transition fonctionnelle pour t_1 puis t_2, c'est-à-dire que nous avons les hypothèses suivantes :

$$\langle V, t_v, t_1\rangle \Rrightarrow \langle V', t'_v, t'_1, W_1\rangle \qquad\qquad \langle V', t'_v, t_2\rangle \Rrightarrow \langle V'', t''_v, t'_2, W_2\rangle$$

avec $W = W_1 \cup W_2$. Les hypothèses d'induction sont au nombre de deux, que nous nommons IH_1 et IH_2. Par hypothèse, nous savons que $(t_1 \ggg t_2)$ est de type $\tau_1 \overset{R}{\rightsquigarrow} \tau_2$ par rapport au contexte $\Re$ ce qui, par décomposition de la propriété de bon typage, nous informe que :

$$\varnothing \cdot \Re \vdash t_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau \qquad\qquad \varnothing \cdot \Re \vdash t_2 : \tau \overset{R_2}{\rightsquigarrow} \tau_2 \qquad\qquad R_1 \cap R_2 = \varnothing$$

avec $R = R_1 \cup R_2$ et τ un type. De plus, nous savons que $\Re$ et V sont cohérents, que t_v est de type τ_1 par rapport à $\Re$, que $halts(t_1)$ et $halts(t_2)$ par décomposition de l'hypothèse (1) et que, via la propriété (4.1c), $halts_{arr}(\Re, t_1)$ et $halts_{arr}(\Re, t_2)$ sont vraies. Grâce à tout cela, nous pouvons appliquer la première hypothèse d'induction IH_1 sur le bon typage de t_1, ce qui nous donne les nouvelles hypothèses suivantes :

1a. chaque cellule associée à une ressource présente dans R_1 est « non-utilisée » dans V ;

2a. $\forall r \in \mathtt{dom}(V) \backslash R_1, V\, r = V'\, r$;

3a. le quadruplet de sortie termine, c'est-à-dire que $halts(V', t'_v, t'_1)$;

4a. le contexte $\Re$ est inclus dans $\Re'$ ainsi que R_1 pour R'_1 ;

5a. $halts_{arr}(\Re', t'_1)$ est vraie ;

6a. $\Re'$ et V' sont cohérents ;

7a. $\varnothing \cdot \Re' \vdash t'_1 : \tau_1 \overset{R'_1}{\rightsquigarrow} \tau$;

8a. $\varnothing \cdot \Re' \vdash t'_v : \tau$;

9a. chaque ressource r dans $R'_1 \backslash R_1$ est dans W_1, mais pas dans l'environnement d'entrée V ;

10a. chaque cellule, dans V', associée à une ressource présente dans R'_1 est « utilisée ».

avec $\Re'$ un contexte de ressources et R'_1 un ensemble de ressources. Pour appliquer l'hypothèse IH_2 il faut que le terme t_2, soit typé par rapport à $\Re'$ et que $halts_{arr}(\Re', t_2)$ soit vraie. Grâce à l'affaiblissement du contexte de ressources, c'est-à-dire le théorème 4.1, la première hypothèse est vraie et grâce à la propriété (4.1f) la seconde l'est aussi. Par conséquent, nous avons les hypothèses suivantes :

1b. chaque cellule associée à une ressource présente dans R_2 est « non-utilisée » dans V' ;

2b. $\forall r \in \mathtt{dom}(V') \backslash R_2, V'\, r = V''\, r$;

3b. le quadruplet de sortie termine, c'est-à-dire que $halts(V'', t''_v, t'_2)$;

4b. le contexte $\Re'$ est inclus dans $\Re''$ ainsi que R_2 pour R'_2 ;

5b. $halts_{arr}(\Re'', t'_2)$ est vraie ;

6b. $\Re''$ et V'' sont cohérents ;

7b. $\varnothing \cdot \Re'' \vdash t'_2 : \tau \overset{R'_2}{\rightsquigarrow} \tau_2$;

8b. $\varnothing \cdot \Re'' \vdash t''_v : \tau_2$;

9b. chaque ressource r dans $R'_2 \backslash R_2$ est dans W_2, mais pas dans l'environnement d'entrée V' ;

10b. chaque cellule, dans V'', associée à une ressource présente dans R'_2 est « utilisée ».

Nous démontrons les sous-buts (1.) jusqu'à (8.) avec $\Re''$ et $(R'_1 \cup R'_2)$. Les deux derniers sous-buts sont similaires aux sous-buts (1.) et (2.) et ne sont donc pas énoncés.

1. Nous démontrons que chaque cellule associée à une ressource présente dans $R_1 \cup R_2$ est « non-utilisée » dans V. Soit r une ressource, si r est présente dans $R_1 \cup R_2$ alors, sachant que $R_1 \cap R_2 = \varnothing$ grâce au typage, nous avons deux cas : soit $r \in R_1$ et $r \notin R_2$, soit $r \notin R_1$ et $r \in R_2$. Dans le premier cas, l'hypothèse (1a.) suffit pour conclure. Dans le second cas, nous savons que r est « non-utilisée » dans V' via l'hypothèses (1b.). Pour que r soit « non-utilisée » dans V, il faut montrer que $r \in \mathtt{dom}(V)$ et que la première transition fonctionnelle n'utilise pas r.

— Pour démontrer que $r \in V$ nous utilisons, les théorèmes 4.4 et 4.8 et les définitions de *halts* et de la cohérence de $\Re$ et V. Via l'application et la préservation du typage, nous pouvons dire qu'il existe une valeur v telle que $t_2 \mapsto^\star v$ et que $\varnothing \cdot \Re \vdash v : \tau \overset{R_2}{\rightsquigarrow} \tau_2$. Ensuite via le théorème 4.4, nous avons $r \in \texttt{dom}(\Re)$. Et finalement, via la correspondance des domaines de $\Re$ et V, nous avons $r \in \texttt{dom}(V)$.

— En appliquant l'hypothèse (2a.) à r nous savons que $V\,r = V'\,r$. Par conséquent, si r n'est pas utilisée dans V' alors elle ne l'est pas dans V.

2. Nous démontrons que $\forall r \in \texttt{dom}(V) \backslash (R_1 \cup R_2), V\,r = V''\,r$. Nous posons r une ressource. Si $r \in \texttt{dom}(V) \backslash (R_1 \cup R_2)$ alors $r \in \texttt{dom}(V)$ et $r \notin R_1$ et $r \notin R_2$. D'après le théorème 4.12, si $r \in \texttt{dom}(V)$ alors $r \in \texttt{dom}(V)$. Grâce aux hypothèses (2a.) et (2b.) nous pouvons conclure ce sous-cas.
3. Nous démontrons que le quadruplet de sortie termine, c'est-à-dire que la propriété $\mathit{halts}(V'', t''_v, t'_1 \ggg t'_2)$ est vraie directement grâce aux hypothèses (3a.) et (3b.).
4. Le contexte $\Re$ est inclus dans $\Re''$ par transitivité via $\Re'$ et les hypothèses (4a.) et (4b.).
5. Grâce à la propriété équation (4.1c), la démonstration est directe.
6. $\Re''$ et V'' sont cohérents par hypothèse.
7. L'expression $(t'_1 \ggg t'_2)$ est de type $\tau_1 \overset{R'_1 \cup R'_2}{\rightsquigarrow} \tau_2$ par rapport à $\Re''$, si :
   (a) $\varnothing \cdot \Re'' \vdash t'_1 : \tau_1 \overset{R'_1}{\rightsquigarrow} \tau$;
   (b) $\varnothing \cdot \Re'' \vdash t'_2 : \tau \overset{R'_2}{\rightsquigarrow} \tau_2$;
   (c) $R'_1 \cap R'_2 = \varnothing$.

   Via le théorème 4.1, nous pouvons affaiblir l'hypothèse (7a.) ce qui nous permet de satisfaire le premier sous-cas et le deuxième sous-cas est vrai par hypothèse. Pour montrer que $R'_1 \cap R'_2 = \varnothing$, nous procédons par contradiction. Nous supposons qu'il existe une ressource r telle que $r \in R'_1$ et $r \in R'_2$ et nous démontrons une contradiction. Via le typage, nous savons que R_1 et R_2 sont disjoints et que, via les hypothèses d'induction, $R_1 \subseteq R'_1$ et $R_2 \subseteq R'_2$. Nous supposons quatre cas :
   (a) $r \in R_1$ et $r \in R_2$: le cas présente une contradiction directe avec nos hypothèses.
   (b) $r \in R_1$ et $r \notin R_2$: Grâce à l'hypothèse (9b.), nous savons que $r \notin V'$. Cependant, l'hypothèse (1a.) stipule que toutes les ressources de R_1 sont inutilisées dans V et donc présentes dans V. Par conséquent, $r \in V$ et, grâce au théorème 4.12, $r \in V'$ ce qui mène à une contradiction.

(c) $r \notin R_1$ et $r \in R_2$: Grâce à l'hypothèse (9a.), nous savons que $r \notin V$. Via l'hypothèse de cohérence entre $\Re$ et V, nous pouvons démontrer $r \notin V$ si $r \notin \Re$. En utilisant, les théorèmes 4.4 et 4.8 et l'hypothèse (3b.) nous pouvons déduire que $r \in \Re$ ce qui rentre en contradiction avec nos hypothèses.

(d) $r \notin R_1$ et $r \notin R_2$: D'après les hypothèses (9a.) et (9b.), r appartient à W_1 et W_2 et n'appartient ni à V ni à V'. Cependant l'hypothèse (10a.), appliquée à $r \in R'_1$, énonce que r est « utilisée » dans V' et donc appartient à V', ce qui nous amène à une contradiction.

8. $\varnothing \cdot \Re'' \vdash t''_v : \tau_2$ est vraie par hypothèse.

□

Après la préservation, il faut démontrer la réactivité du langage, c'est-à-dire que pour toute entrée correcte, il existe un résultat produit. Le théorème 4.16 énonce formellement ce que nous entendons par réactif et la démonstration se base sur le théorème 4.15 qui met en évidence une forme canonique, un sous-ensemble plus restreint que les valeurs et qui ne comporte que les fonctions de signal totalement réduites.

Lemme 4.15 (Forme canonique)**.** *Soient Γ un contexte de variables, v une valeur, $\Re$ un contexte de ressources, τ_1, τ_2 deux types et R un ensemble de ressources. Si v est de type $\tau_1 \overset{R}{\rightsquigarrow} \tau_2$, c'est-à-dire qu'elle satisfait la relation $\Gamma \cdot \Re \vdash v : \tau_1 \overset{R}{\rightsquigarrow} \tau_2$, alors v ne peut être qu'une fonction de signal complètement réduite via la transition d'évaluation, c'est-à-dire, soit $arr\, v'$, $first\, v'$, $v_1 \ggg v_2$, $rsf[r]$ ou $wormhole[r_1, r_2](v_1; v_2)$, où v', v_1 et v_2 sont des valeurs, et r, r_1 et r_2 sont des ressources.*

Démonstration. La démonstration est directe et se fait par induction structurelle sur la propriété de bon typage de v. □

Théorème 4.16 (Réactivité)**.** *Soient $\Re$ un contexte de ressources, V un environnement de ressources, t_v, t deux expressions, τ_1, τ_2 deux types et R un ensemble de ressources, si :*

1. *le triplet d'entrée termine, c'est-à-dire que $halts(V, t_v, t)$;*
2. *$halts_{arr}(\Re, t)$ est vraie ;*
3. *$\Re$ et V sont cohérents ;*
4. *t est de type $\tau_1 \overset{R}{\rightsquigarrow} \tau_2$ et satisfait la relation $\varnothing \cdot \Re \vdash t : \tau_1 \overset{R}{\rightsquigarrow} \tau_2$;*
5. *t_v est de type τ_1 et satisfait la relation $\varnothing \cdot \Re \vdash t_v : \tau_1$;*

6. *chaque cellule dans V associée à une ressource présente dans R est « non-utilisée » ;*

alors il existe un environnement de ressources V', deux expressions t'_v et t' ainsi qu'un ensemble de triplets W tel que la propriété $halts(V', t'_v, t', W)$ est satisfaite et nous avons la relation $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$.

Démonstration. Sachant que t termine, nous savons qu'il existe une valeur v tel que $t \mapsto^\star v$. Nous effectuons une démonstration par induction structurelle sur $t \mapsto^\star v$, ce qui nous donne deux cas : 1. $t = v$ et donc t est une valeur ; 2. il existe t' tel que $t \mapsto t'$. Via la règle `FT-ET-Sf` et par hypothèse d'induction le théorème est vrai pour t'. Dans le premier cas, nous faisons une induction structurelle sur t et, via le théorème 4.15, nous savons qu'il y a cinq sous-cas. Les sous-cas où $t = arr\, t$ et $t = rsf[r]$ sont triviaux, nous démontrons les autres ci-dessous.

— $t = first\, v$: nous réduisons l'expression d'entrée t_v jusqu'à avoir une paire, ce qui est possible, car t_v termine, mais aussi parce que son typage le restreint à un type produit. Le type de t_v se propage à sa réduction en paire via le théorème 4.9. Finalement, nous appliquons l'hypothèse d'induction ce qui clos ce cas.

— $t = v_1 \ggg v_2$: nous démontrons la réactivité de la composition en démontrant la réactivité de ses deux fonctions de signal. Pour v_1, nous appliquons directement l'hypothèse d'induction. Pour v_2, l'application de l'hypothèse nécessite une expression d'entrée t' du bon type. L'expression d'entrée est l'expression de sortie de la transition fonctionnelle appliquée à v_1, cependant aucune hypothèse ne garanti son type. Nous pouvons néanmoins le récupérer via le théorème 4.14. Le contexte sur lequel repose le bon typage de t' est un sur-ensemble du contexte utilisé pour le bon typage de v_2. Nous utilisons donc le théorème 4.1 pour établir le bon typage de v_2 avec le même contexte que t'. Finalement, nous appliquons l'hypothèse d'induction ce qui conclut ce sous-cas.

— $t = wormhole[r_1, r_2](v_1; v_2)$: avec le théorème 4.1, la démonstration se fait immédiatement par hypothèse d'induction.

Dans le deuxième cas, nous appliquons la règle `FT-ET-Sf` ainsi que la préservation du typage à travers la transition d'évaluation, énoncée dans le théorème 4.8, et l'hypothèse d'induction. □

Cette démonstration a motivé le changement de stratégie d'évaluation ainsi que la définition d'une valeur. En effet, outre les blocages induits par la version originale, nous ne pouvions pas raisonner en deux temps comme ce qui est fait ici à cause de la définition de valeur. Pour rappel, dans la proposition originale toute

fonction de signal est une valeur qu'importe ses sous-termes. Dans la démonstration proposée plus haut, nous avons réduit l'expression en une valeur, puis nous avons raisonné par induction structurelle sur cette valeur. Notre hypothèse d'induction est donc applicable uniquement sur des valeurs. Cependant, ce n'est pas le cas dans la définition originale des valeurs. La seule solution autre que modifier la définition aurait été de faire une récursion mutuelle sur t et la transition d'évaluation ce qui est complexe à définir et en démontrer la cohérence. Nous avons donc choisi de modifier la définition de valeur. De plus, la structure de notre démonstration consiste à réduire au maximum notre expression avant d'itérer sur les formes canoniques, ce qui correspond mieux à une stratégie d'évaluation en appel-par-valeur.

Il est maintenant possible de parler de la sûreté de la transition fonctionnelle. La propriété que nous voulons assurer est l'utilisation unique de toute ressource impliquée dans le calcul. Plus spécifiquement, il faut montrer que toutes les ressources présentes dans l'ensemble de ressources porté par le type de la fonction de signal ne soit utilisées qu'au plus une fois. Pour cela, nous utilisons les théorèmes 4.14 et 4.16 et nous définissons le théorème 4.17.

Corollaire 4.17. *Soient $\Re$ un contexte de ressources, V un environnement de ressources, t_v, t deux expressions, τ_1, τ_2 deux types et R un ensemble de ressources, si :*

1. *le triplet d'entrée terminent, c'est-à-dire que $halts(V, t_v, t)$;*
2. *$halts_{arr}(\Re, t)$;*
3. *$\Re$ et V sont cohérents ;*
4. *t est de type $\tau_1 \overset{R}{\rightsquigarrow} \tau_2$ et satisfait la relation $\varnothing \cdot \Re \vdash t : \tau_1 \overset{R}{\rightsquigarrow} \tau_2$;*
5. *t_v est de type τ_1 et satisfait la relation $\varnothing \cdot \Re \vdash t_v : \tau_1$;*
6. *Toutes cellules dans V associées à une ressource dans R sont « non-utilisée »*

alors il existe un contexte de ressources $\Re'$, un ensemble de ressources R', un environnement de ressources V', un ensemble de triplet W et deux expressions t', t'_v tels que :

1. *nous avons la relation $\langle V, t_v, t\rangle \Rrightarrow \langle V', t'_v, t', W\rangle$*
2. *seules les cellules associées à une ressource présente dans R sont mises à jour durant le calcul de l'instant, c'est-à-dire que $\forall r \in \mathtt{dom}(V) \backslash R, V\, r = V'\, r$;*
3. *le contexte $\Re$ est inclus dans $\Re'$ ainsi que R pour R' ;*
4. *chaque ressource r dans $R' \backslash R$ est dans W, mais pas dans l'environnement d'entrée V ;*

5. *chaque cellule, dans V', associée à une ressource présente dans R' est « utilisée ».*

Démonstration. La démonstration est directe et utilise les théorèmes 4.14 et 4.16. □

Ce corollaire énonce que seule les ressources présentes dans le type de la fonction peuvent être utilisées et que les autres restent bien non utilisées. Nous nous reposons sur la structure d'une cellule pour dire que si nous avons une transition fonctionnelle alors il n'est pas possible d'utiliser plus d'une fois une ressource.

4.1.5 Transition temporelle

La transition temporelle applique un programme WORMHOLES point-à-point, via la transition fonctionnelle, à un signal d'entrée tout en gérant les ressources libres ainsi que les ressources liées séparément.

Définition de la transition temporelle

Cette transition est définie dans la figure 4.11 et repose sur la définition de programme donnée ci-dessous. Un programme ne prend pas d'information via le signal d'entrée et n'en produit pas dans le signal de sortie, car toute interaction avec l'extérieur est réduit à l'utilisation de ressources.

Définition 4.6. *Soient $\Re$ un contexte de ressources et P une expression. P est un* programme *s'il est de type Unit $\overset{R}{\rightsquigarrow}$ Unit en satisfaisant la relation $\varnothing \cdot \Re \vdash P : \mathit{Unit} \overset{R}{\rightsquigarrow} \mathit{Unit}$ pour un certain ensemble de ressources R.*

La transition temporelle est une relation, notée $[R, P, W] \xrightarrow{put} [R', P', W']$, entre un triplet d'entrée, composé d'un environnement R, d'un programme P et d'un ensemble de triplets W, et un triplet de sortie, qui est le miroir de l'entrée. Elle est paramétrée par une fonction *put* qui prend une ressource typée dans le contexte par $\langle \tau_1, \tau_2 \rangle$, et une expression de type $\tau_2^\perp$ et produit une expression de type τ_1. Nous notons $\tau^\perp$ le type optionnel et $\perp$ la valeur représentant l'absence de valeur. Intuitivement, la fonction *put* échange la potentielle expression de retour de l'instant courant contre une expression d'entrée pour l'instant suivant. La transition temporelle se découpe en trois phases :

1. La transition débute par l'initialisation du système. Le triplet gauche de la relation, c'est-à-dire $[R, P, W]$, est l'entrée du système. R est un environnement associant des ressources à des expressions. Il contient l'ensemble des

$$\boxed{\begin{array}{rl} V = & \{(r, \boxed{v}) \mid R\, r = v\} \cup \\ & \{(r, \boxed{t}) \mid (r, r', t) \in W\} \cup \{(r', \boxed{\texttt{tt}}) \mid (r, r', t) \in W\} \end{array}}$$

$$\langle V, \texttt{tt}, P\rangle \Rrightarrow \langle V', \_, P', W_{new}\rangle$$

$$W' = \left\{ (r, r', t) \mid \begin{array}{l} ((r, r', t) \in W \cup W_{new} \text{ et } V'\, r' = \boxed{\texttt{tt}}) \text{ ou} \\ ((r, r', \_) \in W \cup W_{new} \text{ et } V'\, r' = \boxed{t}) \end{array} \right\}$$

$$\frac{R' = \{(r, \boxed{put(r, t)}) \mid r \in R \text{ et } V'\, r = \boxed{t} \text{ ou } t = \bot\}}{[R, P, W] \xrightarrow{put} [R', P', W']}\ \texttt{TT-Step}$$

FIGURE 4.11 – Règle de la transition temporelle de WORMHOLES.

ressources libres avec leur expression à l'instant courant. Dans le reste de la section, nous nommons R un environnement de ressources simplifié afin d'éviter toute confusion avec V. P est le programme utilisé pour effectuer le calcul de l'instant. Finalement W est un ensemble de triplets contenant toutes les ressources liées connu au début de l'instant. Pour effectuer un instant, il faut utiliser la transition fonctionnelle. Elle a besoin d'un environnement de ressources V qui est initialisé via R et W.

2. Ensuite, la transition effectue un instant logique via la transition fonctionnelle. En entrée, elle prend l'environnement V initialisé précédemment, une valeur `tt` et le programme P et elle produit un nouvel environnement V', un nouveau programme P' ainsi qu'un ensemble de triplets W_{new}.

3. Finalement, la transition traite les résultats apportés par la transition fonctionnelle. Elle met à jour les triplets contenus dans W et W_{new} via V' et fait l'union des deux ensembles pour produire l'ensemble de triplets de sortie W'. De plus, elle récupère dans R' de nouvelles expressions pour les ressources libres en échange des potentielles expressions des cellules contenus dans V'. Ceci étant fait, R' et W' sont initialisés pour une nouvelle transition.

Remarque 4.3. *L'expression de sortie de la transition fonctionnelle n'est pas utilisée, car ce n'est qu'une expression de type Unit qui ne porte aucune information.*

Exemple 4.15. *Soient P un programme, défini ci-dessous, r_{out} une ressource et $\Re = \{r_{out} : \langle Unit, \mathbb{N}\rangle\}$ un contexte. Le programme prend une entrée de type Unit*

et produit la suite des entiers naturel en commençant par 0.

$$P = \mathit{wormhole}[r_{get}, r_{set}](0;\\ \quad \mathit{rsf}[r_{get}] \ggg \mathit{arr}\,(\lambda x.(x+1, x)) \ggg \mathit{first}\,(\mathit{rsf}[r_{set}]) \ggg \mathit{arr}\,\mathit{snd} \ggg \mathit{rsf}[r_{out}])$$

P est de type Unit $\xrightarrow{\{r_{out}\}}$ *Unit et satisfait la relation* $\varnothing \cdot \Re \vdash P : \mathit{Unit} \xrightarrow{\{r_{out}\}} \mathit{Unit}$. *Avec put une fonction adéquate, nous pouvons définir la relation représentant une étape de transition temporelle comme suit :*

$$V = \{(r_{out}, \boxed{\texttt{tt}})\}$$

$$\langle V, \texttt{tt}, P\rangle \Rrightarrow \langle \{(r_{out}, \boxed{0}), (r_{get}, \boxed{\texttt{tt}}), (r_{set}, \boxed{1})\}, \_, P', \{(r_{get}, r_{set}, 0)\}\rangle$$

$$\frac{W' = \{(r_{get}, r_{set}, 1)\} \qquad R' = \{(r_{out}, \mathit{put}(r_{out}, 0))\}}{[[r_{out} \mapsto \texttt{tt}], P, \varnothing] \xrightarrow{\mathit{put}} [R', P', W']}\ \texttt{TT-Step}$$

Nous notons P' *le programme en sortie tel que :*

$$P' = \mathit{rsf}[r_{get}] \ggg \mathit{arr}\,(\lambda x.(x+1, x)) \ggg \mathit{first}\,(\mathit{rsf}[r_{set}]) \ggg \mathit{arr}\,\mathit{snd} \ggg \mathit{rsf}[r_{out}]$$

Le programme en sortie est de type Unit $\xrightarrow{\{r_{out}, r_{get}, r_{set}\}}$ *Unit. Nous pouvons noter que l'ensemble des ressources ainsi que le contexte de ressources évoluent lors de l'évaluation. En effet, les ressources liées deviennent libres et donc visible par l'extérieur.*

Les deux modifications apportées à la transition temporelle sont liées à la modélisation faite en Rocq et ne relèvent pas d'un problème de formalisation. Dans l'article original, une ressource a un type et « porte » sa cellule mémoire. Cela permet aux auteurs de définir différemment la transition temporelle. La version originale de la règle `TT-Step`, que nous nommons `TT-Step-OG`, est donnée dans la figure 4.12.

Le principe de la transition temporelle ne change pas, elle initialise un environnement V qu'elle donne à la transition fonctionnelle afin d'effectuer l'instant logique et elle finit par mettre à jour les ressources libres et liées. Cependant, la manière d'intégrer l'interaction avec l'environnement global diffère. Au lieu d'avoir R définie comme un environnement de ressources simplifié, R est un ensemble de ressources. La récupération des expressions associées est faite via la fonction *next*. L'envoie des expressions de sortie sont faite via *put* comme dans notre version, mais ce qu'elle retourne est différent. En effet, dans l'article origi-

$$V = \quad \{(r, \boxed{next(r)}) \mid r \in R\} \cup \{(r, \boxed{t}) \mid (r, r', t) \in W\} \cup \{(r', \boxed{\texttt{tt}}) \mid (r, r', t) \in W\}$$

$$\langle V, \texttt{tt}, P\rangle \Rrightarrow \langle V', \texttt{tt}, P', W_{new}\rangle$$

$$W' = \left\{(r, r', t) \mid \begin{array}{l}((r, r', t) \in W \cup W_{new} \text{ et } V'\, r' = \boxed{\texttt{tt}}) \text{ ou} \\ ((r, r', \_) \in W \cup W_{new} \text{ et } V'\, r' = \boxed{t})\end{array}\right\}$$

$$\frac{R' = \{put(r, t') \mid r \in R \text{ et } V'\, r = \boxed{t}\}}{[R, P, W] \longrightarrow [R', P', W']}\ \texttt{TT-Step-OG}$$

FIGURE 4.12 – Règle originale de la transition temporelle.

nal, *put* prend une ressource typée dans le contexte par $\langle \tau_1, \tau_2\rangle$, une expression de type τ_2 et retourne une nouvelle ressource de type $\langle \tau_1, \tau_2\rangle$. Cela permet de renouveler l'ensemble des expressions contenues dans la mémoire à chaque instant. Or, dans la définition de la syntaxe, une ressource n'est qu'un identifiant donc retourner une nouvelle ressource reviendrait à modifier l'identifiant, or cela provoquerait un problème de correspondance de ressources connues par le programme. Par conséquent, le premier changement fut de retirer la fonction *next*, modéliser R comme un environnement et le second fut de prendre en paramètre une fonction *put* qui retourne une expression pour toutes les ressources libres afin de correspondre au mieux à la définition de la syntaxe.

Propriétés de la transition temporelle

Les propriétés satisfaites par la transition fonctionnelle sont directement propagées à la transition temporelle. Nous énonçons donc les théorèmes 4.20 et 4.21 qui étendent la préservation et la réactivité à la transition temporelle. Il nous faut aussi adapter certaines hypothèses comme la cohérence entre le contexte et l'environnement de ressources pour correspondre aux structures utilisées dans cette transition. Pour cela, on définit une cohérence entre un contexte de ressources, un environnement de ressources simplifié et un ensemble de triplets dans la définition 4.7.

Définition 4.7 (Modélisation contexte/entrées)**.** *Soient* $\Re$ *un contexte de ressources,* R *un ensemble de ressources simplifié et* W *un ensemble de triplets.* $\Re$*,* R *et* W *sont* cohérents *si :*

— *leur domaine correspondent, c'est-à-dire* $\texttt{dom}(\Re) = \texttt{dom}(R) \cup \texttt{dom}(W)$ *;*

— $\texttt{dom}(R)$ *et* $\texttt{dom}(W)$ *sont disjoints, c'est-à-dire qu'une ressource ne peut être utilisée de façon libre et liée en même temps ;*

— *pour toute ressource* r *dans* R*, types* τ_1*,* τ_2 *et expression* t*, si* $\Re\, r = \langle \tau_1, \tau_2 \rangle$ *et* $R\, r = t$ *alors* t *est de type* τ_2 *et satisfait la relation* $\varnothing \cdot \Re \vdash t : \tau_2$ *;*

— *pour tout triplet* (r, r', v) *dans* W *et type* τ*, si* v *est de type* τ *alors* $\Re\, r = \langle \mathit{Unit}, \tau \rangle$ *et* $\Re\, r' = \langle \tau, \mathit{Unit} \rangle$*.*

La cohérence des entrées avec le contexte pour la transition temporelle implique la cohérence entre l'environnement de ressources et le contexte pour la transition fonctionnelle comme établit et démontré dans le théorème 4.18.

Lemme 4.18. *Pour tout contexte de ressources* $\Re$*, ensemble de ressources simplifié* R *et ensemble de triplets* W*, si* $\Re$*,* R *et* W *sont cohérents alors* $\Re$ *et* $V = \{(r, \boxed{v}) \mid R\, r = v\} \cup \{(r, \boxed{t}) \mid (r, r', t) \in W\} \cup \{(r', \boxed{\texttt{tt}}) \mid (r, r', t) \in W\}$ *sont cohérents.*

Démonstration. La démonstration se fait par raisonnement au cas par cas de chaque propriété qui compose la cohérence entre $\Re$ et V après déroulement de la définition de cohérence entre $\Re$, R et W. □

Ensuite, nous définissons la terminaison pour le triplet d'entrée et de sortie dans la définition 4.8. Cette définition est une extension des contraintes sur le triplet d'entrées de la transition fonctionnelle.

Définition 4.8 (Terminaison des entrées/sorties)**.** *Soient* P *un programme,* R *un environnement de ressources simplifié et* W *un ensemble de triplets. Si* P *termine ainsi que chaque expression* t *de* W *ou de* R *terminent alors le triplet termine, c'est-à-dire :*

$$\mathit{halts}_{e/s}(R, P, W) = \mathit{halts}(R) \wedge \mathit{halts}(P) \wedge (\forall r \in W, (\mathit{extract}_{set}(W\, r)))$$

où la fonction $\mathit{extract}_{set}$ *prend un triplet et retourne son expression. L'environnement de ressources simplifié est équipé d'une définition similaire à l'environnement de ressources.*

Dans la même logique que la cohérence, la terminaison des entrées de la transition temporelle implique la terminaison des entrées de la transition fonctionnelle. Ce fait est défini et démontré dans le théorème 4.19.

Lemme 4.19. *Pour tout un programme* P*, environnement de ressources simplifié* R *et ensemble de triplets* W*, si la propriété* $\mathit{halts}_{e/s}(R, P, W)$ *est satisfaite alors la propriété* $\mathit{halts}_{\mathit{triple}}(V, \texttt{tt}, P)$ *est satisfaite, avec* V *égal à :*

$$\{(r, \boxed{v}) \mid R\, r = v\} \cup \{(r, \boxed{t}) \mid (r, r', t) \in W\} \cup \{(r', \boxed{\texttt{tt}}) \mid (r, r', t) \in W\}$$

Démonstration. La démonstration se fait par raisonnement sur chaque sous propriété de $halts_{triple}(V, \texttt{tt}, P)$ et après déroulement de la définition 4.8. □

Nous pouvons finalement énoncer et démontrer les propriétés de préservation, voir le théorème 4.20 et de réactivité, voir le théorème 4.21. La sûreté du langage d'un point de vue utilisation des ressources est garantie s'il est possible de produire une sortie, c'est-à-dire que la transition fonctionnelle, via la définition d'une cellule, assure cette propriété. Par conséquent, la réactivité garantie la sûreté. Pour ces deux théorèmes il nous faut cependant supposer la définition 4.9 qui énonce que les expressions provenant de l'extérieur sont bien typée pour un contexte donné et qu'il termine.

Définition 4.9 (Bon comportement de *put*)**.** *Soient $\Re$ un contexte de ressources et R un ensemble de ressources et put une fonction qui prend une ressource et une potentielle expression et retourne une nouvelle expression. La fonction put se* comporte correctement *si pour toute ressource $r \in R$ telle que $\Re\, r = \langle \alpha, \beta \rangle$, avec α et β deux types, et expression t de type α, nous avons une expression $t' = put(r, t)$ telle que :*

- *la propriété $\varnothing \cdot \Re \vdash t' : \beta$ est satisfaite ;*
- *t' termine.*

Théorème 4.20 (Préservation du typage à travers ($\xrightarrow{put}$))**.** *Pour toute fonction put, contexte de ressources $\Re$, ensemble de ressources R, environnements de ressources simplifié $\mathcal{R}$, $\mathcal{R}'$, ensembles de triplets W, W' et programmes P, P', si :*

1. *la fonction put se comporte correctement avec $\Re$ et $\texttt{dom}(R)$;*
2. *$halts_{arr}(\Re, P)$;*
3. *le triplet d'entrée termine, c'est-à-dire que $halts_{e/s}(\mathcal{R}, P, W)$ est satisfait ;*
4. *$\Re$, $\mathcal{R}$ et W sont cohérents ;*
5. *P est de type $Unit \overset{R}{\rightsquigarrow} Unit$, c'est-à-dire qu'il satisfait la relation $\varnothing \cdot \Re \vdash P : Unit \overset{R}{\rightsquigarrow} Unit$;*
6. *la relation $[\mathcal{R}, P, W] \xrightarrow{put} [\mathcal{R}', P', W']$ est satisfaite*

alors il existe $\Re'$ et $\mathcal{R}'$ tels que :

1. *$halts_{arr}(\Re', P')$;*
2. *le triplet de sortie termine, c'est-à-dire que $halts_{e/s}(\mathcal{R}', P', W')$ est satisfait ;*
3. *$\Re'$, $\mathcal{R}'$ et W' sont cohérents ;*
4. *P' est de type $Unit \overset{R'}{\rightsquigarrow} Unit$, c'est-à-dire qu'il satisfait la relation $\varnothing \cdot \Re' \vdash P' : Unit \overset{R'}{\rightsquigarrow} Unit$*

Démonstration. La démonstration se fait en déroulant la définition de transition temporelle afin de faire ressortir la transition fonctionnelle. Avec les théorèmes 4.18 et 4.19, nous déduisons les hypothèses nécessaires à l'utilisation du théorème 4.14. Ce théorème nous permet de définir $\Re'$ et $\mathcal{R}'$ et de démontrer les différents buts excepté pour la terminaison de $\mathcal{R}'$ uniquement et la cohérence de $\Re'$, $\mathcal{R}'$ et W'. En effet, pour ceux-là, la propriété de bon comportement de la fonction *put* est nécessaire. □

Théorème 4.21 (Réactivité)**.** *Pour toute fonction put, contexte de ressources $\Re$, ensemble de ressources R, environnement de ressources simplifié $\mathcal{R}$, ensemble de triplets W et programme P, si :*

1. *la fonction put se comporte correctement avec $\Re$ et $\texttt{dom}(R)$;*
2. *$halts_{arr}(\Re, P)$;*
3. *le triplet d'entrée termine, c'est-à-dire que $halts_{e/s}(\mathcal{R}, P, W)$ est satisfait ;*
4. *$\Re$, $\mathcal{R}$ et W sont cohérents ;*
5. *P est de type $Unit \overset{R}{\rightsquigarrow} Unit$, c'est-à-dire qu'il satisfait la relation $\varnothing \cdot \Re \vdash P : Unit \overset{R}{\rightsquigarrow} Unit$*

alors il existe $\mathcal{R}'$, P' et W' tels que :

1. *le triplet de sortie termine, c'est-à-dire que $halts_{e/s}(\mathcal{R}', P', W')$ est satisfait ;*
2. *la relation $[\mathcal{R}, P, W] \xrightarrow{put} [\mathcal{R}', P', W']$ est satisfaite*

Démonstration. La démonstration se fait en déroulant la définition de transition temporelle et en utilisant les théorèmes 4.16 et 4.20. □

4.2 Formalisation en Rocq

Les nouvelles définitions ainsi que les propriétés énoncées dans la section précédente sont mécanisées dans l'assistant de preuve Rocq. Le projet est disponible sur le dépôt git `https://github.com/JordanIschard/Mechanized-Wormholes`. Il compte environ 2 mille lignes de code (KLdC) de spécifications et 3 KLoC de démonstrations pour un total de presque 300 théorèmes, lemmes, etc. La représentation des ressources dans la mécanisation en Rocq est un challenge que nous développons en tant que cas d'étude dans la section A.3. Dans cette section, nous nous focalisons sur l'organisation du dépôt afin d'orienter le lecteur qui voudrait étudier la version mécanisée de nos définitions. Le projet se décompose en trois dossiers : 1. le dossier `Syntax` ; 2. le dossier `Environment` ; 3. le dossier `Semantics`. Le premier dossier contient les termes, les variables, les ressources ainsi que les

types du langage. Le deuxième dossier contient l'ensemble des environnements, contextes et ensembles nécessaires à la définition de la syntaxe et la sémantique. Pour finir, le troisième dossier contient l'ensemble des transitions, c'est-à-dire la sémantique dynamique, ainsi que le système de type, c'est-à-dire la sémantique statique. Dans la suite, nous nous focalisons sur chaque dossier en donnant un organigramme ainsi qu'une explication concise de ce que contient chaque fichier et à quoi il fait référence dans la section précédente.

4.2.1 Dossier `Syntax`

Le dossier `Syntax` se compose de sept fichiers énumérés ci-dessous. Leur dépendance est donnée par la figure 4.13. Les noms des fichiers sont significatifs de leur contenu. Ce découpage très fin de la syntaxe est dû à l'utilisation de la bibliothèque `DeBrLevel` décrite dans le chapitre A qui nécessite d'encapsuler les définitions des différentes parties de la syntaxe dans des modules.

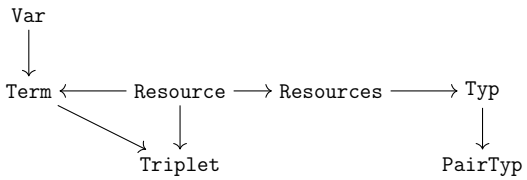


FIGURE 4.13 – Organigramme des modules du dossier `Syntax`.

1. `Var.v` : il contient le module `Var` représentant le domaine des variables.
2. `Resource.v` : il contient un module `Resource`. Il représente le domaine des ressources.
3. `Resources.v` : il contient le module `Resources` qui représentent le domaine des ensembles des ressources.
4. `Term.v` : le fichier se compose d'un unique module `Term`. Il comporte la définition des termes ainsi que des valeurs.
5. `Typ.v` : il se compose des modules `Typ` et `PairTyp`. Le premier module définit l'ensemble des types et le second l'ensemble des paires de types. Les paires de types sont utilisées dans la définition du contexte des ressources.
6. `Cell.v` : il contient le module `Cell` représentant les cellules mémoire de l'environnement de ressources.
7. `Triplet.v` : il contient le module `Triplet`. Ce module représente un triplet composé de deux ressources et d'un terme utilisé dans l'ensemble W.

4.2.2 Dossier `Environment`

Le dossier `Environment` se compose de cinq fichiers énumérés ci-dessous. Leur dépendance est donnée par la figure 4.14. Dans le graphe de dépendance, nous précisons les liens avec le dossier `Syntax`.

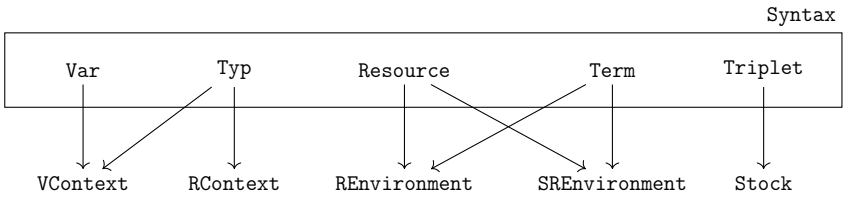


FIGURE 4.14 – Organigramme des modules du dossier `Environment`.

1. `VContext.v` : il contient un module `VContext` représentant le contexte des variables.
2. `RContext.v` : il contient un module `RContext` représentant le contexte des ressources.
3. `REnvironment.v` : il contient le module `REnvironment`. Ce module représente l'environnement de ressources. Il y est aussi défini la terminaison d'un environnement.
4. `SREnvironment.v` : le fichier contient un module `SREnvironment`. Il représente l'environnement de ressources simplifié qui associe une ressource à un terme, et non à une cellule comme `REnvironment`. De plus, ce module contient une fonction nommée `init_globals` qui initialise un environnement de ressources en fonction d'un environnement de ressources simplifié.
5. `Stock.v` : Il contient un module `Stock` représentant l'ensemble de triplets W présent dans la transition fonctionnelle et temporelle. Tout comme `SREnvironment`, il y est défini une fonction `init_locals` qui initialise un environnement de ressources en fonction d'un ensemble de triplets et un environnement de ressources.

4.2.3 Dossier `Semantics`

Le dossier `Semantics` ne regroupe que quatre fichiers représentant la sémantique statique et dynamique du langage. Nous les énumèrons ci-dessous. L'ensemble des lemmes/théorèmes/corollaires et définitions établis dans la section 4.1 sont regroupés dans les tableaux 4.1 et 4.2 avec leur référence dans le manuscrit et leur nom dans la mécanisation en ROCQ.

1. `Type_System.v` : ce fichier contient la définition du système de types, présentée dans les figures 4.4 et 4.5, ainsi que le théorème 4.4, le théorème 4.1 et les théorèmes 4.2 et 4.3.
2. `Evaluation_Transition.v` : il est composé des définitions de la substitution, de la transition d'évaluation, présenté dans la figure 4.8 et de la terminaison d'un terme, voir la définition 4.2. De plus, il contient les théorèmes 4.5 et 4.6, les théorèmes 4.8 et 4.10 et les théorèmes 4.9 et 4.11.
3. `Functional_Transition.v` : ce fichier contient la définition de la transition fonctionnelle (voir la figure 4.9) ainsi que celle de la cohérence entre le contexte et l'environnement (voir la définition 4.1) et les différentes définitions de terminaison, excepté celles des termes (voir les définitions 4.3 à 4.5). Il contient aussi les théorèmes 4.12, 4.13 et 4.15, les théorèmes 4.14 et 4.16 et le théorème 4.17.
4. `Temporal_Transition.v` : il contient la définition de la transition temporelle, présenté dans la figure 4.11 ainsi que la définition de cohérence des entrées (voir la définition 4.7). Les théorèmes 4.20 et 4.21 y sont démontrés.

4.2.4 Sources

Dans ce chapitre, nous avons présenté une variante du langage WORMHOLES. Elle corrige les problèmes trouvés dans la formalisation définie dans l'article de Daniel Winograd-Cort et Paul Hudak [102]. Afin de s'assurer de la véracité de nos résultats, nous avons mécanisé la variante du langage et prouvé ses propriétés dans l'assistant de preuve ROCQ. Ce travail de formalisation est disponible sur le dépôt git `https://github.com/JordanIschard/Mechanized-Wormholes` et compte un peu plus de 7 mille lignes de code au total.

TABLE 4.1 – Correspondance entre les différentes définitions du manuscrit et celles dans ROCQ.

Nom	Référence thèse	Nom ROCQ
Programme	définition 4.6	×
Sémantique statique	figures 4.4 et 4.5	`well_typed`
Substitution	×	`subst`
Transition d'évaluation	figure 4.8	`evaluate`
Multiple transition d'évaluation	×	`multi`
Transition fonctionnelle	figure 4.9	`functional`
Transition temporelle	figure 4.11	`temporal`
Cohérence contexte/env.	définition 4.1	`well_formed_ec`
Cohérence contexte/entrée	définition 4.7	`well_formed_tt`
Terminaison d'un terme	définition 4.2	`halts`
Terminaison des entrées	définition 4.3	`fT_inputs_halts`
Terminaison des sorties	définition 4.4	`fT_outputs_halts`
Terminaison des fcts élevées	définition 4.5	`halts_arr`
Terminaison E/S	définition 4.8	`tT_IO_halts`
Bon comportement *put*	définition 4.9	`put_good_behavior`

TABLE 4.2 – Correspondance entre les différents lemmes du manuscrit et ceux dans ROCQ.

Référence thèse	Nom ROCQ
théorème 4.4	`typing_Re_R`
théorème 4.2	`weakening_Γ`
théorème 4.3	`weakening_ℜ`
théorème 4.5	`evaluate_not_value`
théorème 4.6	`evaluate_deterministic`
théorème 4.8	`evaluate_preserves_typing`
théorème 4.10	`progress_of_evaluate`
théorème 4.9	`multi_preserves_typing`
théorème 4.11	`progress_of_multi_evaluate`
théorème 4.12	`functional_preserves_keys` et `functional_W_props`
équation (4.1a)	`halts_arr_eT`
équation (4.1b)	`halts_arr_first` et `halts_arr_first'`
équation (4.1c)	`halts_arr_comp`
équation (4.1d)	`halts_arr_comp'`
équation (4.1e)	`halts_arr_wh`
équation (4.1f)	`halts_arr_weakening`
théorème 4.14	`functional_preserves_typing`
théorème 4.16	`progress_of_functional`
théorème 4.17	`safety_resources_interaction`
théorème 4.18	`WF_tt_to_WF_ec`
théorème 4.19	`tT_inp_to_fT_inp`
théorème 4.20	`temporal_preserves_typing`
théorème 4.21	`temporal_reactivity`

Chapitre 5

Molholes : un langage fonctionnel réactif avec effets plus permissif

Sommaire

Wormholes offre un moyen simple et élégant de gérer les interactions avec l'environnement à travers des ressources. En effet, grâce aux opérateurs *rsf* et *wormhole*, il est possible d'intégrer des effets de bords ainsi que l'utilisation de références locales naturellement dans la composition des fonctions de signal qui définit un programme. Cependant, le bon comportement des programmes de ce langage nécessite des contraintes d'accès strictes, n'autorisant l'accès aux ressources qu'une seule fois par instant logique. Pour les entrées et les sorties, cette restriction s'aligne à l'hypothèse synchrone. Toutefois, dans le cas de ressources locales, cette restriction peut être trop rigide.

Exemple 5.1. *Soit t une expression* WORMHOLES *définie ci-dessous. La partie de l'expression sous l'opérateur wormhole agit comme un accumulateur qui donne une information complémentaire v à f et stocke une partie du résultat pour l'instant suivant.*

$$\begin{aligned} t = {} & arr\,(\lambda x.(x, ())) \ggg \\ & wormhole[r_{get,r_{set}}](v; second\,(rsf[r_{get}]) \ggg arr\,f \ggg second\,(rsf[r_{set}])) \ggg \\ & \ldots \ggg arr\,g \end{aligned}$$

Modifions légèrement ce programme en remplaçant la fonction g par g'. g' nécessite le résultat complet de f. Par conséquent, il faut mettre toute la séquence de fonctions de signal jusqu'à g' sous la portée de l'opérateur wormhole et adapter le code entre f et g'. En effet, il n'est pas possible de simplement réutiliser la ressource de lecture, car cela serait une violation de la contrainte d'utilisation unique. La version modifiée de t, nommée t', est définie ci-dessous.

$$\begin{aligned} t' = {} & arr\,(\lambda x.(x, ())) \ggg \\ & wormhole[r_{get,r_{set}}](v; second\,(rsf[r_{get}]) \ggg arr\,f \ggg \\ & \qquad second\,(\ldots) \ggg arr\,g' \ggg second\,(rsf[r_{set}])) \end{aligned}$$

Nous revenons au problème initial que WORMHOLES *soulevait, c'est-à-dire qu'une modification locale du programme implique plus de réécriture qu'elle ne le devrait.*

Dans ce chapitre, nous présentons MOLHOLES, un langage qui étend l'approche WORMHOLES en assouplissant les contraintes d'utilisation des ressources locales. MOLHOLES adopte un paradigme à trois ressources composé d'*entrées*, de *sorties* et de ressources *internes*. Il évince la restriction d'accès unique sur les ressources internes, autorisant des lectures et des écritures multiples. Cependant, les ressources d'entrée doivent être lues exactement une fois et les ressources de sortie doivent être écrites exactement une fois. Cette dernière propriété garantit que les programmes sont productifs, c'est-à-dire qu'ils sont capables de produire un résultat à chaque instant. De plus, le langage ne contient pas de terme équivalent à *wormhole*, car nous supposons que les ressources internes seront connus avant l'exécution, et l'opérateur *rsf* est séparé en deux permettant de différencier les lectures des écritures via deux nouveaux opérateurs `Get` et `Set`. En plus du langage, nous démontrons qu'il existe un pont entre MOLHOLES et YAMPA. En effet, nous avons établi une transformation qui va d'un programme MOLHOLES, avec potentiellement des utilisations de ressources, vers un programme YAMPA qui est donc sans effets de bords. Cette transformation préserve bien évidemment

la sémantique du programme de départ.

La section 5.1 définit les domaines sémantiques sur lesquels se base MOLHOLES et YAMPA. La section 5.2 donne une syntaxe et une sémantique à un sous-ensemble de YAMPA nommé YAMPACORE. Le langage MOLHOLES est défini dans la section 5.3. Finalement, l'équivalence entre langages ainsi que celles dans chacun d'eux sont décrites et démontrées dans la section 5.4.

5.1 Domaines sémantiques

Dans la suite de ce chapitre, nous formalisons la sémantique de YAMPA et de MOLHOLES en plus d'établir des transformations. Ces transformations ont lieu à l'intérieur de chacun des langages et entre les deux langages. Chacun d'eux a deux niveaux de syntaxe : le premier niveau consiste en un langage simple permettant de définir des valeurs et des fonctions, tandis que le second niveau forme le langage de domaine spécifique au traitement des flux. Nous définissons un langage hôte abstrait HST qui sert de premier niveau de syntaxe commun au deux langages. Il est axiomatisé au niveau sémantique par un ensemble de valeurs et fonctions typées formant une catégorie cartésienne localement petite $\mathcal{C}_{host}$.

Remarque 5.1. *La contrainte pour $\mathcal{C}_{host}$ d'être cartésienne et localement petite assure la présence des types produits ainsi que des fonctions sur des paires dans le langage hôte, ce qui est nécessaire pour l'utilisation de la famille de morphismes first.*

Les programmes, dans chaque langage, représentent des fonctions synchrones préservant la taille du flux, c'est-à-dire qu'une entrée correspond à exactement une sortie. Leurs comportements se réduisent donc à une application itérative d'une fonction d'étape, unique à chaque langage. Les domaines sémantiques de ces fonctions d'étape peuvent être modélisés comme des coalgèbres pour une famille de foncteurs. Cette famille, indicée par les types de HST, est définie comme suit :

$$\begin{array}{lcll} \mathbf{F}_{A,B}\, X & = & A \to (B \times X) & \\ \mathbf{F}_{A,B} & : & (X \to Y) \to (\mathbf{F}_{A,B}\, X \to \mathbf{F}_{A,B}\, Y) & \\ \mathbf{F}_{A,B}\, f & = & \lambda step\, a.(b, f\, x) & \text{où } (b, x) = step\, a \end{array}$$

Avant d'aller plus loin, nous exprimons ce qu'est une colagèbre dans la définition 5.1.

Définition 5.1 (Coalgèbre)**.** *Étant donné un endofoncteur $\mathbf{F}$ sur une catégorie $\mathcal{C}$, une coalgèbre pour $\mathbf{F}$ (aussi nommée une $\mathbf{F}$-coalgèbre) est une paire (X, f)*

où X est un objet et $f : X \to \mathbf{F}\,X$ un morphisme, tous deux provenant de la catégorie $\mathcal{C}$.

Étant donné deux types A et B et une coalgèbre (X, f) pour $\mathbf{F}_{A,B}$, X représente un type de transformateurs et f est un morphisme qui crée une fonction d'étape à partir d'un transformateur. Intuitivement, cette fonction applique le transformateur à une valeur d'entrée de type A, produisant une valeur de sortie de type B ainsi qu'un nouveau transformateur. Nous omettons parfois l'objet X quand nous définissons une colagèbre. Dans ce cas, nous donnons la fonction f avec son type, ce qui permet de déduire X.

Exemple 5.2. *Nous modélisons un flux par une famille indicée de types Stream A, où A est un type de* Hst*, tel que :*

$$\mathit{Stream}\ A = \texttt{Cons} : A \times \mathit{Stream}\ A \to \mathit{Stream}\ A$$

Nous pouvons définir la coalgèbre (X, f) pour le foncteur $\mathbf{F}_A = X \to A \times X$ où X est le type Stream A et $f = \lambda(\texttt{Cons}\,(x, s)).(x, s)$.

L'exactitude des transformations est établie en appliquant les techniques de démonstration par *bisimulation* entre les coalgèbres de $\mathbf{F}_{A,B}$. Ce sont les techniques générales pour raisonner sur des comportements infinis. Nous établissons ce qu'est une bisimulation dans notre cas dans la définition 5.2.

Définition 5.2 (Bisimulation)**.** *Soient X,Y deux objets d'une même catégorie $\mathcal{C}$ et $step_X : X \to \mathbf{F}_{A,B}\,X$, $step_Y : Y \to \mathbf{F}_{A,B}\,Y$ deux colagèbres. Une bisimulation entre X et Y est une relation R telle que :*

$$\forall x \in X\ y \in Y, step_X\ x\ a = (b, x') \implies \exists y' \in Y, step_Y\ y\ a = (b, y') \wedge R\,x'\,y'$$

Exemple 5.3. *Une bisimulation sur les flux est définie comme une relation R telle que, pour tout s et s' tels que $R\,s\,s'$, nous avons :*

$$s = \texttt{Cons}\,a\,s_1 \implies \exists s_1', s' = \texttt{Cons}\,a\,s_1' \wedge R\,s_1\,s_1'$$

La bisimulation nous offre une méthode puissante pour établir l'équivalence comportementale, c'est-à-dire la *bisimilarité*, à la fois au sein d'un même langage et entre les deux langages. La *bisimularité*, noté $x \sim y$, est la plus grande bisimulation entre X et Y. Intuitivement, la bisimilarité entre deux transformateurs exprime l'idée qu'ils ont le même comportement. Dans le contexte de Yampa et Molholes, cela signifie que, pour toute entrée donnée, les transformateurs produisent la même sortie et que leurs transformateurs mis à jour restent bisimilaires.

Montrer que deux éléments sont bisimilaires revient à construire une bisimulation entre eux. La bisimilarité, entre X et Y, est une relation réflexive, symétrique et transitive comme mentionnée ci-dessous. Par conséquent, la relation $\sim$ est une équivalence si X est égal à Y.

$$x \sim x \tag{5.1a}$$

$$x \sim y \Rightarrow y \sim x \tag{5.1b}$$

$$x \sim y \wedge y \sim z \Rightarrow x \sim z \tag{5.1c}$$

Remarque 5.2. *Il est intéressant de noter que cette notion de bisimulation s'aligne plus étroitement sur la simulation dans la terminologie des algèbres de processus. La bisimulation exige que les deux éléments se simulent l'un l'autre. Cependant, lorsque nous considérons des systèmes décrits par des fonctions totales, les deux notions coïncident.*

5.2 YampaCore

YampaCore est un sous-ensemble minimal de Yampa, capturant ses concepts fondamentaux. Nous avons déjà introduit ce sous-ensemble dans l'état de l'art et plus précisément dans la section 3.3. Dans cette section, nous établissons une sémantique formelle de YampaCore dans le but de l'utiliser comme coalgèbre d'entrée ou d'arrivée pour nos transformations. Nous en profitons pour démontrer via la bisimulation que YampaCore définit effectivement une flèche avec des boucles.

5.2.1 Syntaxe

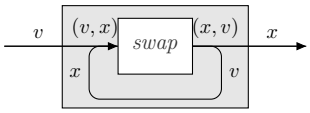
Comme dit dans la section précédente, YampaCore est un langage à deux niveaux : le premier niveau consiste en un langage hôte dont la sémantique est fournie par la catégorie cartésienne $\mathcal{C}_{hst}$, tandis que le second niveau est un langage spécifique au domaine pour le traitement des flux. Les programmes représentent des fonctions totales sur des flux de valeurs, où les flux sont modélisés comme une famille indexée de types *Stream A*, A étant un type du langage hôte, voir exemple 5.2.

Remarque 5.3. *Nous ne supposons pas que les types de flux sont des types du langage hôte, reflétant le fait que les flux ne sont pas traités comme des citoyens de première classe dans* Yampa.

Nous définissons la syntaxe de YampaCore ci dessous, en utilisant des termes typés. Ces termes sont des éléments de l'algèbre $SF\,A\,B$, où A et B sont des types du langage hôte. La définition de SF correspond à la description informelle de Yampa fournie dans la section 3.3. Le terme `Comp` correspond à l'opérateur infixe ($\ggg$).

$$\begin{aligned} SF = \; & \mid \texttt{Arr} : \forall A\,B, (A \to B) \to SF\,A\,B \\ & \mid \texttt{First} : \forall A\,B\,C, SF\,A\,B \to SF\,(A \times C)\,(B \times C) \\ & \mid \texttt{Comp} : \forall A\,B\,C, SF\,A\,B \times SF\,B\,C \to SF\,A\,C \\ & \mid \texttt{Loop} : \forall A\,B\,C, C \times SF\,(A \times C)\,(B \times C) \to SF\,A\,B \end{aligned}$$

Exemple 5.4. *L'expression* $\texttt{Loop}\,(x, \texttt{Arr}\,swap)$ *représente la fonction de signal* $delay\,x$ *qui prend un signal d'entrée et le retourne, mais décalé d'un instant logique. La première valeur de cette fonction est* x*. La représentation graphique est donnée ci-dessous.*



5.2.2 Sémantique

La sémantique de YampaCore se découpe en une fonction d'étape que nous nommons `step` et une fonction sur les flux que nous nommons `run`. La première effectue le calcul d'un instant et la seconde applique la première répétitivement sur un flux. Nous commençons par définir le domaine sémantique des termes. Une fonction de signal dans YampaCore est de type $sf\,A\,B$ et se comporte comme une fonction qui prend une entrée de type A et produit une sortie de type B ainsi qu'une nouvelle version d'elle-même. Soient A et B des types de Hst, le domaine sémantique des termes est défini par le type coinductif :

$$sf\,A\,B = A \to (B \times sf\,A\,B)$$

Ce type constitue la coalgèbre $(sf\,A\,B, id)$ pour le foncteur $\mathbf{F}_{A,B}$ introduit dans la section 5.1. Il existe quatre familles de fonctions de signal dans YampaCore *arr*, *first*, *comp* et *loop* que nous définissons sous forme de fonctions corécursives

ci-dessous.

$$
\begin{array}{lcll}
\mathit{arr}\ f & = & \lambda x.(f\ x, \mathit{arr}\ f) & \\
\mathit{first}\ sf & = & \lambda(x,z).((y,z), \mathit{first}\ sf') & \text{où } (y, sf') = sf\ x \\
\mathit{comp}\ sf_1\ sf_2 & = & \lambda x.(z, \mathit{comp}\ sf_1'\ sf_2') & \text{où } (y, sf_1') = sf_1\ x \text{ et } (z, sf_2') = sf_2\ y \\
\mathit{loop}\ v\ sf & = & \lambda x.(y, \mathit{loop}\ v'\ sf') & \text{où } ((y, v'), sf') = sf\ (x, v)
\end{array}
$$

Maintenant, nous faisons le lien entre les termes et les fonctions de signal qu'ils représentent via la fonction récursive `step` qui plonge les termes dans cette coalgèbre.

$$
\begin{array}{lcl}
\texttt{step} & : & \mathit{SF}\ A\ B \to \mathit{sf}\ A\ B \\
\texttt{step}\ (\texttt{Arr}\ f) & = & \mathit{arr}\ f \\
\texttt{step}\ (\texttt{First}\ \mathit{sf}) & = & \mathit{first}\ (\texttt{step}\ \mathit{sf}) \\
\texttt{step}\ (\texttt{Comp}\ (\mathit{sf}_1, \mathit{sf}_2)) & = & \mathit{comp}\ (\texttt{step}\ \mathit{sf}_1)\ (\texttt{step}\ \mathit{sf}_2) \\
\texttt{step}\ (\texttt{Loop}\ (v, \mathit{sf})) & = & \mathit{loop}\ v\ (\texttt{step}\ \mathit{sf})
\end{array}
$$

Exemple 5.5. *La sémantique de l'expression* $\texttt{Loop}\,(v, \texttt{Arr}\,\mathit{swap})$*, c'est-à-dire la fonction delay est donnée par :*

$$
\begin{aligned}
\texttt{step}\,(\texttt{Loop}\,(v, (\texttt{Arr}\,\mathit{swap}))) &= \mathit{loop}\,v\,(\mathit{arr}\,\mathit{swap}) \\
&= \lambda x.(v, \mathit{loop}\,x\,(\mathit{arr}\,\mathit{swap})) \\
&= \lambda x.(v, \texttt{step}\,(\texttt{Loop}\,(x, (\texttt{Arr}\,\mathit{swap}))))
\end{aligned}
$$

Remarque 5.4. *Il est intéressant de noter que, dans l'exemple précédent, la fonction de signal produite est la même que la fonction de signal originale à l'exception de l'état interne qui passe de* v *à* x*. Cette propriété est vraie pour tout programme* YampaCore *où de multiples mises à jour peuvent être effectuées. Cela simplifie énormément les démonstrations par bisimulation.*

Finalement, nous définissons `run` une fonction corécursive qui prend une fonction de signal *sf* de type $\mathit{sf}\ A\ B$ et produit une fonction qui prend un flux de type $\mathit{Stream}\ A$ et retourne un flux de type $\mathit{Stream}\ B$ en appliquant *sf* point à point.

$$
\begin{array}{lcll}
\texttt{run} & : & \mathit{sf}\ A\ B \to \mathit{Stream}\ A \to \mathit{Stream}\ B & \\
\texttt{run}\ \mathit{sf}\ (\texttt{Cons}\ x\ s) & = & \texttt{Cons}\ y\ (\texttt{run}\ \mathit{sf}'\ s) & \text{où } (y, sf') = sf\ x
\end{array}
$$

Exemple 5.6. *Pour l'expression* $t = \texttt{Loop}\,(0, (\texttt{Arr}\,\lambda(x,y).(y, y+1)))$ *et* s *un flux de type Stream Unit,* $\texttt{run}\,(\texttt{step}\,t)\,s$ *représente le flux de type Stream* $\mathbb{N}$ *isomorphe*

à la liste infini croissante des entiers naturels.

$$
\begin{array}{lcl}
\texttt{run}\,(\texttt{step}\,t)\,s & : & \mathit{Stream}\,\mathbb{N} \\
\texttt{run}\,(\texttt{step}\,t)\,s & = & \texttt{run}\,(\mathit{loop}\,0\;\mathit{arr}\;\lambda(x,y).(y,y+1))\,(\texttt{Cons}\,\texttt{tt}\;s') \\
 & = & \texttt{Cons}\,0\,(\texttt{run}\,(\mathit{loop}\,1\;\mathit{arr}\;\lambda(x,y).(y,y+1))\,(\texttt{Cons}\,\texttt{tt}\;s'')) \\
 & = & \texttt{Cons}\,0\,(\\
 & & \qquad \texttt{Cons}\,1\,(\texttt{run}\,(\mathit{loop}\,2\;\mathit{arr}\;\lambda(x,y).(y,y+1))\,(\texttt{Cons}\,\texttt{tt}\;s'''))) \\
 & = & \ldots
\end{array}
$$

Maintenant que la sémantique est définie, nous démontrons que YAMPACORE forme une flèche avec boucle. Pour cela, nous définissons ce qu'est une bisimulation entre deux fonctions de signal ci-dessous.

Définition 5.3 (Bisimulation dans YAMPACORE)**.** *Une relation R est une bisimulation sur sf A B si pour toutes fonctions de signal sf_1 et sf_2 telle que $R\,sf_1\,sf_2$ nous avons :*

$$sf_1\,x = (y, sf_1') \implies \exists sf_2', sf_2\,x = (y, sf_2') \wedge R\,sf_1'\,sf_2'$$

Pour rappel, la bisimilarité, notée $\sim$, est définie comme la plus grande bisimulation. Le théorème 5.2 énonce que YAMPACORE forme une flèche avec boucle. Sa démonstration nécessite que la relation de bisimilarité soit une congruence. Cette propriété est établie et démontrée dans le théorème 5.1.

Lemme 5.1. *Pour toutes fonctions de signal sf, sf', sf_1, sf_1', sf_2, sf_2' et valeur du langage hôte v, les propriétés suivantes sont respectées :*

- *si $sf_1 \sim sf_1'$ et $sf_2 \sim sf_2'$ alors $comp\;sf_1\;sf_2 \sim comp\;sf_1'\;sf_2'$;*
- *si $sf \sim sf'$ alors $first\;sf \sim first\;sf'$;*
- *si $sf \sim sf'$ alors $loop\,(v, sf) \sim loop\,(v, sf')$*

Démonstration. La démonstration se fait par bisimulation. Pour le premier cas, nous définissons la relation R telle que :

$$R = \{(comp\;sf_1\;sf_2, comp\;sg_1\;sg_2) \mid sf_1 \sim sg_1 \wedge sf_2 \sim sg_2\}$$

Démontrer que R est une bisimulation nous permet de conclure le cas. Par définition, si R est une bisimulation alors elle doit satisfaire la propriété suivante :

$$sf\,x = (y, sf') \implies \exists sg', sg\,x = (y, sg') \wedge R\,sf'\,sg'$$

pour sf, sf', sg et sg' des fonctions de signal et x et y des valeurs du langage hôte. Dans notre cas, $sf = comp\;sf_1\;sf_2$ et $sg = comp\;sg_1\;sg_2$. Par définition de $comp$, il

existe sf'_1, sf'_2 et z tel que $sf_1\, x = (z, sf'_1)$ et $sf_2\, z = (y, sf'_2)$. Par bisimilarité, nous savons qu'il existe sg'_1 et sg'_2 tel que :

1. $sg_1\, x = (z, sg'_1)$ et $sf'_1 \sim sg'_1$;
2. $sg_2\, x = (z, sg'_2)$ et $sf'_2 \sim sg'_2$.

Par conséquent, il existe $sg' = comp\, sg'_1\, sg'_2$ tel que $sg\, x = (y, sg')$ et $R\, sf'\, sg'$. Les deux autres cas suivent la même structure avec, respectivement, les relations $\{(first\, sf, first\, sg) \mid sf \sim sg\}$ et $\{(loop\, v\, sf, loop\, v\, sg) \mid sf \sim sg\}$. □

Théorème 5.2. *Les fonctions arr, first, comp et loop forment une flèche avec boucle.*

Démonstration. La démonstration de chaque loi présente dans les équations (2.10) et (2.12) se fait par bisimulation. Par exemple, pour l'équation (2.10g), nous devons satisfaire la proposition suivante :

$$comp\,(first\,(first\, sf))\,(arr\, assoc) \sim comp\,(arr\, assoc)\,(first\, sf)$$

Pour cela, nous définissons une relation R_{ABC} telle que :

$$\{(comp\,(first\,(first\, sf))\,(arr\, assoc), comp\,(arr\, assoc)\,(first\, sf))\}$$

où sf est de type $sf\,(A \times C)\,(B \times C)$. Démontrer que R_{ABC} est une bisimulation nous permet de conclure le cas. Par définition, si R_{ABC} est une bisimulation alors elle doit satisfaire la propriété suivante :

$$sf_1\, x = (y, sf'_1) \implies \exists sg', sg\, x = (y, sg') \wedge R_{ABC}\, sf'_1\, sg'$$

pour sf_1, sf'_1, sg sg' des fonctions de signal et x et y des valeurs du langage hôte. Dans notre cas, nous avons :

$$\begin{aligned} sf_1 &= comp\,(first\,(first\, sf))\,(arr\, assoc) \\ sg &= comp\,(arr\, assoc)\,(first\, sf) \end{aligned}$$

avec $x = ((x_1, x_2), x_3)$ et $y = (y, (x_2, x_3))$, où x_1, x_2, etc sont des valeurs de *hst* car :

$$(comp\,(first\,(first\, sf))\,(arr\, assoc))\,((x_1, x_2), x_3) = ((y, x_2), x_3, sf_1)$$

Nous savons aussi que :

$$(comp\,(arr\, assoc)\,(first\, sf))\,((x_1, x_2), x_3) = ((y, x_2), x_3, sg)$$

Par définition de R_{ABC}, nous avons $R_{ABC}\ sf_1\ sg$. Les autres cas sont similaires. □

5.3 MOLHOLES

MOLHOLES est un langage fonctionnel réactif simple qui reprend certains concepts du langage WORMHOLES. Tout comme WORMHOLES, ce langage remplace la famille d'opérateurs de boucle *loop* par des fonctions de signal qui lisent et écrivent des ressources. Cependant, contrairement à WORMHOLES, il n'introduit pas de lieurs pour les ressources locales. À la place, nous nous appuyons sur l'état des ressources qui sont classées en tant qu'entrées, sorties ou ressources internes. Les ressources d'entrée et de sortie sont à usage unique, ce qui signifie que nous ne pouvons y accéder qu'une seule fois par instant logique, au contraire des ressources internes qui peuvent être lues et écrites à tout moment. Le langage est équipé d'un système de types qui garantit que les ressources d'entrée et de sortie sont utilisées correctement. Nous commençons par définir la syntaxe du langage ainsi que le concept de programme. Puis nous introduisons la sémantique dynamique suivie par la sémantique statique.

5.3.1 Syntaxe

Les termes du langage MOLHOLES reposent sur le concept de *ressources*. Nous définissons ci-dessous le type *Ref A* qui représente les ressources. Une ressource porte une valeur de type A et un nombre naturel, qui est son identifiant unique.

$$\mathit{Ref} = \texttt{Ref} : \forall A.\ A \to \mathbb{N} \to \mathit{Ref}\ A$$

Une valeur de type *Ref A* est exprimée sous la forme $\texttt{Ref}\ A\ n$, où n est l'identifiant de la ressource. Nous notons $\texttt{id} : \mathit{Ref}\ A \to \mathbb{N}$ la fonction qui retourne l'identifiant d'une ressource. Nous définissons l'ensemble des termes du langage ci-dessous.

$$\begin{aligned}
\mathit{RSF} \quad = \quad & \mid \texttt{Arr} : \forall A\,B.\,(A \to B) \to \mathit{RSF}\ A\ B \\
& \mid \texttt{First} : \forall A\,B\,C.\,\mathit{RSF}\ A\ B \to \mathit{RSF}\ (A \times C)\ (B \times C) \\
& \mid \texttt{Comp} : \forall A\,B\,C.\,\mathit{RSF}\ A\ B \times \mathit{RSF}\ B\ C \to \mathit{RSF}\ A\ C \\
& \mid \texttt{Get} : \forall A\,B.\,\mathit{Ref}\ B \to \mathit{RSF}\ A\ (A \times B) \\
& \mid \texttt{Set} : \forall A\,B.\,\mathit{Ref}\ B \to \mathit{RSF}\ (A \times B)\ A
\end{aligned}$$

Les termes construits avec `Arr`, `First` et `Comp` sont similaires à leurs équivalents dans YAMPACORE. Les opérateurs `Get` et `Set` sont utilisés pour lire et écrire dans les ressources, respectivement. Plus formellement, le terme $\texttt{Get}\ r$ représente une fonction de signal qui prend une entrée x et retourne une paire (x, y), où y est la

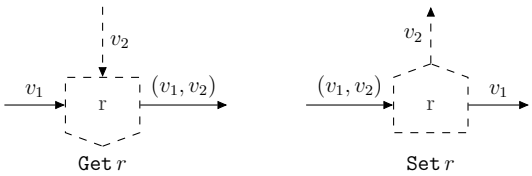


FIGURE 5.1 – Représentation graphique des accès aux ressources.

valeur de la ressource r. Le terme $\texttt{Set}\, r$ est le complémentaire de $\texttt{Get}\, r$, c'est-à-dire qu'il représente une fonction de signal qui prend une entrée (x, y) et retourne x, après avoir mis à jour la ressource r avec la valeur y. Une représentation graphique est fournie dans la figure 5.1.

Exemple 5.7. *Dans* MOLHOLES*, la fonction delay peut s'écrire :*

$$\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\, r, \texttt{Arr}\, \mathit{swap}), \texttt{Set}\, r) : \mathit{RSF}\ A\ A$$

où r : Ref A est une ressource interne dont la première valeur du flux de sortie sera la valeur initiale de r. Sa représentation graphique est donnée dans la figure 5.2, où nous supposons que $a : A$ est la valeur associée à r dans l'environnement.

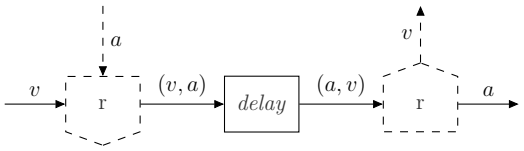


FIGURE 5.2 – Représentation graphique de la fonction *delay* dans MOLHOLES.

Nous adoptons un modèle de programmation dans lequel toutes les interactions avec l'environnement sont effectuées par l'intermédiaire de ressources. Par conséquent, les types d'entrée et de sortie des programmes sont limités au type singleton *Unit*, dont la seule valeur est dénotée par `tt`. Le concept de programme est énoncé dans la définition 5.4.

Définition 5.4 (Programme)**.** *Un programme, dans le langage* MOLHOLES*, se définit par un terme t de type RSF Unit Unit ainsi que par des fonctions qui associent ses ressources d'entrée et de sortie à des types et ses ressources internes à des valeurs initiales typées. Nous notons Prog le type d'un programme et nous*

le définissons comme suit :

$$\begin{array}{rcll} \mathit{Prog} & = & \{ & \texttt{inputs} : \mathit{List\ Type}\ ; \\ & & & \texttt{outputs} : \mathit{List\ Type}\ ; \\ & & & \texttt{internals} : \mathit{List\ Val}_{\mathit{Type}}\ ; \\ & & & \texttt{program} : \mathit{RSF\ Unit\ Unit}\ \} \end{array}$$

où Type représente les types du langage hôte et $\mathit{Val}_{\mathit{Type}}$ *représente les valeurs typées du langage hôte.*

Exemple 5.8. *Nous définissons le programme qui génère la séquence des entiers naturels ci-dessous.* r_0 *est une ressource de sortie et* r_1 *une ressource interne.*

$$\begin{array}{rcl} p_{\mathbb{N}} & = & \{ \\ & & \quad \texttt{inputs} = []; \\ & & \quad \texttt{outputs} = [\mathbb{N}]; \\ & & \quad \texttt{internals} = [(0 : \mathbb{N})]; \\ & & \quad \texttt{program} = (((\texttt{Get}\, r_1 \ggg \\ & & \qquad\qquad \texttt{Arr}\, (\lambda((), n).(((), n), n+1))) \ggg \\ & & \qquad\qquad \texttt{Set}\, r_1) \ggg \texttt{Set}\, r_0) \\ & & \} \end{array}$$

5.3.2 Mémoire et Monade d'état

Le domaine sémantique du langage utilise une monade d'état qui nécessite une mémoire particulière. En effet, le langage contient différents types de ressources ayant chacun leurs droits d'accès ce qui n'est pas gérable avec une monade d'état standard. Par conséquent, nous définissons une mémoire plus complexe qu'un simple état qui nous permet d'autoriser ou non la lecture ou l'écriture en fonction des types de ressources. Nous commençons par définir formellement la mémoire avant d'établir la monade d'état.

Mémoire

Une mémoire, portant le type *Memory*, est une fonction partielle dont le domaine est $\mathbb{N}$ et le codomaine est *Cell*. Le domaine représente les identifiants de ressources qui sont de simples entiers. Le codomaine, quant à lui, représente les *cellules mémoires*. Une cellule mémoire consiste en une paire composée d'un tag, de type *Tag*, et d'une valeur optionnelle du type portée par le tag. Un tag est, lui-même, une paire composée d'un état indiquant l'accessibilité de la ressource et du type de la valeur contenue par la ressource. L'état a trois formes :

— `Internal` : Il représente une ressource interne. La cellule est en lecture et écriture non bornées.

— `Input` b : Il représente une ressource d'entrée. La cellule est en lecture seule et ne peut être lue que si b est vraie.

— `Output` b : Il représente une ressource de sortie. La cellule est en écriture seule et ne peut être écrite que si b est vraie.

Les définitions des différents types énoncés sont regroupées dans la figure 5.3. Nous supposons qu'une mémoire partage les mêmes propriétés, fonctions et notations que les dictionnaires présentés dans la section 2.1.2.

$$
\begin{array}{lcl}
\mathit{Status} & = & \mid \texttt{Internal} : \mathit{Status} \\
 & & \mid \texttt{Input} : \mathit{Bool} \to \mathit{Status} \\
 & & \mid \texttt{Output} : \mathit{Bool} \to \mathit{Status} \\
A^{?} & = & A \mid \texttt{undef} \\
\mathit{Tag} & = & \mathit{Status} \times \mathit{Type} \\
\mathit{Cell} & = & \texttt{Cell} : \forall (t : \mathit{Tag}),\ (\mathit{snd}\, t)^{?} \to \mathit{Cell} \\
\mathit{Memory} & = & \mathbb{N} \rightharpoonup \mathit{Cell}
\end{array}
$$

FIGURE 5.3 – Définition du type *Memory*.

En plus de la fonction d'ajout récupérée de la définition des dictionnaires, nous proposons deux fonctions `read` et `write` qui permettent de lire et écrire dans une mémoire uniquement si les contraintes d'accessibilités des cellules sont satisfaites. Elles sont définies partiellement, c'est-à-dire que nous établissons que les cas où la fonction réussie, dans la figure 5.4.

La lecture d'une cellule mémoire n'est possible que si elle a l'état `Internal` ou `Input true`, c'est-à-dire si la ressource n'a aucune restriction ou qu'une lecture unique est disponible. Cette lecture n'impacte pas la mémoire quand la cellule a l'état `Internal`, mais elle la modifie quand la cellule a l'état `Input true`, car nous remplaçons le booléen associé à `Input` par `false` afin de rendre indisponible la cellule à toute autre lecture.

L'écriture dans une cellule mémoire est permise uniquement si elle a l'état `Internal` ou `Output true`. Comme pour la lecture, avoir l'état `Internal` indique qu'il n'y a aucune restriction sur son accès et, de la même manière que pour les cellules qui ont l'état `Input true`, les cellules qui ont l'état `Output true` se voient modifiées en `Output false` pour interdire une nouvelle écriture.

Remarque 5.5. *Nous avons besoin que le type de la ressource et le type de la cellule, stockée dans le tag, correspondent. Cette contrainte est commune aux deux fonctions et sera garantie via la sémantique statique du langage.*

$$
\begin{array}{lcll}
\texttt{read} & : & \mathit{Ref}\ A \to \mathit{Memory} \rightharpoonup A \times \mathit{Memory} & \\
\texttt{read}\, r\, \sigma & = & \texttt{match}\ \sigma\,(\texttt{id}\, r)\ \texttt{with} & \\
 & & \mid \texttt{Cell}\,(\texttt{Internal}, B)\, x \Rightarrow (x, \sigma) & \text{si } A = B \\
 & & \mid \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, B)\, x & \text{si } A = B \\
 & & \quad \Rightarrow (x, \sigma[\texttt{id}\, r \mapsto \texttt{Cell}\,(\texttt{Input}\,\texttt{false}, B)\,\texttt{undef}]) & \\
 & & & \\
\texttt{write} & : & \mathit{Ref}\ A \to \mathit{Memory} \to A \rightharpoonup A \times \mathit{Memory} & \\
\texttt{write}\, r\, \sigma\, v & = & \texttt{match}\ \sigma\,(\texttt{id}\, r)\ \texttt{with} & \\
 & & \mid \texttt{Cell}\,(\texttt{Internal}, B)\, x & \text{si } A = B \\
 & & \quad \Rightarrow (x, \sigma[\texttt{id}\, r \mapsto \texttt{Cell}\,(\texttt{Internal}, B)\, v]) & \\
 & & \mid \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, \texttt{undef})\, x & \text{si } A = B \\
 & & \quad \Rightarrow (x, \sigma[\texttt{id}\, r \mapsto \texttt{Cell}\,(\texttt{Output}\,\texttt{false}, B)\, v]) &
\end{array}
$$

$$
\begin{aligned}
\texttt{readable}\,(\texttt{Ref}\, A\, n)\, \sigma = (\exists v : A, \sigma\, n = \texttt{Cell}\,(\texttt{Internal}, A)\, v) \vee \\
(\exists v : A, \sigma\, n = \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, A)\, v) \\
\texttt{writable}\,(\texttt{Ref}\, A\, n)\, \sigma = (\exists v : A, \sigma\, n = \texttt{Cell}\,(\texttt{Internal}, A)\, v) \vee \\
(\sigma\, n = \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, A)\,\texttt{undef})
\end{aligned}
$$

FIGURE 5.4 – Fonctions et prédicats pour la mémoire.

Exemple 5.9. *Soient σ une mémoire, définie ci-dessous, ainsi que $r = \texttt{Ref}\,\mathbb{N}\,0$ et $r' = \texttt{Ref}\,\mathbb{N}\,1$ des ressources.*

$$\sigma = [0 \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\, 4][1 \mapsto \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, \mathbb{N})\, 9]$$

Nous pouvons effectuer les opérations de lecture et d'écriture suivantes :

$$
\begin{aligned}
\texttt{write}\, r\, \sigma\, 98 &= \sigma[\texttt{id}\, r \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\, 98] \\
\texttt{read}\, r\, \sigma &= (4, \sigma) \\
\texttt{read}\, r''\, \sigma &= (9, \sigma[\texttt{id}\, r' \mapsto \texttt{Cell}\,(\texttt{Input}\,\texttt{false}, \mathbb{N})\, 98])
\end{aligned}
$$

Il est impossible d'effectuer une écriture avec r', car sa cellule est en lecture seule. Si r était égal à `Ref` *Unit* 0 *alors il ne serait pas utilisable dans σ, car le type de la cellule associée à l'identifiant* 0 *est* $\mathbb{N}$ *alors que r serait de type Unit.*

Nous définissons deux prédicats `readable` et `writable` qui capturent respectivement l'accessibilité en lecture et en écriture d'une cellule en fonction de son état. Intuitivement, le prédicat `readable`, respectivement `writable`, correspond aux hypothèses minimales pour que la fonction `read`, respectivement `write`, pro-

duise un résultat. Nous exprimons ces deux prédicats dans la figure 5.4.

Exemple 5.10. *Soient σ une mémoire, définie ci-dessous, $r = \texttt{Ref}\,\mathbb{N}\,0$, $r' = \texttt{Ref}\,\mathbb{N}\,1$ et $r'' = \texttt{Ref}\,\mathit{Unit}\,3$ des ressources.*

$$\sigma = [0 \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\,4][3 \mapsto \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, \mathit{Unit})\,\texttt{undef}]$$

La seule ressource satisfaisant le prédicat `readable` *est r, car r' n'est pas présente dans σ et r'' représente une ressource de sortie pour σ. Pour le prédicat* `writable`*, les ressources r et r'' la satisfont.*

Comme dit précédemment, les prédicats `readable` et `writable` déterminent si une ressource est lisible ou modifiable. Par conséquent, la fonction de lecture, respectivement d'écriture, est totale pour une ressource $r : \mathit{Ref}\ A$ dans une mémoire σ si le prédicat $\texttt{readable}\,r\,\sigma$, respectivement $\texttt{writable}\,r\,\sigma$, est satisfait. Sous ces contraintes, nous définissons les équations (5.2a) à (5.2e) représentant les différentes interactions possibles entre les opérations de lecture et d'écriture. Ces équations portent l'ensemble d'équations (2.2) établissant les propriétés de l'ajout à nos deux nouvelles fonctions `read` et `write`.

$$\texttt{write}\,r\,\sigma\,v = \texttt{write}\,r\,(\texttt{write}\,r\,\sigma\,v')\,v \tag{5.2a}$$

$$\texttt{write}\,r\,\sigma\,v = \sigma \qquad \text{où } \texttt{read}\,r\,\sigma = (v, \sigma) \tag{5.2b}$$

$$\texttt{read}\,r\,(\texttt{write}\,r\,\sigma\,v) = (v, \sigma) \tag{5.2c}$$

$$\mathit{fst}\,(\texttt{read}\,r\,\sigma) = \mathit{fst}\,(\texttt{read}\,r\,(\mathit{snd}\,(\texttt{read}\,r'\,\sigma))) \quad \text{où } \texttt{id}\,r \neq \texttt{id}\,r' \tag{5.2d}$$

$$\texttt{write}\,r\,(\texttt{write}\,r'\,\sigma\,v')\,v = \texttt{write}\,r'\,(\texttt{write}\,r\,\sigma\,v)\,v' \qquad \text{où } \texttt{id}\,r \neq \texttt{id}\,r' \tag{5.2e}$$

L'équation (5.2a) représente le même comportement que l'équation (2.2a), c'est-à-dire que la seconde écriture masque la première. Les équations (5.2b) et (5.2c) sont deux comportements équivalent à l'équation (2.2c), c'est-à-dire que l'application des fonctions ne change pas la mémoire. Et finalement, les équations (5.2d) et (5.2e) affirment qu'il est possible de permuter deux écritures ou deux lectures comme l'équation (2.2b). Les équations (5.2a) à (5.2c) ne concernent que les ressources internes. Nous démontrons que cet ensemble d'équations est satisfait par nos définitions dans le théorème 5.3.

Lemme 5.3 (Lois des lectures et écritures)**.** *Les opérations* `read` *et* `write` *satisfont les équations* (5.2a) *à* (5.2e)*.*

Démonstration. La démonstration se fait directement par la simplification du but, c'est-à-dire en déroulant les définitions, pour chaque équation. □

Monade d'état

En s'appuyant sur le type *Memory*, nous définissons une monade d'état, nommée *St*, sur la catégorie du langage hôte $\mathcal{C}_{hst}$ et basée sur le foncteur **St** tel que :

$$\begin{aligned} \mathbf{St}\, A &= \mathit{Memory} \rightharpoonup A \times \mathit{Memory} \\ \mathbf{St}\, f &= \lambda h\, \sigma.(f\, a, \sigma') \qquad\qquad \text{où } (a, \sigma') = h\, \sigma \end{aligned}$$

où A est un type du langage hôte. Cette monade est équipée des opérations `return`, `bind`, `get` et `set` présentées dans la figure 5.5. Les deux premières sont les opérations standards des monades et satisfont les lois monadiques présentées dans les équations (2.9a) à (2.9c). L'opérations `get`, respectivement `set`, utilise directement la fonction `read`, respectivement `write`, qui est la fonction précédemment définie pour la lecture, respectivement l'écriture, dans la mémoire.

$$\begin{array}{lcllcl} \texttt{return}_A & :: & A \to \mathit{St}\, A & \texttt{bind}_{AB} & :: & \mathit{St}\, A \to (A \to \mathit{St}\, B) \to \mathit{St}\, B \\ \texttt{return}\, a & = & \lambda\sigma.(a, \sigma) & \texttt{bind}\, m\, f & = & \lambda\sigma.f\, x\, \sigma' \text{ où } (x, \sigma') = m\, \sigma \\ \\ \texttt{get}_A\, r & :: & \mathit{Ref}\, A \to \mathit{St}\, A & \texttt{set}_A\, r\, v & :: & \mathit{Ref}\, A \to A \to \mathit{St}\, \mathit{Unit} \\ \texttt{get}\, r & = & \texttt{read}\, r & \texttt{set}\, r\, v & = & \lambda\sigma.(\texttt{tt}, \texttt{write}\, r\, \sigma\, v) \end{array}$$

FIGURE 5.5 – Opérations sur la monade d'état *St*.

Remarque 5.6. *Les fonctions* `read` *et* `write` *sont totales si leur prédicat respectif est respecté. Par conséquent, l'opérations* `get` r*, respectivement* `set` $r\, v$*, est totale pour les mémoires satisfaisant le prédicat* `readable` r*, respectivement* `writable` r*.*

Nous étendons les lois monadiques avec l'ensemble d'équations (5.3). Elles établissent la relation entre les opérations `get` et `set` et portent les propriétés énoncées dans l'ensemble d'équations (5.2).

Les équations (5.3a) à (5.3d) concernent uniquement les ressources internes. La première équation établit qu'une lecture dont la valeur n'est pas gardée peut être oubliée. La deuxième équation définit que lire et écrire avec la même ressource consécutivement revient à ne rien changer. La troisième permet d'oublier une lecture s'il y a une écriture juste avant. Et enfin la quatrième équation est équivalente à l'équation (5.2a), c'est-à-dire que la dernière écriture masque les précédentes.

Les équations (5.3e) à (5.3h) concernent l'ensemble des ressources. Elles définissent la permutation entre deux lectures, deux écritures, une lecture et une écriture et inversement.

$$\texttt{bind}\,(\texttt{get}\,r)\,(\lambda x.m) = m \tag{5.3a}$$
$$\texttt{bind}\,(\texttt{get}\,r)\,(\texttt{set}\,r) = \texttt{return}\,\texttt{tt} \tag{5.3b}$$
$$\texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt}.\,\texttt{get}\,r) = \texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt}.\,\texttt{return}\,x) \tag{5.3c}$$
$$\texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt}.\texttt{set}\,r\,y) = \texttt{set}\,r\,y \tag{5.3d}$$

$$\texttt{bind}\,(\texttt{get}\,r)\,(\lambda x.\texttt{get}\,r') = \texttt{bind}\,(\texttt{get}\,r')\,(\lambda x.\texttt{bind}\,(\texttt{get}\,r)\,(\lambda y.\texttt{return}\,x)) \tag{5.3e}$$
$$\texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt}.\texttt{set}\,r'\,y) = \texttt{bind}\,(\texttt{set}\,r'\,y)\,(\lambda\texttt{tt}.\texttt{set}\,r\,x) \tag{5.3f}$$
$$\texttt{bind}\,(\texttt{get}\,r)\,(\lambda x.\texttt{set}\,r'\,y) = \texttt{bind}\,(\texttt{set}\,r'\,y)\,(\lambda\texttt{tt}.\texttt{bind}\,(\texttt{get}\,r)\,(\lambda x.\texttt{return}\,\texttt{tt})) \tag{5.3g}$$
$$\texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt}.\texttt{get}\,r') = \texttt{bind}\,(\texttt{get}\,r')\,(\lambda y.\texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt}.\texttt{return}\,y)) \tag{5.3h}$$

Exemple 5.11. *Il est possible de dériver de nombreux résultats en combinant ces lois. Par exemple, une directe application de l'équation* (5.3a) *permet de retirer des accès inutiles dans le cas d'une ressource interne.*

$$\texttt{bind}\,(\texttt{get}\,r)\,(\lambda x.\texttt{get}\,r') = \texttt{get}\,r'$$

Lorsque nous notons $f = g$ dans l'ensemble d'équations (5.3), en réalité nous exprimons l'égalité $f\,\sigma = g\,\sigma$, pour toute mémoire σ. De plus, nous supposons que les prédicats de lisibilité et de modification sont satisfaits pour les lectures et écritures présentent dans l'équation. La validité de ces équations pour notre monade d'état *St* est énoncée et démontrée dans le théorème 5.4.

Lemme 5.4 (Lois additionnelles)**.** *Le foncteur* **St** *équipé des opérations définies dans la figure 5.5 satisfait les lois monadiques énoncées dans les équations* (2.9a) *à* (2.9c) *ainsi que dans les lois additionnelles présentées dans l'ensemble d'équations (5.3).*

Démonstration. La démonstration se fait directement par la simplification du but, c'est-à-dire en déroulant les définitions, pour chaque équation. Nous prenons l'exemple de l'équation (5.3h). La partie gauche de l'équation se simplifie comme suit :

$$
\begin{array}{lll}
 & \texttt{bind}\,(\texttt{set}\,r\,x)\,(\lambda\texttt{tt.set}\,r'\,y) & \\
= & \lambda\sigma.(\lambda\texttt{tt.set}\,r'\,y)\,z\,\sigma' \text{ où } (z,\sigma') = (\texttt{set}\,r\,x)\,\sigma & \text{(déf.)} \\
= & \lambda\sigma.\texttt{set}\,r'\,y\,(\texttt{write}\,r\,\sigma\,x) & \text{(simp.)} \\
= & \lambda\sigma.(\texttt{tt}, \texttt{write}\,r'\,(\texttt{write}\,r\,\sigma\,x)\,y) & \text{(def.)}
\end{array}
$$

La partie droite va suivre la même simplification et nous donner l'équation suivante :

$$\texttt{bind}\,(\texttt{set}\,r'\,y)\,(\lambda\texttt{tt.set}\,r\,x) = \lambda\sigma.(\texttt{tt}, \texttt{write}\,r\,(\texttt{write}\,r'\,\sigma\,y)\,x)$$

Grâce à l'équation (5.2e), nous concluons la démonstration pour ce cas. □

5.3.3 Sémantique dynamique

La sémantique dynamique de MOLHOLES, tout comme celle de YAMPACORE, se divise en deux fonctions `step` et `run`. La première fonction est la fonction d'étape qui nous permet d'évaluer un instant logique et la seconde applique la première sur chaque élément d'un flux. La définition de la monade d'état nous permet d'établir le domaine sémantique des termes. Il est défini par le type :

$$rsf\ A\,B = A \to St\,B$$

Contrairement à la définition du domaine sémantique $sf\ A\,B$ des termes de YAMPACORE, la définition de $rsf\ A\,B$ n'est pas corécursive. Par conséquent, un sous-ensemble d'équivalence entre fonctions de signal peut être démontré sans utiliser les techniques de bisimulation. En effet, si nous prenons deux fonctions de signal rsf_1 et rsf_2 alors nous pouvons démontrer, grâce à l'extensionnalité fonctionnelle, qu'elles sont équivalentes pour une même mémoire initiale. Et donc, si la mémoire initiale est différente pour rsf_1 et rsf_2 alors la démonstration de leur équivalence devra passer par une utilisation de la bisimulation.

Avant de passer à la fonction d'étape, nous définissons la catégorie $\mathcal{C}_{rsf}$ comme une catégorie de Kleisli associée à la monade St. Cette catégorie a pour objet les types du langage hôte et pour morphismes les fonctions de signal de type $rsf\ A\,B$. La famille de morphismes d'identité est donnée par la famille d'opérations `return` et la composition est construite via l'opérateur `bind` comme suit : $g \circ f = \lambda x.\texttt{bind}\,(f\,x)\,g$. Dans cette catégorie, nous définissons les fonctions repré-

sentant les termes du langage.

$$\begin{array}{lcl}
\mathit{arr}\ f & = & \lambda x.\, \texttt{return}\,(f\ x) \\
\mathit{comp}\ \mathit{rsf}_1\ \mathit{rsf}_2 & = & \mathit{rsf}_2 \circ \mathit{rsf}_1 \\
\mathit{first}\ \mathit{rsf} & = & \lambda(x,c).\, \texttt{bind}\,(\mathit{rsf}\ x)\,(\lambda y.\, \texttt{return}\,(y,c)) \\
\mathit{get}\ r & = & \lambda x.\, \texttt{bind}\,(\texttt{get}\ r)\,(\lambda y.\, \texttt{return}\,(x,y)) \\
\mathit{set}\ r & = & \lambda(x,y).\, \texttt{bind}\,(\texttt{set}\ r\ y)\,(\lambda \texttt{tt}.\, \texttt{return}\ x)
\end{array}$$

Le domaine sémantique exprimé, nous définissons la fonction récursive `step` qui plonge les termes de type $\mathit{RSF}\ A\ B$ vers les fonctions de type $\mathit{rsf}\ A\ B$.

$$\begin{array}{lcl}
\texttt{step} & : & \mathit{RSF}\ A\ B \to \mathit{rsf}\ A\ B \\
\texttt{step}\,(\texttt{Arr}\ f) & = & \mathit{arr}\ f \\
\texttt{step}\,(\texttt{Comp}\,(t_1,t_2)) & = & \mathit{comp}\,(\texttt{step}\ t_1)\,(\texttt{step}\ t_2) \\
\texttt{step}\,(\texttt{First}\ t) & = & \mathit{first}\,(\texttt{step}\ t) \\
\texttt{step}\,(\texttt{Get}\ r) & = & \mathit{get}\ r \\
\texttt{step}\,(\texttt{Set}\ r) & = & \mathit{set}\ r
\end{array}$$

Exemple 5.12. *Soient* $t = \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\ r, \texttt{Arr}\ \mathit{swap}), \texttt{Set}\ r)$ *le terme représentant la fonction delay,* $r = \texttt{Ref}\ \mathbb{N}\ 0$ *une ressource de type* $\mathit{Ref}\ \mathbb{N}$ *et* $\sigma = [0 \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\ 15]$ *une mémoire.*

$$\begin{array}{rcl}
(\texttt{step}\ t)\ 9\ \sigma & = & (\mathit{comp}\,(\mathit{comp}\,(\mathit{get}\ r)(\mathit{arr}\ \mathit{swap}))\,(\mathit{set}\ r))\ 9\ \sigma \\
& = & (\mathit{set}\ r \circ \mathit{arr}\ \mathit{swap} \circ \mathit{get}\ r)\ 9\ \sigma \\
& = & (\mathit{set}\ r \circ \mathit{arr}\ \mathit{swap})\ (9,15)\ \sigma \\
& = & (\mathit{set}\ r)\ (15,9)\ \sigma \\
& = & (15, \sigma[r \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\ 9])
\end{array}$$

La lecture de la ressource r *dans la mémoire* σ *n'implique aucune modification de* σ*, car c'est une cellule étiquetée* `Internal`*.*

Nous pouvons maintenant établir entièrement la sémantique en définissant la fonction `run`. Elle prend un programme p : *Prog* et une mémoire initiale σ : *Memory* et produit une fonction agissant comme un transformateur de flux de type $\mathit{Stream}\,(\alpha\,(\texttt{inputs}\ p)) \to \mathit{Stream}\,(\alpha(\texttt{outputs}\ p))$, où α est la fonction qui convertit une liste en tuple tout en préservant son ordre (dans le cas de la liste vide la fonction retourne `tt`). Comme un programme MOLHOLES interagit avec l'environnement uniquement via les ressources, le type des valeurs contenues dans le flux d'entrée, respectivement le flux de sortie, correspondent au tuple associé à la liste des ressources d'entrée, respectivement de sortie, via la fonction α. Par

conséquent, le tuple d'entrée est utilisé pour initialiser la mémoire et le tuple de sortie résulte d'une extraction des valeurs de sortie de la mémoire après évaluation. La fonction `run` repose directement sur trois fonctions : `init`, `pull` et `push` (en plus de la fonction `step`). Pour tout programme p, nous définissons les fonctions $k\,p = |\texttt{internals}\,p|$, $k_{in}\,p = |\texttt{inputs}\,p|$ et $k_{out}\,p = |\texttt{outputs}\,p|$.

$$\begin{array}{lcl} \texttt{run}\,(p : \mathit{Prog}) & : & \mathit{Memory} \rightarrow \\ & & \mathit{Stream}\,(\alpha\,(\texttt{inputs}\,p)) \rightharpoonup \mathit{Stream}\,(\alpha\,(\texttt{outputs}\,p)) \\ \texttt{run}\,p\,\sigma\,(\texttt{Cons}\,i\,s) & = & \texttt{Cons}\,(\texttt{push}\,p\,\sigma')\,(\texttt{run}\,p\,\sigma'\,s) \\ & & \text{où } (\texttt{tt}, \sigma') = \texttt{step}\,(\texttt{program}\,p)\,\texttt{tt}\,(\texttt{pull}\,p\,\sigma\,i) \end{array}$$

La fonction `init`, définie ci-dessous, initialise la mémoire avec les valeurs des ressources internes stockées dans $(\texttt{internals}\,p)$. Par conséquent, la mémoire initiale ne contient pas de cellules pour l'ensemble des ressources du programme. Pour un programme p, nous initialisons la fonction comme suit $\texttt{run}\,p\,(\texttt{init}\,p)$.

$$\begin{array}{lcl} \texttt{init} & : & \mathit{Prog} \rightarrow \mathit{Memory} \\ \texttt{init}\,p\,n & = & \begin{cases} \texttt{Cell}\,(\texttt{Internal}, \tau)\,v & \text{si } k_{in}\,p + k_{out}\,p \leq n \\ & \text{et } n < k_{in}\,p + k_{out}\,p + k\,p \\ & \text{et } v : \tau = (\texttt{internals}\,p)_m \\ & \text{où } m = n - (k_{in}\,p + k_{out}\,p) \\ \texttt{undef} & \text{sinon} \end{cases} \end{array}$$

Ce rôle est attribué à la fonction `pull` définie ci-dessous. En effet, elle récupère le tuple i porté par le flux d'entrée et mets à jour les cellules pour les ressources d'entrées et de sorties. La valeur d'une ressource d'entrée est contenue dans i. La valeur d'une ressource de sorties est toujours `undef` à l'initialisation. Dans la mémoire de retour de `pull`, toutes les ressources sont lisibles ou modifiables.

$$\begin{array}{lcl} \texttt{pull} & : & \mathit{Prog} \rightarrow \mathit{Memory} \rightarrow (\mathit{Val}_{\mathit{Type}} \times \ldots) \times \mathit{Val}_{\mathit{Type}} \rightarrow \mathit{Memory} \\ \texttt{pull}\,p\,\sigma\,i\,n & = & \begin{cases} \texttt{Cell}\,(\texttt{Input true}, \tau)\,v & \text{si } n < k_{in}\,p, \\ & \text{et } \tau = (\texttt{inputs}\,p)_n \\ & \text{et } v : \tau = (\texttt{split}\,i\,(k_{in}\,p))_n \\ \texttt{Cell}\,(\texttt{Output true}, \tau)\,\texttt{undef} & \text{si } k_{in}\,p \leq n < k_{in}\,p + k_{out}\,p \\ & \text{et } \tau = (\texttt{outputs}\,p)_{n - k_{in}\,p} \\ \sigma\,n & \text{sinon} \end{cases} \end{array}$$

En support à la fonction `pull`, nous définissons la fonction `split`. Elle permet

de convertir un tuple en liste selon un entier naturel.

$$
\begin{array}{lcl}
\texttt{split}\, p\, n & : & (\mathit{Val}_{\mathit{Type}} \times \ldots) \to \mathbb{N} \to \mathit{List}\ \mathit{Val}_{\mathit{Type}} \\
\texttt{split}\, p\, n & & \begin{cases} [p] & \text{si } n = 0 \\ [] & \text{si } p = \texttt{tt} \\ (\texttt{split}\, p'\, m) +\!\!+ [v] & \text{si } n = m + 1 \text{ et } p = (p', v) \end{cases}
\end{array}
$$

Finalement, la fonction `push`, définie ci-dessous, récupère toutes les sorties utilisées durant l'évaluation et en produit un tuple. Ce tuple devient la nouvelle tête du flux de sortie.

$$
\begin{array}{lcl}
\texttt{push} & : & \mathit{Prog} \to \mathit{Memory} \to (\mathit{Val}_{\mathit{Type}} \times \ldots) \times \mathit{Val}_{\mathit{Type}} \\
\texttt{push}\, p\, \sigma & = & \alpha \left[v \;\middle|\; \begin{array}{l} \forall n, k_{in}\, p \leq n < k_{in}\, p + k_{out}\, p \text{ et} \\ \sigma\, n = \texttt{Cell}\, (\texttt{Output}\, \texttt{false}, \tau)\, v \end{array} \right]
\end{array}
$$

Remarque 5.7. *La fonction* `run` *est partielle, car elle dépend de la fonction* `step` *qui l'est aussi. Cependant, un programme bien typé garantit un bon comportement des interactions mémoires ce qui rend la fonction* `step` *totale et par extension la fonction* `run`*.*

Le bon comportement de la fonction `run` suppose un ordre précis pour la numérotation des ressources présentes dans le programme. Cela nous permet d'assurer l'identification unique de chaque ressource qui est utilisée lors des transformations. Soit p un programme :

— les identifiants des ressources d'entrées doivent avoir les plus petits identifiants, c'est-à-dire qu'ils sont compris entre 0 inclus et $k_{in}\, p$ exclus ;

— les identifiants des ressources de sorties suivent celle d'entrée, c'est-à-dire qu'ils sont compris entre $k_{in}\, p$ inclus et $(k_{in}\, p + k_{out}\, p)$ exclus ;

— les identifiants des ressources internes sont les plus grands identifiants, c'est-à-dire qu'ils sont compris entre $(k_{in}\, p + k_{out}\, p)$ inclus et $(k_{in}\, p + k_{out}\, p) + k$.

Exemple 5.13. *Reprenons le programme* $p_{\mathbb{N}}$ *générant la suite des entiers naturels de l'exemple 5.8. Nous avons* $k\, p = 1$, $k_{in}\, p = 0$ *et* $k_{out}\, p = 1$*. Dans la suite, nous notons* st_{Unit} *le flux composé uniquement de valeur* `tt`*. Pour une meilleure compréhension, nous échangeons* $(\texttt{init}\, p_{\mathbb{N}})$ *et* $\texttt{pull}\, p_{\mathbb{N}}\, \sigma\, \texttt{tt}$ *par une mémoire équivalente quand cela est possible.*

$$
\begin{aligned}
&\texttt{run}\, p_{\mathbb{N}}\, (\texttt{init}\, p_{\mathbb{N}})\, st_{\mathit{Unit}} \\
=\ &\texttt{run}\, p_{\mathbb{N}}\, ([0 \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{true}, \mathbb{N})\, \texttt{undef}] \\
&\qquad [1 \mapsto \texttt{Cell}\, (\texttt{Internal}, \mathbb{N})\, 0])\, (\texttt{Cons}\, \texttt{tt}\, st_{\mathit{Unit}}) \\
=\ &\texttt{Cons}\, (\texttt{push}\, p_{\mathbb{N}}\, \sigma')\, (\texttt{run}\, p_{\mathbb{N}}\, \sigma'\, st_{\mathit{Unit}}) \\
&\quad \textit{où}\ (\texttt{tt}, \sigma') = \texttt{step}\, (\texttt{program}\, p_{\mathbb{N}})\, \texttt{tt}\, (\texttt{pull}\, p_{\mathbb{N}}\, (\texttt{init}\, p_{\mathbb{N}})\, \texttt{tt}) \\
&\qquad\qquad = \texttt{step}\, (\texttt{program}\, p_{\mathbb{N}})\, \texttt{tt} \\
&\qquad\qquad\qquad ((\texttt{init}\, p_{\mathbb{N}})[0 \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{true}, \mathbb{N})\, \texttt{undef}]) \\
&\qquad\qquad = (\texttt{tt}, (\texttt{init}\, p_{\mathbb{N}})[0 \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{true}, \mathbb{N})\, \texttt{undef}] \\
&\qquad\qquad\qquad\qquad [1 \mapsto \texttt{Cell}\, (\texttt{Internal}, \mathbb{N})\, 1] \\
&\qquad\qquad\qquad\qquad [0 \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{false}, \mathbb{N})\, 0]) \\
=\ &\texttt{Cons}\, (\texttt{push}\, p_{\mathbb{N}}\, \sigma')\, (\texttt{run}\, p_{\mathbb{N}}\, \sigma'\, st_{\mathit{Unit}}) \\
&\quad \textit{où}\ \sigma' = (\texttt{init}\, p_{\mathbb{N}})[1 \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{true}, \mathbb{N})\, \texttt{undef}] \\
&\qquad\qquad\qquad [1 \mapsto \texttt{Cell}\, (\texttt{Internal}, \mathbb{N})\, 1] \\
&\qquad\qquad\qquad [0 \mapsto \texttt{Cell}\, (\texttt{Output}\, \texttt{false}, \mathbb{N})\, 0] \\
=\ &\texttt{Cons}\, (0 : \mathbb{N})\, (\texttt{run}\, p_{\mathbb{N}}\, \sigma'\, st_{\mathit{Unit}})
\end{aligned}
$$

Tout comme les fonctions de signal de YampaCore, les fonctions *arr*, *first* et *comp* dans Molholes satisfont les équations (2.10a) à (2.12c). Ce fait est énoncé et démontré dans le théorème 5.5.

Théorème 5.5. *Les fonctions arr, first et comp forment une flèche.*

Démonstration. La démonstration se fait par un simple raisonnement équationnel. Nous donnons la démonstration de l'équation (2.10d). Les autres cas sont similaires.

$$
\begin{aligned}
\mathit{arr}\, f \ggg \mathit{arr}\, g &= \lambda x.\texttt{bind}\, ((\mathit{arr}\, f)\, x)\, (\mathit{arr}\, g) && \text{(déf.)} \\
&= \lambda x.\lambda\sigma.((\mathit{arr}\, g)\, y\, \sigma') \text{ où } (y, \sigma') = (\mathit{arr}\, f)\, x\, \sigma && \text{(déf.)} \\
&= \lambda x.\lambda\sigma.((\mathit{arr}\, g)\, (f\, x)\, \sigma) && \text{(déf.)} \\
&= \lambda x.\lambda\sigma.(g\, (f\, x), \sigma) && \text{(déf.)} \\
&= \mathit{arr}\, (g \circ f) && \text{(déf.)}
\end{aligned}
$$

□

En plus des règles classiques des flèches, les fonctions *get* et *set* en apportent de nouvelles qui sont rassemblées dans l'ensemble d'équations (5.4). Ces équations

jouent un rôle similaire aux lois de la boucle dans YampaCore. Nous les séparons en trois groupes.

$$arr\ f \ggg get\ r = get\ r \ggg first\ arr\ f \tag{5.4a}$$
$$first\ (get\ r \ggg rsf) = get\ r \ggg arr\ perm \ggg first\ rsf \tag{5.4b}$$
$$first\ (rsf) \ggg get\ r = first\ (rsf \ggg get\ r) \ggg arr\ perm \tag{5.4c}$$
$$get\ r \ggg get\ r' = get\ r' \ggg get\ r \ggg arr\ perm \tag{5.4d}$$
$$get\ r \ggg get\ r = get\ r \ggg arr\ sdup \tag{5.4e}$$

$$set\ r \ggg arr\ f = first\ arr\ f \ggg set\ r \tag{5.4f}$$
$$first\ (rsf \ggg set\ r) = first\ rsf \ggg arr\ perm \ggg set\ r \tag{5.4g}$$
$$set\ r \ggg first\ (rsf) = arr\ perm \ggg first\ (set\ r \ggg rsf) \tag{5.4h}$$
$$set\ r \ggg set\ r' = arr\ perm \ggg set\ r' \ggg set\ r \tag{5.4i}$$
$$set\ r \ggg set\ r = arr\ fst \ggg set\ r \tag{5.4j}$$

$$set\ r \ggg get\ r = arr\ sdup \ggg set\ r \tag{5.4k}$$
$$get\ r \ggg set\ r = arr\ id \tag{5.4l}$$
$$set\ r \ggg get\ r' = get\ r' \ggg arr\ perm \ggg set\ r \tag{5.4m}$$

Le premier groupe, composé des équations (5.4a) à (5.4e), régit la réorganisation des opérations *get*, ce qui permet de les déplacer vers « la gauche » ou de les éliminer. De la même manière, le deuxième groupe, composé des équations (5.4f) à (5.4j), régit la réorganisation des opérations *set*, ce qui permet de les déplacer vers « la droite » ou de les éliminer. Finalement, le troisième et dernier groupe, est composé des équations (5.4k) à (5.4m), régit la réorganisation des opérations *get* et *set* consécutives. Les équations (5.4k) et (5.4l) ne sont valables que pour des ressources internes, alors que l'équation (5.4m) est valable dans tous les cas. Ces lois sont satisfaites par notre domaine sémantique, comme énoncé et démontré dans le théorème 5.6. Ils forment donc une flèche avec des ressources mimant des flèches avec une boucle.

Lemme 5.6. *Les équations* (5.4a) *à* (5.4m) *sont satisfaites pour tous types de ressource. Les équations* (5.4e) *et* (5.4j) *à* (5.4l) *sont satisfaites pour des ressources internes.*

Démonstration. La démonstration est faite par un simple raisonnement équationnel. Nous prenons l'exemple de l'équation (5.4a).

$$
\begin{array}{lll}
 & \mathit{get}\, r \ggg \mathit{first}\ \mathit{arr}\ f & \\
= & \lambda x.\mathtt{bind}\,(\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.\mathtt{return}\,(x,y)))\,(\mathit{first}\ \mathit{arr}\ f) & \text{(déf.)} \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.\mathtt{bind}\,(\mathtt{return}\,(x,y))\,(\mathit{first}\ \mathit{arr}\ f)) & \text{(éq. (2.9c))} \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.(\mathit{first}\ \mathit{arr}\ f)\,(x,y)) & \text{(éq. (2.9a))} \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.(\mathtt{bind}\,(\mathtt{return}\,(f\ x))\,(\lambda z.\mathtt{return}(z,y)))) & \text{(déf.)} \\
= & \lambda x.\mathtt{bind}\,(\mathtt{get}\, r)\,(\lambda y.(\mathtt{return}\,(f\ x,y))) & \text{(éq. (2.9a))} \\
= & \lambda x.\mathit{get}\, r\,(f\ x) & \text{(déf.)} \\
= & \lambda x.\mathtt{bind}\,(\mathtt{return}(f\ x))\ \mathit{get}\, r & \text{(éq. (2.9a))} \\
= & \mathit{arr}\ f \ggg \mathit{get}\, r & \text{(déf.)}
\end{array}
$$

□

5.3.4 Sémantique statique

La sémantique statique pour MOLHOLES assure une utilisation correcte des ressources. Spécifiquement, elle garantit que si un programme est bien typé alors chaque ressource d'entrée est lue exactement une fois et que chaque ressource de sortie est écrite exactement une fois. Avant de définir la sémantique, il faut établir le domaine abstrait sur lequel elle repose. Il consiste en une mémoire abstraite, définie comme une fonction partielle, qui associe des identifiants de ressources à des cellules mémoires abstraites.

$$\mathit{Memory}^{\dagger} = \mathbb{N} \rightharpoonup \mathit{Tag}$$

Ces cellules mémoires sont simplement définies comme des tags, donc de type Tag. Ces mémoires sont des abstractions des mémoires « concrètes » de type Memory. L'ensemble des mémoires concrètes représentées par une mémoire abstraite Σ est définie par $\gamma\,\Sigma$, où γ est la fonction suivante :

$$
\begin{array}{c}
\gamma\,(st,\tau) = \{\mathtt{Cell}\,(st,\tau)\,v \mid \text{si } st \in \mathtt{defined} \text{ alors } v \text{ est de type } \tau \\
\text{sinon } v = \mathtt{undef}\} \\
\text{où } \mathtt{defined} = \{\mathtt{Internal}, \mathtt{Input}\,\mathtt{true}, \mathtt{Output}\,\mathtt{false}\} \\
\gamma\,\Sigma = \{\sigma \mid \mathtt{dom}(\sigma) = \mathtt{dom}(\Sigma) \wedge \forall n.n \in \mathtt{dom}(\sigma) \rightarrow \sigma\, n \in \gamma\,(\Sigma\, n)\}
\end{array}
$$

Cet ensemble contient toutes les mémoires concrètes qui partagent le même domaine que Σ et dont toutes les cellules concrètes sont en relation avec leurs équivalents abstraits dans Σ. De plus, la relation entre cellules mémoires concrètes et abstraites impose que la valeur contenue dans la cellule concrète soit compatible

avec le type du tag quand nous sommes dans un cas où il y a une valeur dedans.

Exemple 5.14. *Soit Σ une mémoire abstraite définie comme suit :*

$$\begin{aligned}\Sigma =&[0 \mapsto (\texttt{Internal}, \mathbb{N})] \\ &[3 \mapsto (\texttt{Output}\,\texttt{true}, \mathbb{N})][2 \mapsto (\texttt{Input}\,\texttt{true}, \mathit{Unit})]\end{aligned}$$

Nous notons σ un modèle de Σ, présent dans $\gamma\,\Sigma$, tel que :

$$\begin{aligned}\sigma =&[0 \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\,12] \\ &[3 \mapsto \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, \mathbb{N})\,\texttt{undef}][2 \mapsto \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, \mathit{Unit})\,\texttt{tt}]\end{aligned}$$

Voici un autre un modèle de Σ, noté σ', présent dans $\gamma\,\Sigma$, tel que :

$$\begin{aligned}\sigma' =&[0 \mapsto \texttt{Cell}\,(\texttt{Internal}, \mathbb{N})\,2] \\ &[3 \mapsto \texttt{Cell}\,(\texttt{Output}\,\texttt{true}, \mathbb{N})\,\texttt{undef}][2 \mapsto \texttt{Cell}\,(\texttt{Input}\,\texttt{true}, \mathit{Unit})\,\texttt{undef}]\end{aligned}$$

Il est possible d'avoir une ressource d'entrée disponible, mais non initialisée.

Afin de porter les conséquences des écritures et lectures dans une mémoire concrète à la mémoire abstraite qui lui correspond, nous définissons les opérations de lecture et d'écriture abstraites ainsi que les équivalents abstraits des prédicats d'accessibilités dans la figure 5.6. Par ailleurs, nous établissons et démontrons dans le théorème 5.7 que les fonctions d'écriture et de lecture, concrètes et abstraites, préservent la relation γ qui existe entre la mémoire concrète et la mémoire abstraite d'entrée aux mémoires de sorties. Dans la suite, nous notons $r \vdash \Sigma$ si le type du tag associé à la ressource r dans la mémoire abstraite Σ correspond au type de r.

Lemme 5.7. *Soient r une ressource de type Ref A, σ une mémoire et Σ une mémoire abstraite telle que $r \vdash \Sigma$:*

- *Si $\sigma \in \gamma\,\Sigma$ et $\texttt{read}^\dagger\,r\,\Sigma = \Sigma'$ alors il existe une valeur v de type A et une mémoire σ' telle que $\texttt{read}\,r\,\sigma = (v, \sigma')$, $\sigma' \in \gamma\,\Sigma'$ et $r \vdash \Sigma'$. De plus, pour toute ressource r' telle que $r' \neq r$ nous avons $\Sigma\,r' = \Sigma'\,r'$.*
- *Si $\sigma \in \gamma\,\Sigma$ et $\texttt{write}^\dagger\,r\,\Sigma = \Sigma'$ alors, pour toute valeur v de type A, il existe une mémoire σ' telle que $\texttt{write}\,r\,\sigma\,v = \sigma'$ avec $\sigma' \in \gamma\,\Sigma'$ et $r \vdash \Sigma'$. De plus, pour toute ressource r' telle que $r' \neq r$ nous avons $\Sigma\,r' = \Sigma'\,r'$.*

Démonstration. La démonstration est immédiate en déroulant les définitions de $\texttt{read}$, $\texttt{write}$, $\texttt{read}^\dagger$ et $\texttt{write}^\dagger$. □

$$
\begin{array}{lcl}
\texttt{read}^\dagger & : & \mathit{Ref}\ A \to \mathit{Memory}^\dagger \rightharpoonup \mathit{Memory}^\dagger \\
\texttt{read}^\dagger\, r\, \Sigma & = & \texttt{match}\, \Sigma\, (\texttt{id}\, r)\, \texttt{with} \\
 & & \mid (\texttt{Internal}, B) \Rightarrow \Sigma \\
 & & \mid (\texttt{Input}\, \texttt{true}, B) \Rightarrow \Sigma\, [\texttt{id}\, r \mapsto (\texttt{Input}\, \texttt{false}, B)] \\
 & & \\
\texttt{write}^\dagger & : & \mathit{Ref}\ A \to \mathit{Memory}^\dagger \rightharpoonup \mathit{Memory}^\dagger \\
\texttt{write}^\dagger\, r\, \Sigma & = & \texttt{match}\, \Sigma\, (\texttt{id}\, r)\, \texttt{with} \\
 & & \mid (\texttt{Internal}, B) \Rightarrow \Sigma \\
 & & \mid (\texttt{Output}\, \texttt{true}, B) \Rightarrow \Sigma[\texttt{id}\, r \mapsto (\texttt{Output}\, \texttt{false}, B)] \\
 & & \\
\texttt{readable}^\dagger\, (\texttt{Ref}\, A\, n)\, \Sigma & = & \Sigma\, n = (\texttt{Internal}, A) \vee \Sigma\, n = (\texttt{Input}\, \texttt{true}, A) \\
\texttt{writable}^\dagger\, (\texttt{Ref}\, A\, n)\, \Sigma & = & \Sigma\, n = (\texttt{Internal}, A) \vee \Sigma\, n = (\texttt{Output}\, \texttt{true}, A)
\end{array}
$$

FIGURE 5.6 – Fonctions et prédicats pour la mémoire abstraite.

Remarque 5.8. *Comme pour les définitions de* `write` *et* `read`*, voir la figure 5.4, le type de la valeur contenue dans la cellule mémoire peut différer du type des ressources. Cependant, nous pouvons garantir une cohérence entre ces types via* $\texttt{readable}^\dagger$ *et* $\texttt{writable}^\dagger$*.*

La sémantique abstraite d'un terme est définie ci-dessous comme une fonction partielle $\texttt{step}^\dagger$ sur les mémoires abstraites. Elle représente la version abstraite de la fonction `step`. Pour les termes de la forme $\texttt{Arr}\, f$, la mémoire abstraite reste inchangée. Pour les termes de la forme $\texttt{Get}\, r$ et $\texttt{Set}\, r$, la sémantique abstraite met à jour l'état des cellules de la mémoire en utilisant les opérations $\texttt{read}^\dagger$ et $\texttt{write}^\dagger$. Finalement, les termes de la forme `First` ou `Comp`, les changements dans la mémoire abstraite se propagent.

$$
\begin{array}{lcll}
\texttt{step}^\dagger & : & & \mathit{RSF} \to \mathit{Memory}^\dagger \rightharpoonup \mathit{Memory}^\dagger \\
\texttt{step}^\dagger\, (\texttt{Arr}\, f)\, \Sigma & = & \Sigma & \\
\texttt{step}^\dagger\, (\texttt{First}\, t)\, \Sigma & = & \Sigma' & \text{si } \texttt{step}^\dagger\, t\, \Sigma = \Sigma' \\
\texttt{step}^\dagger\, (\texttt{Comp}\, (t_1, t_2))\, \Sigma & = & \Sigma' & \text{si } \texttt{step}^\dagger\, t_1\, \Sigma = \Sigma'' \wedge \texttt{step}^\dagger\, t_2\, \Sigma'' = \Sigma' \\
\texttt{step}^\dagger\, (\texttt{Get}\, r)\, \Sigma & = & \Sigma' & \text{si } \texttt{readable}^\dagger\, r\, \Sigma \wedge \texttt{read}^\dagger\, r\, \Sigma = \Sigma' \\
\texttt{step}^\dagger\, (\texttt{Set}\, r)\, \Sigma & = & \Sigma' & \text{si } \texttt{writable}^\dagger\, r\, \Sigma \wedge \texttt{write}^\dagger\, r\, \Sigma = \Sigma'
\end{array}
$$

La propriété de bon typage est établie dans la définition 5.5 et repose sur une fonction $\texttt{init}^\dagger$ que nous énonçons ci-dessous. Intuitivement, un programme est bien typé si $\texttt{step}^\dagger$ retourne une valeur et donc que toutes les vérifications d'accès ont été satisfaites. De plus, nous imposons aussi une utilisation totale des entrées et des sorties. La condition relative aux ressources de sortie est obligatoire

pour garantir la productivité du programme à chaque étape. La condition sur les ressources d'entrée est facultative, mais utile pour s'assurer que le programme consomme toutes ses entrées.

$$
\begin{array}{lcl}
\texttt{init}^\dagger & : & \mathit{Prog} \to \mathit{Memory}^\dagger \\
\texttt{init}^\dagger\, p\, n & = & \begin{cases}
(\texttt{Input}\,\texttt{true}, \tau) & \text{si } n < k_{in}\, p \text{ et } \tau = (\texttt{inputs}\, p)_n \\
(\texttt{Output}\,\texttt{true}, \tau) & \text{si } k_{in}\, p \leq n < k_{in}\, p + k_{out}\, p \\
 & \text{et } \tau = (\texttt{outputs}\, p)_{n-(k_{in}\, p)} \\
(\texttt{Internal}, \tau) & \text{si } k_{in}\, p + k_{out}\, p \leq n < k_{in}\, p + k_{out}\, p + k\, p \\
 & \text{et } (v : \tau) = (\texttt{internals}\, p)_m \\
 & \text{et } m = n - (k_{in}\, p + k_{out}\, p) \\
\text{undefined} & \text{sinon}
\end{cases}
\end{array}
$$

La fonction $\texttt{init}^\dagger$, n'est pas exactement la fonction abstraite équivalente à `init`. En effet, en plus d'initialiser les ressources internes, elle initialise les ressources d'entrée et de sortie afin qu'elles soient respectivement lisible et modifiable. Une version abstraite de `pull` n'est pas nécessaire, car nous n'avons pas de valeur à attribuer dans les cellules abstraites.

Définition 5.5 (Bien typé)**.** *Un programme p est bien typé s'il existe une mémoire abstraite Σ telle que $\Sigma = \texttt{step}^\dagger\,(\texttt{program}\, p)\,(\texttt{init}^\dagger\, p)$ et que les conditions suivantes sont réunies :*

— *Pour tout entier naturel n, tel que $n < k_{in}\, p$, la cellule associée à n est utilisée, c'est-à-dire que nous avons $\Sigma\, n = (\texttt{Input}\,\texttt{false}, \cdot)$*

— *Pour tout entier naturel n, tel que $k_{in}\, p \leq n < k_{in}\, p + k_{out}\, p$, nous avons $\Sigma\, n = (\texttt{Output}\,\texttt{false}, \cdot)$*

Exemple 5.15. *Nous démontrons que le programme $p_{\mathbb{N}}$, défini dans l'exemple 5.8, est bien typé. Nous commençons par définir la mémoire abstraite Σ, ci-dessous, en raisonnement sur $\texttt{step}^\dagger\,(\texttt{program}\, p_{\mathbb{N}})\,(\texttt{init}^\dagger\, p_{\mathbb{N}})$ avec $k\, p = 1$, $k_{in}\, p = 0$ et $k_{out}\, p = 1$. Σ étant définie, nous devons satisfaire les deux conditions sur l'état de fin des entrées et sorties. Dans notre cas, il y a qu'une sortie et elle est bien utilisée.*

$$
\begin{aligned}
\Sigma &= \mathtt{step}^\dagger\,(\mathtt{program}\,p_{\mathbb{N}})\,(\mathtt{init}^\dagger\,p_{\mathbb{N}}) \\
&= \mathtt{step}^\dagger\,(\mathtt{program}\,p_{\mathbb{N}})\,([1 \mapsto (\mathtt{Internal}\,\mathbb{N})][0 \mapsto (\mathtt{Output}\,\mathtt{true}\,\mathbb{N})]) \\
&= [0 \mapsto (\mathtt{Internal}\,\mathbb{N})] \\
&\qquad [0 \mapsto (\mathtt{Output}\,\mathtt{true}\,\mathbb{N})][0 \mapsto (\mathtt{Output}\,\mathtt{false}\,\mathbb{N})] \\
&\quad \textit{où } \mathtt{readable}^\dagger\,1\,([1 \mapsto (\mathtt{Internal}\,\mathbb{N})][0 \mapsto (\mathtt{Output}\,\mathtt{true}\,\mathbb{N})])\ \textit{tient} \\
&\quad \textit{où } \mathtt{writable}^\dagger\,1\,([1 \mapsto (\mathtt{Internal}\,\mathbb{N})][0 \mapsto (\mathtt{Output}\,\mathtt{true}\,\mathbb{N})])\ \textit{tient} \\
&\quad \textit{où } \mathtt{writable}^\dagger\,0\,([1 \mapsto (\mathtt{Internal}\,\mathbb{N})][0 \mapsto (\mathtt{Output}\,\mathtt{true}\,\mathbb{N})])\ \textit{tient}
\end{aligned}
$$

Grâce à la sémantique statique, il est possible d'établir les propriétés de progression, de correction de typage et de réactivité du langage MOLHOLES. Le théorème 5.8 énonce que pour tout terme si la sémantique d'étape abstraite produit un résultat alors la sémantique d'étape ne peut pas bloquer et doit donc produire une valeur, ce qui résume la propriété de progression.

Théorème 5.8. *Soient t : RSF $A\,B$ un terme, où A et B sont deux types du langage hôte, σ une mémoire et $a : A$. Pour toute mémoire abstraite Σ, telle que $\sigma \in \gamma\,\Sigma$, s'il existe Σ', telle que $\Sigma' = \mathtt{step}^\dagger\,t\,\Sigma$, alors il existe σ' telle que :*

- $\mathtt{step}\,t\,\sigma\,a = (b, \sigma')$*, pour un certain* $b : B$
- $\sigma' \in \gamma\,\Sigma'$

Démonstration. La démonstration se fait par induction structurelle sur t.

1. $\mathtt{step}\,(\mathtt{Arr}\,f)\,\sigma\,a = (f\,a, \sigma)$ et $\mathtt{step}^\dagger\,(\mathtt{Arr}\,f)\,\Sigma = \Sigma$. Par hypothèse, nous garantissons que $\sigma \in \gamma\,\Sigma$.
2. Par hypothèse d'induction, nous avons $a = (a', c)$, $\mathtt{step}\,(t)\,\sigma\,a' = (b, \sigma')$ et $\sigma' \in \gamma\,\Sigma'$. Il existe donc (b, c) et σ' tels que :

$$\mathtt{step}\,(\mathtt{First}\,t)\,\sigma\,(a', c) = ((b, c), \sigma') \text{ et } \sigma' \in \gamma\,\Sigma'$$

3. Par hypothèse d'induction, nous avons :

$$\mathtt{step}\,(t_1)\,\sigma\,a = (b, \sigma') \text{ et } \sigma' \in \gamma\,\Sigma'$$
$$\mathtt{step}\,(t_2)\,\sigma'\,b = (c, \sigma'') \text{ et } \sigma'' \in \gamma\,\Sigma''$$

   Par conséquent, il existe σ'' et c tels que $\mathtt{step}\,(\mathtt{Comp}\,(t_1, t_2))\,\sigma\,a = (c, \sigma'')$ et $\sigma'' \in \gamma\,\Sigma''$.
4. Par hypothèse, nous savons que $\mathtt{step}^\dagger\,(\mathtt{Get}\,r)\,\Sigma = \mathtt{read}^\dagger\,r\,\Sigma$. Via le théorème 5.7, nous savons qu'il existe une mémoire σ' et une valeur v telle que

$\texttt{read}\, r\, \sigma = (v, \sigma')$ et $\sigma' \in \gamma\,(\texttt{read}^\dagger\, r\, \Sigma)$. Par conséquent, il existe σ' et (a, v) telle que :

$$\texttt{step}\,(\texttt{Get}\, r)\, \sigma\, a = ((a, v), \sigma') \text{ et } \sigma' \in \gamma\,(\texttt{read}^\dagger\, r\, \Sigma)$$

5. Même logique que pour le cas précédent, mais pour le terme $\texttt{Set}\, r$.

□

Le théorème 5.9 énonce que pour tout programme bien typé et entrée du bon type, la sémantique dynamique ne peut pas échouer.

Théorème 5.9. *Soient p un programme bien typé, i un tuple d'entrée de type $(\alpha\,(\texttt{inputs}\, p))$ et σ une mémoire telle que $\texttt{pull}\, p\, \sigma\, i \in \gamma\,(\texttt{init}^\dagger\, p)$. Il existe une mémoire σ' telle que :*

— $\texttt{step}\,(\texttt{program}\, p)\, \texttt{tt}\,(\texttt{pull}\, p\, \sigma\, i) = (\texttt{tt}, \sigma')$

— $\texttt{push}\, p\, \sigma' : (\alpha\,(\texttt{outputs}\, p))$

— $\texttt{pull}\, p\, \sigma'\, i' \in \gamma\,(\texttt{init}^\dagger\, p)$, *pour tout i' de type $(\alpha\,(\texttt{inputs}\, p))$*

Démonstration. D'après la définition 5.5 et le théorème 5.8, il existe une mémoire σ' et une valeur v : *Unit* telle que :

$$\texttt{step}\,(\texttt{program}\, p)\, \texttt{tt}\,(\texttt{pull}\, p\, \sigma\, i) = (v, \sigma')$$

où $\sigma' \in \gamma\, \Sigma$ et $\Sigma = \texttt{step}^\dagger\,(\texttt{program}\, p)\,(\texttt{init}^\dagger\, p)$. v se réduit en $\texttt{tt}$, ce qui nous donne la forme attendue. Maintenant, sachant que le programme est bien typé, nous sommes sûr que toutes les ressources de sortie ont été utilisées. Cela implique que le tuple $\texttt{push}\, p\, \sigma'$ aura exactement le type $(\alpha\,(\texttt{outputs}\, p))$, car toute cellule en écriture aura le tag $(\texttt{Output}\, false)$. Finalement, via le théorème 5.7, nous savons que σ' préserve le domaine de σ il n'y a donc pas de nouveaux identifiants de ressources. De plus, les ressources internes sont toujours dans un état qui satisfait $\texttt{init}^\dagger$ laissant uniquement le problème des ressources d'entrées et de sorties. Sachant que $\texttt{pull}$ initialise ces derniers, nous établissons que $\texttt{pull}\, p\, \sigma'\, i' \in \gamma\,(\texttt{init}^\dagger\, p)$, pour tout i' de type $(\alpha\,(\texttt{inputs}\, p))$. □

Le théorème 5.10 est une conséquence directe du théorème 5.9. Il établit que la fonction $\texttt{run}$ ne peut pas échouer, c'est-à-dire que le programme produit à chaque instant une valeur de sortie pour une valeur d'entrée.

Corollaire 5.10. *Soit p un programme bien typé, pour tout flux d'entrée is de type Stream $(\alpha\,(\texttt{inputs}\, p))$, il existe un flux de sortie os de type Stream $(\alpha\,(\texttt{outputs}\, p))$ tel que :*

$$\texttt{run}\, p\,(\texttt{init}\, p)\, is = os$$

Démonstration. La démonstration se fait par bisimulation sur le flux d'entrée is et grâce au théorème 5.9. □

5.4 Transformations de programmes

La transformation de programme MOLHOLES vers YAMPACORE ne se fait pas directement sur n'importe quel programme. En effet, nous nous restreignons à la forme normale de chacun des langages afin de simplifier le raisonnement. Dans le cas de YAMPACORE, une définition de forme normale existe déjà. Intuitivement, tout programme YAMPACORE peut se réduire en une boucle réactive avec une fonction f élevée au rang de fonction de signal via l'opérateur *arr* en sous terme.

Exemple 5.16. *Soit t un programme* YAMPACORE*, défini ci-dessous. Ce programme additionne un entier initialisé à 0 à la valeur du flux et incrémente une fois sur deux cette valeur.*

$$
\begin{aligned}
t = \texttt{Loop}\,(0,&\\
&\texttt{Loop}\,(\texttt{true},\\
&\quad \texttt{Comp}\,(\texttt{First}\,(\texttt{Arr}\,(\lambda(x,y).(x+y,y))),\\
&\qquad \texttt{Arr}\,(\lambda((x,y),z).\mathit{if}\ z\ \mathit{then}\ ((x,y+1),\texttt{false})\\
&\qquad\qquad \mathit{else}\ ((x,y),\texttt{true})))))
\end{aligned}
$$

Sémantiquement, il est équivalent au programme t' suivant :

$$
\begin{aligned}
t' = \texttt{Loop}\,((0,\texttt{true}),&\\
&\texttt{Arr}\,(\lambda(x,(y,z)).\mathit{if}\ z\ \mathit{then}\ ((x+y,y+1),\texttt{false})\\
&\qquad \mathit{else}\ ((x+y,y),\texttt{true})))
\end{aligned}
$$

t' est ce que nous appellerons la forme normale de t dans la suite et ce que nous utiliserons dans la transformation de programmes.

Dans le cas de MOLHOLES, nous définissons une forme normale. Intuitivement, un programme MOLHOLES peut se simplifier en une fonction f élevée au rang de fonction de signal via l'opérateur *arr* et entourée par deux niveaux de lecture/écriture : les internes et les externes. Cette forme normale permet d'extraire les effets à « l'extérieur » du programme.

Exemple 5.17. *Soit p le programme défini ci-dessous. Ce programme* MOLHOLES *a un comportement similaire au programme présenté dans l'exemple précédent. Cette fois-ci la valeur d'entrée est récupérée via la ressource* 0 *qui correspond*

à l'unique ressource d'entrée et sa valeur est renvoyée via la ressource de sortie 3*, les ressources internes* 1 *et* 2 *permettent respectivement de récupérer l'entier naturel* 0 *et le booléen* `true`*. Le terme* `Second` *est le dual du* `First` *et peut être défini comme suit* $\texttt{Second}\, t = \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Arr}\, \mathit{swap}, \texttt{First}\, t), \texttt{Arr}\, \mathit{swap})$*.*

$$
\begin{aligned}
p \;=\; \{\;& \texttt{inputs} = [\mathbb{N}];\\
& \texttt{outputs} = [\mathbb{N}];\\
& \texttt{internals} = [(0 : \mathbb{N}), (\texttt{true} : \mathit{Bool})];\\
& \texttt{program} = \texttt{Get}\, 2 \ggg \texttt{Second}\,(\\
&\qquad \texttt{Get}\, 0 \ggg \texttt{Arr}\,(\lambda(x, y).(x, x + y)) \ggg \texttt{Set}\, 1 \ggg\\
&\qquad \texttt{Get}\, 3 \ggg\\
&\qquad \texttt{Arr}\,(\lambda(x, y).\mathit{if}\ y\ \mathit{then}\ (x + 1), \texttt{false}\ \mathit{else}\ (x, \texttt{true})) \ggg\\
&\qquad \texttt{Set}\, 3) \ggg \texttt{Set}\, 2\}
\end{aligned}
$$

Ce programme admet une forme normale p'*, définie ci-dessous. Nous pouvons constater que tous les appels aux ressources se font au début et à la fin du programme, laissant le cœur du programme totalement sans effets de bords.*

$$
\begin{aligned}
p' \;=\; \{\;& \texttt{inputs} = [\mathbb{N}];\\
& \texttt{outputs} = [\mathbb{N}];\\
& \texttt{internals} = [(0 : \mathbb{N}), (\texttt{true} : \mathit{Bool})];\\
& \texttt{program} = \texttt{Get}\, 0 \ggg \texttt{Get}\, 2 \ggg \texttt{Get}\, 3 \ggg\\
&\qquad \texttt{Arr}\,(\lambda(((\texttt{tt}, x), y), z).\\
&\qquad\quad \mathit{if}\ z\ \mathit{then}\ (((\texttt{tt}, x + y), y + 1), \texttt{false})\\
&\qquad\qquad \mathit{else}\ (((\texttt{tt}, x + y), y), \texttt{true})) \ggg\\
&\qquad \texttt{Set}\, 3 \ggg \texttt{Set}\, 2 \ggg \texttt{Set}\, 1\}
\end{aligned}
$$

Finalement, la transformation de l'un à l'autre revient à garder la fonction de signal interne, c'est-à-dire le terme $\mathit{arr}\, f$, pour une certaine fonction f, à remplacer les boucles dans YampaCore par des ressources internes et inversement et enfin à faire correspondre le tuple du flux d'entrée dans un programme YampaCore avec les ressources globales dans un programme Molholes.

Exemple 5.18. *Le programme* YampaCore t *et le programme* Molholes p *sont équivalents sémantiquement. Leurs formes normales respectives, c'est-à-dire* t' *et* p'*, partagent quasiment la même fonction principale et les ressources locales de* p' *correspondent aux boucles de* t'*. La seule différence entre les deux fonctions principales est la présence de la valeur* `tt` *dans le tuple d'entrée et de sortie de la fonction principale de* p'*.*

Dans cette section, nous démontrons une équivalence sémantique entre les

programmes MOLHOLES et YAMPACORE. Nous commençons par définir formellement la forme normale pour le langage YAMPACORE et nous démontrons que tout programme est équivalent à une forme normale par bisimulation. Ensuite, nous définissons la notion de forme normale pour un programme MOLHOLES et nous établissons que tout programme est équivalent à une forme normale. Enfin, nous exprimons et démontrons l'existence d'une transformation de MOLHOLES vers YAMPACORE.

5.4.1 Forme normale dans YAMPACORE

La forme normale d'un terme pour le langage YAMPACORE est donnée dans la définition ci-dessous. Intuitivement, un terme en forme normale est une fonction entourée d'une unique boucle. La boucle stocke les valeurs locales et l'entièreté des modifications appliquées au flux d'entrée se réduisent à une unique fonction élevée en fonction de signal via *arr*. Dans la suite, nous construisons l'équivalence nécessaire afin d'exprimer le fait que tout terme a une forme normale.

Définition 5.6 (Forme normale)**.** *Soient t une expression, v une valeur du langage hôte et f une fonction du langage hôte. L'expression t est en* forme normale *si et seulement si :*

$$t = \texttt{Loop}\,(v, (\texttt{Arr}\,f))$$

Nous avons déjà défini la bisimulation entre fonction de signal dans la définition 5.3. Nous l'utilisons dans la définition qui suit afin d'établir une relation $\equiv$ qui nous permet de raisonner sur une équivalence entre termes.

Définition 5.7 (Equivalence de termes)**.** *Soit $\equiv$ une relation sur les termes telle que $t_1 \equiv t_2$ si et seulement si $\texttt{step}\,t_1 \sim \texttt{step}\,t_2$, où t_1 et t_2 sont deux termes.*

Un résultat direct de cette équivalence est le théorème 5.11 qui stipule que $\equiv$ est une congruence sur les termes. Nous pouvons maintenant définir et démontrer le théorème 5.12 qui énonce l'existence d'une forme normale pour tout programme via la relation $\equiv$, et donc par bisimulation.

Lemme 5.11. *Pour tous termes t, t', t_1, t'_1, t_2, t'_2 et valeur du langage hôte v, les propriétés suivantes sont respectées :*

- *si $t_1 \equiv t'_1$ et $t_2 \equiv t'_2$ alors $\texttt{Comp}\,(t_1, t_2) \equiv \texttt{Comp}\,(t'_1, t'_2)$;*
- *si $t \equiv t'$ alors $\texttt{First}\,t \equiv \texttt{First}\,t'$;*
- *si $t \equiv t'$ alors $\texttt{Loop}\,(v, t) \equiv \texttt{Loop}\,(v, t')$*

Démonstration. La démonstration est immédiate via le théorème 5.1. □

Théorème 5.12. *Pour tout terme t, il existe une valeur v et une fonction f dans le langage hôte tel que $t \equiv \texttt{Loop}\,(v, \texttt{Arr}\,f)$.*

Démonstration. La démonstration se fait par induction structurelle sur le terme t et la définition de $\equiv$.

— $\texttt{Arr}\,f \equiv \texttt{Loop}\,(\texttt{tt}, (\texttt{Arr}\,(\lambda(x,y).(f\ x, \texttt{tt}))))$

— Si $t \equiv \texttt{Loop}\,v\,(\texttt{Arr}\,f)$ alors $\texttt{First}\,t \equiv \texttt{Loop}\,v\,(\texttt{Arr}\,g)$, où

$$g = \lambda((a,d),c).\texttt{let}\,(b,c') = f\,(a,c)\,\texttt{in}\,((b,d),c')$$

— Si $t_1 \equiv \texttt{Loop}\,d\,(\texttt{Arr}\,f_1)$ et $t_2 \equiv \texttt{Loop}\,e\,(\texttt{Arr}\,f_2)$ alors

$$\texttt{Comp}\,(t_1,t_2) \equiv \texttt{Loop}\,(d,e)\,(\texttt{Arr}\,g)$$

où $g = \lambda(a,(c_1,c_2)).\texttt{let}\,(b,c_1') = f_1\,(a,c_1)\,\texttt{in}\,\texttt{let}\,(c,c_2') = (b,c_2)\,\texttt{in}\,(c,(c_1',c_2'))$.

— Si $t \equiv \texttt{Loop}\,d\,(\texttt{Arr}\,f)$ alors $\texttt{Loop}\,c\,t \equiv \texttt{Loop}\,(c,d)\,(\texttt{Arr}\,g)$, où

$$g = \lambda(a,(c,d)).\texttt{let}\,((b,c'),d') = f\,((a,c),d)\,\texttt{in}\,(b,(c',d'))$$

Dans chaque cas, la bisimilarité est établie au niveau sémantique en considérant la relation qui relie les côtés gauche et droit de l'équation, après généralisation de toutes les variables, et en s'assurant que les sous-expressions sont bisimilaires. Par exemple, dans le cas de *first*, on prend la relation $R = \{(\mathit{first}\,sf, \mathit{loop}\,v\,(\mathit{arr}\,g)) \mid sf \sim \mathit{loop}\,v\,(\mathit{arr}\,f)\}$, où g est défini comme ci-dessus, et on prouve qu'il s'agit d'une bisimulation. Nous posons x et z deux valeurs du langage hôte telles que :

$$(\mathit{first}\,sf)\,(x,z) = ((y,z), \mathit{first}\,sf') \text{ où } (y,sf') = sf\,x$$

Par hypothèse, nous savons que $sf \sim \mathit{loop}\,v\,(\mathit{arr}\,f)$, par conséquent, nous avons $(\mathit{loop}\,v\,(\mathit{arr}\,f))\,x = (y, sf'')$ et $R\,sf'\,sf''$. Sachant que $\mathit{arr}\,f$ est invariant lors de l'évaluation, nous pouvons en déduire que :

$$(\mathit{loop}\,v\,(\mathit{arr}\,f))\,x = (y, \mathit{loop}\,v'\,(\mathit{arr}\,f)) \text{ où } (y,v') = f\,(x,v)$$

Maintenant, nous montrons que $(\mathit{loop}\,v\,(\mathit{arr}\,g))\,(x,z) = ((y,z), \mathit{loop}\,v'\,(\mathit{arr}\,g))$.

$$\begin{aligned}(\mathit{loop}\,v\,(\mathit{arr}\,g))\,(x,z) &= (y, \mathit{loop}\,v'\,sf') && \text{où } ((y,v'),sf') = \mathit{arr}\,g\,((x,z),v)\\ &= ((y,z), \mathit{loop}\,v'\,(\mathit{arr}\,g)) && \text{où } (y,v') = f\,(x,v)\end{aligned}$$

Par définition de R, nous avons $R\,(\mathit{first}\,sf')\,(\mathit{loop}\,v'\,(\mathit{arr}\,g))$ ce qui clos notre démonstration pour ce cas. □

5.4.2 Forme normale dans MOLHOLES

Un terme t peut contenir un ensemble de ressources qui peut être divisé en deux : les lectures et les écritures. Nous définissons $\texttt{refs}_{get}$, respectivement $\texttt{refs}_{set}$, une fonction qui prend un terme t et retourne la liste ordonnée sans doublons des ressources utilisées pour une lecture, respectivement pour une écriture. La forme normale pour les termes de MOLHOLES est énoncée dans la définition 5.8. Tout comme la forme normale de YAMPACORE, le cœur d'un terme en forme normale est une simple fonction f élevée en fonction de signal via $\texttt{Arr}\, f$. Cependant, les valeurs locales ne sont pas simulées par une boucle, mais par des ressources (il en va de même pour les entrées qui ne sont pas directement données au programme). Par conséquent, le terme $\texttt{Arr}\, f$ est entouré par les accès en lecture et en écriture. Plus précisément, si nous prenons le terme de gauche à droite alors nous avons les ressources d'entrée, les ressources internes en lecture, la fonction de signal $arr\, f$, puis les ressources internes en écriture et enfin les ressources de sortie.

Définition 5.8 (Forme normale)**.** *Soient t un terme, f une fonction du langage hôte. t est en forme normale si et seulement si :*

$$t = \texttt{Get}\, r_0 \ggg \ldots \ggg \texttt{Get}\, r_{n_g-1} \ggg \texttt{Arr}\, f \ggg \texttt{Set}\, r'_0 \ggg \ldots \ggg \texttt{Set}\, r'_{n_s-1}$$

Nous notons $n_g = |\texttt{refs}_{get}(t)|$, respectivement $n_s = |\texttt{refs}_{set}(t)|$, la taille de la liste de ressources utilisées pour une lecture, respectivement une écriture et r_i une ressource en lecture, respectivement r'_i pour une ressource en écriture, présent à la $i^{ème}$ position dans $\texttt{refs}_{get}(t)$, respectivement $\texttt{refs}_{set}(t)$.

Nous mettons en relation deux termes MOLHOLES via l'opérateur infixe $\equiv$. Cette relation est définie dans la définition 5.9 et forme une congruence comme énoncé et démontré dans le théorème 5.13.

Définition 5.9 (Équivalence de termes)**.** *Soit $\equiv$ une relation sur les termes telle que $t_1 \equiv t_2$ si et seulement si $\texttt{step}\, t_1 = \texttt{step}\, t_2$, où t_1 et t_2 sont deux termes de* MOLHOLES*.*

Exemple 5.19. *Soient $add = \lambda(x, y).x + y$ une fonction qui prend une paire d'entiers naturels et retourne leur somme et r, r' deux ressources. Nous avons l'équivalence suivante :*

$$\texttt{Arr}\, dup \ggg \texttt{Set}\, r' \ggg \texttt{Get}\, r \ggg \texttt{Arr}\, add \equiv \texttt{Get}\, r \ggg \texttt{Arr}\, f \ggg \texttt{Set}\, r'$$

où $f = \lambda(x, y).(add\, x\, y, x)$.

Lemme 5.13. *Pour tous termes t, t', t_1, t'_1, t_2 et t'_2, les propriétés suivantes sont respectées :*

— *si* $t_1 \equiv t'_1$ *et* $t_2 \equiv t'_2$ *alors* $\texttt{Comp}\,(t_1, t_2) \equiv \texttt{Comp}\,(t'_1, t'_2)$ *;*

— *si* $t \equiv t'$ *alors* $\texttt{First}\,t \equiv \texttt{First}\,t'$

Démonstration. La démonstration est immédiate via la définition 5.9. □

Un premier résultat, que nous énonçons dans le théorème 5.14, est qu'un terme sans accès en lecture ou en écriture à la mémoire est équivalent à une fonction signal $\texttt{Arr}\,f$ pour un certain f. En effet, nous nous limitons, dans ce cas, à trois termes : $\texttt{Arr}\,f$, $\texttt{Comp}\,(t_1, t_2)$ et $\texttt{First}\,t$. Via l'ensemble d'équations (2.10), nous savons que la composition et la fonction de signal *first* peuvent être réécrites pour « passer » sous un terme `Arr`.

Lemme 5.14. *Pour tout terme t ne contenant pas d'accès à une ressource, il existe f tel que $t \equiv \texttt{Arr}\,f$.*

Démonstration. La démonstration se fait par induction structurelle sur t et utilise les équations (2.10d) et (2.10h). □

Dans la suite, nous utilisons les fonctions définies dans la figure 5.7. Elles nous permettent de définir des termes paramétrés par des entiers ou des listes de ressources. La compréhension des résultats établis dans le reste de la section dépend de la bonne compréhension de ces différentes notations, par conséquent nous allons les décrire successivement avec des exemples d'utilisation.

La fonction seq_l, respectivement seq_r, permet de définir une expression contenant un terme précédé, respectivement suivi, d'une séquence de termes `Get` ou de termes `Set` en fonction d'une liste de ressources. Le choix du type d'accès, c'est-à-dire lecture ou écriture, est donné en paramètre via la lettre `G` pour la lecture et via la lettre `S` pour l'écriture. Lorsqu'il n'est pas nécessaire de spécifier nous donnons `GS` en paramètre. Ces fonctions rendent accessible une définition plus rigoureuse de la forme normale et une notation plus concise dans les démonstrations.

Exemple 5.20. *Soient t un terme et r_1, r_2 et r_3 des ressources, voici quelques exemples d'utilisation de seq_l et seq_r ainsi que ce qu'ils représentent concrètement.*

$$seq_l(\texttt{GS}, l, t) = \begin{cases} t & \text{si } l = [] \\ \texttt{Comp}\,(seq'_l(\texttt{GS}, l), t) & \text{sinon} \end{cases}$$

$$seq'_l(\texttt{GS}, l) = \begin{cases} \texttt{Arr}\ id & \text{si } l = [] \\ \texttt{Comp}\,(seq'_l(\texttt{GS}, l'), \texttt{Get}\ r) & \text{si } l = l' +\!\!+ [r] \text{ et } \texttt{GS} = \texttt{G} \\ \texttt{Comp}\,(seq'_l(\texttt{GS}, l'), \texttt{Set}\ r) & \text{si } l = l' +\!\!+ [r] \text{ et } \texttt{GS} = \texttt{S} \end{cases}$$

$$seq_r(\texttt{GS}, l, t) = \begin{cases} t & \text{si } l = [] \\ \texttt{Comp}\,(seq_r(\texttt{GS}, l', t), \texttt{Get}\ r) & \text{sinon, où } l = r :: l' \text{ et } \texttt{GS} = \texttt{G} \\ \texttt{Comp}\,(seq_r(\texttt{GS}, l', t), \texttt{Set}\ r) & \text{sinon, où } l = r :: l' \text{ et } \texttt{GS} = \texttt{S} \end{cases}$$

$$stack(n, t) = \begin{cases} t & \text{si } n = 0 \\ \texttt{First}\,(stack(m, t)) & \text{sinon, où } n = m + 1 \end{cases}$$

$$perm_l(n, t) = \begin{cases} t & \text{si } n = 0 \\ \texttt{Comp}\,(perm'_l(0, m), t) & \text{sinon, où } n = m + 1 \end{cases}$$

$$perm'_l(n, m) = \begin{cases} stack(m, \texttt{Arr}\ perm) & \text{si } n \geq m \\ \texttt{Comp}\,(perm'_l(n + 1, m), stack(n, \texttt{Arr}\ perm)) & \text{sinon} \end{cases}$$

$$perm_r(n, t) = \begin{cases} t & \text{si } n = 0 \\ \texttt{Comp}\,(perm_r(m, t), stack(m, \texttt{Arr}\ perm)) & \text{sinon,} \\ & \text{où } n = m + 1 \end{cases}$$

FIGURE 5.7 – Notations concises pour des termes paramétrés.

$$\begin{aligned} seq_l(\texttt{GS}, [], t) &= t \\ seq_l(\texttt{G}, [r_1, r_2], t) &= \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\ r_1, \texttt{Get}\ r_2), t) \\ seq_l(\texttt{S}, [r_1], t) &= \texttt{Comp}\,(\texttt{Set}\ r_1, t) \\ seq_r(\texttt{GS}, [], t) &= t \\ seq_r(\texttt{G}, [r_1, r_2], t) &= \texttt{Comp}\,(\texttt{Comp}\,(t, \texttt{Get}\ r_2), \texttt{Get}\ r_1) \\ seq_r(\texttt{S}, [r_2, r_3], seq_l(\texttt{G}, [r_1], t)) &= \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\ r_1, t), \texttt{Set}\ r_3), \texttt{Set}\ r_2) \end{aligned}$$

La fonction *stack* encapsule un terme t dans n termes `First`, des exemples sont proposés ci-dessous. Cette fonction est utile lors du réarrangement d'un terme,

car nous cherchons à faire sortir les lectures et écritures à l'« extérieur » ce qui va demander beaucoup d'application des équations (5.4a) et (5.4f) et qui va, par conséquent, encapsuler de nombreuses fois les termes de la forme $\texttt{Arr}\, f$ sous des `First`.

Exemple 5.21. *Soient t un terme et n un entier naturel, voici quelques exemples d'utilisation de stack.*

$$\begin{aligned} \mathit{stack}(0, t) &= t \\ \mathit{stack}(3, t) &= \texttt{First}\,(\texttt{First}\,(\texttt{First}\, t)) \\ \mathit{stack}(n+1, t) &= \texttt{First}\,(\mathit{stack}(n, t)) \end{aligned}$$

La fonction perm_l, respectivement perm_r, produit une séquence de fonctions de permutations, élevées en terme via `Arr`, encapsulées dans des opérateurs `First` qui précède, respectivement qui succède à, un terme donné en paramètre. Cette fonction permet de représenter la sortie d'une séquence d'accès d'un terme de la forme $\texttt{First}\, t$. Chaque accès va générer une permutation lors sa sortie, voir équations (5.4b) et (5.4g). Pour récupérer une séquence composée uniquement d'accès, il faut faire passer les lectures ou écriture au-dessus de chaque permutation.

Exemple 5.22. *Soit t un terme, voici quelques exemples d'utilisation de perm_l et perm_r ainsi que ce qu'ils représentent concrètement.*

$$\begin{aligned} \mathit{perm}_l(0, t) &= t \\ \mathit{perm}_l(2, t) &= \texttt{Comp}\,(\texttt{Comp}\,(\texttt{First}\,(\texttt{Arr}\, \mathit{perm}), \texttt{Arr}\, \mathit{perm}), t) \\ \mathit{perm}_r(0, t) &= t \\ \mathit{perm}_r(2, t) &= \texttt{Comp}\,(\texttt{Comp}\,(t, \texttt{Arr}\, \mathit{perm}), \texttt{First}\,(\texttt{Arr}\, \mathit{perm})) \end{aligned}$$

Nous présentons, ci-dessous, un ensemble de résultats qui expriment le comportement des fonctions précédemment définies ainsi que les interactions entre elles. Excepté les équations (5.5a) à (5.5f), les autres équations sont des extensions de certains résultats de l'ensemble d'équations (5.4). Les équations (5.5g) et (5.5j) ne sont vraies que si t' est un terme qui ne contient pas d'effets. Les équations (5.5o) et (5.5p) sont établies pour toute ressource r n'appartenant pas

à la liste rs. Toutes ces équations sont démontrées par le théorème 5.15.

$$seq_l(\mathtt{G}, rs +\!\!+ [r], t) \equiv seq_l(\mathtt{G}, rs, \mathtt{Comp}\,(\mathtt{Get}\,r, t)) \tag{5.5a}$$
$$seq_l(\mathtt{S}, rs +\!\!+ [r], t) \equiv seq_l(\mathtt{S}, rs, \mathtt{Comp}\,(\mathtt{Set}\,r, t)) \tag{5.5b}$$
$$seq_l(\mathtt{GS}, rs, seq_l(\mathtt{GS}, rs', t)) \equiv seq_l(\mathtt{GS}, rs +\!\!+ rs', t) \tag{5.5c}$$
$$seq_r(\mathtt{GS}, rs, seq_r(\mathtt{GS}, rs', t)) \equiv seq_r(\mathtt{GS}, rs +\!\!+ rs', t) \tag{5.5d}$$
$$seq_l(\mathtt{GS}, rs, perm_r(n, t)) \equiv perm_r(n, seq_l(\mathtt{GS}, rs, t)) \tag{5.5e}$$
$$seq_r(\mathtt{GS}, rs, perm_l(n, t)) \equiv perm_l(n, seq_r(\mathtt{GS}, rs, t)) \tag{5.5f}$$

$$seq_r(\mathtt{G}, rs, t') \equiv seq_l(\mathtt{G}, rs, stack(|rs|, t')) \tag{5.5g}$$
$$seq_l(\mathtt{S}, rs, t') \equiv seq_r(\mathtt{S}, rs, stack(|rs|, t')) \tag{5.5h}$$
$$seq_l(\mathtt{G}, rs, \mathtt{Comp}\,(stack(|rs|, t'), t)) \equiv \mathtt{Comp}\,(t', seq_l(\mathtt{G}, rs, t)) \tag{5.5i}$$
$$seq_r(\mathtt{S}, rs, \mathtt{Comp}\,(t, stack(|rs|, t'))) \equiv \mathtt{Comp}\,(seq_r(\mathtt{S}, rs, t), t') \tag{5.5j}$$

$$perm_l(n, \mathtt{Comp}\,(\mathtt{Get}\,r, \mathtt{Comp}\,(\mathtt{Aperm}, t))) \equiv \mathtt{Comp}\,(\mathtt{Get}\,r, perm_l(n+1, t)) \tag{5.5k}$$
$$perm_r(n, \mathtt{Comp}\,(t, \mathtt{Comp}\,(\mathtt{Aperm}, \mathtt{Set}\,r))) \equiv \mathtt{Comp}\,(perm_r(n+1, t), \mathtt{Set}\,r) \tag{5.5l}$$
$$\mathtt{First}\,(seq_l(\mathtt{G}, rs, t)) \equiv seq_l(\mathtt{G}, rs, perm_l(|rs|, \mathtt{First}\,t)) \tag{5.5m}$$
$$\mathtt{First}\,(seq_r(\mathtt{S}, rs, t)) \equiv seq_r(\mathtt{S}, rs, perm_r(|rs|, \mathtt{First}\,t)) \tag{5.5n}$$
$$seq_l(\mathtt{G}, rs, \mathtt{Comp}\,(perm_r(|rs|, \mathtt{Get}\,r), t)) \equiv \mathtt{Comp}\,(\mathtt{Get}\,r, seq_l(\mathtt{G}, rs, t)) \tag{5.5o}$$
$$seq_r(\mathtt{S}, rs, \mathtt{Comp}\,t, (perm_l(|rs|, \mathtt{Set}\,r))) \equiv \mathtt{Comp}\,(seq_r(\mathtt{G}, rs, t), \mathtt{Set}\,r) \tag{5.5p}$$
$$\mathtt{Comp}\,(\mathtt{Get}\,r, perm_r(n, t)) \equiv perm_r(n, \mathtt{Comp}\,(\mathtt{Get}\,r, t)) \tag{5.5q}$$
$$\mathtt{Comp}\,(\mathtt{Set}\,r, perm_r(n, t)) \equiv perm_r(n, \mathtt{Comp}\,(\mathtt{Set}\,r, t)) \tag{5.5r}$$

Lemme 5.15. *Les fonctions définies dans la figure 5.7 respectent l'ensemble d'équations (5.5). Les équations* (5.5g) *et* (5.5j) *ne sont respectées que si* t' *ne contient pas d'accès mémoire.*

Démonstration. La démonstration pour chaque équation se fait par induction soit sur une liste, soit sur un entier naturel, tout dépend des fonctions impliquées. Par exemple, l'équation (5.5m) se fait par induction sur l'inverse de la liste rs. Si la liste est vide alors la fonction seq_l ainsi que la fonction $perm_l$ retourne juste le terme. Par conséquent, l'équation est vraie par réflexivité. Nous supposons que l'équation est vraie pour la liste rs et nous montrons qu'elle est vraie pour $rs +\!\!+ [r]$,

avec r un entier naturel.

$$
\begin{array}{lll}
 & \texttt{First}\,(seq_l(\texttt{G}, rs, t)) & \\
\equiv & \texttt{First}\,(\texttt{Comp}\,(seq'_l(\texttt{G}, rs' +\!\!+ [r]))\,t) & \text{(déf)} \\
\equiv & \texttt{First}\,(\texttt{Comp}\,(\texttt{Comp}\,(seq'_l(\texttt{G}, rs)\,(\texttt{Get}\,r)))\,t) & \text{(déf)} \\
\equiv & \texttt{First}\,(\texttt{Comp}\,(seq'_l(\texttt{G}, rs))\,(\texttt{Comp}\,(\texttt{Get}\,r)\,t)) & \text{(éq. 2.10c)} \\
\equiv & \texttt{Comp}\,(seq'_l(\texttt{G}, rs))\,(perm_l(|rs|, \texttt{First}\,(\texttt{Comp}\,(\texttt{Get}\,r)\,t))) & \text{(hyp. ind.)} \\
\equiv & \texttt{Comp}\,(seq'_l(\texttt{G}, rs))\,(perm_l(|rs|, & \\
 & \qquad \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\,r)\,\texttt{Arr}\,perm)\,(\texttt{First}\,t))) & \text{(éq. 5.4a)} \\
\equiv & \texttt{Comp}\,(seq'_l(\texttt{G}, rs))\,(perm_l(|rs|, & \\
 & \qquad \texttt{Comp}\,(\texttt{Get}\,r)\,(\texttt{Comp}\,(\texttt{Arr}\,perm)\,(\texttt{First}\,t)))) & \text{(éq. 2.10c)} \\
\equiv & \texttt{Comp}\,(seq'_l(\texttt{G}, rs))\,(\texttt{Comp}\,(\texttt{Get}\,r)\,(perm_l(|rs|+1, \texttt{First}\,t))) & \text{(éq. 5.5k)} \\
\equiv & \texttt{Comp}\,(\texttt{Comp}\,(seq'_l(\texttt{G}, rs))\,\texttt{Get}\,r)\,(perm_l(|rs|+1, \texttt{First}\,t)) & \text{(éq. 2.10c)} \\
\equiv & seq_l(\texttt{G}, rs +\!\!+ [r], perm_l(|rs +\!\!+ [r]|, \texttt{First}\,t)) & \text{(déf.)}
\end{array}
$$

□

Finalement, nous définissons le théorème 5.18 qui énonce que tout terme à une forme normale équivalente. Ce théorème se base sur les résultats intermédiaires définis précédemment, c'est-à-dire le théorème 5.14 et l'ensemble des équations (5.5), ainsi que deux autres lemmes qui sont utilisés dans le cas de la composition. Le théorème 5.16 établit qu'une séquence de lectures et une séquence d'écritures peuvent être interverties autour d'un terme sans effets.

Lemme 5.16. *Pour toute fonction g et listes de ressources rs_1 et rs_2, il existe une fonction f telles que :*

$$seq_r(\texttt{G}, rs_1, seq_l(\texttt{S}, rs_2, \texttt{Arr}\,g)) \equiv seq_r(\texttt{S}, rs'_2, seq_l(\texttt{G}, rs'_1, \texttt{Arr}\,f))$$

avec $rs'_1 = rs_1 \backslash rs_2$ et $rs'_2 = rs_2 \backslash rs_1$.

Démonstration. La démonstration se fait par induction structurelle sur rs_1. □

Le théorème 5.17 énonce que deux séquences triées de lecture, respectivement d'écriture, consécutives peuvent être rassemblées en une seule séquence triée. C'est une propriété plus forte que celles énoncées dans les équations (5.5c) et (5.5d).

Lemme 5.17. *Pour toute fonction g et listes de ressources triées rs_1 et rs_2, il existe une fonction f telle que :*

$$seq_l(\texttt{G}, rs_1, seq_l(\texttt{G}, rs_2, \texttt{Arr}\,g)) \equiv seq_l(\texttt{G}, rs, \texttt{Arr}\,f)$$
$$seq_r(\texttt{S}, rs_1, seq_r(\texttt{S}, rs_2, \texttt{Arr}\,g)) \equiv seq_r(\texttt{S}, rs, \texttt{Arr}\,f)$$

avec rs la concaténation des listes rs_1 et rs_2 triée.

Démonstration. Pour chaque équation, la démonstration se fait par induction sur rs_1. Si rs_1 est vide alors rs se simplifie en rs_2 qui est, par hypothèse, triée. Par conséquent, il existe g qui satisfait l'équation. Intuitivement, le cas général se réduit à placer au bon endroit dans la séquence rs une ressource r appartenant à rs_1. Grâce aux équations (5.5c) et (5.5d), nous découpons une séquence afin d'isoler la position que doit avoir r et via les équations (5.5o) et (5.5p) nous déplaçons r après ou avant (cela dépend du type d'accès) la séquence. Si cette ressource est déjà présente dans rs_2 alors nous utilisons l'équations (5.4e) et (5.4j), pour remplacer un accès par une fonction de duplication ou de suppression (cela dépend encore du type d'accès). Finalement, nous ramenons toutes les fonctions de signal sans effets qui ont été créées par les déplacements de r, en plus de celles que va générer le réarrangement, vers $\texttt{Arr}\, g$ pour produire une nouvelle fonction f via la théorème 5.14 qui satisfait l'équation. □

Théorème 5.18. *Pour tout terme t, il existe une fonction f provenant du langage hôte tel que* $t \equiv seq_r(\texttt{S}, \texttt{refs}_{set}(t), seq_l(\texttt{G}, \texttt{refs}_{get}(t), \texttt{Arr}\, f))$*, c'est-à-dire :*

$$t \equiv \texttt{Get}\, r_0 \ggg \ldots \ggg \texttt{Get}\, r_{n_g-1} \ggg \texttt{Arr}\, f \ggg \texttt{Set}\, r_0 \ggg \ldots \ggg \texttt{Set}\, r_{n_s-1}$$

où $n_g = |\texttt{refs}_{get}(t)|$ *et* $n_s = |\texttt{refs}_{set}(t)|$.

Démonstration. La démonstration se fait par induction structurelle sur t.

— $t = \texttt{Arr}\, g$: nous avons $\texttt{refs}_{get}(t) = \texttt{refs}_{set}(t) = \{\}$, ce qui simplifie le but en $\exists f, \texttt{Arr}\, g \equiv \texttt{Arr}\, f$. Le cas est donc direct en définissant f par g.

— $t = \texttt{First}\, t'$: nous savons par hypothèse d'induction que :

$$\exists f, t' \equiv seq_r(\texttt{S}, \texttt{refs}_{set}(t), seq_l(\texttt{G}, \texttt{refs}_{get}(t), \texttt{Arr}\, f))$$

Nous posons f et nous substituons t' dans $\texttt{First}\, t'$. Il faut maintenant faire passer le `First` sous la séquence de `Get` et de `Set`.

$$\begin{aligned}
&\texttt{First}\,(seq_r(\texttt{S}, \texttt{refs}_{set}(t), seq_l(\texttt{G}, \texttt{refs}_{get}(t), \texttt{Arr}\, f))) \\
\equiv\ & seq_r(\texttt{S}, \texttt{refs}_{set}(t), \\
& \quad perm_r(n_s, \texttt{First}\,(seq_l(\texttt{G}, \texttt{refs}_{get}(t), \texttt{Arr}\, f)))) && \text{(éq. 5.5}n\text{)} \\
\equiv\ & seq_r(\texttt{S}, \texttt{refs}_{set}(t), \\
& \quad perm_r(n_s, seq_l(\texttt{G}, \texttt{refs}_{get}(t), perm_r(n_g, \texttt{First}\,(\texttt{Arr}\, f))))) && \text{(éq. 5.5}m\text{)} \\
\equiv\ & seq_r(\texttt{S}, \texttt{refs}_{set}(t), \\
& \quad seq_l(\texttt{G}, \texttt{refs}_{get}(t), perm_r(n_s, perm_r(n_g, \texttt{First}\,(\texttt{Arr}\, f))))) && \text{(éq. 5.5}e\text{)}
\end{aligned}$$

où $n_g = |\mathtt{refs}_{get}(t)|$ et $n_s = |\mathtt{refs}_{set}(t)|$. Via le théorème 5.14, nous définissons $\mathtt{Arr}\, g \equiv perm_r(n_s, perm_r(n_g, \mathtt{First}\,(\mathtt{Arr}\, f)))$. Par conséquent, il existe la fonction g telle que :

$$\mathtt{First}\, t \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t), \mathtt{Arr}\, g))$$

— $t = \mathtt{Get}\, r$: nous définissons f comme la fonction d'identité id. Par l'équation (2.10b) nous savons que :

$$\mathtt{Get}\, r \equiv seq_r(\mathtt{S}, [], seq_l(\mathtt{G}, [r], \mathtt{Arr}\, id)) \equiv \mathtt{Comp}\,(\mathtt{Get}\, r)\,(\mathtt{Arr}\, id) \equiv (\mathtt{Get}\, r)$$

— $t = \mathtt{Set}\, r$: nous définissons f comme la fonction d'identité id. Par l'équation (2.10a) nous savons que :

$$\mathtt{Set}\, r \equiv seq_r(\mathtt{S}, [r], seq_l(\mathtt{G}, [], \mathtt{Arr}\, id)) \equiv \mathtt{Comp}\,(\mathtt{Arr}\, id)\,(\mathtt{Set}\, r) \equiv (\mathtt{Set}\, r)$$

— $t = \mathtt{Comp}\, t_1\, t_2$: nous savons par hypothèse d'induction que :

$$\exists f_1, t_1 \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t_1), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t_1), \mathtt{Arr}\, f_1))$$
$$\exists f_2, t_2 \equiv seq_r(\mathtt{S}, \mathtt{refs}_{set}(t_2), seq_l(\mathtt{G}, \mathtt{refs}_{get}(t_2), \mathtt{Arr}\, f_2))$$

Nous posons f_1, f_2 et nous substituons t_1 et t_2 dans t. De plus, nous définissons $r_{g_1} = \mathtt{refs}_{get}(t_1)$, $r_{s_1} = \mathtt{refs}_{set}(t_1)$, $r_{g_2} = \mathtt{refs}_{get}(t_2)$ et $r_{s_2} = \mathtt{refs}_{set}(t_2)$.

$$t = \mathtt{Comp}\,(seq_r(\mathtt{S}, r_{s_1}, seq_l(\mathtt{G}, r_{g_1}, \mathtt{Arr}\, f_1)), seq_r(\mathtt{S}, r_{s_2}, seq_l(\mathtt{G}, r_{g_2}, \mathtt{Arr}\, f_2)))$$

Grâce à l'associativité de la composition, c'est-à-dire l'équation (2.10c), et le neutre de la composition, c'est-à-dire les équations (2.10a) et (2.10b), nous pouvons réécrire ce terme comme suit :

$$\begin{aligned} t \equiv \mathtt{Comp}\,(\mathtt{Comp}\,(&seq_l(\mathtt{G}, r_{g_1}, \mathtt{Arr}\, f_1),\\ &seq_r(\mathtt{S}, r_{s_1}, seq_l(\mathtt{G}, r_{g_2}, \mathtt{Arr}\, id))),\\ &seq_r(\mathtt{S}, r_{s_2}, \mathtt{Arr}\, f_2)) \end{aligned}$$

D'après le théorème 5.16, il existe une fonction f telle que :

$$\begin{aligned} t \equiv \texttt{Comp}\,(\texttt{Comp}\,(&seq_l(\texttt{G}, r_{g_1}, \texttt{Arr}\, f_1),\\ &seq_l(\texttt{G}, r'_{g_2}, seq_r(\texttt{S}, r'_{s_1}, \texttt{Arr}\, f))),\\ &seq_r(\texttt{S}, r_{s_2}, \texttt{Arr}\, f_2)) \end{aligned}$$

avec $r'_{s_1} = r_{s_1} \backslash r_{g_2}$ et $r'_{g_2} = r_{g_2} \backslash r_{s_1}$. Via les équations (5.5g) à (5.5i), l'associativité et le neutre une fois de plus, nous pouvons réécrire ce terme comme suit :

$$\begin{aligned} t \equiv \texttt{Comp}\,(&seq_l(\texttt{G}, r_{g_1}, seq_l(\texttt{G}, r'_{g_2}, stack(|r'_{g_2}|, \texttt{Arr}\, f_1))),\\ &seq_r(\texttt{S}, r_{s_2}, seq_r(\texttt{S}, r'_{s_1}, stack(|r'_{s_1}|, \texttt{Arr}\, f_2)))) \end{aligned}$$

Via le théorème 5.14, nous définissons f'_1, f'_2 telles que $\texttt{Arr}\, f'_1 \equiv stack(|r'_{g_2}|, \texttt{Arr}\, f_1)$ et $\texttt{Arr}\, f'_2 \equiv stack(|r'_{s_1}|, \texttt{Arr}\, f_2)$. Finalement, nous appliquons sur chaque sous-terme de la composition le théorème 5.17 ce qui nous donne deux nouvelles fonctions f''_1 et f''_2 ainsi que deux listes r_g et r_s. Toujours grâce à l'associativité de la composition, nous pouvons réécrire t comme suit :

$$t \equiv seq_r(\texttt{S}, r_s, seq_l(\texttt{G}, r_g, \texttt{Comp}\,(\texttt{Arr}\, f''_1, \texttt{Arr}\, f''_2)))$$

L'équation (2.10e) nous permet de faire passer la composition sous le terme `Arr`, il existe donc $f = f''_2 \circ f''_1$ qui satisfait l'équivalence.

□

5.4.3 Transformation de MOLHOLES vers YAMPACORE

La transformation d'un programme MOLHOLES vers une expression YAMPACORE nécessite la forme normale dans chacun des langages ainsi qu'une bisimulation les mettant en relation. Pour simplifier la définition de la bisimulation et de la transformation, nous n'allons pas nous baser sur la forme normale de MOLHOLES, mais sur une version plus simple qui réduit aussi les ressources à une par type, c'est-à-dire une pour les entrées, une pour les sorties et une pour les internes.

Définition 5.10. *Un programme p est dit en* forme simplifiée *si et seulement si :*

$$
\begin{aligned}
p \;=\; \{ & \\
& \texttt{inputs} = [I]; \\
& \texttt{outputs} = [O]; \\
& \texttt{internals} = [(v : C)]; \\
& \texttt{program} = \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Get}\,0, \texttt{Get}\,2), \texttt{Arr}\,f), \texttt{Set}\,2), \texttt{Set}\,1) \\
\} &
\end{aligned}
$$

où I, O et C sont des types du langage hôte et $f : ((Unit \times I) \times C) \to ((Unit \times O) \times C)$ une fonction.

Un programme sous cette forme correspond mieux à un programme YAMPA-CORE en forme normale. En effet, l'unique ressource d'entrée correspond au flux d'entrée, l'unique ressource de sortie, au flux de sortie et enfin l'unique ressource interne correspond à la boucle unique.

Exemple 5.23. *Reprenons le programme p', avec* `program` p' *en forme normale, de l'exemple 5.17. La forme simplifiée équivalente à p' est :*

$$
\begin{aligned}
p' \;=\; \{\; & \texttt{inputs} = [\mathbb{N}]; \\
& \texttt{outputs} = [\mathbb{N}]; \\
& \texttt{internals} = [(0, \texttt{true}) : \mathbb{N} \times \mathit{Bool}]; \\
& \texttt{program} = \texttt{Get}\,0 \ggg \texttt{Get}\,2 \ggg \\
& \qquad \texttt{Arr}\,(\lambda((\texttt{tt}, x), (y, z)). \\
& \qquad\quad \mathit{if}\ z\ \mathit{then}\ ((\texttt{tt}, x + y), (y + 1, \texttt{false})) \\
& \qquad\qquad \mathit{else}\ ((\texttt{tt}, x + y), (y, \texttt{true}))) \ggg \\
& \qquad \texttt{Set}\,2 \ggg \texttt{Set}\,1\}
\end{aligned}
$$

Nous pouvons noter que l'agencement du tuple est légèrement différente de la forme normale.

La forme simplifiée d'un programme MOLHOLES lui est équivalente uniquement d'un point de vue fonctions de flux, c'est-à-dire que pour un même flux d'entrée, les deux programmes vont produire le même flux de sortie. En effet, les ressources, et par conséquent la mémoire, ne sont pas les mêmes. Donc le raisonnement équationelle n'est pas suffisant pour établir une relation d'équivalence. Nous avons besoin de définir la bisimulation entre deux programmes. Soit p un programme, nous définissons C_p l'ensemble des mémoires tel que :

$$C_p = \{\sigma \mid \forall i : (\alpha(\texttt{inputs}\,p)), \texttt{pull}\,p\,\sigma\,i \in \gamma\,(\texttt{init}^\dagger\,p)\} \tag{5.6}$$

Cette ensemble nous permet de définir la paire (C_p, f_p), où f_p est défini ci-dessous, qui forme une coalgèbre pour un programme p et le foncteur $\mathbf{F}_{A,B}$ où $[A] = \texttt{inputs}\, p$ et $[B] = \texttt{outputs}\, p$.

$$f_p = \lambda\sigma\, a.(\texttt{push}\, p\, \sigma', \sigma') \qquad \text{où } (\texttt{tt}, \sigma') = \texttt{step}\, (\texttt{program}\, p)\, \texttt{tt}\, (\texttt{pull}\, p\, \sigma\, a)$$

Définition 5.11 (Bisimulation entre programmmes)**.** *Soient p et p' deux programmes et I, O deux types tels que :*

$$\begin{array}{rclcl} I & = & \alpha\, (\texttt{inputs}\, p) & = & \alpha\, (\texttt{inputs}\, p') \\ O & = & \alpha\, (\texttt{outputs}\, p) & = & \alpha\, (\texttt{outputs}\, p') \end{array}$$

Une relation R est une bisimulation entre C_p et $C_{p'}$ si pour toutes mémoires $\sigma \in C_p$ et $\sigma' \in C_{p'}$, valeur d'entrée $a : I$, valeur de sortie $b : O$ telle que $R\, \sigma\, \sigma'$ nous avons :

$$f_p\, \sigma\, a = (b, \sigma_1) \implies \exists \sigma_1', f_{p'}\, \sigma'\, a = (b, \sigma_1') \wedge R\, \sigma_1\, \sigma_1'$$

La bisimilarité est notée $\sim$ comme pour YampaCore. Nous établissons qu'un programme en forme simple est définissable pour tout programme bien typé dans le théorème 5.19. Ce théorème nécessite deux fonctions `gather` et `flatten` qui sont définies ci-dessous. Pour les établir, nous définissons l'inverse de la fonction α, noté β, qui prend un tuple et forme une liste.

$$\texttt{gather}\, n\, m\, p = \begin{cases} \texttt{tt} & \text{si } p = \texttt{tt} \\ p & \text{si } n + m = 0 \\ p & \text{si } n >= |\beta\, p| \text{ ou } m >= |\beta\, p| \\ ((\alpha\, l_1, \alpha\, l_2), \alpha\, l_3) & \text{avec } (l_2, l_3) = \texttt{splitn}\, (|l| - m, l) \\ & \text{et } (l_1, l) = \texttt{splitn}\, (|\beta\, p| - (n + m), \beta\, p) \end{cases}$$

$$\texttt{flatten}\, p = \begin{cases} \texttt{tt} & \text{si } p = \texttt{tt} \\ \alpha\, (\beta\, p_1 \mathbin{+\!\!+} \beta\, p_2 \mathbin{+\!\!+} \beta\, p_3) & \text{si } p = ((p_1, p_2), p_3) \\ p & \text{sinon} \end{cases}$$

Intuitivement, `gather`, pour les entiers n et m, découpe le tuple d'entrée p en trois sous-tuples respectivement de taille $|\beta\, p| - (n + m)$, n et m. La fonction `flatten` inverse ce processus. Elles permettent de faire le lien entre la fonction f du programme en forme normale et la fonction g du programme simplifié. En

effet, dans le premier cas, le type du tuple d'entrée est :

$$((((\texttt{tt},\ \overbrace{v_1),\ldots,v_k}^{\text{valeurs d'entrées}}),\ \overbrace{v'_1),\ldots,v_n}^{\text{valeurs internes}})$$

Dans le second cas, le tuple d'entrée de g aura le type suivant :

$$((\texttt{tt},\ \overbrace{(v_1,\ldots,v_k)}^{\text{valeurs d'entrées}}),\overbrace{(v'_1,\ldots,v_n)}^{\text{valeurs internes}})$$

Le tuple d'entrée de g regroupe les entrées et les internes dans des sous-tuples alors que f non. Il en va de même pour les sorties, par conséquent il faut entourer la fonction f par nos fonctions définies précédemment pour créer la fonction g. Nous définissons, ci-dessous, une fonction *simplify* qui prend un programme en forme normale et retourne un programme en forme normale simplifiée.

$$\begin{array}{lcl}
\mathit{simplify} & : & \mathit{Prog} \to \mathit{Prog} \\
\mathit{simplify}\, p & = & \{ \\
 & & \texttt{inputs} = [\alpha\,(\texttt{inputs}\, p)]; \\
 & & \texttt{outputs} = [\alpha\,(\texttt{outputs}\, p)]; \\
 & & \texttt{internals} = [\alpha\,(\texttt{internals}\, p)]; \\
 & & \texttt{program} = \texttt{Comp}\,(\texttt{Comp}\,(\texttt{Comp}\,(\texttt{Comp}\,(\\
 & & \qquad\qquad \texttt{Get}\,0, \texttt{Get}\,2), \\
 & & \qquad\qquad \texttt{Arr}\,(\texttt{flatten} \circ f \circ (\texttt{gather}\,(k_{in}\, p)\,(k\, p))), \\
 & & \qquad\qquad \texttt{Set}\,2), \texttt{Set}\,1) \\
 & & \}
\end{array}$$

avec $\texttt{program}\, p = \mathit{seq}_r(\texttt{S}, rs, \mathit{seq}_l(\texttt{G}, rs', \texttt{Arr}\, f))$ et rs,rs' deux listes de ressources. Nous pouvons maintenant énoncer le théorème 5.19 qui stipule que pour tout programme p en forme normale et bien typé, $\mathit{simplify}\, p$ et p sont équivalents.

Théorème 5.19. *Pour tout programme en forme normale bien typé p,*

$$(\texttt{init}\, p) \sim (\texttt{init}\,(\mathit{simplify}\, p))$$

Démonstration. Soient C_p l'ensemble des mémoires pour le programme p, voir l'équation (5.6) pour la définition, et $C_{\mathit{simplify}\, p}$ l'ensemble des mémoires pour le programme $(\mathit{simplify}\, p)$, nous définissons la relation $R = \{(\sigma, \sigma') \mid \sigma \in C_p \text{ et } \sigma' \in C_{\mathit{simplify}\, p}\}$. Démontrer ce théorème revient à prouver que R est une bisimulation. □

La transformation qui va suivre a aussi besoin de la bisimulation. Pour un

programme p, nous définissons la bisimulation entre les coalgèbres (C_p,f_p) et ($sf\ A\ B$,id).

Définition 5.12 (Bisimulation entre langages)**.** *Soient p un programme, $A = \alpha\,(\texttt{inputs}\,p)$ et $B = \alpha\,(\texttt{outputs}\,p)$ deux types. Une relation R est une bisimulation entre C_p et $sf\ A\ B$ si pour toute mémoire $\sigma \in C_p$, valeur d'entrée $a : A$, valeur de sortie $b : B$ et expression en forme normale $sf : sf\ A\ B$ telle que $R\,\sigma\ sf$ nous avons :*

$$\textit{si } f_p\,\sigma\,a = (b, \sigma') \textit{ alors il existe } sf' \textit{ telle que } sf\ a = (b, sf') \textit{ et } R\,\sigma'\ sf'$$

avec $sf = \texttt{Loop}\,(c, \texttt{Arr}\,f)$, pour c une valeur typée et f une fonction dans Hst*, et $c = \alpha\,(\texttt{internals}\,p)$.*

Nous avons maintenant tout le nécessaire défini pour établir la transformation. Un programme p en forme simplifiée contient une unique ressource d'entrée, de sortie et interne. Cette structure correspond à celle d'une expression YampaCore en forme normale qui prend un flux d'entrée, produit un flux de sortie et se repose sur un tuple de valeur associé à la boucle.

Exemple 5.24. *Reprenons le terme $\texttt{Loop}\,v\,(\texttt{Arr}\,swap) : SF\ A\ A$, qui représente la fonction qui décale l'entrée d'un instant en commençant par donner la valeur v de type A, pour un certain type A. Un programme* Molholes *équivalent serait le suivant :*

$$\begin{aligned} p \;=\; \{\;& \texttt{inputs} = [A];\\ & \texttt{outputs} = [A];\\ & \texttt{internals} = [(v : A)];\\ & \texttt{program} = \texttt{Get}\,0 \ggg \texttt{Get}\,2 \ggg\\ & \qquad\qquad \texttt{Arr}\,(\lambda((), a, a').((), a', a)) \ggg\\ & \qquad\qquad \texttt{Set}\,2 \ggg \texttt{Set}\,1\} \end{aligned}$$

Le signal d'entrée est de type *Unit* dans un programme et il en va de même pour le signal de sortie. Par conséquent, la fonction f dans un programme simplifié p sera de type :

$$\textit{Unit} \times \alpha(\texttt{inputs}\,p) \times \alpha(\texttt{internals}\,p) \to \textit{Unit} \times \alpha(\texttt{outputs}\,p) \times \alpha(\texttt{internals}\,p)$$

Il faut donc prendre en considération cette légère différence pour faire correspondre p avec une expression YampaCore de type :

$$\alpha(\texttt{inputs}\,p) \times \alpha(\texttt{internals}\,p) \to \alpha(\texttt{outputs}\,p) \times \alpha(\texttt{internals}\,p)$$

Nous commençons par définir, ci-dessous, une fonction *translate* qui prend un programme simplifié p et retourne une expression YampaCore.

$$\begin{aligned} &\mathit{translate}\,(p : \mathit{Prog}) : \mathit{Memory} \to \mathit{SF}\,(\alpha(\texttt{inputs}\,p))\,(\alpha(\texttt{outputs}\,p)) \\ &\quad \mathit{translate}\,p\,\sigma = \texttt{Loop}\,(\alpha\,(\mathit{getInternals}\,\sigma), \texttt{Arr}\,(\lambda(x,y).f\,(\texttt{tt},x),y)) \end{aligned}$$

avec

$$\texttt{program}\,p = \texttt{Get}\,0 \ggg \texttt{Get}\,2 \ggg \texttt{Arr}\,f \ggg \texttt{Set}\,2 \ggg \texttt{Set}\,1$$

et *getInternals* une fonction qui prend une mémoire et retourne la liste des valeurs contenues dans des cellules étiquetée `Internal` rangées dans l'ordre croissant de leur identifiant. L'expression YampaCore produite par cette fonction est, après évaluation, encore définissable via la fonction *translate*. Nous capturons cette propriété dans le théorème 5.20.

Lemme 5.20. *Pour tout programme bien typé en forme normale simplifiée p et mémoire σ, si* $\texttt{step}\,\mathit{translate}\,p\,\sigma = (v, sf)$*, pour v une valeur dans* Hst *et sf une fonction de signal, alors il existe σ' tel que $sf = \mathit{translate}\,p\,\sigma'$.*

Démonstration. Nous supposons qu'il existe v et sf tel que $\texttt{step}\,\mathit{translate}\,p\,\sigma = (v, sf)$. Si nous réduisons le terme $\mathit{translate}\,p\,\sigma$ nous avons le terme suivant :

$$\texttt{Loop}\,(\alpha\,(\mathit{getInternals}\,\sigma), \texttt{Arr}\,(\lambda(x,y).f\,(\texttt{tt},x),y))$$

avec f étant la fonction au coeur de $(\texttt{program}\,p)$. L'observation clé ici est que le sous-terme $\texttt{Arr}\,(\ldots)$ est invariant durant l'évaluation. Par conséquent, la seule modification provient de $(\alpha\,(\mathit{getInternals}\,\sigma))$ la valeur interne à la boucle. Cette valeur dépend directement des valeurs internes à σ. Donc il existe un σ' tel que $sf = \mathit{translate}\,p\,\sigma'$. □

Maintenant, nous énonçons dans le théorème 5.21 que cette fonction *translate* produit effectivement une expression équivalent par rapport à la définition de bisimilarité précédente.

Théorème 5.21 (Transformation de Molholes vers YampaCore)**.** *Pour tout programme en forme normale simplifié et bien typé p,*

$$(\texttt{init}\,p) \sim \texttt{step}\,(\mathit{translate}\,p\,(\texttt{init}\,p))$$

Démonstration. Nous définissons $A = \texttt{inputs}\,p$ et $B = \texttt{outputs}\,p$. Soient (C_p, f_p) et $(sf\,A\,B, id)$ deux coalgèbres pour le foncteur $\mathbf{F}_{A,B}$ telles que $(\texttt{init}\,p) \in C_p$ et $(\mathit{translate}\,p\,(\texttt{init}\,p)) \in sf\,A\,B$. La démonstration se fait par bisimulation sur la relation $R = \{(\sigma, \texttt{step}\,(\mathit{translate}\,p\,\sigma)) \mid \sigma \in C_p\}$. □

Ce résultat s'étend à l'ensemble de MOLHOLES grâce aux théorèmes 5.18 et 5.19 qui nous assurent qu'il existe une forme normale simplifiée pour tout terme. De plus, nous pouvons dériver ce résultat pour les fonctions `run` comme l'énonce et le démontre le théorème 5.22.

Corollaire 5.22. *Pour tout programme en forme normale simplifié et bien typé p,*

$$\texttt{run}\, p\, (\texttt{init}\, p)\, s \sim \texttt{run}\, (\mathit{translate}\, p\, (\texttt{init}\, p))$$

avec s le flux d'entrée de type $(\alpha\, (\texttt{inputs}\, p))$ et $\sim$ dénotant la bisimilarité entre flux.

Démonstration. La démonstration se fait par bisimulation sur les flux et grâce au théorème 5.21. □

5.5 Conclusion

Dans ce chapitre, nous avons présenté MOLHOLES, un langage qui étend l'approche de WORMHOLES pour introduire des effets de bords tout en préservant une compositionnalité typique des langages FRP avec flèches. MOLHOLES contient, en plus des familles d'opérateurs classiques des Flèches, une famille d'opérateur pour la lecture et une pour l'écriture qui reposent sur un concept de ressource. Ces ressources sont divisées en trois types d'accès : entrée, sortie et interne. Les entrées et sorties maintiennent la contrainte d'utilisation unique proposée dans WORMHOLES, alors que les internes peuvent être écrites et lues plusieurs fois.

Nous avons fourni un système de types qui garantit la bonne utilisation des ressources dans le programme ainsi qu'une sémantique dynamique reposant sur une monade d'état. Nous avons démontré que tout programme bien typé est réactif.

En plus de la définition du langage, nous avons établi des transformations entre MOLHOLES et YAMPACORE, un sous-ensemble de YAMPA. En effet, nous avons démontré qu'un programme MOLHOLES est équivalent à un terme YAMPACORE, et donc sans effets de bords, grâce à des techniques de démonstration par bisimulation.

Chapitre 6

Conclusion

6.1 Contributions

Dans ce manuscrit, nous avons présenté des travaux autours de la formalisation de langage de programmation fonctionnelle réactive avec flèches et effets de bords. Ils se découpent en trois contributions :

1. une formalisation mécanisée en Rocq d'un langage AFRP avec effets nommée Wormholes ;
2. une bibliothèque Rocq de représentation de variables via les *niveaux* de De-Bruijn ;
3. un nouveau langage AFRP avec effets nommé Molholes qui étend l'approche proposée dans Wormholes.

Tout d'abord, nous avons proposé une variation du langage Wormholes proposé par D. Winograd-Cort et P. Hudak en 2012 qui corrige des problèmes critiques de la formalisation initiale comme la perte de la progression ou encore une faille dans le système de types. Pour s'assurer que notre variante corrige effectivement les erreurs trouvées tout en n'en provoquant pas d'autres, nous avons mécanisé le langage en Rocq. La formalisation est accessible dans le dépôt git suivant :

`https://github.com/JordanIschard/Mechanized-Wormholes`

Ensuite, pour mener à bien la formalisation de notre variante de Wormholes, il a fallu gérer les identifiants de ressources présents dans le langage via une représentation pour éviter la capture de variables. Par rapport aux représentations existantes, nous avons décidé d'utiliser les niveaux de De-Bruijn. D'abord intégré dans la formalisation de Wormholes, nous avons ensuite extrait les fonctionnalités uniquement liées aux niveaux pour définir notre bibliothèque de gestion

d'identifiants nommée `DeBrLevel` qui est disponible pour la version 8.20.1 de Coq sur le dépôt suivant :

`https://github.com/JordanIschard/DeBrLevel`

Elle consiste en une collection de modules et d'interfaces qui permet de définir des structures nivelées, c'est-à-dire des structures contenant des niveaux. De plus, elle contient un ensemble de structures prédéfinies comme les listes, les ensembles ou encore les dictionnaires.

Finalement, nous avons présenté Molholes un langage AFRP qui étend l'approche de Wormholes en donnant plus de liberté quant à l'utilisation des ressources tout en préservant les propriétés typiques d'un langage AFRP. Molholes contient, en plus des familles d'opérateurs classiques des flèches, une famille d'opérateur pour la lecture et une pour l'écriture qui reposent sur un concept de ressource. Ces ressources sont divisées en trois types d'accès : entrée, sortie et interne. Les ressources d'entrées et de sorties maintiennent la contrainte d'utilisation unique proposée dans Wormholes, alors que les ressources internes peuvent être écrites et lues plusieurs fois.

Nous avons fourni un système de types qui garantit la bonne utilisation des ressources dans le programme ainsi qu'une sémantique dynamique reposant sur une monade d'état. Nous avons démontré que tout programme bien typé est réactif.

En plus de la définition du langage, nous avons établi des transformations entre Molholes et YampaCore, un sous-ensemble de Yampa. En effet, nous avons démontré qu'un programme Molholes est équivalent à un terme YampaCore, et donc sans effets de bords, grâce à des techniques de démonstration par bisimulation.

6.2 Perspectives

La bibliothèque `DeBrLevel` est encore loin d'être optimale, elle ne contient pas de tactiques sur mesure et pourrait jouir d'un plus grand nombre de structures nivelées prédéfinies. De plus, notre bibliothèque n'a qu'un seul cas d'utilisation. Il serait donc intéressant dans un premier temps d'améliorer et d'enrichir la bibliothèque et dans un second temps de la mettre à l'épreuve sur d'autres formalisations. Le PoplMark challenge [1] pourrait être un bon cas d'étude.

Le langage Molholes a été initialement motivé par le développement d'une bibliothèque de programmation réactive pour le langage OCaml, intégrant le

1. `https://www.seas.upenn.edu/~plclub/poplmark/`

modèle de programmation de Yampa avec la nature impure de OCaml, qui n'a pas encore été faite. Par conséquent, une première perspective pour ce langage serait cela.

Pour la suite, nous prévoyons de formaliser nos résultats en utilisant l'assistant de preuve Rocq. Notre objectif à long terme est de vérifier la correction des programmes par rapport à des spécifications comportementales, potentiellement exprimées en logique temporelle, permettant la génération de programmes réactifs certifiés. Nous espérons que le domaine sémantique de Molholes est suffisamment expressif pour définir la sémantique de langages similaires, c'est-à-dire des langages qui combinent l'approche du flux de données avec des calculs efficaces.

Une autre direction de recherche consiste à étendre Molholes avec des constructions supplémentaires. Les combinateurs présentés dans cet article ne couvrent qu'un sous-ensemble restreint de fonctions de signal, ce qui simplifie la bisimulation en raison du comportement de mise à jour limité dans les étapes de calcul. Nous souhaitons étudier les combinateurs Yampa au-delà de cette restriction et analyser leur impact sur les preuves de bisimulation. En outre, nous prévoyons d'explorer les extensions du langage qui améliorent l'expressivité, y compris d'autres effets de calcul, tels que la gestion des exceptions, afin d'améliorer la compatibilité avec OCaml. L'interaction entre les ressources et les références d'OCaml serait particulièrement intéressante.

Annexe A

Bibliothèque Rocq pour les niveaux de De-Bruijn

Sommaire

Représenter une variable dans un langage de programmation est un point critique dans une formalisation. Dans un langage aussi simple que le lambda-calcul non-typé, voir la section 2.2, il est possible d'avoir des problèmes de *capture de noms* à cause de la substitution.

Exemple A.1. *Soient x, y deux variables et $t = \lambda x.(y\,x)$, $t' = (x\,x)$ deux termes. Dans t la variable x est liée et la variable y est libre. Dans t' la variable x est libre. Substituer y par t' dans t produit le terme $\lambda x.((x\,x)\,x)$. Il n'est plus possible de distinguer si x est la variable libre de t' ou la variable liée de t, par conséquent*

x est capturée *par l'abstraction. Cela peut être évité si nous renommons x par une autre variable z dans t', ce qui produit le terme $\lambda x.(z\,z)\,x$ sans ambiguïté.*

Présentée dans la section 2.2, l'α-équivalence est une relation qui permet de raisonner sur des classes de termes. Intuitivement, cela revient à raisonner à « renommage près ». Cette équivalence est majoritairement utilisée dans les articles sur le lambda-calcul.

Cependant, dans le cadre d'une formalisation plus stricte, ou d'une version mécanisée du lambda-calcul dans un assistant de preuve comme Rocq, il est difficile de raisonner avec l'α-équivalence. En effet, dans le cas de l'assistant de preuve Rocq, l'ensemble des termes se définit naturellement comme un type inductif qui a un schéma d'induction pré-défini avec l'égalité de base de l'assistant de preuve. Cette égalité est une égalité structurelle, par conséquent, une démonstration sur un terme, en utilisant les mécanismes de Rocq, utilise l'égalité syntaxique ce qui pousse à la définition d'une représentation qui contourne la nécessité des classes d'équivalences et se concentre sur une égalité syntaxique.

La communauté scientifique a donc cherché une représentation des variables qui évite toute capture. Plusieurs ont été proposées [25,26,38] et les plus anciennes ont été définies par N.G De Bruijn [38].

Dans ce chapitre, nous faisons un état de l'art concis des représentations de variables les plus utilisées. Ensuite nous présentons une bibliothèque Coq qui permet de gérer des identifiants dans des langages avec lieurs via les *niveaux*. Finalement, nous proposons un cas d'utilisation de cette bibliothèque dans la section A.3.

A.1 État de l'art

Dans cette section, nous proposons une explication succincte de trois représentations d'identifiants majoritairement utilisée dans la formalisation de langages que nous avons considéré lors de notre mécanisation du langage Wormholes détaillée dans la section A.3.

A.1.1 Représentations de De-Bruijn

En 1972, N. G. De Bruijn définit, pour le lambda-calcul, deux mesures permettant d'annoter les variables de manière à éviter le problème de capture : la *distance* et le *niveau*. Pour rappel, une variable est soit libre, soit liée. Afin de simplifier les explications, nous ne considérons que les termes clos, c'est-à-dire que

toute variable est associée à une abstraction dans le terme. Nous commençons par définir la mesure *distance* puis celle de *niveau*.

Distance

La représentation des termes avec la *distance* consiste à identifier chaque occurrence de variable avec sa distance entre elle et l'abstraction à laquelle elle est liée. La *distance* d'une occurrence de variable est le nombre d'abstractions qui « sépare » l'occurrence de la variable de l'abstraction qui la déclare. Plus formellement, nous parcourons le chemin entre la variable et son abstraction dans l'arbre syntaxique abstrait tout en comptant le nombre de nœuds associés à une abstraction.

Exemple A.2. *Soit* $t = \lambda x.x\ ((\lambda y.y\ x)\ (\lambda z.(z\ (\lambda w.w\ x))))$ *une lambda expression. Nous annotons l'arbre syntaxique de cette expression, donné dans la Figure A.1, avec les distances correspondantes à chaque occurrence de variables. Cela nous donne l'expression traduite en représentation par distance suivante :*

$$\lambda.(0\ (\lambda.(0\ 1)\ \lambda.(0\ \lambda.(0\ 2))))$$

L'occurrence la plus à gauche de x *a pour distance* 0*, car aucune abstraction n'est présente dans le chemin entre elle et son abstraction. Par contre, l'occurrence la plus à droite de* x *est sous deux abstractions : celle qui déclare* z *et celle qui déclare* w*. Par conséquent, cette occurrence a pour distance* 2*.*

La mesure de distance correspond à ce que nous appellons aujourd'hui les *indices* de De-Bruijn. Cette représentation simple permet de modéliser tous types d'identifiants dans des langages plus ou moins complexe [11, 37, 65, 98]. De plus, il sert de base dans la représentation *localement non-nommée* que nous présentons dans la section suivante.

Niveau

À la différence d'un indice, le niveau d'une occurrence de variable ne dépend pas de sa position, mais de la position de l'abstraction qui la déclare dans le terme. En effet, le *niveau* d'une variable est le nombre d'abstractions qui « sépare » la racine du terme de l'abstraction qui déclare la variable. Par conséquent, toutes les occurrences d'une même variable correspondent au même niveau, ce qui fait de cette représentation un témoin spécifique de l'α-équivalence. Plus formellement, nous attribuons à tous les nœuds représentant une abstraction, dans l'arbre syntaxique d'un terme, le niveau courant. À la racine, ce niveau est 0. Il est ensuite incrémenté à chaque abstraction rencontrée.

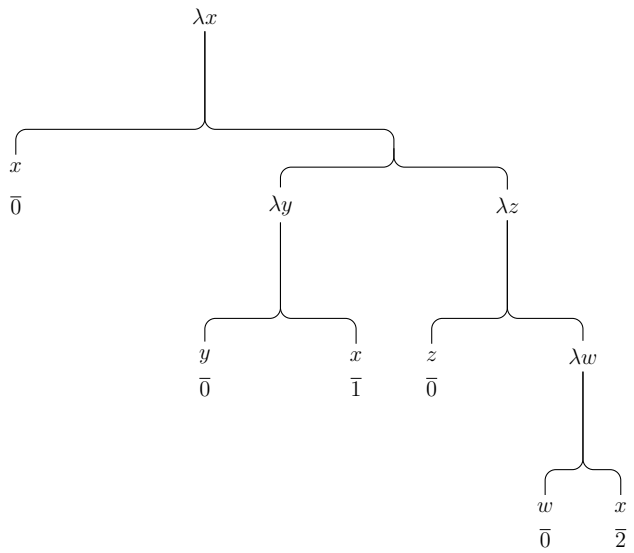


FIGURE A.1 – Représentation arborescente de $\lambda x.x\ ((\lambda y.y\ x)\ (\lambda z.(z\ (\lambda w.w\ x))))$ annotée avec les $\overline{distance}$.

Exemple A.3. *Reprenons l'expression* $t = \lambda x.x\ ((\lambda y.y\ x)\ (\lambda z.(z\ (\lambda w.w\ x))))$ *de l'exemple A.2. Cette fois-ci, nous annotons l'arbre syntaxique de* t*, donné dans la Figure A.2, par des niveaux. La représentation nous donne l'expression suivante :*

$$\lambda.(0\ (\lambda.(1\ 0)\ \lambda.(1\ \lambda.(2\ 0))))$$

L'abstraction qui déclare x *est dans le niveau* 0*, par conséquent,* x *est associé au niveau* 0 *et le niveau courant passe à* 1 *sous l'abstraction.*

Remarque A.1. *Un même niveau peut être attribué à deux variables différentes, c'est le cas de* z *et* y *dans l'expression de l'exemple A.3. Cela arrive quand il y a une application de deux abstractions. Ce n'est pas dérangeant, car les variables liées, ayant le même niveau, appartiennent à deux branches distinctes de l'arbre syntaxique.*

Dans la littérature, peu de mentions des niveaux sont faites explicitement [71]. En effet, cette mesure n'a pas eu le même succès que les indices. Une potentielle raison à cela est qu'elle n'est pas centrale dans l'article original de N.G. De-Bruijn qui se focalise sur la distance.

A.1.2 Représentations localement non-nommée

Dans l'article qui introduit les indices et les niveaux [38], N.G. De-Bruijn suggère que les variables libres n'ont pas besoin d'être gérées sous forme d'indices

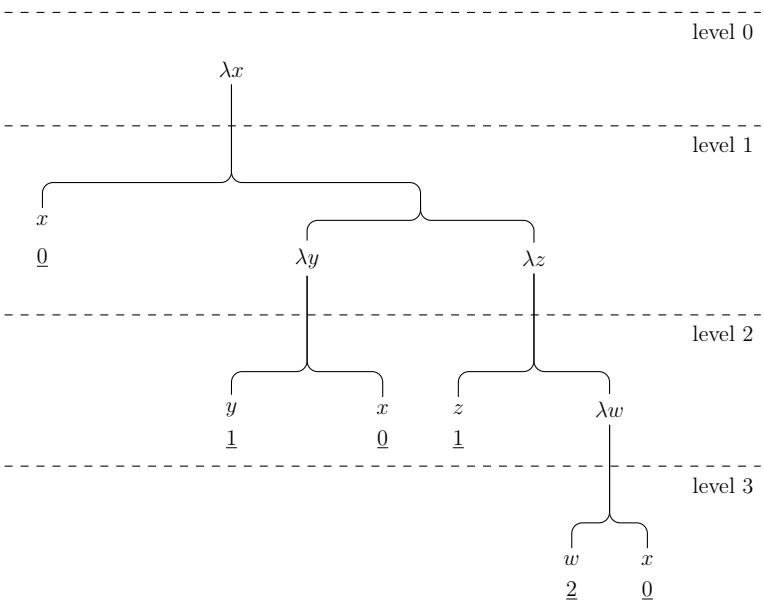


FIGURE A.2 – Représentation arborescente de $\lambda x.x\ ((\lambda y.y\ x)\ (\lambda z.(z\ (\lambda w.w\ x))))$ annotée avec les niveaux.

et peuvent être prises directement dans un ensemble de noms. Les variables liées sont donc « non-nommées » au profit d'indices pour gérer leur portée.

Exemple A.4. *Soit $t = \lambda x.x\,y$ un terme du lambda calcul. Dans la représentation localement non-nommée, x est remplacée par un indice et y reste un nom, ce qui donne le terme $\lambda.0\,y$.*

Lors de l'ouverture d'une abstraction, principalement lors de la β-réduction ou durant la vérification du typage d'un terme, les indices associés à celle-ci sont remplacés par un nom de variable « neuf », c'est-à-dire pris en dehors des noms de variables libres déjà présents. Grâce à cela, la capture de variables est évitée, car le nommage des variables liées est fait uniquement quand cela est nécessaire et en tenant compte du contexte.

Cette représentation a été majoritairement adoptée par la communauté dans le cadre de la mécanisation de sémantiques de langages [3, 8, 25, 45, 69]. De plus, elle est implémentée dans plusieurs assistants de preuves comme ROCQ [95] ou encore ISABELLE/HOL [76].

A.1.3 Syntaxe abstraite paramétrée

Lorsque nous mécanisons un langage dans un assistant de preuve, il faut choisir le niveau *d'intégration* du langage dans l'assistant. Nous nous posons la question

suivante : Est-ce que les termes du langage sont entièrement définis indépendamment du métalangage proposé par l'assistant ou est-ce que les termes du langage sont construits avec le métalangage ?

Exemple A.5. *Prenons les fonctions de signal de* Yampa. *Il y a deux façons de les modéliser en* Rocq. *Soit nous définissons les quatre constructeurs* et *un langage interne (imaginons que le langage interne soit un lambda-calcul non typé), cela donnerait :*

```
Section Deep_arrow.
  Variable variable : Set.

  Inductive intern : Set :=
    | var : variable -> intern
    | app : intern -> intern -> intern
    | abs : variable -> intern -> intern.

  Inductive sf : Set :=
    | Arr   : intern -> sf
    | First : sf -> sf
    | Comp  : sf -> sf -> sf
    | Loop  : intern -> sf -> sf.
End Deep_arrow.
```

Soit nous définissons les quatre constructeurs en utilisant le métalangage comme langage interne, ce qui donnerait :

```
Section Shallow_arrow.
  Inductive sf' : Type -> Type -> Type :=
    | Arr' A B : (A -> B) -> sf' A B
    | First' A B C : sf' A B -> sf' (A * C) (B * C)
    | Comp' A B C : sf' A B -> sf' B C -> sf' A C
    | Loop' A B C : C -> sf' (A * C) (B * C) -> sf' A B.
End Shallow_arrow.
```

Dans le premier cas, nous parlons d'*intégration profonde*. Elle est nécessaire dans les cas où le metalangage ne peut pas être utilisé pour définir en partie le langage, cependant toute la gestion des identifiants est à gérer manuellement. Dans le second cas, il s'agit d'*intégration de surface*. Elle permet de déléguer une partie de l'évaluation au métalangage en plus d'éviter la gestion des identifiants. Il n'est cependant pas toujours possible ou simple de passer par ce genre d'intégration.

En 2008, Adam Chlipala présente dans un article [26] un moyen de définir une syntaxe abstraite qui délègue la gestion des identifiants au métalangage. Cette représentation s'appelle *PHOAS*, ce qui signifie « *Parametric Higher-Order Abstract Syntax* ». Intuitivement, dans un lambda calcul, il reprend l'intégration de surface uniquement pour les abstractions. Voici, ci-dessous, la définition d'un lambda calcul non typé via la représentation *PHOAS*. Cet exemple provient du site de A. Chlipala `http://adam.chlipala.net/cpdt/html/Hoas.html`.

```
Section term.
  Variable var : Set.

  Inductive term : Set :=
  | UVar : var -> term
  | UApp : term -> term -> term
  | UAbs : (var -> term) -> term.
  End term.

  Definition Term := forall var, term var.
End term.
```

L'abstraction est définie comme une fonction dans le métalangage de Rocq, c'est-à-dire Gallina. Cette représentation a été utilisée dans de nombreuses formalisations [6, 35, 55, 105], montrant sa polyvalence. Cependant, par rapport aux autres représentations dont nous avons parlé précédemment, celle-ci ne propose pas une structuration des noms de variables pour éviter les captures de noms, mais délègue ce problème au système dans lequel le langage est modélisé, car il sait déjà le faire.

A.2 La bibliothèque `DeBrLevel`

La bibliothèque `DeBrLevel` a été développée dans le cadre de cette thèse pour les besoins de la mécanisation d'une variante du langage Wormholes. Cette mécanisation nous sert de cas d'étude dans la section A.3. La bibliothèque permet de définir différentes structures utilisant des niveaux de De-Bruijn. Elle est disponible dans son entièreté dans le répertoire git `https://github.com/JordanIschard/DeBrLevel`. Elle est découpée en quatre dossiers :

— le dossier `Core` : il contient toutes les interfaces et modules présentés dans la section A.2.2 ;

— le dossier `Datatype` : il contient les interfaces et modules pour les paires, les listes et les types optionnelles ;

— le dossier `Map` : il contient les interfaces et modules pour les dictionnaires qui sont détaillées dans la section A.2.3 ;

— le dossier `Set` : il contient les interfaces et modules pour les ensembles.

La modélisation des niveaux passe par la création de propriétés et de fonctions pour les manipuler dans différents contextes. Dans cette section, nous commençons par présenter la fonction et la propriété minimale qui nous permet de raisonner sur les niveaux. Ensuite nous décrivons le cœur de la bibliothèque avec son architecture et son fonctionnement. Finalement, nous développons une des structures prédéfinies : le dictionnaire.

A.2.1 Concepts autour des niveaux

Dans les explications de la section A.1.1, nous avons décrit, pour le lambda-calcul, le fonctionnement des niveaux de De-Bruijn sur des termes clos. Pour rappel, un terme est dit clos si toutes ses variables sont liées à une abstraction. De ce fait, nous avons supposé que le *niveau de base*, c'est-à-dire le niveau sur lequel nous nous basons pour attribuer le niveau des variables liées, est 0. Ce choix provient du lien entre le *niveau de base* et l'*ensemble des variables libres*. En effet, soit $t = \lambda x.t_1$ un terme, durant l'application de la représentation sur t, le passage sous l'abstraction incrémente de 1 le niveau de base de t_1. Sachant que x devient libre dans celui-ci, nous pouvons en déduire que le niveau de base représente le nombre de variables libres potentielles (nous précisons « potentielle », car x peut ne pas avoir d'occurrence dans t_1). En résumé, la représentation d'un terme est paramétrée par un niveau qui est égal à 0 s'il ne contient pas de variables libres. Ce niveau qui détermine la représentation du terme est appelé « niveau de base » dans la suite.

Exemple A.6. *Reprenons le terme* $t = \lambda x.x\ ((\lambda y.y\ x)\ (\lambda z.(z\ (\lambda w.w\ x))))$ *dont l'arbre syntaxique abstrait est donnée dans la figure A.2. Sa représentation, nommée* t_1*, est* $\lambda.(0\ (\lambda.(1\ 0)\ \lambda.(1\ \lambda.(2\ 0))))$ *en prenant* 0 *comme niveau de base. Nous n'avons donc considéré aucune variable libre.*

Si nous gardons t_1 *et que nous l'interprétons avec* 1 *au lieu de* 0*, c'est-à-dire que nous considérons qu'il y a potentiellement une variable libre, alors le sens de l'expression se voit changé. En effet, la variable* 0 *devient libre, la variable* 1 *devient la variable liée de la première abstraction et la variable* 2 *de la seconde. Remis sous représentation classique, nous aurions :*

$$\lambda x.a\ ((\lambda y.x\ a)\ (\lambda z.(x\ (\lambda w.z\ a))))$$

avec a une variable libre.

L'ensemble des termes, dans cette représentation, n'ont pas forcément de sens en fonction du niveau de base utilisé. En effet, la variable 0 n'a pas d'interprétation si le niveau de base est 0, car la variable n'est pas associée à une abstraction, ce qui la rend libre, mais le niveau de base est 0 ce qui signifie qu'il n'y a pas de variables libres. Cependant, avec le niveau de base 1 la variable 0 est interprétable. Nous capturons cette propriété dans un prédicat de *bonne formation* qui assure qu'un terme n'est pas incohérent. Nous disons que « t est bien formé sous n », avec t un terme et n un niveau, si t et n satisfont le prédicat et nous le notons $n \Vdash t$. Dans le cas du lambda calcul, un terme est bien formé sous un certain niveau s'il satisfait les règles présentées dans la figure A.3.

$$\frac{x < n}{n \Vdash x}\ \texttt{WF-Var} \qquad \frac{n+1 \Vdash t}{n \Vdash \lambda.t}\ \texttt{WF-Lam} \qquad \frac{n \Vdash t_1 \quad n \Vdash t_2}{n \Vdash t_1\ t_2}\ \texttt{WF-App}$$

FIGURE A.3 – Règles de bonne formation d'une expression du lambda calcul.

Exemple A.7. *Soient $t = \lambda x.(x\ y)$ un terme du lambda calcul. Son équivalent dans la représentation avec comme niveau de base 1 est $t_1 = \lambda.(1\ 0)$. Nous montrons la dérivation de la propriété de bonne formation de t_1 avec 0 et 1 dans la figure A.4.*

Remarque A.2. *Si un terme t est bien formé sous un niveau n, c'est-à-dire que $n \Vdash t$ est satisfait, alors tout niveau m supérieur à n satisfait ce prédicat. C'est une conséquence de la règle* `WF-Var`.

Dans le lambda-calcul, il est possible de substituer les occurrences d'une variable par un terme dans un terme cible. La substitution, et plus généralement toute fonction qui réécrit un terme, est une fonction critique dans la gestion des niveaux, car l'interprétation du terme résultant peut être faussée.

Exemple A.8. *Soient $t = \lambda.(0\,1)$ et $t_1 = \lambda.0$ deux termes, avec t bien formé sous 1 et t_1 sous 0. Si nous substituons 1 dans t par t_1 alors le résultat est $\lambda.0\,(\lambda.0)$. Dans le terme résultant, il y a deux occurrences de zéro, la variable liée de t_1 a été capturée par l'abstraction de t.*

$$\dfrac{\dfrac{\dfrac{\neg(1<1)}{1 \nVdash 1}\ \texttt{WF-Var} \quad \dfrac{0<1}{1 \Vdash 0}\ \texttt{WF-Var}}{1 \nVdash (1\ 0)}\ \texttt{WF-App}}{0 \nVdash \lambda.(1\ 0)}\ \texttt{WF-Lam} \qquad \dfrac{\dfrac{\dfrac{1<2}{2 \Vdash 1}\ \texttt{WF-Var} \quad \dfrac{0<1}{2 \Vdash 0}\ \texttt{WF-Var}}{2 \Vdash (1\ 0)}\ \texttt{WF-App}}{1 \Vdash \lambda.(1\ 0)}\ \texttt{WF-Lam}$$

FIGURE A.4 – Exemple de dérivation de la propriété de bonne formation du terme $\lambda.(1\ 0)$ sous 0 et 1.

Pour éviter ce problème, il faut pouvoir modifier les niveaux dans un terme. Nous définissons une fonction *shift* qui prend en paramètre un seuil n, un décalage m et un terme t et retourne t où tous les niveaux k supérieurs ou égaux au seuil n sont remplacés par $m + k$. Notons $[\triangleright\ n \mid m]$ la fonction $(\mathit{shift}\, n\, m)$. Le tableau A.1 fournit des exemples d'utilisation de cette fonction. Une opération analogue est aussi nécessaire pour la représentation avec les indices. Si $n = 0$ alors l'incrémentation s'appliquera sur tous les niveaux inconditionnellement. Cependant, si n est le niveau de base utilisé pour la traduction en représentation par niveaux du terme alors la fonction aura un impact uniquement sur ses variables liées.

TABLE A.1 – Exemples d'application de *shift* sur des lambda-termes.

repr. *classique* repr. par *niveau*	$\lambda x.(x\ x)$ $\lambda.(0\ 0)$	$\lambda x.((\lambda y.(x\ y))\ x)$ $\lambda.((\lambda.(0\ 1))\ 0)$	$\lambda x.\lambda y.((\lambda z.(x\ z))\ (x\ y))$ $\lambda.\lambda.((\lambda.(0\ 2))\ (0\ 1))$
appl. de $[\triangleright\ 0 \mid 0]$	$\lambda.(0\ 0)$	$\lambda.((\lambda.(0\ 1))\ 0)$	$\lambda.\lambda.((\lambda.(0\ 2))\ (0\ 1))$
appl. de $[\triangleright\ 0 \mid 3]$	$\lambda.(3\ 3)$	$\lambda.((\lambda.(3\ 4))\ 3)$	$\lambda.\lambda.((\lambda.(3\ 5))\ (3\ 4))$
appl. de $[\triangleright\ 1 \mid 0]$	$\lambda.(0\ 0)$	$\lambda.((\lambda.(0\ 1))\ 0)$	$\lambda.\lambda.((\lambda.(0\ 2))\ (0\ 1))$
appl. de $[\triangleright\ 1 \mid 3]$	$\lambda.(0\ 0)$	$\lambda.((\lambda.(0\ 4))\ 0)$	$\lambda.\lambda.((\lambda.(0\ 5))\ (0\ 4))$
appl. de $[\triangleright\ 2 \mid 2]$	$\lambda.(0\ 0)$	$\lambda.((\lambda.(0\ 1))\ 0)$	$\lambda.\lambda.((\lambda.(0\ 4))\ (0\ 1))$

A.2.2 Présentation du cœur de la bibliothèque

Nous appelons *structure* (ou *élément*) *nivelée* une structure qui contient des niveaux, que ce soit dans le cadre de son utilisation pour représenter des identifiants, par exemple des termes d'un langage, ou dans le cadre d'un stockage, par exemple un contexte dans un système de type ou un environnement. Dans notre bibliothèque, un *niveau* est représenté par un entier naturel. Par conséquent, un niveau ne peut descendre en dessous de 0, doit pouvoir être incrémenté, décrémenté et deux niveaux doivent pouvoir être comparés. L'ensemble des niveaux est représenté par le module `Lvl` défini ci-dessous.

```
Module Lvl <: Orders.OrderedTypeFull := OrdersEx.Nat_as_OT.
```

Le cœur de la bibliothèque `DeBrLevel` consiste en une collection d'interfaces dont chacune représente des contraintes supplémentaires sur la structure nivelée représentée. Plus précisément, il y a quatre interfaces principales : la première contient la propriété de bonne formation, la deuxième la fonction d'incrémentation, la troisième contient les deux précédentes pour définir une structure nivelée et enfin une quatrième qui est une variante de la troisième. Nous montrons et expliquons les quatre interfaces dans cette section.

Propriété de bonne formation

Une structure nivelée doit respecter une propriété de *bonne formation*. Cette propriété est notée `Wf` et doit être définie par l'utilisateur via le module type `IsWF` ci-dessous. `Wf` prend en paramètre un niveau et un élément nivelé `t`, qui a au moins une égalité. La définition de `Wf` doit satisfaire une propriété d'affaiblissement, notée `Wf_weakening`, qui stipule que si un élément t est bien formée sous un niveau m alors tout niveau n supérieur à m satisfait la propriété de bonne formation pour t. C'est une conséquence directe de la remarque A.2.

```
Module Type IsWF (Import T : EqualityType).
  Parameter Wf : Lvl.t -> t -> Prop.
  Parameter Wf_weakening :
    forall (m n: Lvl.t) (t: t), Lvl.le m n -> Wf m t -> Wf n t.

  #[export] Declare Instance Wf_iff :
    Proper (Lvl.eq ==> eq ==> iff) Wf.
End IsWF.
```

Nous définissons la propriété `Wf` pour une variable ci-dessous. Cette définition vient de la règle `WF-Var` donnée en amont. En effet, la seule contrainte de bonne formation d'une variable est d'être strictement inférieure à n.

```
Definition Wf (n m: Lvl.t) := m < n.
```

Exemple A.9. *Soit* `t` *un type inductif modélisant un lambda-calcul non typé ci-dessous. Les variables sont représentées par des niveaux, par conséquent la définition d'une abstraction ne contient plus de variables.*

```
Inductive t : Type :=
| tm_var : Lvl.t -> t
| tm_app : t -> t -> t
| tm_abs : t -> t.

Inductive Wf : Lvl.t -> t -> Prop :=
| wf_var n m : m < n ->  Wf n (tm_var m)
| wf_app n e1 e2 : Wf n e1 -> Wf n e2 ->  Wf n (tm_app e1 e2)
| wf_abs n e1 : Wf (S n) e1 -> Wf n (tm_abs e1).
```

Une application est bien formée sous n si chaque sous-terme est bien formé. Une variable est bien formée si elle est strictement inférieure à n. Et enfin une abstraction est bien formée sous n si son sous-terme est bien formé sous $n+1$.

Fonction d'incrémentation

Comme suggéré dans la section A.2.1, il est nécessaire d'avoir une fonction permettant d'incrémenter des niveaux selon un certain seuil. Pour cela, nous avons défini une interface nommée `HasShift` qui contient la signature de la fonction d'incrémentation ainsi que les propriétés qu'elle doit satisfaire.

```
Module Type HasShift (Import T : EqualityType).
  Parameter shift : Lvl.t -> Lvl.t -> t -> t.

  Section specs.

  Variable m n k p : Lvl.t.
  Variable t t' : t.

  Parameter shift_zero_refl: eq (shift m 0 t) t.

  Parameter shift_trans :
    eq (shift m n (shift m k t)) (shift m (n + k) t).

  Parameter shift_permute :
    eq (shift m n (shift m k t)) (shift m k (shift m n t)).

  Parameter shift_unfold :
    eq (shift m (n + k) t) (shift (m + n) k (shift m n t)).

  Parameter shift_unfold_1 :
    n <= k -> k <= p ->
    eq (shift k (p - k) (shift n  (k - n) t)) (shift n (p - n) t).

  Parameter shift_eq_iff :
    eq t t' <-> eq (shift m k t) (shift m k t').

  #[export] Declare Instance shift_eq :
    Proper (Lvl.eq ==> Lvl.eq ==> eq ==> eq) shift.

  End specs.
End HasShift.
```

La première propriété, c'est-à-dire `shift_zero_refl`, établit qu'incrémenter

de zéro revient à ne rien changer dans la structure nivelée, qu'importe son seuil. Les propriétés `shift_trans` et `shift_permute` définissent les interactions possibles entre deux fonctions d'incrémentation *ayant le même seuil*. Les deux propriétés `shift_unfold` et `shift_unfold_1` établissent comment il est possible de « dérouler » une fonction d'incrémentation en plusieurs. Finalement, la propriété `shift_eq_iff` énonce que deux structures nivelées égales le sont toujours après application de la *même* fonction d'incrémentation. Pour un niveau, la fonction d'incrémentation est définie comme suit :

```
Definition shift (n m k : Lvl.t) := if (n <= k) then m + k else k.
```

Exemple A.10. *Soit* `t` *notre définition du lambda-calcul de l'exemple A.9, la fonction d'incrémentation se définit comme suit :*

```
Fixpoint shift (n m: Lvl.t) (t: t) :=
  match t with
    | tm_var x => if (n <= x) then m + x else x
    | tm_abs t1 => tm_abs (shift n m t1)
    | tm_app t1 t2 => tm_app (shift n m t1) (shift n m t2)
  end.
```

Les arguments ne sont pas impactés par le passage sous une abstraction, car cette fonction n'a pas pour but de prendre en considération la bonne formation ou non du terme.

Structure nivelée

Une structure nivelée comporte, au minimum, une fonction d'incrémentation, une propriété de bonne formation et des propriétés qui contraignent l'interaction entre les deux. Tout cela est rassemblé dans une interface nommée `IsLeveled` que nous fournissons ci-dessous.

```
Module Type IsLeveled (Import T : EqualityType) <: HasShift T
                                                <: IsWF T.
  Include HasShift T <+ IsWF T.

  Section specs.

  Variable m n k p : Lvl.t.
  Variable t : t.
```

```
  Parameter shift_preserves_wf_gen :
    k <= p -> m <= n -> k <= m -> p <= n -> p - k = n - m ->
    Wf m t -> Wf n (shift k (p - k) t).

  Parameter shift_preserves_wf :
    Wf k t -> Wf (k + p) (shift k p t).

  Parameter shift_preserves_wf_1 : m <= n ->
    Wf m t -> Wf n (shift m (n - m) t).

  End specs.
End IsLeveled.
```

Les trois lemmes sont des variantes d'une même propriété qui établit que la propriété de bonne formation est préservée pour certaines applications de la fonction d'incrémentation.

Exemple A.11. *Soit* `t` *un terme, défini ci-dessous, bien formé sous 1. Le niveau 1 représente la variable associée à l'abstraction et le niveau 0 est une variable libre.*

```
t = tm_abs (tm_app (tm_var 0) (tm_var 1))
```

Si nous appliquons `shift 1 3` *à* `t`*, nous avons le résultat suivant :*

```
shift 1 3 t = tm_abs (tm_app (tm_var 0) (tm_var 4))
```

Selon le lemme `shift_preserves_wf`*, le résultat est bien formé sous 4. Le niveau 0 représente toujours une variable libre, mais la variable liée est maintenant représentée par 4. Nous ne perdons pas le sens malgré le passage de 1 à 4.*

Une structure nivelée n'est pas forcément un ensemble de termes contenant l'équivalent d'une abstraction. En général, lors du développement d'un projet avec des niveaux, il faut des structures pour stocker des données qui peuvent utiliser la représentation par niveaux. Ne pas avoir de lieurs dans la structure permet d'avoir une contrainte plus forte entre la fonction d'incrémentation et la propriété de bonne formation. Dans ce cas, le niveau de bonne formation est strictement supérieur à tous les niveaux présents dans la structure. Par conséquent, toutes applications d'une fonction d'incrémentation avec un seuil égal au niveau de bonne formation ne modifient pas le terme. Cette propriété est encapsulée dans une interface nommée `IsBindlessLeveled` que nous fournissons ci-dessous.

```
Module Type IsBindlessLeveled
```

```
            (Import T : EqualityType) <: IsLeveled T.
  Include IsLeveled T.

  Parameter shift_wf_refl :
    forall m k t, Wf m t -> eq (shift m k t) t.
End IsBindlessLeveled.
```

Remarque A.3. *Cette propriété est aussi respectée pour tout seuil supérieur ou égal à* `m`*.*

Afin de faciliter l'utilisation de ces deux interfaces dans différents cas, nous avons créé deux alias `IsLvL` et `IsBdlLvl` respectivement pour `IsLeveled` et `IsBindlessLeveled`. De plus, `T`, étant le module qui représente la structure, est limité à l'interface `EqualityType` dans nos définitions précédentes, mais peut s'étendre à d'autres. Pour les utiliser, nous avons prédéfini des variantes en fonction de l'interface instanciée par `T` qui suivent la convention de nommage suivante :

`(IsLvl|IsBdlLvl)(ET|DT|OT)(WL)?`

Le suffixe `ET` signifie que `T` est contient une égalité. Il peut être remplacé par `DT` quand `T` est un type décidable et `OT` quand il est ordonné. Il est aussi possible de spécifier si l'égalité de la structure nivelée peut se ramener à l'égalité de base de Rocq en rajoutant `WL` à la fin du nom de l'interface. L'ensemble des variantes est donné dans le tableau A.2.

Exemple A.12. *Par exemple l'interface* `IsBdlLvlOTWL`*, si instanciée, représente une structure nivelée ordonnée sans lieur dont l'égalité implique celle de base de* Rocq*.*

Le premier module à instancier une de ces interfaces est le module `Level` qui représente un niveau avec la fonction d'incrémentation, la propriété de bonne formation et les propriétés qui leur sont associées. Plus précisément, il étend le module `IsBdlLvlOTWL`.

A.2.3 Exemple de structure prédéfinie : Dictionnaire

La structure nivelée la plus développée pour la gestion des niveaux dans la bibliothèque `DeBrLevel` est le dictionnaire. Nous nous basons sur la bibliothèque `MMaps` disponible sur le dépôt git `https://github.com/coq-community/mmaps` pour la modélisation sans niveau des dictionnaires. Cette structure permet de définir facilement des contextes et des environnements, ce qui correspond au besoin

TABLE A.2 – Variantes des interfaces `IsLvl` et `IsBdlLvl`.

Interface	Avec lieur	Sans lieur	Avec égalité	Décidable	Ordonné	Implique l'égalité de Leibniz
`IsLvlET`	x		x			
`IsLvlETWL`	x		x			x
`IsBdlLvlET`		x	x			
`IsBdlLvlETWL`		x	x			x
`IsLvlDT`	x		x	x		
`IsLvlDTWL`	x		x	x		x
`IsBdlLvlDT`		x	x	x		
`IsBdlLvlDTWL`		x	x	x		x
`IsLvlOT`	x		x	x	x	
`IsLvlOTWL`	x		x	x	x	x
`IsBdlLvlOT`		x	x	x	x	
`IsBdlLvlOTWL`		x	x	x	x	x

d'une formalisation d'un langage. Avant de définir des interfaces pour gérer les dictionnaires nivelées, nous avons étendu les `MMaps` dans une collection d'interfaces et de modules préfixés par le nom `MapExt`. Cette extension nous a permis d'ajouter quelques propriétés complémentaires ainsi que des fonctions non présentes dans la bibliothèque initiale. Celle qui nous intéresse le plus pour la suite est la fonction `new_key`. Définie uniquement pour des dictionnaires qui ont pour clé des entiers naturels, elle prend un dictionnaire m et retourne la plus grande clé de m incrémenté de 1 (si m est vide alors elle retourne 0). Plus précisément, nous définissons `new_key` en fonction de `max_key` comme suit :

```
Definition max_key (m : t) : nat :=
  foldkeys Nat.max m 0.

Definition new_key (m : t) : nat :=
  if M.is_empty m then 0 else S (max_key m).
```

Nous avons différencié trois familles de dictionnaires : 1. celles qui contiennent des structures nivelées uniquement pour les clés ; 2. celles qui contiennent des structures nivelées uniquement pour les valeurs ; 3. celles qui contiennent des structures nivelées pour les clés et les valeurs. Nous ajoutons à cela une dernière famille qui est une spécification de la première famille où les clés sont des niveaux. Ces familles suivent la convention de nommage suivante :

`(IsLvl|IsBdlLvl)Map(K|D|KD|LVL|LVLD)Interface`

Le choix entre `IsLvl` et `IsBdlLvl` informe si les éléments nivelées du dictionnaire comportent ou non des lieurs. `K` signifie que les clés sont nivelées, `D` signifie que les valeurs sont nivelées et `LVL` signifie que les clés sont des niveaux. `K` et `LVL` ne sont pas équivalent, car `K` signifie une structure nivelée qui peut être plus complexe qu'un niveau, une paire de niveau par exemple. L'ensemble des interfaces définies est résumé dans le tableau A.3. Le module qui instancie une de ces interfaces suit la convention de nommage suivante :

`Make(Lvl|BdlLvl)Map(K|D|KD|LVL|LVLD)`

Exemple A.13. *Soit* `Lambda` *le module implémentant les lambda-termes. Il instancie l'interface* `IsLvlET`*. L'ensemble des contextes qui lie une variable avec un terme peut être défini par le module suivant :*

```
Module Contxt := MakeLvlMapLVLD Lambda.
```

TABLE A.3 – Variantes des interfaces pour les dictionnaires.

Interface	Avec lieur	Sans lieur	Clé nivelée	Valeur nivelée	Niveau comme clé
IsLvlMapKInterface	x		x		
IsLvlMapDInterface	x			x	
IsLvlMapKDInterface	x		x	x	
IsLvlMapLVLInterface	x		x		x
IsLvlMapLVLDInterface	x		x	x	x
IsBdlLvlMapKInterface		x	x		
IsBdlLvlMapDInterface		x		x	
IsBdlLvlMapKDInterface		x	x	x	
IsBdlLvlMapLVLInterface		x	x		x
IsBdlLvlMapLVLDInterface		x	x	x	x

Pour chaque famille, la fonction d'incrémentation et la propriété de bonne formation fonctionnent de la même manière :

— appliquer une fonction d'incrémentation sur un dictionnaire revient à l'appliquer sur tous ses éléments nivelés (clés et/ou valeurs).

— un dictionnaire est bien formée sous un certain niveau n si tous les éléments nivelés qui la compose le sont.

Exemple A.14. *Par exemple, pour le module* `MakeBdlLvlMapD`*, la fonction d'incrémentation et la propriété de bonne formation sont définies comme suit :*

```
Definition Wf (n : Lvl.t) (m : t) :=
M.fold (func k d acc => Data.Wf n d /\ acc) m True.

Definition shift (n p : Lvl.t) (m: t) :=
M.fold (fun k d acc => M.add k (Data.shift n p d) acc) m M.empty.
```

L'ensemble des modules proposent des lemmes qui mettent en relation `shift` et `WF` avec le reste des fonctions et propriétés sur les dictionnaires, c'est-à-dire l'ajout, la suppression, l'appartenance, la recherche, etc. Si le lecteur souhaite voir plus en profondeur cette structure, elle se trouve dans le dossier `Map` de la bibliothèque.

A.3 Cas d'étude : WORMHOLES mécanisé

Dans le chapitre 4, nous avons défini une variante du langage WORMHOLES. Afin de nous assurer que nos modifications corrigent effectivement les erreurs identifiées dans l'article [102] de D. Winograd-Cort et P. Hudak tout en préservant les propriétés du langage, nous avons décidé de formaliser dans l'assistant de preuve ROCQ cette variante [59]. La formalisation [60] est disponible sur git à l'adresse suivante :

`https://github.com/JordanIschard/Mechanized-Wormholes`.

Le langage WORMHOLES contient des variables et des identifiants de ressources. Un identifiant de ressources peut être libre ou lié à un opérateur *wormhole*. Une mauvaise gestion des identifiants de ressources dans la formalisation peut provoquer une capture de noms.

Exemple A.15. *Soient* $t_1 = \lambda x.wormhole[r, r_w](0; x \ggg rsf[r])$ *et* $t_2 = rsf[r]$ *deux termes. Dans* t_1*, les ressources* r *et* r_w *sont toutes les deux liées à l'opérateur*

wormhole, au contraire de t_2, pour laquelle r est une ressource libre. Appliquer t_2 à t_1 donne le résultat suivant :

$$\begin{aligned} t_1\, t_2 &= \lambda x.\mathit{wormhole}[r, r_w](0; x \ggg \mathit{rsf}[r])\,(\mathit{rsf}[r]) \\ &= [x := \mathit{rsf}[r]]\mathit{wormhole}[r, r_w](0; x \ggg \mathit{rsf}[r]) \\ &= \mathit{wormhole}[r, r_w](0; \mathit{rsf}[r] \ggg \mathit{rsf}[r]) \end{aligned}$$

Dans le terme résultant, la ressource r provenant de t_2 devient liée au constructeur wormhole.

Nous avons choisi d'utiliser les niveaux de *De-Bruijn* pour représenter les ressources du langage WORMHOLES. Ce choix est motivé par une tentative infructueuse d'utiliser la représentation localement non-nommée et plusieurs observations de notre part. Tout d'abord, la définition de la représentation localement non-nommée décrite dans l'article d'A. Charguéraud [25] propose la règle de typage de l'abstraction suivante :

$$\frac{(\forall x, x \notin L \to \Gamma[x \to \alpha] \vdash t^x : \beta)}{\Gamma \vdash \lambda.t : \alpha \to \beta}\ \texttt{Ty-Abs-LN}$$

La notation t^x signifie que la variable x remplace tous les indices associés à l'abstraction $\lambda.t$. Nous voyons que l'hypothèse de bon typage du sous-terme est quantifié par rapport à une liste d'identifiant. L'adapter à l'opérateur *wormhole* donne la règle suivante :

$$\frac{\begin{array}{l}\Gamma \cdot \Re \vdash t_1 : \tau \\ (\forall r_{get}\, r_{set}, \quad r_{get} \notin L \wedge r_{set} \notin L \wedge r_{set} \neq r_{get} \to \\ \qquad \{r_{set}; r_{get}\} \cap (\mathit{refs}\ \alpha \cup \mathit{refs}\ \beta) = \varnothing \wedge R = R' \backslash \{r_{get}; r_{set}\} \\ \qquad \wedge\ \Gamma \cdot \Re[r_{get} \mapsto \langle \mathit{Unit}, \tau \rangle][r_{set} \mapsto \langle \tau, \mathit{Unit} \rangle] \vdash t_2 : \alpha \overset{R'}{\rightsquigarrow} \beta)\end{array}}{\Gamma \cdot \Re \vdash \mathit{wormhole}[r_{get}, r_{set}](t_1; t_2) : \alpha \overset{R}{\rightsquigarrow} \beta}\ \texttt{Ty-Wh-LN}$$

En plus de l'hypothèse de bon typage du sous-terme, il y a aussi toutes les contraintes sur les ressources liées et sur la formation de R. Le problème de cette règle est que le schéma d'induction se casse à cause de la conjonction contenue sous le quantificateur universel. Par conséquent, nous avons opté pour une représentation plus simple que nous pourrions adapter plus facilement aux identifiants de ressources. Le choix entre indices et niveaux a été fait au regard de deux observations. Premièrement, les identifiants de ressources ne se substituent pas lorsque nous « ouvrons » un terme $\mathit{wormhole}[r_r, r_w](t_1; t_2)$, c'est-à-dire lors de l'application de la règle `FT-Wh`. Par conséquent, utiliser les indices qui sont

définis en fonction de leur lieur (dans le lambda-calcul c'est l'abstraction et dans WORMHOLES c'est l'opérateur *wormhole*) semble peu naturel, alors que les niveaux, eux, se basent sur la racine du terme et le nombre de lieurs « traversés ». Deuxièmement, dans la littérature, peu de personnes se sont penchées sur les niveaux de *De-Bruijn*, par conséquent, la création d'une bibliothèque ROCQ pour son utilisation semblait être un apport intéressant pour la communauté.

Les identifiants de ressource sont donc des *niveaux*, c'est-à-dire des *entiers naturels*. Dans cette section, nous montrons une partie des modifications apportées par la gestion des niveaux aux règles et définitions présentées dans la section 4.1. Nous nous focalisons principalement sur la définition de la syntaxe, la sémantique, la substitution et la transition fonctionnelle. Dans la suite, les noms de motifs dans les définitions inductives sont légèrement modifiées par rapport à la formalisation accessible sur le dépôt afin de rendre plus lisible ce que nous présentons dans cette section.

A.3.1 Syntaxe

La syntaxe regroupe les variables, les ressources, les termes et les types. Des explications sur la syntaxe sont données dans la section 4.1.1. Les variables sont représentées par des entiers naturels, c'est-à-dire qu'ils proviennent du module `Nat_as_OT`, et les ressources sont, comme dit précédemment, des niveaux provenant du module `Level`. L'ensemble des termes est défini dans le module `Term` qui instancie l'interface `IsLvlDTWL`. Il est modélisé par un type inductif `t` présenté ci-dessous.

```
Inductive t : Type :=
  | unit   : t
  | var    (x : nat) : t
  | abs    (x : nat) (t : t) : t
  | app    (t1 t2 : t) : t
  | pair   (t1 t2 : t) : t
  | fix    (t : t) : t
  | fst    (t : t) : t
  | snd    (t : t) : t
  | arr    (t : t) : t
  | first  (t : t) : t
  | rsf    (r : Lvl.t) : t
  | comp   (t1 t2 : t) : t
  | wh     (t1 t2 : t) : t
```

Les ressources liées à un opérateur *wormhole*, ici représenté par `wh`, ne sont plus associées syntaxiquement, c'est-à-dire qu'elles ne sont plus présentes dans la définition d'un terme constitué de l'opérateur *wormhole*, car elles seront déduites grâce à la propriété de bonne formation que nous lui associons. Afin d'étayer ce propos, nous donnons la définition de bonne formation pour les termes ci-dessous.

```
Inductive Wf : Lvl.t -> t -> Prop :=
  | wf_unit (k : Lvl.t) : Wf k unit
  | wf_var  (k : Lvl.t) (x : nat) : Wf k (var x)
  | wf_abs  (k : Lvl.t) (x : nat) (t : t) :
        Wf k t -> Wf k (abs x t)

  | wf_app  (k : Lvl.t) (t1 t2 : t) :
        Wf k t1 -> Wf k t2 -> Wf k (app t1 t2)
  | wf_pair (k : Lvl.t) (t1 t2 : t) :
        Wf k t1 -> Wf k t2 -> Wf k (pair t1 t2)

  | wf_fix   (k : Lvl.t) (t : t) : Wf k t -> Wf k (fix t)
  | wf_fst   (k : Lvl.t) (t : t) : Wf k t -> Wf k (fst t)
  | wf_snd   (k : Lvl.t) (t : t) : Wf k t -> Wf k (snd t)
  | wf_arr   (k : Lvl.t) (t : t) : Wf k t -> Wf k (arr t)
  | wf_first (k : Lvl.t) (t : t) : Wf k t -> Wf k (first t)

  | wf_rsf (k : Lvl.t) (r : resource) :
        Resource.Wf k r ->  Wf k (rsf r)

  | wf_comp (k : Lvl.t) (t1 t2 : t) :
        Wf k t1 -> Wf k t2 -> Wf k (comp t1 t2)
  | wf_wh   (k : Lvl.t) (t1 t2 : t) :
        Wf k t1 -> Wf (S (S k)) t2 -> Wf k (wh t1 t2).
```

Soit `t = wh t1 t2` un terme, avec `t1` et `t2` ses sous-termes. Les ressources liées à `t` dépendent du niveau de bonne formation. Si l'entier `n` satisfait la propriété `Wf n t` alors les ressources liées seront `n` et `n+1`. Cela est cohérent avec le fait qu'un opérateur *wormhole* est lié à deux ressources et la définition de `Wf` reflète cela en incrémentant de deux le niveau de `t2`, indiquant donc que le niveau courant et le niveau courant plus 1 sont réservés aux ressources liées.

Exemple A.16. *Soit* `t` *le terme suivant :*

```
t = wh unit (first (comp (arr (abs 0 (pair unit 0))) (rsf 1)))
```

Ce terme est bien formé sous `0`, c'est-à-dire que `Wf 0 t` *est satisfait. L'identifiant de la ressource de « lecture » du lieur est `0` et celui de l'« écriture » est `1`. Nous pouvons noter que ce terme est aussi bien formé pour tout entier naturel, cependant sons sens change. En effet, si nous regardons `t` en supposant qu'il est bien formé pour `1` alors la ressource de « lecture » deviendra `1`. Par conséquent,* `rsf 1` *ne signifie pas la même interaction en fonction du niveau que nous utilisons pour l'interpréter.*

Pour compléter le module `Term`, nous fournissons la fonction d'incrémentation. Intuitivement, elle se propage dans les sous-termes et applique la fonction d'incrémentation des niveaux sur la ressource `r` lorsqu'elle arrive à un terme de la forme `rsf r`.

```
Fixpoint shift (lb k : Lvl.t) (e : t) : t :=
  match e with
    | rsf r => rsf (Lvl.shift lb k r)

    | abs x t => abs x (shift lb k t)
    | fix t   => fix   (shift lb k t)
    | fst t   => fst   (shift lb k t)
    | snd t   => snd   (shift lb k t)
    | arr t   => arr   (shift lb k t)
    | first t => first (shift lb k t)

    | app  t1 t2 => app  (shift lb k t1) (shift lb k t2)
    | pair t1 t2 => pair (shift lb k t1) (shift lb k t2)
    | comp t1 t2 => comp (shift lb k t1) (shift lb k t2)
    | wh   t1 t2 => wh   (shift lb k t1) (shift lb k t2)
    | _ => e
  end.
```

La modélisation des types suit la même procédure. Le module des types, nommé `Typ`, instancie l'interface `IsBdlLvlDTWL`. Ici, nous précisons uniquement le type inductif représentant les types.

```
Module Resources <: IsBdlLvlFullOTWL := MakeLVL.

Module Typ <: IsBdlLvlDTWL.
  Inductive t : Type :=
    | unit : t
    | func  (a b : t) : t
```

```
    | prod  (a b : t) : t
    | sfunc (a b : t) (R : Resources.t) : t.
  ...
End Typ.
```

Pour rappel, `Bdl` signifie « bindless » ce qui veut dire que la structure en question ne contient pas de lieur. En effet, l'ensemble des types n'en contient pas, ce qui pose la question de l'utilité de l'instanciation d'une interface de niveau sur le module, cependant le type des fonctions de signal porte un ensemble de ressources ce qui l'oblige à instancier une interface de gestion des niveaux. Cet ensemble provient du module `Resources` qui instancie l'interface `IsBdlLvlFullOTWL` et étend le module `MakeLVL` qui, pour rappel, est un module prédéfini dans `DeBrLevel` qui implémente les ensembles de niveaux.

A.3.2 Sémantique statique

La sémantique statique met en relation un terme et un type en fonction de deux contextes : un contexte de variables et un contexte de ressources. De plus, un terme bien typé garantit une utilisation correcte de ses ressources. La section 4.1.2 fournit les règles de la sémantique statique ainsi que des explications sur son fonctionnement. Dans un premier temps, nous décrivons la modélisation des contextes. Dans un second temps, nous montrons les modifications apportées à la sémantique statique.

Contextes

Les deux types de contextes sont modélisés par des dictionnaires (voir section A.2.3). Un contexte de variables ne contient pas de niveaux dans ses clés, car une variable est seulement un entier naturel, mais ses valeurs en portent, car ce sont des types. Par conséquent, le module `VContext`, représentant les contextes de variables, étend le module `MakeBdlLvlMapD`.

```
Module VContext <: IsBdlLvlET.
  Include MakeBdlLvlMapD Var Typ.
  ...
End VContext.
```

De son côté, un contexte de ressources a des niveaux comme clés et des paires de types comme valeurs. Nous utilisons donc le module `MakeBdlLvlMapLVLD` pour définir le module `RContext` représentant les contextes de ressources.

```
Module RContext <: IsBdlLvlET.
  Include MakeBdlLvlMapLVLD PairTyp.
  ...
End RContext.
```

De plus, l'ensemble des paires de types est définie directement en utilisant le module `IsBdlLvlPairETWL` prédéfini dans la bibliothèque pour gérer des paires de structures nivelées.

```
Module PairTyp <: IsBdlLvlETWL := IsBdlLvlPairETWL Typ Typ.
```

Le module `RContext` est équipé d'une fonction `new_key` prédéfinie dans le module qu'il implémente. Cette fonction prend un contexte de ressources et retourne soit 0 si le contexte est vide soit le plus grand identifiant de ressource plus 1. Cela nous donne un niveau de bonne formation pour toutes les ressources présentes dans l'ensemble des clés d'un contexte.

Règles

En plus de garantir le bon typage d'une expression, la modélisation de cette propriété, nommé `well_typed`, contient des contraintes supplémentaires dans la sémantique statique afin de vérifier aussi la bonne formation de l'expression et de son type associé en fonction de ses contextes. Plus précisément, nous souhaitons pouvoir démontrer cet énoncé :

```
Theorem well_typed_implies_Wf (G : VContext.t) (Re : RContext.t)
                              (t : Term.t) (a : Typ.t) :

  Wf (new_key Re) G ->  Wf (new_key Re) Re ->
  well_typed G Re t a -> Wf (new_key Re) t /\ Wf (new_key Re) a.
```

Intuitivement, cela revient à dire que tous les composants de la propriété de bon typage sont d'accord sur le nombre de ressources libres dans l'expression. Seulement trois des treize règles du système de type énoncées dans les figures 4.4 et 4.5 sont modifiées pour accomplir cet objectif. Elles sont données ci-dessous.

```
Inductive well_typed : VContext.t -> RContext.t ->
                                     Term.t -> Typ.t -> Prop :=

  | wt_abs (G : VContext.t) (Re : RContext.t)
           (x : nat) (t : Term.t) (a b : Typ.t) :
```

```
    well_typed (add x a G) Re t b ->
    Wf (new_key Re) a -> well_typed G Re (abs x t) (func a b)

  | wt_first (G : VContext.t) (Re : RContext.t)
             (R : Resource.t) (t : Term.t) (a b c : Typ.t) :

    well_typed G Re t (sfunc a b R) -> Wf (new_key Re) c ->
    well_typed G Re (first t) (sfunc (prod a c) (prod b c) R)

  | wt_wh (G : VContext.t) (Re : RContext.t)
          (R R' : Resource.t) (t1 t2 : Term.t) (a b c : Typ.t) :

    (R = R' \ {(new_key Re);(S (new_key Re))}) ->
    Wf (new_key Re) a -> Wf (new_key Re) b ->
    well_typed G Re t1 c ->
    well_typed G (add (S (new_key Re)) (c,unit)
                   (add (new_key Re) (unit,c) Re))
               t2 (sfunc a b R') ->
    well_typed G Re (wh t1 t2) (sfunc a b R)
  ...
```

Dans les règles `wt_abs` et `wt_first`, qui sont respectivement la règle de typage de l'abstraction et de la fonction de signal `first`, nous ajoutons une contrainte de bonne formation sur le type choisi arbitrairement. Dans la première règle, c'est `a` qui ne dépend pas du type associé à `t` et qui est ajouté dans le contexte. Dans le cas de la seconde règle, il s'agit de `c`. N'ayant aucune information pouvant être récupéré du typage des sous-termes dans ces deux cas, nous spécifions explicitement la contrainte.

Dans la règle `wt_wh`, qui modélise la règle de bon typage d'un terme *wormhole*, il y a deux modifications. La première est l'utilisation de `new_key` pour définir les deux ressources liées au terme. C'est l'implication directe de l'utilisation des niveaux. La seconde est l'ajout des contraintes de bonne formation de `a` et `b` selon `(new_key Re)`. En soi, il n'est pas nécessaire d'ajouter cette contrainte si nous souhaitons uniquement garantir la bonne formation du terme et de son type. Cependant, il est important de noter que les contraintes sur les ressources que peuvent contenir `a` et `b` ne sont pas présentes. Pour rappel, afin de corriger l'échappement des ressources de leur portée, nous avons ajouté une contrainte qui stipule que les types ne peuvent pas contenir les ressources liées au terme *wormhole*. Dans notre cas, `a` et `b` ne doivent pas contenir `(new_key Re)` et `(S`

`(new_key Re)`). Mais grâce à notre contrainte de bonne formation, cette propriété est garantie, car pour une structure nivelée sans lieur comme les types, dire qu'un type est bien formée sous n revient à dire que toutes ses ressources sont strictement inférieures à n.

A.3.3 Transition d'évaluation

La transition d'évaluation est une sémantique opérationnelle à petit pas qui nécessite une fonction de substitution. La fonction de substitution « classique » ne peut pas être utilisée comme telle dans la formalisation. En effet, tout terme est cohérent en fonction d'un niveau et la fonction de substitution se doit de garantir sa cohérence.

Exemple A.17. *Soient deux termes* `t1`, `t2` *définis ci-dessous et bien formés sous* `0`.

```
t1 = wh unit (comp (rsf 0) x)
t2 = wh unit (rsf 0)
```

Nous reprenons la notation de la substitution du lambda-calcul. Avec une substitution naïve nous avons :

```
[x := t2] t1 = wh unit (comp (rsf 0) (wh unit (rsf 0)))
```

La substitution doit préserver le niveau de bonne formation ce qui implique que le résultat doit aussi être bien formé sous `0`. *Malheureusement, la ressource liée* `0` *de* `t2` *a été capturée par le lieur de* `t1`.

Il faut donc définir une fonction de substitution personnalisée qui, en plus des paramètres classiques, prend deux niveaux n et m. Le niveau n représente le niveau de bonne formation du terme substituant, ici `v`. Le niveau m sert de compteur, lorsque nous entrons dans un terme *wormhole*, ce compteur est incrémenté de deux car deux niveaux sont réservés aux ressources liées qui sont inconnues pour le substituant.

```
Fixpoint subst (n m : Lvl.t) (x : nat) (v t : t) : t :=
match t with
  | var y   => if (x =? y) then shift n m v else t
  | abs y t => if (x =? y) then abs y t
                           else abs y ([x := v ~ n - m] t)

  | fst t   => fst ([x := v ~ n - m] t)
```

```
  | snd t   => snd ([x := v ~ n - m] t)
  | fix t   => fix ([x := v ~ n - m] t)
  | arr t   => arr ([x := v ~ n - m] t)
  | first t => first ([x := v ~ n - m] t)

  | app t1 t2  => app ([x := v ~ n - m] t1) ([x := v ~ n - m] t2)
  | pair t1 t2 => pair ([x := v ~ n - m] t1) ([x := v ~ n - m] t2)
  | comp t1 t2 => comp ([x := v ~ n - m] t1) ([x := v ~ n - m] t2)
  | wh t1 t2   => wh ([x := v ~ n - m] t1)
                     ([x := v ~ n - {S (S m)}] t2)

  | _ => t
end
where "'[' x ':=' v '~' n '-' m ']' t" := (subst n m x v t).
```

Exemple A.18. *Reprenons les termes* `t1` *et* `t2` *définis dans l'exemple précédent. En supposant que* `t2` *est bien formé sous* 0 *et que le compteur est aussi à* 0*, nous pouvons définir la substitution suivante :*

```
  [x := t2 ~ 0 - 0] wh unit (comp (rsf 0) x)
= wh ([x := t2 ~ 0 - 0] unit) ([x := t2 ~ 0 - 2] (comp (rsf 0) x))
= wh unit (comp ([x := t2 ~ 0 - 2] rsf 0) ([x := t2 ~ 0 - 0]  x))
= wh unit (comp (rsf 0) (shift 0 2 (wh unit (rsf 0))))
= wh unit (comp (rsf 0) (wh (shift 0 2 unit) (shift 0 2 (rsf 0))))
= wh unit (comp (rsf 0) (wh unit (rsf 2)))
```

Le résultat, toujours bien formé sous 0*, contient bien deux ressources différentes :* 0 *et* 2*. Le niveau* 0 *étant lié au lieur réactif externe et* 2 *à l'interne.*

Remarque A.4. *Il est intéressant de noter que nous pourrions aussi incrémenter le substituant à chaque rencontre d'un opérateur wormhole. De cette manière, le niveau* m *n'est plus nécessaire.*

Les règles de la transition d'évaluation ne changent quasiment pas, par rapport à celles exprimées dans la section 4.1.3, excepté la présence d'un niveau k. Dans la formalisation, cette transition est un type inductif noté `evaluate k t t'`. Ce niveau k est nécessaire pour une utilisation cohérente de la fonction de substitution. En effet, la fonction de substitution est utilisée dans les règles $\texttt{ET-Fix}_v$ et $\texttt{ET-App}_v$ et nécessite donc de fournir les niveaux n et m. La propriété `value(v)` établit que `v` est une valeur du langage.

```
Inductive evaluate : Lvl.t -> Term.t -> Term.t -> Prop :=
  | eT_appv (k : Lvl.t) (x : nat) (t v : Term.t) :
    value(v) -> evaluate k (app (abs x t) v) ([x:= v ~ k - 0] t)

  | eT_fixv (k : Lvl.t) (f : nat) (t : Term.t) :
    evaluate k (fix (abs f t)) ([f := (fix (abs f t)) ~ k - 0] t)
  ...
```

Le niveau m est initialisé à 0, car il représente le nombre de ressources liées rencontrées durant la descente dans le terme. Cependant, il n'est pas possible de déduire n, le niveau de bonne formation du terme courant. Par conséquent, cette information doit être portée par la transition, d'où la présence de k. Dans le cas de l'évaluation du second sous-terme de *wormhole* il faut incrémenter ce niveau k, car deux ressources liées deviennent libres. Cette contrainte est capturée dans la règle suivante :

```
Inductive evaluate : Lvl.t -> Term.t -> Term.t -> Prop :=
    ...
  | eT_wh2 (k : Lvl.t) (v t t' : Term.t) :
    value(v) -> evaluate (S (S k)) t t' ->
                evaluate k (wh v t) (wh v t')
```

La transition d'évaluation préserve le typage et progresse. Pour rappel, une sémantique opérationnelle *progresse* si pour tout terme bien typé, soit c'est une valeur, soit il est possible d'effectuer une étape d'évaluation. Ces faits sont établis dans le théorème 4.7 et les théorèmes 4.8 et 4.10 présentés dans la section 4.1.3. La version mécanisée du théorème de progression ne diffère pas de la version déjà présentée. À la différence de la préservation du typage après substitution, et plus généralement après évaluation, qui doit contenir une hypothèse de bonne formation supplémentaire pour chaque contexte. Cela permet d'utiliser le théorème `well_typed_implies_Wf`. En plus des propriétés du langage, il faut définir le comportement de la transition d'évaluation et de la substitution avec la propriété de bonne formation et la fonction d'incrémentation. La première propriété que nous établissons est la préservation de la bonne formation par la substitution et l'évaluation.

```
Lemma subst_preserves_Wf
      (m n : Lvl.t) (y : nat) (v t : Term.t) :
  m >= n -> Wf m t -> Wf n v -> Wf m ([y := v ~ n - {m - n}] t).
Proof. (* omit *) Qed.
```

```
Lemma evaluate_preserves_wf_term (k : Lvl.t) (t t' : Term.t) :
  Wf k t -> evaluate k t t' -> Wf k t'.
Proof. (* omit *) Qed.
```

La seconde propriété que nous vérifions est la propagation de la fonction d'incrémentation aux composants de la substitution ainsi que ceux de l'évaluation.

```
Lemma subst_shift (p k m n : Lvl.t) (y : nat) (v t : Term.t) :
  p <= k ->
  shift p m ([y := v ~ k - n] t) =
  [y := (shift p m v) ~ (k + m) - n] (shift p m t).
Proof. (* omit *) Qed.

Corollary evaluate_shift (m n : Lvl.t) (t t' : Term.t) :
  m <= n -> evaluate m t t' ->
  evaluate n (shift m (n - m) t) (shift m (n - m) t').
Proof. (* omit *) Qed.
```

A.3.4 Transition fonctionnelle

La transition fonctionnelle est une sémantique opérationnelle à grand pas qui représente l'exécution d'un terme sur des entrées durant un instant logique. Dans la formalisation, cette transition est modélisée par une propriété inductive notée `functional (V,st,sf) (V1,st',sf',W)`. La section 4.1.4 détaille son fonctionnement et établit les différentes propriétés qui lui sont associées. Cette transition se repose sur les termes du module `Term` ainsi que deux environnements : l'environnement de ressources et le stockage. Nous commençons par définir ces environnements avant de passer aux règles.

Environnements

Un environnement de ressources, noté V, est un dictionnaire qui met en relation des ressources et des cellules. Une cellule est une structure qui encapsule un terme avec un statut utilisé ou non. Une cellule est modélisée comme un type inductif dans un module `Cell` qui instancie l'interface `IsLvlDTWL`, qui est celle qu'instancie `Term`.

```
Module Cell <: IsLvlDTWL.
  Inductive t : Type :=
    | inp : Term.t -> t
    | out : Term.t -> t.
```

```
  ...
End Cell.
```

L'ensemble des environnements de ressources est donc modélisé par un module nommé `Env` qui instancie l'interface `IsLvlET`.

```
Module Env <: IsLvlET.
  Include MakeLvlMapLVLD Cell.
  ...
End Env.
```

Un stockage est un ensemble de triplets, noté W, qui contient des triplets composés de deux ressources et d'un terme. Il y a deux possibilités pour modéliser un triplet : soit nous utilisons le module `IsLvlPairETWL`, une variante du module utilisé pour le module `PairTyp`, deux fois afin de produire un module représentant des triplets ; soit nous définissons directement un module qui instancie `IsLvlDTWL` avec comme structure un triplet. Nous avons choisi la seconde possibilité, cela évite la création d'un module intermédiaire. Il y a donc un module `Triplet` qui est défini comme suit :

```
Module Triplet <: IsLvlET.
  Definition t : Type := Lvl.t * Lvl.t * Term.t.
  ...
End Triplet.
```

Dans la formalisation, un stockage est modélisé comme une liste de triplets. Le module `Stock` représente cela et instancie l'interface `IsLvlET` et utilise le module `IsLvlListETWL`, un module prédéfini dans la bibliothèque pour des listes de structures nivelées.

```
Module Stock <: IsLvlET.
  Include IsLvlListETWL Triplet.
  ...
End Stock.
```

Règles

La particularité de cette transition est qu'elle retire les lieurs réactifs contenus dans le terme évalué tout en préservant les ressources, contrairement aux variables dans le lambda-calcul, qui sont substituées lors de la β-réduction. Par conséquent, les ressources liées, cachées par la portée de leur lieur, deviennent toutes libres et visibles pour l'ensemble du terme. La transition fonctionnelle est donc critique

pour la gestion des identifiants de ressources, car elle demande une attention toute particulière pour maintenir une cohérence dans le terme et éviter une capture de noms.

Exemple A.19. *Si nous n'appliquons aucune modification aux règles de cette transition, nous pouvons perdre la propriété de bonne formation. Soit* `t` *un terme défini ci-dessous.*

```
t = comp (wh unit (comp (rsf 0) (rsf 1)))
         (wh unit (comp (rsf 0) (rsf 1)))
```

`t` *est un terme bien formé sous* `0`*, par conséquent dans chacun des sous-termes de la composition les ressources liées sont définies par* `0` *et* `1`*. Lorsque qu'une exécution de la transition fonctionnelle est effectuée sur ce terme, le terme résultant, que l'on note* `t1`*, est égal à :*

```
t1 = comp (comp (rsf 0) (rsf 1)) (comp (rsf 0) (rsf 1))
```

`0` *et* `1` *du premier sous-terme wormhole deviennent indistinguable des ressources du second.*

Comme pour le système de types, nous établissons des niveaux de bonne formation pour garder une cohérence globale. Pour cela, nous considérons que le triplet d'entrée, c'est-à-dire `(V,st,sf)`, est bien formée selon `(new_key V)` et nous supposons que le quadruplet de sortie, c'est-à-dire `(V1,st',sf',W)`, fait de même avec `V1`. Plus précisément, nous cherchons à satisfaire le lemme ci-dessous :

```
Lemma functional_preserves_Wf (V V1 : Env.t) (W : Stock.t)
                              (st st' sf sf' : Term.t) :
  functional (V,st,sf) (V1,st',sf',W) ->
  Wf (new_key V) V -> Wf (new_key V) st -> Wf (new_key V) sf ->

  Wf (new_key V1) V1 /\ Wf (new_key V1) st' /\
  Wf (new_key V1) sf' /\ Wf (new_key V1) W /\
  (new_key V) <= (new_key V1).
```

Intuitivement, cela revient à dire que si tous les composants d'entrée sont d'accord sur le nombre de ressources libres alors les composants de sortie le sont aussi et qu'il ne peut pas y en avoir moins. Cela modifie les règles de la composition, de *first* et de *wormhole* présentées dans la figure 4.9. Prenons comme exemple la règle du *first*.

```
Inductive functional :
          (Env.t * Term.t * Term.t) ->
          (Env.t * Term.t * Term.t * Stock.t) -> Prop :=

  | fT_first (v1 v1' v2 t t' : Term.t) (a : Typ.t)
             (V V1 : Env.t) (W : Stock.t) :

    functional (V,v1,t) (V1,v1',t',W) ->
    functional (V,pair v1 v2,first t)
               (V1,
                pair v1' (shift (new_key V)
                                ((new_key V1) - (new_key V)) v2),
                first t',W)
  ...
```

Nous pouvons voir une application de la fonction d'incrémentation sur `v2` avec `(new_key V)` comme seuil et `((new_key V1) - (new_key V))` comme incrément. Le terme `v2` est initialement présent dans les composants d'entrée et est donc bien formé sous `(new_key V)`. Cependant, il se retrouve aussi dans les composants de sortie qui sont eux bien formés sous `(new_key V1)`. Par conséquent, il y a un problème de cohérence, car les autres composants de sortie peuvent potentiellement connaitre plus de ressources libres que `v2`. Pour corriger cela, nous incrémentons toutes les ressources liées de `v2` par la différence entre le nombre de ressources libres dans `V1` et dans `V`.

A.4 Conclusion

Dans ce chapitre, nous avons décrit la bibliothèque Rocq dédiée à la gestion des niveaux de De-Bruijn dans le contexte d'une mécanisation de sémantique d'un langage. Elle se nomme `DeBrLevel` et consiste en une collection de modules et d'interfaces qui permet de définir des structures nivelées. Elle est disponible sur le répertoire git `https://github.com/JordanIschard/DeBrLevel` et contient un ensemble de structures prédéfinies comme les listes, les ensembles ou encore les dictionnaires. Nous avons utilisé cette bibliothèque dans le cadre de la mécanisation d'une variante de Wormholes afin de gérer les identifiants de ressources.

Bibliographie

[1] Laurent Arditi, Amar Bouali, Hedi Boufaied, Gael Clave, Mourad Hadjchaib, Laure Leblanc, and Robert Simone. Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP. 10 1999.

[2] Andrea Asperti and Giuseppe Longo. *Categories, types and structures - an introduction to category theory for the working computer scientist*. 1991.

[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 3–15, New York, NY, USA, 2008. ACM.

[4] Patrick Bahr. Modal FRP for all : Functional reactive programming without space leaks in Haskell. *J. Funct. Program.*, 32 :e15, 2022.

[5] Patrick Bahr, Christian Graulund, and Rasmus Møgelberg. Simply RaTT : A fitch-style modal calculus for reactive programming without space leaks, 2019.

[6] Patrick Bahr and Graham Hutton. Beyond trees : Calculating graph-based compilers Functional Pearl. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024.

[7] Manuel Bärenz and Ivan Perez. Rhine : FRP with type-level clocks. *SIGPLAN Not.*, 53(7) :145–157, September 2018.

[8] Nick Benton and Vasileios Koutavas. A mechanized bisimulation for the nu-calculus. Technical Report MSR-TR-2008-129, September 2008.

[9] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proc. Inst. Electr. Electron. Eng.*, 79(9) :1270–1282, 1991.

[10] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proc. Inst. Electr. Electron. Eng.*, 91(1) :64–83, 2003.

[11] Stefan Berghofer. A solution to the PoplMark challenge using de Bruijn indices in Isabelle/HOL. *J. Autom. Reason.*, 49(3) :303–326, 2012.

[12] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[13] Gérard Berry. *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, chapter The foundations of Esterel, page 425–454. MIT Press, Cambridge, MA, USA, 2000.

[14] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. ESTEREL : a formal method applied to avionic software development. *Sci. Comput. Program.*, 36(1) :5–25, 2000.

[15] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, Berlin, Heidelberg, 2004.

[16] Y. Biannic, Eric Nassor, Emmanuel Ledinot, and Sylvan Dissoubray. Uml object specification for real-time software. *Real-time Systems - RTS*, 01 2000.

[17] Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Programming Language Design and Implementation (PLDI)*. ACM, June 2017.

[18] Timothy Bourke and Marc Pouzet. Zélus : A Synchronous Language with ODEs. In Calin Belta and Franjo Ivančić, editors, *HSCC - 16th International Conference on Hybrid systems : computation and control*, International Conference on Hybrid systems : computation and control, pages 113–118, Philadelphia, United States, Apr 2013. ACM.

[19] Frédéric Boussinot. Reactive C : an extension of C to program reactive systems. *Softw. - Pract. Exp.*, 21(4) :401–428, 1991.

[20] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *IEEE Trans. Softw. Eng.*, 22(4) :256–266, 1996.

[21] Daniel Bünzli. React-OCaml. `https://erratique.ch/software/react`, 2009.

[22] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL)*, pages 178–188. ACM, 1987.

[23] Paul Caspi and Marc Pouzet. A functional extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.

[24] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair reactive programming. In Suresh Jagannathan and Peter Sewell, editors, *Principles of Programming Languages (POPL)*, pages 361–372. ACM, 2014.

[25] Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3) :363–408, 2012.

[26] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming (ICFP)*, New York, NY, USA, 2008. ACM.

[27] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2) :346–366, 1932.

[28] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2) :56–68, 1940.

[29] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6 : A formal language for embedded critical software development. In *Theoretical Aspects of Software Engineering (TASE)*, pages 1–11, 2017.

[30] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *Programming Languages and Systems (ESOP)*, volume 3924 of *LNCS*, pages 294–308. Springer, 2006.

[31] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2) :95–120, 1988.

[32] Antony Courtney. Frappé : Functional reactive programming in Java. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages (PADL)*, volume 1990 of *LNCS*, pages 29–44. Springer, 2001.

[33] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.

[34] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18, New York, NY, USA, 2003. Association for Computing Machinery (ACM).

[35] Bruno C. d. S. Oliveira and Andres Löh. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, page 87–96, New York, NY, USA, 2013. ACM.

[36] Frédéric Dabrowski and Jordan Ischard. Functional reactive programmong with effects, a more permissive approach. working paper or preprint, March 2025.

[37] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 211–225, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[38] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Studies in logic and the foundations of mathematics*, 133 :375–388, 1972.

[39] François Xavier Dormoy. SCADE 6 a model based solution for safety critical software development. In *ERTS 2008 proceedings*, toulouse, France, January 2008.

[40] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2) :231–294, 1945.

[41] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE transactions on software engineering*, 25(3) :291–308, 1999.

[42] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*, pages 263–273. ACM, 1997.

[43] Conal M. Elliott. Push-pull functional reactive programming. In Stephanie Weirich, editor, *International Symposium on Haskell*, pages 25–36. ACM, 2009.

[44] William A. Howard et al. The formulae-as-types notion of construction. *To HB Curry : essays on combinatory logic, lambda calculus and formalism*, 44 :479–490, 1980.

[45] Jacques Garrigue. A certified implementation of ML with structural polymorphism. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems (APLAS)*, APLAS'10, page 360–375, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1) :176–210, 1935.

[47] Roger Godement. Topologies algébrique et théorie des faisceaux. *Actualites Scientifiques et Industrielles 1252*, 1958.

[48] Georges Gonthier. Formal proof—the four- color theorem. 2008.

[49] Christian Uldal Graulund, Dmitrij Szamozvancev, and Neel Krishnaswami. Adjoint reactive GUI. *Computing Research Repository (CoRR)*, abs/2010.12338, 2020.

[50] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with Signal. *Proc. Inst. Electr. Electron. Eng.*, 79(9) :1321–1336, 1991.

[51] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proc. Inst. Electr. Electron. Eng.*, 79(9) :1305–1320, 1991.

[52] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer.

[53] Curry B. Haskell. The inconsistency of certain formal logics. *Journal of Symbolic Logic*, 7(3) :115–117, 1942.

[54] Chris Heunen and Bart Jacobs. Arrows, like Monads, are monoids. *Electronic Notes in Theoretical Computer Science (ENCS)*, 158 :219–236, 2006.

[55] Noriko Hirota and Kenichi Asai. Formalizing a correctness property of a type-directed partial evaluator. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification*, PLPV '14, page 41–46, New York, NY, USA, 2014. ACM.

[56] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming*, volume 2638 of *LNCS*, pages 159–187. Springer, 2002.

[57] John Hughes. Generalizing monads to arrows. *Sci. Comput. Program.*, 37(1-3) :67–111, 2000.

[58] Jordan Ischard. DeBrLevel - version 1.0.1, November 2024.

[59] Jordan Ischard, Frédéric Dabrowski, Jules Chouquet, and Frédéric Loulergue. A mechanized formalization of an FRP language with effects. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, SAC '25, page 1431–1439, New York, NY, USA, 2025. ACM.

[60] Jordan Ischard, Frédéric Dabrowski, Jules Chouquet, and Frédéric Loulergue. Sémantique mécanisée de Wormholes. `https://github.com/JordanIschard/Mechanized-Wormholes`, 2025.

[61] Bart Jacobs, Chris Heunen, and Ichiro Hasuo. Categorical semantics for arrows. *Journal of Functional Programming*, 19(3–4) :403–438, 2009.

[62] Alan Jeffrey. LTL types FRP : linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *Programming Languages meets Program Verification (PLPV)*, pages 49–60, Philadelphia, PA, USA, 2012. ACM.

[63] Alan Jeffrey. Functional reactive programming with liveness guarantees. In Greg Morrisett and Tarmo Uustalu, editors, *International Conference on Functional Programming (ICFP)*, pages 233–244. ACM, 2013.

[64] Wolfgang Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electron. Notes Theor. Comput. Sci.*, 286 :229–242, 2012.

[65] Fairouzand Kamareddine and Alejandro Ríos. A lambda-calculus à la de bruijn with explicit substitutions. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages : Implementations, Logics and Programs*, pages 45–62, Berlin, Heidelberg, 1995. Springer.

[66] Anders Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23 :113–120, 12 1972.

[67] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In Greg Morrisett and Tarmo Uustalu, editors, *International Conference on Functional Programming (ICFP)*, pages 221–232. ACM, 2013.

[68] Joachim Lambek and Philip J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1988.

[69] Xavier Leroy. A locally nameless solution to the POPLmark challenge. Research Report RR-6098, INRIA, 2007.

[70] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[71] Pierre Lescanne and Jocelyne Rouyer-Degli. Explicit substitutions with de bruijn's levels. In Jieh Hsiang, editor, *Rewriting Techniques and Applications (RTA)*, pages 294–308, Berlin, Heidelberg, 1995. Springer.

[72] Louis Mandel and Marc Pouzet. ReactiveML : a reactive extension to ML. In Pedro Barahona and Amy P. Felty, editors, *Principles and Practice of Declarative Programming (PPDP)*, pages 82–93, Lisbon, Portugal, 2005. ACM.

[73] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax : a programming language for Ajax applications. *SIGPLAN Not.*, 44(10) :1–20, October 2009.

[74] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

[75] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 51–64, New York, NY, USA, 2002. Association for Computing Machinery.

[76] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[77] Ross Paterson. A new notation for Arrows. *SIGPLAN Not.*, 36(10) :229–240, oct 2001.

[78] Ivan Perez. Back to the future : time travel in FRP. *SIGPLAN Not.*, 52(10) :105–116, September 2017.

[79] Ivan Perez. Fault tolerant functional reactive programming (functional pearl). *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

[80] Ivan Perez. The beauty and elegance of functional reactive animation. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, FARM 2023, page 8–20, New York, NY, USA, 2023. Association for Computing Machinery.

[81] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In *International Symposium on Haskell*, Haskell 2016, page 33–44, New York, NY, USA, 2016. ACM.

[82] Ivan Perez and Henrik Nilsson. Testing and debugging functional reactive programming. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[83] John Peterson, Gregory Hager, and Paul Hudak. Language for declarative robotic programming. volume 2, pages 1144 – 1151 vol.2, 02 1999.

[84] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion : Controlling robots with Haskell. 07 1999.

[85] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

[86] Benjamin C. Pierce. *Basic category theory for computer scientists*. MIT Press, Cambridge, MA, USA, 1991.

[87] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, 1977.

[88] Alastair Reid, John Peterson, Gregory Hager, and Paul Hudak. Prototyping real-time vision systems : An experiment in DSL design. In *Proceedings of the 21st international conference on Software engineering*, pages 484–493, 1999.

[89] Bertrand Russell. Les paradoxes de la logique. *Revue de Métaphysique et de Morale*, 14(5) :627–650, 1906.

[90] Lindley S., Walder P., and Yallop J. The arrow calculus. *J. Funct. Program.*, 20 :51–69, 2010.

[91] Rodrigo C. M. Santos, Guilherme F. Lima, Francisco Sant'Anna, Roberto Ierusalimschy, and Edward Hermann Haeusler. A memory-bounded, deterministic and terminating semantics for the synchronous programming language Céu. In Zheng Zhang and Christophe Dubach, editors, *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–18, Philadelphia, PA, USA, 2018. ACM.

[92] Abhiroop Sarkar and Mary Sheeran. Hailstorm : A statically-typed, purely functional language for IoT applications. In *Principles and Practice of Declarative Programming (PPDP)*, New York, NY, USA, 2020. ACM.

[93] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In Graham Hutton and Andrew P. Tolmach, editors, *International Conference on Functional programming (ICFP)*, pages 23–34. ACM, 2009.

[94] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, pages 184–207, Berlin, Heidelberg, 1996. Springer.

[95] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`.

[96] Atze van der Ploeg and Koen Claessen. Practical principled FRP : forget the past, change the future, FRPNow ! In Kathleen Fisher and John H. Reppy, editors, *International Conference on Functional Programming (ICFP)*, pages 302–314. ACM, 2015.

[97] Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs. In *Journées Francophones des Langages Applications (JFLA)*, 2017.

[98] Jérôme Vouillon. A solution to the PoplMark challenge based on de Bruijn indices. *J. Autom. Reason.*, 49(3) :327–362, 2012.

[99] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In Monica S. Lam, editor, *Programming Language Design and Implementation (PLDI)*, pages 242–252, Vancouver, British Columbia, Canada, 2000. ACM.

[100] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In Benjamin C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 146–156, Firenze (Florence), Italy, 2001. ACM.

[101] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 155–172. Springer, 2002.

[102] Daniel Winograd-Cort and Paul Hudak. Wormholes : Introducing effects to FRP. In *International Symposium on Haskell*, New York, NY, USA, 2012. ACM.

[103] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. Virtualizing real-world objects in FRP. In *Practical Aspects of Declarative Languages (PADL)*, volume 7149 of *LNCS*, pages 227–241. Springer, 2012.

[104] Dana N. Xu and Siau-Cheng Khoo. Compiling real time functional reactive programming. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, ASIA-PEPM '02, page 83–93, New York, NY, USA, 2002. ACM.

[105] Urara Yamada and Kenichi Asai. Certifying CPS transformation of let-polymorphic calculus using PHOAS. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 375–393, Cham, 2018. Springer International Publishing.

Jordan Ischard

Sémantique de langages de programmation fonctionnels réactifs avec effets

La programmation réactive fonctionnelle (FRP) est un paradigme fonctionnel élaboré dans le but de simplifier la programmation de systèmes qui interagissent continuellement avec leur environnement. YAMPA est une bibliothèque particulièrement importante de ce paradigme. Elle permet aux utilisateurs de construire des fonctions de signal qui traitent de manière synchrone les flux d'entrée pour produire des flux de sortie. Malgré une construction concise et élégante de programme dans YAMPA, la gestion des entrées/sorties est un point difficile qui a amené au questionnement sur les effets de bords dans un langage FRP à la YAMPA. Parmi les solutions proposées par la communauté scientifique, nous nous sommes focalisés sur le langage WORMHOLES défini par Winograd-Cort et Hudak en 2012. Ce langage ajoute des constructions supplémentaires à YAMPA permettant d'introduire naturellement des effets dans un programme. Nous avons remarqué des problèmes dans la formalisation ainsi que des limitations non-nécessaires sur l'utilisation des effets. Dans cette thèse, nous proposons une mécanisation en ROCQ d'une variante du langage WORMHOLES corrigeant les problèmes de la formalisation papier initiale et nous étendons ses concepts dans un nouveau langage nommé MOLHOLES.
Mots clés : programmation réactive, sémantique, effets, assistant de preuves

Semantics of functional reactive programming languages with effects

Functional Reactive Programming (FRP) is a functional programming paradigm designed for systems interacting with their environment. An important branch of this paradigm was introduced by Hudak *et al.* in 2002 via the HASKELL library named YAMPA. It allows users to construct signal functions that synchronously process input streams to produce output streams. While this library facilitates concise and robust coding, managing I/O is cumbersome which led to questions about side effects for YAMPA-like languages. Among the solutions proposed by the scientific community, we focused on the WORMHOLES language defined by Winograd-Cort and Hudak in 2012. This language adds constructs to YAMPA which allows us to simply introduce side effects into a program. We noticed problems in the formalization, as well as unnecessary limitations on the use of effects. In this thesis, we propose a mechanization in ROCQ of a variant of WORMHOLES correcting the problems found and extending its concepts in a new language named MOLHOLES.
Keywords : reactive programming, semantics, effects, proof assistant

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)