Inférer et vérifier les tailles de tableaux avec des types polymorphes

Jean-Louis Colaço[1], Baptiste Pauget[1,3], and Marc Pouzet[2,3]

[1] Ansys, Toulouse, France
jean-louis.colaco@ansys.com
baptiste.pauget@ansys.com
[2] Ecole normale supérieure – PSL university, Paris, France
marc.pouzet@ens.fr
[3] INRIA, Paris, France

Résumé

Cet article présente un système de vérification et d'inférence statique des tailles de tableaux dans un langage fonctionnel strict statiquement typé. Plutôt que de s'en remettre à des types dépendants, nous proposons un système de types proche de celui de ML. Le polymorphisme sert à définir des fonctions génériques en type et en taille. L'inférence permet une écriture plus légère des opérations classiques de traitement du signal — application point-à-point, accumulation, projection, transposée, convolution — et de leur composition ; c'est un atout clef de la solution proposée. Pour obtenir un bon compromis entre expressivité du langage des tailles et décidabilité de la vérification et de l'inférence, notre solution repose sur deux éléments : (i) un langage de types où les tailles sont des polynômes multi-variés et (ii) l'insertion de points de coercition explicites entre tailles dans le programme source. Lorsque le programme est bien typé, il s'exécute sans erreur de taille en dehors de ces points de coercition. Deux usages de la proposition faite ici peuvent être envisagés : (i) la génération de code défensif aux points de coercition ou, (ii) pour les applications critiques ou intensives, la vérification statique des coercitions en les limitant à des expressions évaluables à la compilation ou par d'autres moyens de vérification formelle.

L'article définit le langage d'entrée, sa sémantique dynamique, son système de types et montre sa correction. Il est accompagné d'une implémentation en OCaml, dont le code source est accessible publiquement.

1 Introduction

On s'intéresse ici à la programmation, dans un langage de haut niveau, de systèmes temps réel soumis à des contraintes de sûreté fortes, comme ceux que l'on trouve dans l'avionique, le rail et l'automobile (e.g., commande de vol, freinage, moteur électrique). Le langage Scade [4], par exemple, est utilisé depuis plus de vingt ans pour programmer du logiciel temps réel embarqué critique. Il hérite des principes et du style de programmation de Lustre [10] : on écrit un modèle idéal synchrone dit *zero-délai* en composant des fonctions de suites et on confie au compilateur le soin de vérifier des propriétés de sûreté du programme source — par exemple qu'il est déterministe et qu'il peut être exécuté en temps et mémoire calculables statiquement — et de générer le code exécutable. La confiance donnée à son compilateur permet d'éviter d'avoir à démontrer *a posteriori* que le code est correct vis-à-vis du programme source. Elle s'appuie sur une description formelle précise de la sémantique du langage et de son compilateur. [1]

Les applications temps réel modernes combinent du code de contrôle complexe (avec un niveau d'imbrication important d'automates hiérarchiques) et du calcul intensif utilisant des

[1] Il ne s'agit cependant pas d'un compilateur prouvé formellement comme c'est le cas pour CompCert [18].

tableaux. Cet article étudie ce dernier aspect. On cherche à exprimer ces calculs dans le cadre d'un langage purement fonctionnel tel que Scade, en offrant une expressivité suffisante et dont la sûreté peut être assurée à la compilation par des moyens modulaires peu coûteux.

L'utilisation de tableaux dans un langage de programmation introduit des accès dynamiques à la mémoire dont la sûreté n'est en général pas vérifiée par le compilateur. Ces accès doivent respecter les bornes des tableaux, sous peine de produire, au mieux un arrêt à l'exécution, au pire une corruption silencieuse de la mémoire. Cette propriété peut être assurée de différentes façons : par l'ajout de code défensif et la levée d'exceptions (comme le fait OCaml, par exemple), en saturant la valeur de l'indice [9] ou en ajoutant une valeur par défaut [4] — deux solutions suivies dans plusieurs compilateurs synchrones (e.g., Heptagon, Lustre V6 et Scade). Il en résulte une moindre efficacité du code généré et l'introduction potentielle de code mort.[2] Assurer à la compilation l'absence d'accès hors des bornes d'un tableau apporte une garantie de sûreté et offre de meilleures perspectives d'optimisations. C'est ce que permet notamment le langage Spark [1], construit sur le langage Ada.

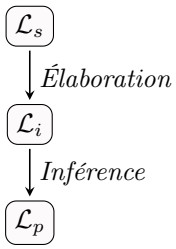
Les applications ciblées proviennent du traitement du signal et de l'IA (en particulier, l'implémentation de réseaux de neurones). Les opérations de base sont celles de l'algèbre linéaire (addition, multiplication sur des tableaux multi-dimentionnels, etc.). Certains algorithmes requièrent une forme limitée de récursivité sur les tailles de tableaux : c'est le cas de la transformée de Fourier rapide, ce qui dépasse l'expressivité actuelle de Scade. D'autres, comme l'échantillonnage et le filtrage, nécessitent des relations non linéaires entre les tailles. Dans un langage explicitement typé, la spécification des tailles dans les types conduit à des annotations longues et répétées. Afin d'y remédier, il est nécessaire d'offrir un moyen d'inférer automatiquement les types et les tailles en s'assurant de leur bonne utilisation, c'est-à-dire de l'absence d'accès aux éléments d'un tableau en dehors de leur domaine.

L'expression d'opérations sur des tableaux dans un langage purement fonctionnel ainsi que la vérification par typage de leur bonne utilisation sont deux questions anciennes qui ont été étudiées largement. Nous y revenons dans la section 7. Plutôt que de s'en remettre à l'utilisation d'un système de types dépendants, la solution proposée s'appuie sur deux éléments : (i) un langage de types avec polymorphisme où les tailles sont des polynômes multi-variés. Le polymorphisme sert à définir des fonctions génériques en type et en taille. L'inférence permet une écriture plus légère des opérations classiques de traitement du signal et de leur composition. (ii) L'insertion de points de coercition explicites entre tailles dans le programme source. Lorsque le programme est bien typé, il s'exécute sans erreur de taille en dehors de ces points de coercitions. Pour des applications critiques, la vérification statique des coercitions pourra être faite a posteriori en les limitant à des expressions évaluables à la compilation ou en utilisant d'autres moyens de vérification formelle.

Cette étude est menée sur un langage purement fonctionnel et elle s'appuie sur un prototype indépendant de Scade dont le code source et les exemples sont accessibles ici : `https://gitlab.inria.fr/parkas/jfla-2022`. Ce langage ne contient pas de constructions temporelles et se concentre sur les opérations de manipulation de tableaux. L'article est organisé ainsi : la section 2 donne un aperçu général de la contribution proposée. Le langage et sa sémantique sont définis dans la section 3. Le système de types est défini dans la section 4. La section 5 détaille l'inférence des types et des tailles. Un langage de plus haut niveau est esquissé dans la section 6. Nous discutons des travaux connexes et concluons dans la section 7.

[2] Ce dernier point n'est pas anodin : la couverture du code, activité exigée pour la certification d'applications critiques, est plus délicate à assurer en présence de code mort ne pouvant pas être couvert par un cas de test.

2 Vue d'ensemble



La proposition faite dans cet article définit trois langages dont l'articulation est résumée ci-contre. Le langage de *profondeur* $\mathcal{L}_p$, explicitement typé, pour lequel une sémantique sera définie, contient les constructions nécessaires à l'introduction de tailles dans les types. Ce langage représente les tableaux par des fonctions dont le domaine est un intervalle commençant à zéro. Pour rendre les annotations optionnelles, une inférence des types et des tailles est construite sur un langage *intermédiaire* $\mathcal{L}_i$, variante de $\mathcal{L}_p$. Enfin, le langage de *surface* $\mathcal{L}_s$ apporte une couche de commodités syntaxiques. Il couvre en particulier les notations usuelles de tableaux et s'élabore vers $\mathcal{L}_i$.

Afin d'illustrer la représentation des tailles dans les types, les exemples suivants sont écrits dans $\mathcal{L}_i$. Le lecteur impatient trouvera dans la Figure 8 ces fonctions écrites dans $\mathcal{L}_s$.

Tableaux intentionnels. Dans de nombreux langages inventés pour le calcul scientifique (dont Sisal [6], pour n'en citer qu'un), les manipulations explicites d'indices sont remplacées par des opérateurs sur les tableaux, appelés aussi combinateurs [15, 13]. Ils fournissent des schémas d'accès prédéfinis toujours corrects. Le langage Scade [4] suit la même approche. Certaines primitives nécessitent cependant que des tailles coïncident, ce qui ne peut être imposé sans exprimer les tailles dans les types. Dans $\mathcal{L}_p$, le langage des tailles est séparé de celui des expressions et possède son propre ensemble de variables. Ainsi, l'application point-à-point d'une fonction (`map`), sa version binaire (`map2`) et la réduction de tableaux (`fold`), trois opérateurs qui s'expriment en Scade, auront ici le schéma de type suivant :

$$\texttt{val map} : \forall \iota.\ \forall \alpha, \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\beta$$
$$\texttt{val fold} : \forall \iota.\ \forall \alpha, \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\,\beta \to \alpha$$
$$\texttt{val map2} : \forall \iota.\ \forall \alpha, \beta, \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\beta \to \texttt{[}\iota\texttt{]}\,\gamma$$

Le second argument de chacun de ces itérateurs, de type <ι> (lire *taille* ι), est appelé *paramètre de taille*. Il s'agit d'un entier dont la valeur est égale à celle de la *variable de taille* ι. Pour un type τ, le type [ι]τ est celui des tableaux de taille ι d'éléments de type τ. L'application partielle `map` f 42 attend donc un tableau de taille 42 uniquement. Ces types sont universellement quantifiés en tailles (ι) et en types (α, β, γ). Contrairement à un système avec types dépendants où `map` aurait pour signature $\forall \alpha, \beta.\ (\alpha \to \beta) \to \Pi\, n\,\texttt{:int}.\ \texttt{[}n\texttt{]}\,\alpha \to \texttt{[}n\texttt{]}\,\beta$, les arguments de taille n'ont pas besoin d'être nommés et liés dans les types. Dans $\mathcal{L}_i$, le produit scalaire s'écrit :

$$\texttt{let dot\_product} : \_ = \lambda u\texttt{:}\_.\ \lambda v\texttt{:}\_.\ \texttt{fold}\ (\texttt{+})\ \texttt{<\_>}\ 0\ (\texttt{map2}\ (\texttt{*})\ \texttt{<\_>}\ u\ v)$$

Les trois premiers "trous" (`_`) symbolisent des types à inférer. Les deux suivants sont contenus dans la construction <$\cdot$> qui extrait la valeur d'une taille et qu'il faudra ici inférer. Ces paramètres de taille des itérateurs n'ont pas besoin d'être spécifiés car leur valeur est contrainte par la taille des tableaux. Le langage de surface $\mathcal{L}_s$ permet d'omettre ces emplacements de tailles à inférer (<_>). Pour la fonction `dot_product`, l'inférence détermine le type $\forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int} \to \texttt{int}$. Supposons maintenant que l'on dispose d'une primitive `window` définissant une fenêtre glissante de taille κ, de pas 1 et dont le schéma de type est :

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to \texttt{[}\iota + \kappa - 1\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\texttt{[}\kappa\texttt{]}\,\alpha$$

Ici, la variable de taille ι n'apparaît pas comme paramètre de taille. Elle devra donc être déduite des types. De plus, la relation entre les tailles des tableaux utilise des opérations arithmétiques. Il est ainsi possible de définir une convolution de noyau k :

$$\texttt{let convolution} : \_ = \lambda k{:}\_.\ \lambda i{:}\_.\ \texttt{map}\ (\texttt{dot\_product}\ k)\ \texttt{<\_>}\ (\texttt{window}\ \texttt{<\_>}\ i)$$

Cette fois encore, l'inférence parvient à compléter les parties manquantes, instancier les déclarations référencées et trouve le type $\forall \iota, \kappa.\ \texttt{[}\kappa\texttt{]int} \to \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota - \kappa + 1\texttt{]int}$. Cependant, une expression telle que `fold` $(+)$ `<_>` 0 (`[_]`2) (où `[_]`2 est un tableau de taille indéfinie dont tous les éléments valent 2) est ambiguë : la taille du tableau, dont dépend la valeur de l'expression, n'est pas définie. Pour lever cette ambiguïté, l'un des deux trous `<_>` ou `[_]` doit être spécifié.

Tableaux extensionnels. Lorsque les opérateurs intentionnels ne suffisent pas ou pour les programmer, il est nécessaire d'accéder explicitement aux éléments des tableaux. Dans $\mathcal{L}_i$, l'itérateur `map` peut être défini ainsi :

$$\texttt{let map} : \_ = \lambda f{:}\_.\ \lambda n{:}\texttt{<}\iota\texttt{>}.\ \lambda X{:}\_.\ \lambda i{:}\texttt{[}\iota\texttt{]}.\ f\ (X\ i)$$

Le type du dernier argument, $\texttt{[}\iota\texttt{]}$ (lire *indice de taille* ι) représente les entiers positifs strictement inférieur à la valeur de la taille ι. Les tableaux étant vus comme des fonctions, le type $\texttt{[}\eta\texttt{]}\tau$ est en réalité une abréviation pour le type $\texttt{[}\eta\texttt{]} \to \tau$. L'application $X\ i$ lit donc le tableau X de taille ι à l'indice i, et respecte par construction les bornes. Ici, le paramètre n sert uniquement à introduire un argument permettant de contraindre la variable de taille ι.

Définir l'opérateur de fenêtre glissante requiert un calcul sur les indices :

$$\texttt{let window} : \_ = \lambda k{:}\texttt{<}\kappa\texttt{>}.\ \lambda X{:}\texttt{[}\iota + \kappa - 1\texttt{]}\ \_.\ \lambda i{:}\texttt{[}\iota\texttt{]}.\ \lambda j{:}\texttt{[}\kappa\texttt{]}.\ X\ (i + j \triangleright \texttt{[\_]})$$

L'accès au tableau X utilise une *coercition*, notée $\triangleright$. Pour un terme e et un type τ, $e \triangleright \tau$ symbolise un test sur les tailles apparaissant dans les types, laissant uniquement au typage la vérification que le type de e et τ ont la même structure. Dans la définition de la fonction `window`, pour que l'application soit correcte, la valeur de $i + j$ doit être bornée supérieurement par une taille. Il est inutile de la spécifier, d'où l'écriture $X\ (i + j \triangleright \texttt{[\_]})$, car elle peut être déduite du type de X. Les coercitions permettent d'utiliser des opérations complexes qui dépassent l'expressivité du langage des tailles. Les erreurs qui en résultent doivent être éliminées par un autre mécanisme que le typage.

3 Langage de profondeur

L'élimination statique des erreurs liées aux tableaux s'appuie sur la définition d'un langage d'expressions, purement fonctionnel, d'ordre supérieur, explicitement typé, doté de constructions polymorphes et évalué strictement. Il a pour but de traiter tailles et types sur un pied d'égalité, tant du point de vue du système de types que de l'inférence. Ce langage est volontairement restreint aux constructions essentielles de manipulation de tableaux.

Sans recourir à un langage aussi abstrait que HM(X)[24], nous garderons de la souplesse dans la définition d'*opérateurs* (l'équivalent des destructeurs de HM(X)[25]). Cette paramétrisation requiert trois composants — syntaxe, sémantique et typage — qui seront définis précisément.

3.1 Syntaxe

La syntaxe de $\mathcal{L}_p$, le langage explicitement typé, est résumée dans la Figure 1. Par la suite, n désignera un entier relatif. Les tailles, les types et leurs variables seront désignés par des lettres grecque, tandis que les termes et leurs variables seront nommés par des lettres latines.

η	::=	*Tailles*	e	::=	*Termes*
	ι	variable		x	variable
	n	constante		$e\ e$	application
	η `+` η	somme		$\lambda x{:}\tau.\ e$	abstraction
	η `*` η	produit		`true` | `false`	booléen
				n	entier
τ	::=	*Types*		o	opérateur
	α	variable		e `[`η`]`	application de taille
	`<`η`>`	singleton		e `[`τ`]`	application de type
	`[`η`]`	intervalle		$\Lambda\iota.\ e$	abstraction de taille
	`int`	entier		$\Lambda\alpha.\ e$	abstraction de type
	`bool`	booléen		`fix` $x{:}\sigma = e$	point-fixe
	$\tau \to \tau$	fonction		`let` $x{:}\sigma = e$ `in` e	définition locale
				`let size` $\iota = e$ `in` e	déf. de taille
σ	::=	*Schémas de type*		`<`η`>`	taille
	τ	type simple		$e \triangleright \tau$	coercition
	$\forall\iota.\ \sigma$	quantif. de taille		`case` e `then` e `else` e	déf. par cas
	$\forall\alpha.\ \sigma$	quantif. de type		`.`	branche morte

Figure 1: Syntaxe abstraite du langage explicitement typé $\mathcal{L}_p$

$$\langle \forall\iota.\ \sigma \rangle = \overline{\langle\sigma\rangle}^{\iota} \qquad \langle \lambda x{:}\tau.\ e \rangle = \langle\tau\rangle \cup \overline{\langle e\rangle}^{x}$$
$$\langle \forall\alpha.\ \sigma \rangle = \overline{\langle\sigma\rangle}^{\alpha} \qquad \langle \texttt{fix}\ x{:}\sigma = e \rangle = \langle\sigma\rangle \cup \overline{\langle e\rangle}^{x}$$
$$\langle \Lambda\iota.\ e \rangle = \overline{\langle e\rangle}^{\iota} \qquad \langle \texttt{let}\ x{:}\sigma = e_1\ \texttt{in}\ e_2 \rangle = \langle\sigma\rangle \cup \langle e_1\rangle \cup \overline{\langle e_2\rangle}^{x}$$
$$\langle \Lambda\alpha.\ e \rangle = \overline{\langle e\rangle}^{\alpha} \qquad \langle \texttt{let size}\ \iota = e_1\ \texttt{in}\ e_2 \rangle = \langle e_1\rangle \cup \overline{\langle e_2\rangle}^{\iota}$$

Figure 2: Variables libres des objets de la syntaxe. $\overline{S}^{v}$ désigne l'ensemble $S \setminus \{v\}$. Dans les autres cas, $\langle o \rangle$ est construit inductivement par union des variables libres des sous-objets.

Espaces de noms, variables libres et substitutions. Les tailles, types et termes possèdent chacun leur propre espace de noms de variables, respectivement $\mathcal{V}_\eta$, $\mathcal{V}_\tau$ et $\mathcal{V}_e$, supposés disjoints. Pour un objet de la syntaxe o, $\langle o \rangle \in \mathcal{V}_\eta \cup \mathcal{V}_\tau \cup \mathcal{V}_e$ désigne l'ensemble des *variables libres*, défini dans la Figure 2 et $\langle o \rangle_e$ (*resp.* $\langle o \rangle_\eta$, $\langle o \rangle_\tau$) l'ensemble $\langle o \rangle \cap \mathcal{V}_e$ (*resp.* $\mathcal{V}_\eta$, $\mathcal{V}_\tau$). Un objet o est *clos* si $\langle o \rangle = \emptyset$. Les substitutions sont définies pour les différentes catégories syntaxiques et les couples variable/élément. Elle seront notées uniformément $\cdot\{\cdot/\cdot\}$. Ainsi, $e\{\eta/\iota\}$ désigne la substitution dans le terme e de toutes les occurrences libres de la variable de taille ι par la taille η, y compris dans les tailles et types contenus dans e.

Tailles, types et schémas. L'ensemble des tailles $\mathcal{P} := \mathbb{Z}[\mathcal{V}_\eta]$ est celui des polynômes multivariés à coefficients entiers. Cette classe d'expressions arithmétiques présente l'avantage de posséder une forme normale sous la forme d'une somme pondérée de produits de variables, ce qui permet de comparer symboliquement des tailles structurellement différentes (*eg.* $(\iota - 1)^2 - 1 \ = \ \iota * (\iota - 2)$). Ce langage suffit pour exprimer des propriétés des tailles des opérations de tableaux pour les applications visées.

En plus des types usuels du λ-calcul simplement typé (variables, types entier et booléen, fonctions), deux *raffinements* du type entier, au sens de [8, 31, 20, 7], introduisent les tailles dans

les types. Pour un entier n, $<n>$ désigne le singleton $\{n\}$, tandis que $[n]$ représente l'intervalle $[\![0, n-1]\!]$. Pour une taille générique, ces types dépendent de la valuation des variables de taille. Le langage des types ne contient pas de constructeur de type pour les tableaux : le type «tableau de taille η d'éléments de type τ» est représenté par $[\eta] \to \tau$, noté également $[\eta]\tau$[3]. Le type singleton $<\eta>$ permet aux expressions et aux tailles de communiquer [30, 7] et il aide l'inférence par l'introduction de contraintes sur les tailles (section 5).

Suivant les restrictions introduites dans les langages de la famille ML, les types sont quantifiés universellement et doivent être en forme prénexe selon des variables de taille ($\forall\iota.\ \sigma$) et de type ($\forall\alpha.\ \sigma$). Par la suite, nous factoriserons les notations pour les types et les règles de typage (quantification, abstraction, application ou substitution) en écrivant : $\forall\iota|\alpha.\ \sigma$.

Termes. Le cœur *a la* ML de notre langage fonctionnel (variables, abstraction, applications et déclarations locales polymorphes) est enrichi d'un opérateur de point-fixe polymorphe —$\texttt{fix}\ x{:}\sigma = e$—, extension très étudiée [21, 22] qui importe particulièrement pour la définition de fonctions récursives sur des tableaux (voir l'annexe D).

Les seules valeurs de base sont les entiers et les booléens. La composante syntaxique de la paramétrisation des opérateurs est constituée d'un ensemble d'identifiants noté $\mathcal{O}$, supposé distinct de celui des variables. Cet ensemble contient *a minima* les opérations arithmétiques entières (en particulier l'addition et la multiplication), les comparaisons entières et une égalité polymorphe. Ces opérateurs seront notés avec les symboles usuels, et leur application écrite en forme infixe.

Afin de définir une notion de portée des variables de taille et de type, le polymorphisme est rendu explicite par des abstractions de taille et de type (*généralisations*) —$\Lambda\iota|\alpha.\ e$— et les applications correspondantes (*instanciations*) —$e\ [\eta|\tau]$—.

Bien que séparés, le monde des tailles et celui des expressions communiquent. La valeur d'une taille peut être utilisée comme terme —$<\eta>$—. L'inverse est plus compliqué et s'exprime par la quantification existentielle d'une variable de taille —$\texttt{let size}\ \iota = e_1\ \texttt{in}\ e_2$—.

Pour échapper au système de types lorsque ses capacités de résolution formelle ne suffisent pas, une coercition —$e \triangleright \tau$—, semblable à celle de FUTHARK [12], insère une vérification *post-typage* des raffinements. Enfin, une définition par cas —$\texttt{case}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$—, qui peut être rendue partielle par un terme d'erreur —. — sur l'une des branches, conditionne l'évaluation d'expressions, par exemple pour les cas terminaux de fonctions récursives.

Précédence et notations. Les expressions seront implicitement parenthésées par les règles suivantes : les abstractions et généralisations ont la plus faible précédence, suivies des introductions de variables (de taille, de type ou d'expression). Viennent ensuite la définition par cas, puis la coercition et enfin les application et instanciations, associatives à gauche. Ainsi, $e_1\ e_2\ e_3$ désigne $(e_1\ e_2)\ e_3$ et $\lambda x{:}\tau.\ e\ x \triangleright \tau' =$ a le sens de $\lambda x{:}\tau.\ ((e\ x) \triangleright \tau')$.

3.2 Sémantique à grands pas

La sémantique de $\mathcal{L}_p$ —$e \rightsquigarrow v^\star$— associe à certaines expressions closes une *valeur*. L'ensemble des valeurs (noté $\mathcal{V}$) est le sous-ensemble des expressions présenté dans la Figure 3, augmenté d'une valeur spécifique $\star$ (lire «erreur») qui représente une erreur *post-typage* explicite. Ces erreurs correspondent à des propriétés que l'on peut vérifier statiquement par des mécanismes autres que le typage, en imposant par exemple aux expressions dont elles dépendent d'être évaluables à la compilation.

[3] Utilisée dans Futhark[13], cette notation convient aux tableaux multidimensionnels : $[\eta_1, \eta_2]\tau := [\eta_1]\,[\eta_2]\,\tau$

Opérateurs. Le volet sémantique de la paramétrisation des opérateurs, fortement inspiré de celle des destructeurs de HM(X)[25], comprend une fonction d'arité $\nu : \mathcal{O} \longrightarrow \mathbb{N}^*$ ainsi qu'une relation de réduction $\overset{o}{\rightsquigarrow}$.

En plus des constructeurs (n, `true`, `false`, λ, Λ), l'ensemble des valeurs est constitué des applications partielles d'opérateurs instanciés (la classe *Instance d'opérateur*) dont les arguments ont été réduits à des valeurs. Pour une instance d'opérateur p, l'arité de l'opérateur instancié sera notée $\nu(p)$. Les applications totales d'opérateurs sont réduites à l'aide de $\overset{o}{\rightsquigarrow}$. Cette sémantique des opérateurs est supposée totale sur le domaine de valeurs défini par leur type (voir section 4), et ne peut pas produire $\star$. Cela exclut par exemple la détection des divisions par zéro, ou des racines carrées de nombres négatifs. Comme dans [15], seules les erreurs *structurelles* sont représentées par $\star$.

p	::=	$o \mid p\ [\eta] \mid p\ [\tau]$	*Instance d'op.*
v	::=		*Valeur*
		$\mid n \mid$ `true` $\mid$ `false`	constantes
		$\mid p\ v_1\ \ldots\ v_i,\quad i < \nu(p)$	app. partielle
		$\mid \lambda x{:}\tau.\ e$	abstraction
		$\mid \Lambda\alpha.\ e$	abstraction de type
		$\mid \Lambda\iota.\ e$	abstraction de taille
$v^\star$	::=	$v \mid \star$	*Valeur optionnelle*

Figure 3: Valeurs et expressions

Erreurs et réduction. Le système de réduction associé est présenté dans la Figure 4. Afin d'alléger la formalisation, les règles *étoilées* (de la forme $\frac{\_\ \ldots\ \_}{\star}$) propagent implicitement les erreurs : si l'un des antécédents est réduit à $\star$, l'expression entière est réduite elle aussi à $\star$. Cela correspond à l'ajout de règles partielles dont la dernière prémisse et l'expression totale sont réduites à $\star$, (voir l'annexe A).

Propriétés. Cette sémantique impose une évaluation stricte : les arguments des applications de fonctions ou d'opérateurs sont réduits avant d'être substitués (règles β-RED et TOTOP). Les règles relatives à la définition par cas (TCASE, FCASE) n'évaluent, elles, qu'une seule des deux branches ce qui permet d'empêcher la propagation d'erreurs depuis la branche non choisie. Associée à la construction de branche morte, dont la sémantique produit $\star$ (règle ERR), cette définition par cas permet d'exprimer des gardes : la sémantique de `case` P `then` e `else .` n'évaluera l'expression e qui si le prédicat booléen P est valide.

Ce système de réduction est partiel : certains termes ne peuvent pas être réduits, on parlera de termes *bloqués*. En effet, les règles β-RED, INST, CTRUE, CFALSE et LETS restreignent la forme de la valeur obtenue par la réduction de l'une de leurs sous-expressions. Enfin, les règles de la sémantique des coercitions requièrent une adéquation de la valeur et du type.

Raffinements et coercitions. La présence de types raffinés empêche la définition d'une sémantique indépendante des types. En effet, ces raffinements discriminent des valeurs de même forme (type de base sous-jacent identique) et doivent être vérifiés en plusieurs endroits. Par exemple, nous ne souhaitons pas donner de sémantique au terme $(\lambda x{:}[4].\ e)\ 8$ où l'argument 8 n'est pas du type de la variable x. De manière générale, il est nécessaire pour chaque substitution de s'assurer que la valeur remplaçante satisfait le raffinement du type de la variable substituée. Cela concerne les règles β-RED, LET et FIX.

Pour restreindre les occurrences de $\star$ aux coercitions *explicites*, deux degrés sont distingués : les coercitions *faibles* $\text{—}\triangleright_w\text{—}$ ($\triangleright$ dans la syntaxe de $\mathcal{L}_i$), peuvent échouer et produire $\star$ tandis que les coercitions *fortes* $\text{—}\triangleright_s\text{—}$ (introduites par les règles de typages β-RED, LET et FIX) n'ont de sémantique que si elles réussissent. Étant fixé un degré k, les règles de réduction des coercitions $\text{—}v \triangleright_k \sigma \rightsquigarrow v^*\text{—}$ vérifient dans les cas de base le respect des raffinements (CSIZE,

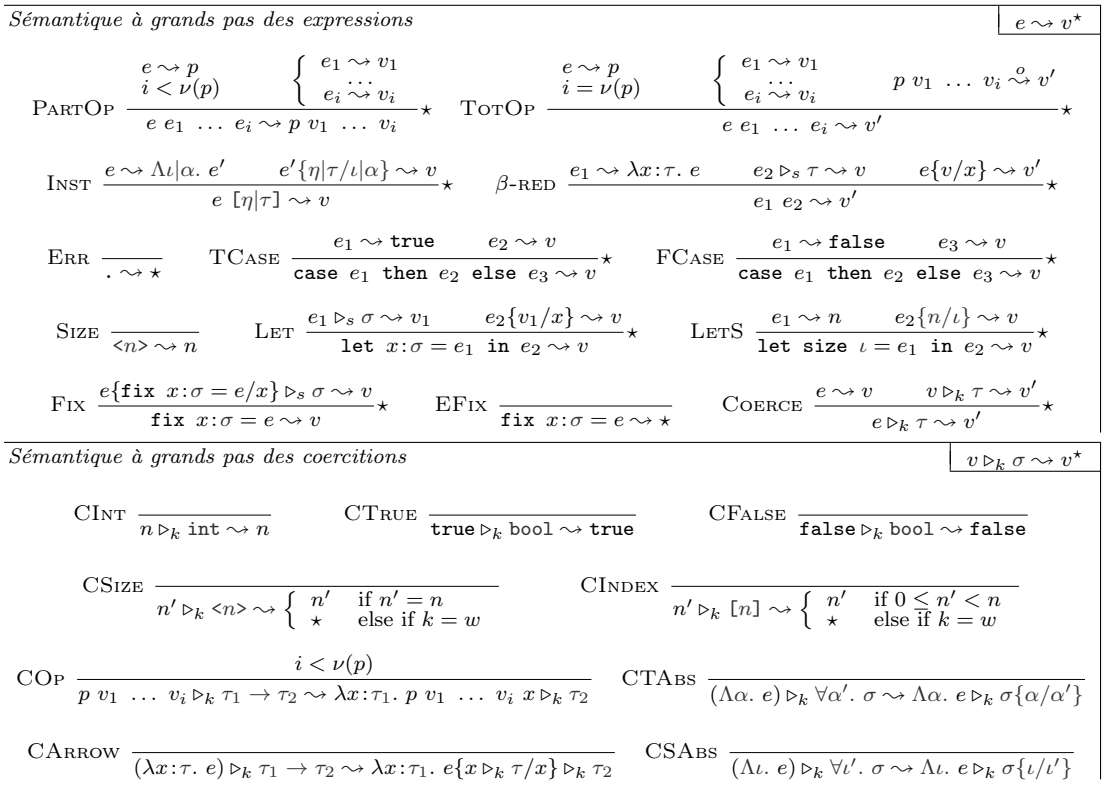


Figure 4: Règles de déduction de la sémantique de $\mathcal{L}_p$

CIndex), et ne peuvent échouer que si $k = w$. Pour les abstractions, applications partielles d'opérateurs et généralisations, les vérifications sont retardées à l'appel ou instanciation.

Rappelons notre objectif initial d'empêcher les accès incorrects aux tableaux. Par leur encodage sous la forme de fonctions, une valeur de type $[\eta]\,\tau$ est de la forme $\lambda i{:}\,[\eta]\,.\ e$. L'accès aux éléments est représenté par l'application, qui assure le respect des bornes. L'existence d'une sémantique garantit donc la correction des accès.

Récursion et convergence. Pour les distinguer des termes bloqués, ceux qui divergent doivent posséder une sémantique[4]. Pour cela, la règle EFix arrête la réduction par la levée d'une erreur et peut s'appliquer à n'importe quelle occurrence d'un point-fixe. La détection à la compilation de l'erreur produite correspond aux usages récursions statiques, déroulées à la compilation, qui sont les seuls autorisés dans l'embarqué critique pour borner la pile et le temps d'exécution.

Les règles Fix et EFix rendent la sémantique de $\mathcal{L}_p$ non déterministe, car elles permettent d'interrompre des récursions bornées. Il est aisé de vérifier que les autres règles sont dirigées par la syntaxe, ce qui permet d'énoncer une propriété de déterminisme affaibli, prouvée dans l'annexe A : il existe au plus une valeur v ($v \neq \star$) telle que $e \rightsquigarrow v$.

[4]Dans les sémantiques à petits pas, les termes divergents produisent une suite infinie de réductions élémentaires, ce qui les distinguent des termes bloqués, dont la réduction s'achève sans trouver de valeur.

4 Système de types

Une discipline de types, inspirée de celle d'Hindley et de Milner [14] (HM), sélectionne des termes pour lesquels une sémantique existe, c'est à dire des valeurs ou des expressions pouvant être réduites, possiblement à $\star$.

4.1 Jugements et déductions

Environnement. Les expressions sont typées dans un environnement Γ défini comme un triplet $(\Gamma_\eta, \Gamma_\tau, \Gamma_e)$ où Γ_η (*resp.* Γ_τ) collecte les variables de taille (*resp.* type) définies et Γ_e associe à certaines variables d'expression un schéma de type. Dans la suite, nous supposerons que les termes ont été convenablement renommés afin d'éviter les conflits de noms. L'environnement n'est donc pas ordonné et nous noterons uniformément $\Gamma, x : \sigma$; Γ, ι ou Γ, α l'extension de la composante appropriée, implicitement supposée ne pas contenir la variable insérée x, ι ou α.

Jugements. Le jugement de typage —$\Gamma \vdash e : \sigma$— se lit «dans l'environnement Γ, l'expression e a le schéma de type σ». Cette relation suppose implicitement la bonne formation de σ et de e, c'est à dire que leurs variables libres doivent être définies dans Γ. La relation de sous-typage —$\tau_1 <: \tau_2$— introduit ou élimine (selon la variance) des raffinements, tandis que celle de coercition —$\sigma_1 \approx \sigma_2$— introduit une égalité de types ignorant les tailles. Dans la suite, les environnements vides seront omis. Les règles de déductions de ces relations sont présentées dans la Figure 5.

Opérateurs. Dernière facette de notre paramétrisation, chaque opérateur o se voit attribuer un schéma de type, noté $\sigma(o)$. Celui-ci permet de définir l'ultime hypothèse sur le domaine de la sémantique $\overset{o}{\rightsquigarrow}$: pour chaque opérateur instancié p, elle doit être définie pour toutes les valeurs $v_1, \ldots, v_{\nu(p)}$ telles que $\vdash p\ v_1\ \ldots\ v_{\nu(p)} : \tau$ et vérifier $p\ v_1\ \ldots\ v_{\nu(p)} \overset{o}{\rightsquigarrow} v \implies \vdash v : \tau$.

Déductions Dans $\mathcal{L}_p$, le polymorphisme est explicite. Les constructions associées sont typées à l'aide des règles Gen et Inst. Elles généralisent et instancient en taille et en type.

La quantification existentielle de variables de taille, typée par la règle LetSize est locale car la variable introduite ne peut pas apparaître librement dans le type de l'expression, puisque celui-ci est bien formé dans l'environnement de typage non étendu.

La règle Fix permet une récursion polymorphe. En effet, la variable introduite possède un schéma de type. Elle pourra donc être instanciée différemment lors de ses utilisations récursives.

L'égalité de types, nécessaire par exemple dans les règles SRefl et CRefl, requiert l'identité formelle des tailles qui y apparaissent. Ainsi, les types $[\iota]$ et $[2 - \iota]$ sont différents, bien que toute instanciation telle que $\iota = 1$ les rende identiques.

Gestion des raffinements Le sous-typage permet uniquement d'insérer ou de supprimer des raffinements. Bien que sémantiquement correcte si $\eta_1 \leq \eta_2$ est formellement prouvable, la relation $[\eta_1] <: [\eta_2]$ n'est pas valide. Cette restriction permet à l'inférence des tailles de résoudre des égalités plutôt que des inégalités. Les raffinements peuvent également être modifiés par les coercitions, qui permettent de vérifier : (i) que des valeurs **entières** satisfont des raffinements (règles CSize et CIndex), ou (ii) que les tailles effectives d'un type, c'est à dire après instanciation, coïncident avec les tailles attendues (règles Coerce et relation $\tau_1 \approx \tau_2$).

Les constantes entières reçoivent le type $\langle n \rangle$. Elles peuvent donc être utilisées comme paramètres de taille mais pas à la pace d'indices. Il est nécessaire, pour obtenir des valeurs de

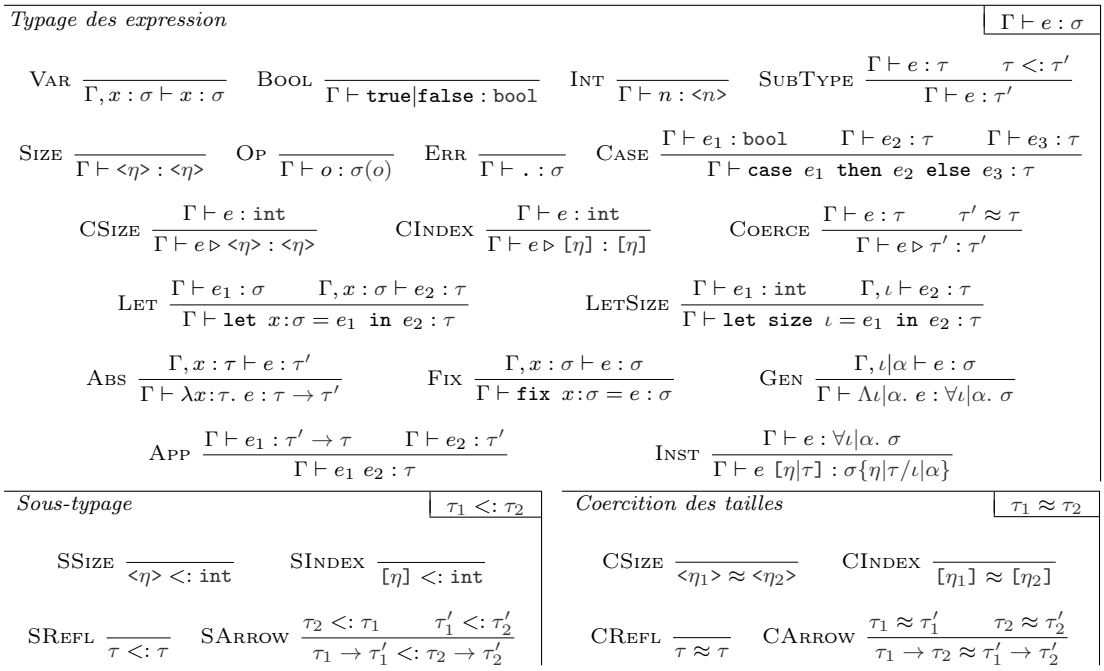


Figure 5: Système de types de $\mathcal{L}_p$

de type $[\eta]$ d'utiliser une coercition. Le système de type présenté ne contraint pas le signe des tailles. Des tableaux de taille négative peuvent donc être déclarés. Ils ne pourront cependant pas être lus (et se comportent donc comme des tableaux vides), car une coercition vers un indice de taille négative échouera systématiquement.

Calculs sur les tailles. Afin de limiter le nombre de coercitions nécessaires pour unifier différentes tailles, le typage des opérations arithmétiques `+`, `-` et `*` (désigné par $\bullet$ dans ce paragraphe, pour les expressions et les tailles) peut-être étendu par la règle ci-contre qui effectue les opérations simultanément sur les expressions et les tailles.

Typage des expression (extension) $\Gamma \vdash e : \sigma$

$$\text{ADD,SUB,MUL}\ \frac{\Gamma \vdash e_1 : <\eta_1> \qquad \Gamma \vdash e_2 : <\eta_2>}{\Gamma \vdash e_1 \bullet e_2 : <\eta_1 \bullet \eta_2>}$$

4.2 Propriétés du typage

Équivalence de types. Suivant la définition de [22], un schéma de type σ' est une *instance générique* de σ s'il existe une substitution des variables liées de σ ne capturant aucune de ses variables libres permettant d'obtenir σ', à l'ordre des quantifications près. Lorsque deux schémas de type sont des instances génériques l'un de l'autre, ils sont dits *équivalents*. Pour les types de ML, dont l'égalité est structurelle, cette relation coïncide avec un renommage des variables liées. Lorsque les tailles sont considérées, leur égalité extensionnelle élargit cette équivalence. Les types de $\mathcal{L}_s$ ne sont donc pas uniques : bien qu'ils soient structurellement différents, les deux schémas de type suivants sont équivalents.

$$\texttt{val concat} : \forall \iota_1, \iota_2.\ \forall \alpha.\ [\iota_1]\alpha \to [\iota_2]\alpha \to [\iota_1 + \iota_2]\alpha$$
$$\texttt{val concat} : \forall \iota_1, \iota_2.\ \forall \alpha.\ [\iota_1]\alpha \to [\iota_2 - \iota_1]\alpha \to [\iota_2]\alpha$$

Types principaux L'utilisation de polynômes comme langage de tailles permet des contraintes non linéaires pour lesquelles plusieurs solutions peuvent exister. Cela empêche l'existence d'un type le plus général. Ainsi, la fonction (sans intérêt) suivante, où ι doit être inférée

$$\texttt{let zero} : <\iota> \to <0> = \lambda n{:}<\iota>.\ (n-1) * (n-2)$$

est correctement typé pour $\iota = 1$ ou $\iota = 2$. Les deux types de `zero` qui en résultent ($<1> \to <0>$ et $<2> \to <0>$) ne sont ni équivalents ni ne sont des instances génériques d'un même schéma de type. L'inférence, détaillée dans la section 5, rejettera une telle définition.

Correction. Ce système de types vérifie deux propriétés : le type est préservé par réduction et le typage est correct vis-à-vis de la sémantique. Elles s'expriment formellement par :

$$\forall e\ \sigma\ v,\quad \left.\begin{array}{r}\vdash e : \sigma\\ e \rightsquigarrow v\end{array}\right\} \Longrightarrow \vdash v : \sigma \qquad \text{(Préservation)}$$

$$\forall e\ \sigma,\ \vdash e : \sigma \implies \exists v^\star \in \mathcal{V},\ e \rightsquigarrow v^\star \qquad \text{(Correction)}$$

La preuve de ces propriétés (voir l'annexe A) utilise une extension générique de sémantiques à grands pas développée dans [5]. Comme énoncé sous-section 3.2, l'existence d'une sémantique garantit une exécution sans erreur (en particulier aux accès de tableaux) à l'exception des valeurs $\star$ produites par certaines constructions. Le typage implique donc la même propriété.

5 Inférence

Les annotations de types deviennent encombrantes lorsque les expressions de tailles grossissent ; il est souhaitable de les inférer. Cependant, au vu de la richesse de notre système de types, impossible d'étendre ici les propriétés fondamentales dont jouit l'inférence pour la discipline de types de HM [14] (existence de types principaux, complétude et correction). En effet, la récursion polymorphe seule rend l'inférence indécidable [11]. De plus, l'utilisation de raffinements, réminiscence de types dépendants [30], limite là aussi les perspectives de reconstruction des types. Enfin, la manipulation d'expressions arithmétiques non linéaires laisse, elle aussi, peut d'espoir de solution totale au problème d'inférence. L'inférence cherchera seulement un type *pertinent*, qu'il sera possible d'ajuster par l'ajout d'annotations.

5.1 Aperçu de l'inférence

Comme le font les différents travaux sur l'inférence de types étendus [30, 17, 26], notre stratégie de reconstruction de types procède graduellement : (i) Les types (sans raffinement) sont inférés, par unification structurelle. (ii) Les raffinements du type entier sont sélectionnés par propagation locale des annotations. (iii) Les tailles sont inférées par résolution de contraintes.

Langage $\mathcal{L}_i$. Les termes implicitement typés sont exprimés dans le langage $\mathcal{L}_i$, variante du langage $\mathcal{L}_p$. Les *trous* (`_`) laissés à la place de tailles ou de types représentent des variables libre à inférer. Les termes sont implicitement généralisés aux `let` et les instanciations explicites sont interdites. Les généralisations peuvent apparaître, mais sont dotées d'une sémantique légèrement différente de celles de $\mathcal{L}_p$ (voir sous-section 5.3).

Inférence par collecte de contraintes. Les trois passes d'inférence ont été implémentées selon le même schéma : le terme est parcouru afin d'en extraire des contraintes dont le système

résultant est résolu en certaines constructions du langage [17]. Cela permet de traiter de façon similaire les tailles et les types, tant pour l'unification que pour la généralisation et l'instanciation. La principale limitation de cette stratégie tient à une localisation des erreurs plus approximative. Si elle n'apporte rien à l'inférence des types[5] pour laquelle l'ordre des contraintes n'a pas d'importance, la collecte des contraintes de taille offre une vue d'ensemble du système qui aide à leur résolution.

Les contraintes provenant de contextes de sous-typage sont distinguées de celles qui émanent des coercitions. Les variables libres, qui proviennent des *trous*, sont également collectées. Aux points de généralisation (`let`, abstractions de taille ou type), le système de contraintes est résolu, par des processus d'unification propres à chaque phase, présentés ci-dessous.

5.2 Résolution de contraintes

(i) Types. Première étape du processus d'inférence, les contraintes sont résolues pour les types sans raffinements. Pour cela, un type $\overline{\texttt{int}}$ désigne le type entier dont le raffinement reste à déterminer. La résolution de contraintes procède par unification structurelle des types, qui échoue selon les conditions habituelles (*e.g.* inégalité des constructeurs de tête, types cycliques). Ici, les contraintes provenant de sous-typage et de coercitions sont traitées uniformément.

À l'issue de cette passe d'inférence, les occurrences du type $\overline{\texttt{int}}$ sont remplacées par des variables de type fraîches, qui seront inférées dans la phase suivante.

(ii) Raffinements. Cette seconde phase choisit l'un des types `int`, `<_>` ou `[_]` pour chaque variable de type introduite à la place de $\overline{\texttt{int}}$ par l'inférence des types. Les tailles qui apparaissent dans les raffinements ne sont pas inférées. La définition suivante permettra d'illustrer la résolution des contraintes. Elle applique une négation booléenne à chaque élément d'un tableau :

$$\texttt{let not\_array} : \_ = \lambda X : \_.\ \lambda i : [\_].\ \texttt{not}\ (X\ i)$$

L'inférence des types produit le terme suivant, dont les contraintes sont résumées ci dessous :

$$\texttt{let not\_array} : (\alpha \to \texttt{bool}) \to \beta \to \texttt{bool} = \lambda X : \gamma \to \texttt{bool}.\ \lambda i : [\_].\ \texttt{not}\ (X\ i)$$

$$[\_] <: \gamma \tag{1}$$

$$(\gamma \to \texttt{bool}) \to [\_] \to \texttt{bool} <: (\alpha \to \texttt{bool}) \to \beta \to \texttt{bool} \tag{2}$$

La Contrainte 1 provient de l'application qui impose que le type de i soit un sous-type de celui du domaine de X. La Contrainte 2 est induite par la déclaration qui requiert que le type de l'expression soit un sous-type de l'annotation. Par la règle SARROW cette dernière se décompose en deux contraintes : $\gamma <: \alpha$ et $\beta <: [\_]$.

Les contraintes de la forme $\alpha <: \texttt{<\_>}$; $\alpha <: [\_]$ ou $\texttt{int} <: \alpha$ imposent le raffinement de la variable α. Elles sont considérées en premier afin de réduire le nombre de variable à inférer. Dans l'exemple, seule la variable β peut être définie de cette façon.

Remplacer les variables restantes par `int` produirait un terme bien raffiné dont le types ne serait pas aussi général qu'attendu. Pour résoudre les contraintes restantes (ici, $[\_] <: \gamma$ et $\gamma <: \alpha$), les variables qui n'apparaissent à droite ou à gauche que d'une seule inégalité sont remplacées par l'autre membre. Cette propagation locale de raffinements permet d'inférer le type `[_]bool` $\to$ `[_]bool` pour `idBArray`, sans laquelle il se verrait attribuer le type moins général (`int` $\to$ `bool`) $\to$ `[_]bool`, et ne serait pas utilisable avec un tableau.

[5] Les algorithmes d'unification destructrice obtiennent les mêmes résultats et localisent mieux les erreurs.

(iii) Tailles. Bien qu'il ne soit pas possible de résoudre formellement un système d'équations polynomiales entières, les relations entre les tailles de tableaux dans les fonctions sont en pratique simples. Pour présenter les stratégies de résolution les primitives suivantes seront utilisées :

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota + \kappa - 1]\,\alpha \to [\iota]\,[\kappa]\,\alpha$$
$$\texttt{val sample} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to [(\iota - 1) * \kappa + 1]\,\alpha \to [\iota]\,\alpha$$

La fenêtre glissante, présentée ci-dessus, contraint la taille des tableaux d'entrée et de sortie de façon à imposer une lecture complète de l'entrée. De manière similaire, `sample` extrait un élément sur κ, en lisant le premier et le dernier. La condition de divisibilité est ici encodée dans la taille de l'entrée. La composition de ces fonctions définit l'opérateur d'échantillonnage général `pack`, illustré dans la Figure 6, qui sélectionne ι fenêtres de taille κ, régulièrement espacées et couvrant les extrémités du tableau d'entrée de taille ν. Pour garantir cette propriété, le pas δ entre les fenêtres doit vérifier la relation $\delta * (\iota - 1) = \nu - \kappa$. Étant donnée la définition implicitement typée :

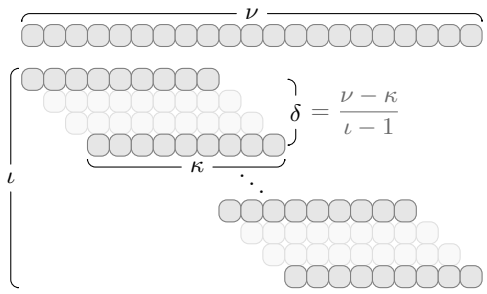


Figure 6: Opérateur `pack`

$$\texttt{let pack} : \_ = \lambda x : \_.\ \texttt{sample <\_> (window <\_> } x\texttt{)}$$

le terme suivant, instancié explicitement, est produit par les deux premières phases d'inférence :

$$\texttt{let pack} : [\nu]\,\alpha \to [\iota]\,[\kappa]\,\alpha = \lambda x : [\nu_i]\,\alpha.$$
$$\texttt{sample } [\iota_s]\ [\kappa_s]\ [[\kappa'_w]\,\alpha]\ \texttt{<}\kappa_1\texttt{>}\ (\texttt{window } [\iota_w]\ [\kappa_w]\ [\alpha]\ \texttt{<}\kappa_2\texttt{>}\ x)$$

Il est associé aux contraintes de taille résumées ici sous la forme d'égalité pour plus de lisibilité :

$$\kappa_w = \kappa_2 \qquad \nu_i = \iota_w + \kappa_w - 1 \qquad \kappa_w = \kappa'_w \qquad \kappa'_w = \kappa$$
$$\kappa_s = \kappa_1 \qquad \iota_w = (\iota_s - 1) * \kappa_s + 1 \qquad \iota_s = \iota \qquad \nu_i = \nu$$

Elles sont toutes de la forme $v - P = 0$ où $v \in \mathcal{V}_\eta$, $P \in \mathcal{P}$ et $v \notin \langle P \rangle_\eta$[6]. Pour de telles contraintes, la **stratégie *d'élimination de variables isolées*** substitue v par P dans le système ce qui n'en modifie pas les solutions. Elle permet ici d'inférer le type :

$$\texttt{val pack} : \forall \iota, \kappa, \delta.\ \forall \alpha.\ [\iota * \delta - \delta + \kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha$$

où la contrainte énoncée sur les tailles et le pas apparaît dans le type. S'il est théoriquement possible d'étendre cette stratégie à des contraintes de la forme $(v - P) * Q$ où $Q \in \mathcal{P}$ ne possède pas de racines entières, une telle factorisation n'est pas facilement dérivable algorithmiquement et ne suffirait pas pour certaines contraintes non linéaires.

L'opérateur `split`, restriction de `pack` où les fenêtres forment une partition du tableau d'entrée (transformation d'un vecteur en matrice) est défini par une annotation de type :

$$\texttt{let split} : [\iota * \kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha = \texttt{pack}$$

Après instanciation de `pack` par des variables ι_p, κ_p et δ_p, et substitution des variables isolées, l'unique contrainte $(\iota - 1) * \delta_p + \kappa = \iota * \kappa$ est obtenue. Celle-ci (équivalente à $(\iota - 1) * (\delta_p - \kappa) = 0$) est satisfaite si $\iota = 1$ ou $\delta_p = \kappa$. Choisir l'une ou l'autre des égalités pour en dériver une substitution réduit l'ensemble de solutions du système original.

La **stratégie *d'unification de polynômes semblables*** choisit, pour des contraintes c

[6] Ce type de contraintes provient en particulier des variables libres introduites dans les *trous*.

vérifiant : $\exists!\iota_1, \iota_2 \in \mathcal{V}_\eta,\ c\{\iota_1/\iota_2\} = 0$, de substituer ι_2 par ι_1 (δ_p par κ dans l'exemple). Cela permet de simplifier des contraintes de la forme $\eta_1 = \eta_2$, où les deux tailles ne diffèrent structurellement que d'une seule variable. Ces contraintes résultent notamment d'expressions partielles des tailles. Lorsqu'elles ne sont pas pertinentes, l'inférence sera orientée par l'ajout d'annotations. Algorithmiquement, l'exploration exhaustive des paires de variables est envisageable car chaque contrainte ne possède que peu de variables libres. Cette stratégie permet à la fonction `split` d'obtenir le type :

$$\texttt{val split} : \forall \iota, \kappa.\ \forall \alpha.\ [\iota * \kappa]\alpha \to [\iota][\kappa]\alpha$$

Les contraintes issues de coercitions sont considérées en dernier car ces vérifications post-typage n'ont pas d'obligation à être résolues. Plutôt que de rassembler ces primitives de manipulations de tableaux dans une bibliothèque, elles sont en pratique prédéfinies et traitées spécialement, ce qui réduit le nombre de coercitions à vérifier et permet des optimisations.

5.3 Polymorphisme

Le langage propose un polymorphisme unifié de taille et de type. À la manière des langages de la famille ML, les expressions sont généralisées aux déclarations et les variables sont instanciées à leurs utilisations. De plus, comme le fait le langage OCaml [19, Chapter 5], ces mécanismes implicites sont complétés par plusieurs constructions. (i) Des *annotations de schéma de types* dans les déclarations locales et points-fixe. Elles sont nécessaires pour les récursions polymorphes. (ii) Des *abstractions explicites* de type et de taille introduisant des variables localement abstraites dans un terme. Ces variables seront donc généralisées à la déclaration englobante. (iii) Des *variables anonymes* de taille et de type (préfixées par ‘ dans la syntaxe concrète). Elles sont généralisées aux déclarations globales uniquement.

Contrairement au cas des types[7], les variables de taille généralisées qui n'apparaissent pas dans le type rendent des expressions non définies (car elles ne peuvent pas être instanciées explicitement). Ainsi la déclaration suivante est rejetée :

$$\texttt{let even} : \_ = \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{[\_]}2)$$

car, la taille du tableau (et de l'itération), dont dépend la valeur de l'expression, n'est pas contrainte. L'ajout d'un paramètre de taille produit le type $\forall \iota.\ \texttt{<}\iota\texttt{>} \to \texttt{int}$, ce qui permet de contraindre la variable ι et rend le terme valide :

$$\texttt{let even} : \_ = \lambda n{:}\texttt{<\_>}.\ \texttt{fold}\ (+)\ n\ 0\ (\texttt{[\_]}2)$$

6 Langage de tableaux

Le langage de plus haut niveau $\mathcal{L}_s$ a pour but d'introduire les annotations nécessaires à l'inférence. Il cherche également à se rapprocher des constructions de tableaux de Scade [4]. Son étude est encore incomplète et sa forme sera uniquement esquissée dans cette partie.

Différenciation des arguments. L'ajout des annotations de raffinements est structurée par une séparation syntaxique de trois types d'arguments. Avec les contractions habituelles entre déclarations et abstractions (n-aires), la concaténation de deux tableaux s'exprime par :

```
let concat «n,p» (X:[n]_, Y:[p]_) [i<n+p] =
   case i < n then X[i] else Y[i-n]
```

[7] Des variables de type non contraintes proviennent de code mort, ce que l'on peut vouloir interdire également.

Paramètres de taille	*Arguments généraux*	*Indices de tableaux*
λ«n». e	$\lambda(x:\tau).\ e$	$\lambda[i<\eta].\ e$
$\Lambda n.\ \lambda n{:}\texttt{<}n\texttt{>}.\ e$	$\lambda x{:}\tau.\ e$	$\lambda i{:}[\eta].\ e$

(a) Abstractions

	Paramètres de taille	*Arguments généraux*	*Indices de tableaux*
$\mathcal{L}_s$	f «η»	$f\ (e)$	f [e]
$\mathcal{L}_i$	f <η>	$f\ e$	$f\ (e \triangleright$ [_])

(b) Applications

Figure 7: Élaboration de $\mathcal{L}_s$ dans $\mathcal{L}_i$

```
let window «k» (X:[k+'n-1]_) [i<'n, j<k] = X [i+j]
let dot_product (u,v) = fold (+) (0, map (*) (u,v))
let convolution (k,i) = map (dot_product (k,_)) (window (i))
```

Figure 8: Convolution dans $\mathcal{L}_s$

Suivant les règles de la Figure 7, ce terme est élaboré vers la définition suivante de $\mathcal{L}_i$:

$$\texttt{let concat} : \_ = \Lambda n.\ \lambda n{:}\texttt{<}n\texttt{>}.\ \Lambda p.\ \lambda p{:}\texttt{<}p\texttt{>}.\ \lambda X{:}[n]\,\_.\ \lambda Y{:}[p]\,\_.\ \lambda i{:}[n+p].$$
$$\texttt{case}\ i<n\ \texttt{then}\ X\ (i \triangleright [\_])\ \texttt{else}\ Y\ (i-n \triangleright [\_])$$

L'inférence détermine le type attendu $\forall \iota_1, \iota_2.\ \forall \alpha.\ \texttt{<}\iota_1\texttt{>} \to \texttt{<}\iota_2\texttt{>} \to [\iota_1]\alpha \to [\iota_2]\alpha \to [\iota_1+\iota_2]\alpha$. L'abstraction de *paramètres de taille* (`«n,p»`) introduit simultanément une variable localement abstraite de taille et une variable d'expression du même nom, permettant une utilisation transparente des variables dans les termes et les tailles. Les accès aux tableaux (applications d'*indices de tableaux*) insèrent une coercition vers le type `[_]`, indice dont la taille sera déterminée par l'inférence.

Stratification des types. $\mathcal{L}_s$ distingue les types de données (contenant des tableaux) de ceux des fonctions, de la forme $\texttt{<}\eta_1,\ldots,\eta_k\texttt{>} \to \tau_1^i,\ldots,\tau_m^i \to \tau_1^o,\ldots,\tau_n^o$, où les types du domaine et du codomaine sont des types de données. Cette restriction empêche l'ordre supérieur et la currification qui, bien qu'explorés dans [2, 3], ne sont pas utilisés dans les langages synchrones pour l'embarqué critique. Cela ouvre deux perspectives pour le langage :

(i) La définition d'une notion d'*arité* de fonction, grâce à laquelle la spécification des paramètres de taille est optionnelle. Une passe d'inférence détermine les arités des fonctions et insère le nombre d'arguments nécessaires (sans en spécifier les tailles).

(ii) Des *itérateurs n-aires*, qui unifient les définitions précédentes (`map`, `map2`) et possèdent un argument fonctionnel supplémentaire en tête.

Les exemples introduits dans la section 2 écrits dans $\mathcal{L}_s$ sont présentés dans la Figure 8. La syntaxe `dot_product (k,_)` désigne l'application partielle de `dot_product k`. Dans ces termes, les paramètres de taille laissés à l'inférence sont totalement omis.

7 Travaux connexes, discussion et perspectives

L'étude des types dépendants généraux [31] et leur restriction aux tableaux [30] offre un cadre formel pour l'ajout de prédicats complexes dans les types, au prix d'annotations supplémentaires. Un usage restreint aux tableaux permet d'en simplifier l'utilisation. La gestion des tailles dans le langage proposé dans cet article s'articule autour des éléments suivants.

(i) *Un système élémentaire de raffinement des types.* Plusieurs techniques d'inférence de raffinements généraux ont été proposées [8, 17, 26] à l'aide d'outils de résolution de contraintes

(solveurs SMT). Ici, les seuls raffinements singletons ($<\eta>$) et intervalles ($\lfloor\eta\rceil$) suffisent à exprimer des utilisations intentionnelles et extensionnelles des tableaux. En l'absence de sous-typage entre raffinements qui nécessiterait des preuves d'implications, cette restriction permet de vérifier formellement les raffinements et de les propager simplement pendant l'inférence.

(ii) *Un langage des tailles séparé de celui des termes.* Il permet de spécifier des tailles par du polymorphisme proche de celui des types plutôt que par des types dépendants. Ce langage, constitué de polynômes multivariés à coefficients entiers, est adapté aux manipulations formelles grâce à la forme canonique des tailles, en particulier pour l'inférence. Il étend l'expressivité d'autres systèmes [30, 29] et permet d'exprimer des opérations non-linéaires nécessaires aux algorithmes de traitement du signal ou pour écrire un réseau de neurones artificiels. Plutôt que de considérer les tailles comme des constantes (explicites ou abstraites) ou des variables, comme c'est le cas dans [12], les tailles sont ici spécifiées par des contraintes entre méta-variables.

(iii) *Des obligations de preuve explicites (coercitions).* Ces vérifications pallient aux limites du système de types dans la manipulation formelle des tailles. Elles permettent de dépasser les restrictions liées au langage des tailles. Leur résolution est indépendante de la première phase d'analyse statique (inférence, vérification). Comme dans [15], plusieurs implémentations sont envisageables : code défensif, méthodes formelles plus avancées, stratification de l'exécution, etc. Ces coercitions proviennent en particulier des utilisations extensionnelles (par indices) des tableaux, auxquelles il est préférable de substituer des opérations intentionnelles, qui n'en nécessitent pas et apportent de meilleures perspectives de compilation et d'optimisation. Les travaux sur λ^H [7] et SaC [28] proposent des coercitions similaires, introduites systématiquement aux applications par la compilation. Comme pour Futhark [12], nous avons choisi de les rendre explicites afin d'augmenter les garanties apportées par le typage : les erreurs de coercitions ne peuvent provenir que de celles qui sont écrites explicitement dans le programme.

La solution proposée ressemble à l'ajout de dimensions dans les types [16]. Un polymorphisme sur les variables de dimensions y est considéré. La structure du langage des dimensions est cependant plus simple que dans notre cas. Il ne contient qu'une seule opération interne associative et commutative, la multiplication, ce qui permet de définir des types principaux et préserve les propriétés de l'inférence de HM. Les difficultés posées par ce système de type enrichi sont similaires aux nôtres : pas d'unicité des types principaux et le besoin de récursion polymorphe, voire de types dépendants.

Plusieurs travaux autour de SaC proposent des idées qu'il serait certainement possible d'adapter. (i) Dans [29], le polymorphisme de forme (nombre de dimensions) est exploré. Construit sur des types dépendants, il permet de définir des algorithmes génériques sur le nombre de dimensions, grâce un double niveau de description des tableaux par leur rang (nombre de dimensions) et leur forme (taille de chaque dimension). (ii) La définition de tableaux par compréhension est étudiée dans [27]. L'inférence des tailles y est guidée par les accès aux tableaux et évite la spécification des tailles lors des manipulations extensionnelles.

Les perspectives de ce travail sont multiples. Il faudra s'assurer de la compatibilité des constructions de tableaux avec les constructions temporelles propres aux langages synchrones. La mise en place d'un système de types permettant de stratifier l'évaluation, à la manière de [23] est également envisagée. Elle permettrait de restreindre les coercitions impossibles à vérifier à des expressions évaluables à la compilation. Enfin, l'étude de la compilation efficace vers des cibles parallèles utilisées pour le calcul intensif embarqué est prévue.

Bibliographie

[1] John Gilbert Presslie Barnes. *High integrity software: the spark approach to safety and security: sample chapters*. Pearson Education, 2003.

[2] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[3] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM international Conference on Embedded Software*, pages 230–239, 2004.

[4] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

[5] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In *ESOP*, pages 169–196, 2020.

[6] John T Feo, David C Cann, and Rodney R Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[7] Cormac Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[8] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, 1991.

[9] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, Beijing, June 12-13 2012. ACM. Best paper award.

[10] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[11] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[12] Troels Henriksen and Martin Elsman. Towards size-dependent types for array programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 1–14, 2021.

[13] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[14] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[15] C Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, Citeseer, 1996.

[16] Andrew Kennedy. Dimension types. In *European Symposium on Programming*, pages 348–362. Springer, 1994.

[17] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *European Symposium on Programming*, pages 505–519. Springer, 2007.

[18] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[19] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.11. *Available electronically at https://coq. inria. fr/refman*, 2020.

[20] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements.

ACM SIGPLAN Notices, 38(9):213–225, 2003.

[21] Lambert Meertens. Incremental polymorphic type checking in B. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 265–275, 1983.

[22] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, pages 217–228. Springer, 1984.

[23] Hanne R Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of computer programming*, 10(2):139–176, 1988.

[24] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and practice of object systems*, 5(1):35–55, 1999.

[25] François Pottier and Didier Rémy. The essence of ML type inference. 2005.

[26] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[27] Artjoms Sinkarovs, Sven-Bodo Scholz, Robert Stewart, and Hans-Nikolai Vießmann. Recursive array comprehensions in a call-by-value language. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017, Bristol, UK, August 30 - September 01, 2017*, pages 5:1–5:12, 2017.

[28] F. Tang and C. Grelck. User-defined shape constraints in Sac. In R. Hinze, editor, *24th International Symposium on Implementation and Application of Functional Languages (IFL'12), Oxford, UK*. University of Oxford, 2012.

[29] Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.

[30] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, 1998.

[31] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.

Annexes

A Propriétés de la sémantique

Propagation d'erreurs Nous détaillons ici la construction des règles de propagation d'erreurs pour les règles *étoilées* de la sémantique (Figure 4).

La règle $\dfrac{e_1 \rightsquigarrow v_1 \quad e_2 \rightsquigarrow v_2 \quad \dots \quad e_n \rightsquigarrow v_n}{e \rightsquigarrow v}\star$ introduit les règles de propagation suivantes :

$$\frac{e_1 \rightsquigarrow \star}{e \rightsquigarrow \star} \qquad \frac{e_1 \rightsquigarrow v_1 \quad e_2 \rightsquigarrow \star}{e \rightsquigarrow \star} \qquad \dots \qquad \frac{e_1 \rightsquigarrow v_1 \quad e_2 \rightsquigarrow v_2 \quad \dots \quad e_n \rightsquigarrow \star}{e \rightsquigarrow \star}$$

Dans le cas des règles TCase et FCase, les deux premières règles de propagations sont identiques

$$\frac{e_1 \rightsquigarrow \star}{\texttt{case } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \star}$$

Elles sont considérées indiscernables et n'introduisent pas de non déterminisme dans la dérivation de la sémantique.

Déterminisme affaibli. À cause des règles liées au point-fixe, la sémantique définie sous-section 3.2 n'est pas déterministe. Cependant, elle ne peut associer à une expression deux valeurs distinctes différentes de $\star$:

$$\forall e,\ \forall v_1, v_2 \in \mathcal{V} \setminus \{\star\},\ e \rightsquigarrow v_1 \wedge e \rightsquigarrow v_2 \implies v_1 = v_2$$

Preuve. Supposons $v_1, v_2 \neq \star$ vérifiant $e \rightsquigarrow v_1 \wedge e \rightsquigarrow v_2$. Par induction sur les arbres de déductions (finis) des sémantiques, et en raisonnant par cas sur la forme des expressions.

- Point fixe. `fix` $x{:}\sigma = e'$
  Deux règles peuvent s'appliquer, Fix et EFix. Par hypothèses, $v_1 \neq \star$, $v_2 \neq \star$ donc la déduction utilise des instances de la règle Fix (identiques)
- Définition par cas. `case` e_1 `then` e_2 `else` e_3
  Si $e_1 \rightsquigarrow \star$ alors $e \rightsquigarrow \star$ donc, par hypothèse, $e_1 \rightsquigarrow v$, $v \neq \star$. Par induction, dans les deux cas, la condition se réduit vers des valeurs identiques donc une seule règle s'applique.
- Autres cas : dirigés par la syntaxe (1 seule règle applicable), et toutes les prémisses définies uniquement par la conséquence.

B Propriétés du typage

La preuve de la correction du typage par rapport à une sémantique à grands pas ne peut pas être dérivée des propriétés habituelles de préservation du type et de progrès de la réduction, utiles pour les sémantiques à petit pas, car il n'y a pas de différence entre termes bloqués et divergents. Cependant, une analyse générale des conditions de correction pour des sémantiques à grands pas [5] permet de déduire ces résultats de propriétés similaires. Dans le cas des sémantiques non déterministes, il est utile de distinguer deux sortes de corrections, nommées selon l'article cité précédemment :

- *Soundness-must* : aucune réduction possible n'est bloquée
- *Soundness-may* : une réduction possible au moins n'est pas bloquée

Pour la sémantique de $\mathcal{L}_p$, nous prouverons la correction forte (*Soundness-must*). Cette preuve nécessite les lemmes usuels suivants, et les propriétés (S1), (S2) et (S3), détaillées ci-dessous.

B.1 Lemmes initiaux des preuves sur la sémantique

Lemme d'inversion (Dérivation des propriétés des prémisses sous réserve de typage)
Soit Γ, e, σ tel que $\Gamma \vdash e : \sigma$

1. Si $e = n$, alors $\sigma \in \{\texttt{int}, \texttt{<\_>}, \texttt{[\_]}\}$
2. Si $e = x$, alors $x : \sigma \in \Gamma_e$
3. Si $e = \texttt{true}|\texttt{false}$, alors $\sigma = \texttt{bool}$
4. Si $e = \lambda x{:}\tau.\ e'$, alors $\sigma = \tau_1 \to \tau_2$ et $\Gamma, x : \tau \vdash e' : \sigma$
5. Si $e = e_1\ e_2$, alors $\sigma = \tau$ il existe τ' tel que $\Gamma, x : \tau \vdash e_2 : \tau' \to \tau$ et $\Gamma, x : \tau \vdash e_1 : \tau'$
6. ...

Lemme de substitution (Préservation du type par substitution)

$$\left.\begin{array}{rl} \Gamma, x : \sigma' & \vdash e : \sigma \\ \Gamma & \vdash e' : \sigma' \end{array}\right\} \Longrightarrow \Gamma \vdash e\{e'/x\} : \sigma$$

Preuve. Un arbre de dérivation fini de $\Gamma \vdash e\{e'/x\} : \sigma$ est obtenu en replaçant dans une dérivation finie de $\Gamma, x : \sigma' \vdash e : \sigma$ toutes les occurrences de la règle VAR (en nombre fini) par une dérivation de $\Gamma \vdash e' : \sigma'$ (finie). Il faut cependant modifier les environnements pour chaque occurence car ils peuvent être étendus. Cela est valide grâce au lemme suivant : le typage est préservé par extension de l'environnement. En effet, les variables ajoutées sont fraîches et ne peuvent masquer de variables existantes.

Lemme de forme canonique (Dériver la former des valeurs de leur type)
Pour tout schéma de type σ, on note $\{\!|\sigma|\!\} = \{v \in \mathcal{V} | \vdash v : \sigma\}$. Alors,

$$\begin{aligned} \{\!|\texttt{int}|\!\} &= \{n \mid n \in \mathbb{Z}\} \\ \{\!|\texttt{bool}|\!\} &= \{\texttt{true}, \texttt{false}\} \\ \{\!|\cdot \to \cdot|\!\} &= \{\lambda \cdot : \cdot . \ \cdot\} \cup \{p \ v_1 \ \ldots \ v_k \mid k < \nu(p)\} \\ \{\!|\forall \cdot \, . \ \cdot|\!\} &\subset \{\Lambda \cdot \, . \ \cdot\} \cup \{p\} \end{aligned}$$

Note, pour les fonctions, il est nécessaire de considérer les opérateurs partiellement appliqués. Pour les types polymorphes, seule l'inclusion est vérifiée si certains opérateurs sont monomorphes.
Preuve. Analyse par cas sur la forme des valeurs.

B.2 Propriétés du système

Les sémantiques construites automatiquement dans [5] permettent de prouver la correction et la préservation du type à partir de trois propriétés locales détaillées ci-dessous. Ces propriétés ne nécessitent pas d'induction (laquelle est construite dans la preuve générique).

Pour les présenter (restreinte dans le cadre de notre sémantique et notre système de type), la notation d'instance de règle en lignes suivante est utilisée :

$$(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e) \equiv \frac{e_1 \rightsquigarrow v_1 \qquad \ldots \qquad e_n \rightsquigarrow v_n \qquad e_{n+1} \rightsquigarrow v_{n+1}}{e \rightsquigarrow v_{n+1}}$$

Où $e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n$ sont les *prémisses* et $e_{n+1} \rightsquigarrow v_{n+1}$ est la *continuation*, qui produit la valeur résultant de la réduction. Pour les règles ne possédant pas cette forme, une continuation triviale $v_{n+1} \rightsquigarrow v_{n+1}$ peut être ajoutée.

(S1) Préservation locale Pour toute instance $(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e)$, si $\vdash e : \sigma$, alors il existe $\sigma_1, \ldots, \sigma_{n+1}$ avec $\sigma_{n+1} = \sigma$ tels que pour tout $k \in [\![1, n+1]\!]$,

$$\text{si pour tout } h < k, \vdash v_h : \sigma_h, \text{ alors } \vdash e_k : \sigma_k$$

Preuve. Par cas sur les instances de règles (en utilisant le lemme d'inversion).

- Règle β-RED. On suppose qu'il existe σ tel que $\vdash e : \sigma$ et que la dérivation de typage commence par la règle APP.

Par le lemme d'inversion, $\sigma = \tau$ et $e = e_1\ e_2$
On propose le type $\tau' \to \tau$ pour e_1 et τ pour e_2.

- ...

(S2) Progrès existentiel Pour tout $e \notin \mathcal{V}$, si il existe σ tel que $\vdash e : \sigma$, alors il existe une instance de règle de la forme $(j_1, \ldots, j_n, j_{n+1}, e)$
Preuve. Traitement par cas trivial sur les expressions

(S3) Progrès universel Pour toute règle $(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e)$, si il existe σ tel que $\vdash e : \sigma$, alors pour tout $k \in [\![1, n+1]\!]$,

si pour tout $h < k$, $e_h \rightsquigarrow v_h$ et $e_k \rightsquigarrow v$, alors il existe une règle $(j'_1, \ldots, j'_{n'}, j_{n'+1}, e')$ telle que
$\forall h < k,\ j'_h = j_h$, $e' = e$, et $j_k = e'' \rightsquigarrow v$

Montrer que les sous-expressions s'évaluent vers des résultats qui satisfont leur utilisation (forme).
Preuve.

- TCase, FCase. Par le typage (règle Case), $\vdash e_1 : \texttt{bool}$. Donc par le lemme d'inversion $e_1 \rightsquigarrow \texttt{true}|\texttt{false}|\star$. Donc l'une des deux règles s'applique. Pour la continuation, v peut être instancié librement.

- ...

Théorème L'article [5] montre les implications suivantes, qui vérifie les résultats annoncés dans la sous-section 4.2 :

$$(S1) \implies \text{(Préservation du type)}$$
$$(S1) + (S2) + (S3) \implies \text{(Correction du typage)}$$

C Calculs sur les indices

Introduire des règles de calculs sur les indices pour éviter des coercitions ne semble pas pertinent. Bien qu'il soit possible d'ajouter les règles dérivées des inclusions suivantes (dans lesquelles les types sont vus comme leur ensemble de valeurs) :

$$[\eta_1] + [\eta_2] \subset [\eta_1 + \eta_2 - 1] \qquad [\eta_1] + \texttt{<}\eta_2\texttt{>} \subset [\eta_1 + \eta_2]$$
$$[\eta_1] * [\eta_2] \subset [\eta_1 * \eta_2 - \eta_1 - \eta_2] \qquad [\eta_1] * \texttt{<}\eta_2\texttt{>} \subset [\eta_1 * \eta_2 - \eta_2 - 1]$$

il ne sera pas possible de gérer de soustractions, car le type `[_]` ne permet pas de définir de borne inférieure non nulle. De plus, le cas de la concaténation échappera également au système de types puisqu'il requiert la prise en compte des gardes de la définition par cas.

La vérification de ces propriétés relève d'une interprétation abstraite (à l'aide d'intervalles, par exemple). Par soucis de simplicité du système de types, ces extensions ont été écartées, au profit de l'ajout de primitives considérées comme correctes (`window`, `sample`, ...), qu'il est donc inutile de vérifier et dont l'expressivité est semblable.

Sauf à utiliser directement des indices, il est donc nécessaire d'insérer des coercitions pour accéder aux tableaux. Celles-ci seront plus facilement vérifiables du fait des informations de tailles.

D FFT

On suppose disposer d'un type `complex`. Et des opérations associées.

```
val exp : complex → complex
val (+) : complex → complex → complex
val (*) : complex → complex → complex
val (/) : complex → complex → complex
```

La transformée de Fourier rapide s'écrit alors :

```
let div : _ = λn:_.
   (fix f:_ = λi:_.
      case i*i > n then 1 else
      case n%i = 0 then i else f (i+1)
   ) 2

let mat_vec : _ = λA:_. λu:_. map (λx:_. dot_product x u) A
let fft : _ =
   fix f:∀ι. <ι> → [ι]_ → [ι]_ = λn:_. λx:_.
      let tw:_ = λk:[_]. λj:[_]. exp (2i*pi*k*j/n) in
      let size κ = div n in
      case <κ> = 1 then mat_vec tw x else
         let size δ = n/<κ> in
         let x:_ = x ▷ [κ*δ]_ in
         let x:_ = split x in
         let x:_ = map (f <_>) <κ> x in
         let x:_ = transpose (map2 (map2 (*) <_>) <_> tw x) in
         let x:_ = map (f <_>) <δ> x in
         let x:_ = flatten x in
         x ▷ [ι]_
```

Dans le langage de surface (et en syntaxe concrete), on écrit :

```
let mat_vec (A,u) = map (dot_product (_,u)) (A)
let rec fft «n» (x:[n]_): [n]_ =
   let tw [k,j] = exp (2i*pi*k*j/n) in
   let size k = div (n) in
   case k = 1 then mat_vec (tw,x) else
      let size d = n/k in
      let x = (x : [k*d]_) in
      let x = split (x) in
      let x = map fft «k» (x) in
      let x = transpose (map (map (*)) (tw,x)) in
      let x = map fft «d» (x) in
      let x = flatten (x) in
      (x : [n]_)
```

Remarques : (i) Il est nécessaire de préciser les occurrences de n dans le type de la valeur (fonction) récursive définie. (ii) le tableau tw est instancié de deux façons : `[`ι`][`ι`]complex` et `[`κ`][`δ`]complex`