université
PARIS-SACLAY

NNT : 2018SACLY018

Vérification des résultats de l'inférence de types du langage OCaml

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'École Nationale Supérieure des Techniques Avancées

École doctorale n°580 : Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Paris, le 23 octobre 2018, par

Pierrick Couderc

Composition du Jury :

Mme. Catherine Dubois Professeure, Université Paris-Saclay (– ENSIIE, Samovar)	Présidente du Jury
M. Emmanuel Chailloux Professeur, Sorbonne Université (– LIP6, APR)	Rapporteur
M. Jacques Garrigue Professeur, Université de Nagoya (– Graduate School of Mathematics)	Rapporteur
M. Sylvain Conchon Professeur, Université Paris-Saclay (– LRI, VALS)	Examinateur
M. Alain Frisch Directeur technique, Lexifi	Examinateur
M. Michel Mauny Directeur de Recherche, Inria Paris (– Gallium)	Directeur de thèse
M. Fabrice Le Fessant Directeur technique, OcamlPro	Co-Directeur de thèse

Thèse de doctorat

Remerciements

Septembre 2014, j'annonce à Çagdas que mon dossier CIFRE est envoyé : *"Mais pourquoi tu fais une thèse ? Tu aurais dû me demander avant ! Ne fais pas de thèse !"*. Çagdas, tu avais raison. À ceux qui auront le courage de lire ces quelques lignes et se poseraient la question : une telle sagesse ne saurait être perdue.

Il s'est passé 4 ans depuis, ça n'a pas toujours été facile, mais j'aimerai d'abord remercier Michel, qui a accepté alors d'encadrer mon travail et de m'acceuillir à l'ENSTA. Même s'il n'a pas toujours été facile de se coordonner, je sais que tout mon travail n'aurait pas pu être aussi avancé sans ses conseils et ses connaissances pointues sur le sujet.

J'aimerais remercier Fabrice, à la fois pour m'avoir donné ma chance dès mon stage de Master 1, mais aussi d'avoir continué à croire en mes capacités au point de me laisser tenter ma chance avec un doctorat, et par la suite chez OCamlPro.

Je remercie bien entendu mes rapporteurs Jacques Garrigue et Emmanuel Chailloux. Jacques, pour ses conseils et son avis tout au long de ma thèse, et pour avoir su m'aiguiller lors de ma lecture du compilateur. Emmanuel, d'abord pour avoir appuyé mon choix vers cette voie durant ma dernière année de Master, mais aussi pour ces discussions diverses, qu'elles soient scientifiques ou non, durant toutes ces années. Je n'oublie pas mes examinateurs Sylvain Conchon, Alain Frisch et Catherine Dubois qui, sans le savoir, ont contribué à mon travail, que ce soit en m'enseignant OCaml et en me donnant la passion pour la compilation, que ce soit pour des explications et des évolutions du compilateur lui-même, ou pour leurs travaux de recherche qui m'auront grandement inspiré.

Bien sûr, je ne peux oublier tous mes collègues d'OCamlPro, qui m'ont acceuilli toutes ces années, et avec qui j'espère pouvoir continuer à travailler dans ces conditions si particulières : Muriel, Cagdas, Pierre, Louis, Mohamed, Thomas, Michaël, Vincent, Alain, Raja, Maxime, Steven, Guillaume, David et Albin, et bien sûr Grégoire et Benjamin, petits anges partis trop tôt pour voguer vers d'autres horizons. Merci à vous, car sans le savoir, vous aurez chacun apporté votre pierre à cette centaine de pages qui résument mes 4 dernières années.

Je n'y étais pas souvent, mais je n'oublie pas mes quelques collègues de l'U2IS à l'ENSTA : Benoît, François, Alexandre et Julien. J'en retiendrai beaucoup de science sur l'art de la contrepèterie, de la cueillette de champignons, du ramassage des noisettes et de la dégustation de fromage.

En revanche, s'il y a des gens que je ne suis pas sûr de vouloir remercier, ce sont bien mes amis eux-même doctorants. Bien que notre groupe œuvre pour la science, je pense qu'il s'est aussi beaucoup diversifié vers des sujets de discussion, parfois houleuses, diverses et variées, qui n'auront pas toujours fait avancer la science. Merci Rémy, tu auras réussi à terminer premier cette course vers la soutenance. Merci Mattias pour ta mauvaise foi légendaire, même si, et je ne l'admettrai qu'ici, tu auras souvent eu raison. Merci Albin, petit être barbu disparu le 30 septembre dernier pour laisser place à ton jumeau maléfique et imberbe. Je n'oublie pas mes anciens camarades de M2, qui auront parfois aussi décidé de faire une thèse, et parfois eu la sagesse de s'arrêter là : Pierre, Roven, Inigo, Vincent, Mat(t)hieu.

Enfin, je n'oublie pas mes amis qui eux n'auront vu que tout ça de loin, sans comprendre ce que je faisais, voire parfois même en se demandant si je travaillais vraiment. Merci Romain, Eva, Alexandre. Merci à mes deux camarades de brassage, Jérémy, Geoffroy, et bien sûr Clémence et

Pascaline, pour nous avoir subi. Un petit mot pour Julien, Mickaël et Matthieu, qui avait appris avec effroi mon choix de faire une thèse, d'abord, et en OCaml, ensuite.

Merci à ma mère qui aura suivi tout ça de loin, à mon beau-père, à mon frère, qui m'ont apporté leur support. Et à mon père, qui, j'en suis sûr, aurait été fier de moi.

Finalement, et c'est sans doute le plus important pour moi, car c'est toi qui m'aura subit le plus ces 4 années, qui aura supporté mon mauvais caractère, mes angoisses, qui a su me réconforter, qui a su m'aider, qui a toujours été là lorsque j'en avais besoin. Je ne pourrai jamais te remercier assez, Julie.

Table des matières

Introduction

Quiconque a déjà hérité de l'antique voiture familiale le sait : la maintenir en état relève parfois plus du miracle que du bon sens. Les réparations successives qu'elle aura subies, parfois malines, parfois hasardeuses, auront rendu toute modification compliquée, et chaque réparation entrainant parfois d'autres problèmes *a priori* improbables. Imaginez réparer l'un des véhicules issus du cerveau malade des célèbres britanniques de *Top Gear* : personne n'oserait y toucher [57].

Au risque de décevoir le lecteur de cette thèse amateur de voitures, ce manuscrit ne parlera ni de mécanique, ni d'automobile. Tout comme cette antique voiture ayant subit les nombreuses réparations et améliorations de ses propriétaires successifs, nous allons ici nous intéresser à un compilateur dont l'implémentation date d'une vingtaine d'années et qui a été entretenu au fil du temps [1] : OCaml. La maintenance d'un programme écrit par un tiers est un problème connu, la logique derrière une implémentation étant souvent très personnelle. L'histoire est connue et n'est pas à refaire, mais intéressante pour remettre en contexte la problématique de cette thèse. OCaml est un compilateur pour un dialecte de ML [32, 47]. Il s'agit à l'origine de CAML [45], une première version écrite au milieu des années 80 dont l'intérêt était la compilation de langages fonctionnels, et en particulier ML, vers la Machine Abstraite Catégorique [44, 19]. Une nouvelle implémentation sera proposée cinq ans plus tard, sous le nom de *Caml-Light* [68], une version plus rapide compilant vers une machine virtuelle et possédant notamment un glaneur de cellules générationnel [23]. OCaml en est le successeur : il y ajoute un système de modules avec foncteurs applicatifs [41] et une couche d'objets avec de l'héritage et du sous-typage [60]. Le langage connaît de nombreux ajouts de fonctionnalités par la suite : variants polymorphes [28], modules de première classe, types algébriques gardés [30] ou encore arguments de fonctions labellisés [2]. Pour résumer brièvement, le compilateur OCaml est le résultat de l'évolution des techniques de compilation et de typage issues de 30 ans de recherche, dont une vingtaine sur la même implémentation. Une partie de ces ajouts ont été formalisés et souvent publiés, et le tout implémenté et maintenu par divers développeurs. L'implémentation de l'inférence de types est intelligente et conçue pour être rapide, et par conséquent écrite parfois de manière complexe et difficilement compréhensible, à tel point qu'il peut être compliqué pour quelqu'un de non investi dans son développement d'en comprendre les détails. Le système de types n'est pas formalisé dans son entièreté mais est plutôt réparti entre divers articles de recherche et paragraphes du manuel. Cela a plusieurs conséquences :

1. ∧Contrairement aux trois présentateurs de la BBC, les différents développeurs et mainteneurs de celui-ci sont parfaitement sains d'esprit.

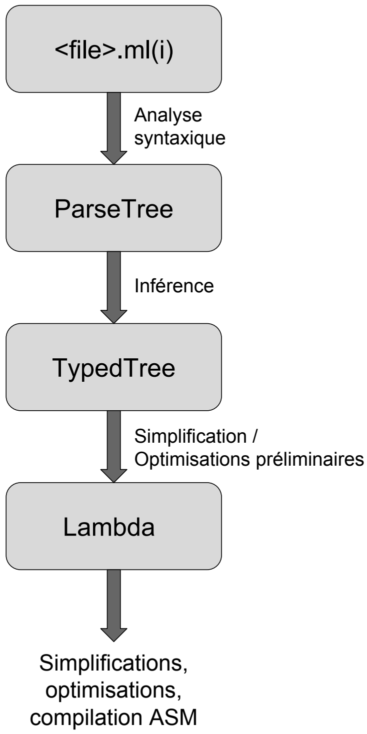

FIGURE 1 – Représentation de la chaîne de compilation d'OCaml (jusqu'au Lambda)
La chaîne de compilation dédiée au langage source est relativement classique. Seul un langage intermédiaire est typé : le Typedtree, qui servira de base pour tout le travail de vérification.

— il n'existe pas de règles formelles sur lesquelles s'appuyer pour décrire et implémenter le système de types complet d'OCaml ;
— il n'existe de fait pas de standard pour le langage : le compilateur constitue le standard.

Disposer de règles formelles permettrait de comprendre la théorie implémentée dans le système de types et de s'assurer que les évolutions respectent ce formalisme. De plus, les évolutions du langage pourraient être accompagnées d'une évolution de ce formalisme. Le travail de cette thèse est d'essayer de combler le vide représenté par cette conséquence en proposant une formalisation d'un sous-ensemble de ce système de types, sous la forme d'un ensemble de règles de typage du langage. Ce travail n'a en revanche pas vocation à résoudre le problème de la standardisation, relevant plutôt d'un comité des concepteurs et utilisateurs du langage, mais pourrait être une brique vers un tel processus.

Ce travail de formalisation s'appuie sur les différents articles décrivant les évolutions apportées au langage, mais aussi l'implémentation de celui-ci dans sa version 4.02. Plutôt que d'écrire un système de types pour un langage avec inférence, le travail de reconstruction se base sur l'un des langages intermédiaires du compilateur : les arbres de syntaxe annotés, ou *Typedtree* (voir figure 1 pour la représentation de la chaîne de compilation d'OCaml). Cette structure de données, que l'on désignera sous le nom de *TAST* (pour *Type-annotated Abstract Syntax Tree*) tout au long de ce manuscrit, est le résultat de l'inférence : il s'agit d'un dérivé de l'arbre résultant de l'analyse syntaxique dont le moteur d'inférence de types a annoté chacun des nœuds avec des informations, en particulier le type inféré pour celui-ci. Le principe de cette thèse est de considérer ce TAST comme un arbre de preuve de typage du programme, et d'écrire un ensemble de règles pour en vérifier la cohérence. La cohérence d'un nœud est donc définie en fonction de l'expression qu'il représente et ses sous-nœuds : l'annotation d'un nœud représentant cette expression est valide si ses sous-nœuds sont eux-même cohérents et si les conditions inhérentes à l'expression sont respectées. Cette vérification de cohérence peut se présenter sous la forme de règles qui respectent le système de types. Cette présentation peut alors être lue comme étant une spécification de ce système.

Une deuxième approche de ce travail de thèse n'est pas seulement l'écriture d'un système de types pour un sous-ensemble d'OCaml, mais aussi de simplifier certaines composantes issues du compilateur. L'inférence utilise énormément l'unification tout au long du typage des programmes. Celle-ci est impérative et destructive (elle modifie en place des types), ce qui rend fastidieux le processus de retour en arrière et plus difficile l'interprétation des erreurs de types. Cette implémentation impérative a cependant l'avantage d'être rapide, ce qui peut être un composant essentiel pour la compilation [2]. De plus, l'unification a l'avantage et l'inconvénient d'être capable d'effectuer plusieurs vérifications en même temps : il s'agit d'un mécanisme intéressant du point de vue de l'implémentation et de la réutilisation de code mais problématique dans le cadre de la formalisation où le but est de rendre le système de types le plus clair possible. A cet effet, pour la vérification de cohérence, l'unification est remplacée par un ensemble d'opérations dédiées à la vérification. En effet, l'unification est conçue pour l'inférence et n'est donc pas appropriée pour la vérification. Toutes les opérations implicites réalisées par l'unification doivent être remplacées par des versions explicites : équivalence, instanciation, filtrage de types et vérification de bonne formation.

Finalement, ce travail de formalisation de la vérification de cohérence se veut aussi proche que possible d'un système exécutable : le but est de s'assurer que chacune de ces règles est effectivement implémentable et non simplement théorique. Le résultat pratique est un logiciel de vérification de cohérence de Typedtree (lui-même implémenté en OCaml), dont le code est dérivé directement des règles. Un tel vérificateur permet notamment de s'assurer de la cohérence des programmes acceptés par le moteur d'inférence, et donc de vérifier sa correction.

Le premier chapitre de ce manuscrit est un état de l'art des différentes techniques de vérification de l'inférence de types, tels que le typage des langages intermédiaires à des fins de correction, ou l'écriture de typeurs dans des assistants de preuves tels Coq et HOL pour s'assurer que l'implémentation respecte le formalisme d'origine. L'intérêt ici est de remettre ce travail dans un contexte où la vérification de programmes est une sujet fortement étudié. On y verra néanmoins que la technique proposée dans ce manuscrit est encore différente de toutes celles présentées, ce qui s'explique par la contrainte de vérifier un langage existant en se basant simplement sur les résultats de sa compilation, sans devoir le réécrire ou le formaliser dans un langage d'assistant de preuve, ce qui s'avèrerait fastidieux étant donné la densité d'OCaml.

Le second chapitre est dédié à l'étude de cette technique sur un langage que l'on pourra

2. ∧Avec l'ajout du langage intermédiaire Flambda dédié à l'optimisation, les éventuels problèmes de lenteur de l'inférence de types deviennent marginaux à côté du temps de compilation rallongé par les passes d'optimisation.

considérer comme idéal, en d'autres termes que ses constructionsurl contiennent l'ensemble des informations nécessaires à la vérification de l'inférence. On y présente donc une version de MiniML (sans références) dont chaque nœud est annoté, ainsi que ce qui sera l'un des objectifs de la vérification des résultats du typage d'OCaml : la décomposition de l'unification en différentes opérations. Cette décomposition sera essentielle pour comprendre le rôle joué par l'unification dans l'inférence et donc l'ensemble des propriétés qu'elle garantie. En plus de ces règles de vérification d'arbres de syntaxe annotés est présentée une sémantique opérationnelle de ce langage. On verra notamment que celle-ci n'est pas si simple à définir à cause de la présence des annotations de types et des réductions de types qu'il sera nécessaire d'effectuer pour éventuellement prouver la préservation du typage. Ce chapitre est accompagné d'une présentation en OCaml d'un vérificateur pour cette version de MiniML, ce qui servira de support à l'idée présentée dans cette thèse qui est d'écrire le système de vérification de manière algorithmique. Cette implémentation sera une transcription directe des règles de vérification modulo le choix des structures de données pour représenter les termes, les types et les environnement. A cette implémentation est également présentée une formalisation en Coq de ces règles de vérification. Cette formalisation sera fortement basée sur les travaux de Charguéraud [12] sur la représentation des variables libres et liées et sur la bibliothèque LibLN [13] basée sur ses travaux et dédiée à la preuve de langages. Cette formalisation permettra de valider l'ensemble des propriétés extraites de la décomposition de l'unification, et pourra servir à des futurs travaux sur une éventuelle preuve de sureté de ce système de types vis-à-vis de la sémantique proposée. Le reste de ce manuscrit se consacre à adapter la méthode présentée précédemment au TAST d'OCaml

D'abord, le troisième chapitre est une présentation formelle du Typedtree, de ses composantes et aborde quelques détails issus de l'unification nécessaires pour la compréhension de la structure de donnée. Ce chapitre permet de comprendre exactement le langage qui sera étudié et ses différences avec OCaml, différences qu'il est nécessaire de comprendre par la suite de ce manuscrit. Il s'agit d'un chapitre qui peut être lu comme une forme d'état de l'art.

Ensuite, le quatrième chapitre aborde la formalisation de la vérification de cohérence du Typedtree d'OCaml pour le cœur du langage, autrement dit tout ce qui ne concerne ni modules ni objets. Encore une fois, il est important de rappeler que le langage traité ne sera pas OCaml mais le Typedtree. Il s'agira donc d'adapter les concepts présentés au second chapitre, en particulier la décomposition de l'unification qui vérifie implicitement beaucoup plus de propriétés que dans le cas de MiniML. On verra également que ce langage n'est pas parfait et qu'il sera nécessaire de retrouver certaines informations inférées par le typeur mais pas annotées dans le Typedtree, tel que la quantification des schémas de types, ce qui potentiellement problématique pour la vérification d'instanciation. Ce chapitre présente notamment des extraits du vérificateur accompagnés de la règle associée, pour montrer la correspondance directe entre le code et la règle, comme cela a été fait pour MiniML. Il est important de noter que toutes ces règles sont effectivement implémentées sous la forme d'un vérificateur de types pour les Typedtree de la version 4.02.3 d'OCaml. Le choix de la version étant un choix arbitraire et non lié à une quelconque difficulté. Le vérificateur est disponible à l'adresse : `github.com/OCamlPro-Couderc/ocp-typechecker`. Les règles de vérifications telles que décrites dans ce chapitre pourront être vue comme une présentation formelle d'un large sous-ensemble du système de types dOCaml.

Le chapitre suivant présente une sémantique opérationnelle pour le langage. Celui-ci reprend les idées énoncés pour la sémantique de MiniML et l'applique au Typedtree. Ce chapitre est en revanche purement théorique et n'est donc pas implémenté dans un évaluateur. Néanmoins il est intéressant de pouvoir raisonner sur une telle sémantique pour pouvoir définir et comprendre le système de types. De plus, dans le cas d'une future preuve de sureté, un tel travail serait nécessaire.

Finalement, la conclusion de ce manuscrit s'attarde sur les potentiels développements de cette technique de vérification et les perspectives d'utilisation, et résume brièvement l'ensemble du travail présenté dans les chapitres précedents.

Enfin, en annexe est donnée une implémentation Coq d'un ML sans références mais avec couples et une forme de filtrage de motifs, dont les règles de typage sont écrites directement dans l'assistant de preuve ainsi qu'un vérificateur, dont la propriété de correction de l'algorithme vis-à-vis de la spécification du système de types est prouvée. Une seconde annexe reprend les différentes règles de vérification et de la sémantique dynamique et les compile en un seul document.

Chapitre 1

Etat de l'art

Ce bref chapitre présente un tour d'horizon des différentes techniques de vérification de cohérence de programmes par rapport à la spécification d'un système de type. Il s'attarde sur les différentes solutions mises en place pour vérifier l'inférence de types *a posteriori*.

La vérification de programmes déjà compilés (ou au moins typés) peut avoir lieu de multiples manières. Par exemple, le programme peut-être transformé vers un autre langage et son système de types encodé dans celui-ci. Une telle méthode permet de formaliser et prouver un langage qu'on dira "*noyau*", et laisser à ce sous-langage le soin de s'occuper de la propriété de sûreté du langage de surface. L'ensemble des transformations jusqu'à l'assembleur peuvent elles aussi être typées, garantissant alors la sûreté sur l'ensemble de la chaîne de compilation. Une autre solution est de dériver une preuve au moment de la compilation pour accompagner le programme et vérifier certaines propriétés *a posteriori*. Une solution est la transformation de programmes vers des langages de preuves : le soin de la preuve est laissé au programmeur mais celui-ci n'a pas besoin de l'écrire dans le langage de l'assistant de preuve en question.

1.1 Vérification de l'inférence dans les langages fonctionnels

La vérification de correction de l'inférence peut être intégrée dans le processus de compilation. La solution la plus évidente est alors de compiler un arbre de syntaxe typé vers un langage intermédiaire lui-même typé, et de vérifier un tel langage. Un tel processus a été implémenté pour des langages tels que SML dans les compilateurs TIL ([64]) et CakeML, mais aussi pour la vérification de programmes Haskell dans le compilateur *ghc*.

1.1.1 *Typed Intermediate Languages*

Le compilateur *TIL* [65] est l'une des premières occurrences de compilateur dont l'ensemble de la chaîne de compilation propage les informations de types issues de l'inférence. L'utilisation de représentations intermédiaires typées garantit que toutes les transformations de programmes sont correctes et conservent les types. Ainsi, il est possible d'écrire des passes d'optimisations en s'assurant que celles-ci ne génèreront pas d'erreurs de types à l'exécution, mais également que ces informations de types peuvent orienter les-dites optimisations, qui n'auraient pu être effectuées autrement. De cette manière, il est possible de vérifer l'inférence de types *a posteriori*, dès la compilation du langage de haut niveau vers le premier langage intermédiaire.

Une chaîne de compilation entièrement typée permet également d'assurer une meilleure maintenabilité du code et de ses évolutions. En effet, être capable de typer et vérifier chacune des étapes de compilation rajoute une garantie que toute modification entraînant une erreur de types à l'exécution d'un programme sera rattrapée par l'une des représentations intermédaires. De plus, cela force toute évolution de système de types de surface (SML) à être compilée de manière sûre et à être correctement spécifiée. *TIL* sera suivi par *TIL(T)*[16, 15], une évolution du compilateur prenant en compte l'ensemble de SML, apportant notamment des améliorations du langage intermédaire [54] utilisant Système F_ω, système de types présenté dans la section suivante.

TAL* : *Typed assembly language Un des travaux connexe à *TIL* est sans aucun doute celui sur *Typed Assembly Language* [51], une chaîne de compilation transformant un langage basé sur Système F vers un assembleur typé. Ce travail formalise les différentes étapes de transformation d'un lambda-calcul polymorphe vers plusieurs représentations intermédiaires typées, jusqu'à un assembleur RISC lui-même typé. Cette chaîne de compilation comprend une traduction en passage par continuation, une conversion des clotures typée, introduisant des types existentiels pour représenter les variables libres embarquées dans la clôture, et enfin une représentation intermédiaire permettant l'allocation mémoire, avant transformation vers l'assembleur typé.

1.1.2 Haskell : Core et Système FC

Haskell [55] est un langage fonctionnel dit "pur" : il n'est pas possible d'effectuer des effets de bords. Il possède également un mécanisme de polymorphisme *ad-hoc* [36] (on parlera également de surcharge), autrement dit plusieurs implémentations d'une même fonction pour des types différents. À la compilation, une implémentation est choisie en fonction du type de ses arguments. Un tel mécanisme permet principalement de spécialiser certaines opérations en fonction d'un type donné. A cela s'ajoute un système de *kinding*, soit une manière de classifier les types et de rendre le langage de types plus expressif. Pour accompagner ceux-ci, le langage possède un système de familles de types, autrement dit des fonctions de types vers types. Enfin, tout comme OCaml, Haskell est capable d'exprimer des types algébriques gardés. L'ensemble de ces constructions rendent alors le système de types et l'inférence assez complexe à mettre en oeuvre et à vérifier. La solution adoptée pour vérifier un tel langage est la compilation vers un langage intermédiaire lui-même typé explicitement, où toutes les égalités de types sont de premier ordre : Système F_C.

Système F_C [62, 46, 67, 66] est une extension de F_ω, lui-même une extension de Système F avec classification des types à l'aide d'un système de *kinding*. F_C ajoute dans le langage de kinds un mécanisme d'égalités de types, ainsi qu'un opérateur de coercions. Le langage possède également des types algébriques accompagnés d'une construction pour effectuer du filtrage. Ce langage intermédiaire fait suite à une version de Système F dans lequel sont directement ajoutés les GADTs, langage intermédiaire introduit pour gérer l'ajout de cette construction dans le langage source (Haskell).

Une égalité est de la forme $\tau_1 \sim \tau_2$. Les égalités étant des kinds, il est alors possible de les utiliser pour classifier des types. Ainsi, le type

$$\forall a.\forall b.\forall c :: a \sim b.\, a \to b$$

est le type d'une fonction qui, pour deux types a et b quelconque et un type c représentant une égalité de type, prend une valeur de type a en argument et retourne un type b. Ce type c est un témoin d'équivalence entre les types a et b.

La coercion est une construction située au niveau des expressions, de la forme $e \blacktriangleright \gamma$, où γ est un type de kind $\tau_1 \sim \tau_2$. Ainsi, si l'expression e est de type τ_1, l'application de la coercion est une valeur de type τ_2. On peut donc écrire la fonction du type donné précédemment :

```
val cast = /\a. /\b. \/c :: a ~ b. \x : a. x ► c
```

Ce kind et cette construction permettent alors d'exprimer les types algébriques gardés sans nécessiter leur ajout dans le langage. Prenons alors un exemple de témoin de type, que l'on retrouvera par la suite comme exemple de cette construction :

```
data T :: * => * where
| Int : ∀ a. ∀ c :: a ~ Int. T a
| Bool : ∀ a. ∀ c :: a ~ Bool. T a
| Pair : ∀ a b p. ∀ c :: p ~ (a * b). T a -> T b -> T p
```

Chacun des constructeurs de type `T a` est alors un témoin de l'égalité du paramètre `a` à un type concret donné. On peut ainsi écrire la fonction suivante :

```
f : ∀ a: *. T a → a
  = /\a. \w : T a.
  case w where
  | Int (c : a ~ Int) -> 0 ► (sym c)
  | Bool (c : a ~ Bool) -> True ► (sym c)
  | Int (l : *) (r : *) (c : a ~ (r * l))
        (tx : T l) (ty : T r) ->
    let x : l = f [l] tx in
    let y : r = f [r] ty in
    (x * y) ► (sym c)
```

Puisque l'égalité de type est contenue dans le constructeur et est de première classe, il est possible de la récupérer à l'aide d'un filtrage sur chacun des constructeurs possibles. Pour que le filtrage soit correct, il faut que toutes les branches soient de même type. On peut alors retourner une valeur du type du témoin voulu, mais dont le type est le résultat de l'application de la coercion. *sym* est une fonction opérant une symétrie sur l'opérateur d'égalité.

A cela s'ajoute la construction `axiom C : t1~ t2`, permettant de déclarer des égalités de types. Celle-ci permet alors de supporter les familles de types ainsi que les types associés aux classes de types. La particularité de cette construction est d'introduire des égalités sans être vérifiées. Cela pose alors un problème de sûreté : par exemple, il est tout à fait possible de définir une égalité de la forme `axiom Bad : Int ~ String`, et donc par la suite d'utiliser des entiers là où des chaînes de caractères seraient attendues, grâce à l'opération de coercion. Néanmoins, on suppose que ce langage intermédiaire n'est jamais écrit "par un humain" mais toujours une traduction issue du langage source. On peut s'assurer par construction que de tels axiomes ne peuvent être produits par le compilateur et donc garder une forme de cohérence dans le programme.

Ce procédé de transformer un programme typé vers un langage plus simple tout en conservant les types et en les compilant vers un système plus restreint permet de vérifier après typage que l'inférence est correcte. L'intérêt est donc de vérifier un système restreint, sur lequel il est possible de prouver plus de propriétés, mais surtout dans un langage intermédiaire qui sera stable dans le temps : ses évolutions ne sont motivées que par des changements dans le système de types du langage source qui ne seraient pas exprimables dans le langage intermédiaire actuel. De plus, il est possible d'utiliser les types propagés pour effectuer des optimisations plus complexes. Un autre avantage de cette approche est que le langage intermédiaire et son vérificateur de types associés sont intégrés à la chaîne de compilation. Ainsi, les évolutions du langage source sont toujours propagées dans le langage intermédiaire, et celui-ci est maintenu au même niveau que l'ensemble du compilateur.

En revanche, cette approche n'est pas toujours idéale. Tout d'abord, dans le cadre d'un vérificateur concret, et donc de résultats sur des programmes autres que "jouets", les erreurs issues de l'inférence sont identifiées dans le langage intermédiaire et non dans le langage source. Par exemple, il est probable que deux constructions de haut niveau se compilent vers une même construction intermédiaire avec un type légèrement différent. Il devient alors plus complexe de réfléchir en fonction du typage du langage source, puisque les types ont eux-mêmes été compilés vers un autre système. De plus, il ne résoud pas le problème de donner une spécification formelle du système de types de son langage source, mais une spécification de la traduction de ce langage vers une représentation intermédiaire, avec la spécification de celle-ci.

1.1.3 MLton

MLton est un compilateur optimisant pour SML, dont chaque étape de transformation du programme est optimisée. Chacun des langages intermédiaire est lui-même typé, à l'exception de la dernière représentation "Machine", évidemment proche du langage machine. Après analyse syntaxique, le programme est traduit sous la forme de "CoreML", une représentation proche de ML dans laquelle les foncteurs et modules ont été éliminés grâce notamment à une passe d'alpha-conversion rendant tous les noms uniques, et donc ne nécessitant plus l'utilisation de modules. Une étape de beta-réducation a lieu pour les foncteurs, avant d'être effectivement éliminés comme des modules classiques. Les abréviations de types sont également expansées. CoreML est une représentation intermédiaire dans laquelle les types sont propagés, mais le compilateur n'est pas capable d'en vérifier la cohérence.

Cette représentation intermédiaire est ensuite traduite vers une nouvelle représentation appelée "XML" [1], une représentation du programme où tous les nœuds sont annotés, et la quantification explicite : chaque valeur (ou fonction) est annotée avec une liste de variables, celles-ci étant les variables généralisées par la-dite valeur (ou fonction). Réciproquement, les instantiations de types polymorphes sont explicites, et comme nous le verrons plus tard, ont lieu exclusivement au niveau des occurrences de variables. La transformation de CoreML vers XML est principalement utilisée pour la compilation du filtrage de motifs et pour séparer les élements dits dynamiques (les valeurs) des élements statiques (les déclarations de types algébriques) : ceux-ci sont alors tous remontés au début du programme. Cette représentation intermédiaire possède un vérificateur de

1. ∧A ne pas confondre avec le format de description *eXtensible Markup Language.*

types qui n'est présent que pour permettre de s'assurer de la correction de la transformation et potentiellement de l'inférence. Celui-ci n'est d'ailleurs pas activé par défaut.

Enfin, le programme est ensuite monomorphisé : il s'agit du langage SXML. À ce niveau du programme, chaque valeur polymorphe est dupliquée pour chaque instance à laquelle elle apparaît. La monomorphisation est également appliquée sur l'ensemble des types algébriques. SXML est ensuite tranformé vers une représentation sous forme SSA [2], elle-même typée. S'ensuivent un ensemble de transformations jusqu'au langage "Machine" et à l'assembleur.

L'étape de vérification du langage intermédiaire XML est proche de l'approche étudiée dans cette thèse : le programme est annoté par un type à chaque nœud, et la vérification s'assure de la cohérence d'un nœud sachant les annotations de ses nœuds fils. Néanmoins, le langage est plus restreint que le TypedTree d'OCaml. Au delà des constructions supplémentaires que compte OCaml, les modules et foncteurs ont déjà été éliminés, les abréviations de types ont été expansées, ce qui réduit alors la complexité du système de types qui doit être mis en oeuvre pour vérifier un tel langage. La différence principale que l'on cherchera à traiter est surtout l'absence de quantifications des variables de types polymorphes et d'instanciations explicites des schémas de types, informations présentes dans le cas de XML. Cette vérification effectuée par MLton permet donc de s'assurer de la correction de l'inférence, mais ne peut être considérée comme une implémentation du système de types de SML, comme l'est un vérificateur de types pour le TypedTree d'OCaml.

1.2 Vérification de l'inférence : machine virtuelles typées

Certains langages permettent la compilation vers une machine virtuelle : cela permet une meilleure portabilité des programmes puisque la gestion des différents systèmes est déférée à la machine virtuelle. Pour rendre celles-ci plus robustes, il est possible de leur envoyer un *bytecode* typé qui sera vérifié avant sont exécution, et ainsi en vérifier sa cohérence. Cela permet entre autres d'envoyer des programmes avec une forme de garantie sans avoir besoin de fournir les sources pour s'assurer que le programme est effectivement bien typé.

Java [33] est compilé vers une machine virtuelle : la *Java Virtual Machine* [43]. Celle-ci effectue de base une étape de vérification de cohérence avant de charger du bytecode, et rejette tout code qui pourrait causer une erreur de types. Stephen Freund [26] présente une formalisation du système de types pour un sous-ensemble de la JVM. Ce système de de types et le vérificateur associé permettent entre autres de s'assurer que des objets sont bien initialisés avant d'être utilisés, en plus de vérifier la compilation des méthodes dont la compilation peut-être affectée par l'utilisation d'exceptions et donc du gestionnaire d'erreurs. Certains aspects du langage ne sont pas gérés, comme les modificateurs `final` ou `static`, la concurrence ou la vérification des `packages`, le mécanisme d'espaces de noms. La vérification se fait en trois étapes distinctes : la génération du flot de contrôle, l'identification des parties polymorphes du programme et l'inférence de types à l'aide d'une analyse de flots de données. Cette analyse va plus loin que la simple vérification de cohérence, et permet par exemple de vérifier l'utilisation correcte des objets, et permet également d'éliminer des vérifications de pointeurs non nuls à certains endroits du programme. On notera des expérimentations [5] de la compilation de SML vers la machine virtuelle Java, qui donnera

2. ∧Static Single Assignement.

lieu par la suite à SML.NET[6], son équivalent pour la machine virtuelle **.Net**[61]. Ce compilateur pour Standard ML propose un ensemble langages intermédiaires tous typés, permettant alors de garantir que le typage est conservé à chaque transformation.

OCaml est également compilé vers une machine virtuelle dérivée de ZINC [42]. Celle-ci ne fait que très peu de vérifications et le *bytecode* reçu n'est pas typé : si on se réfère la chaîne de compilation en figure 1, les types sont abandonnés à l'étape de compilation du Typedtreevers le Lambda. Or, le bytecode est compilé à partir du lambda : il devient donc difficile de donner un type au bytecode. Un travail de vérification de cohérence a été effectué en ce sens, pour permettre d'ajouter des garanties de sûreté au code executé par la machine virtuelle [1]. Celui-ci modifie le bytecode pour y ajouter des informations de types minimales, permettant une forme d'inférence locale. Pour éviter d'alourdir le bytecode, seuls les labels sont annotés avec leur type, leur arité et le type de leurs variables libres. Ces informations sont alors suffisantes pour une forme d'inférence locale et suffit à vérifier la sûreté du bytecode. Il est à noter qu'il existe une implémentation d'une chaîne de compilation pour OCaml capable de compiler un programme pour la machine virtuelle **.Net** : OCamIL[10, 50]. Ce travail réutilise une partie de la chaîne de compilation d'OCaml, en proposant deux approches : l'une propage les types depuis le Typedtree, l'autre les reconstruit une fois fois arrivé à la représentation intermédiaire *Clambda* [3]. Celui-ci ne propose néanmoins pas de système de types ni même de vérification des langages intermédaires dans lequel sont propagés les types : ceux-ci sont essentiellement utilisés pour la machine **.Net**, donc le langage est typé.

1.3 Programmes avec preuve embarquée

La technique de programmes avec preuves embarquées [53] permet d'écrire des programmes capables d'importer du code à l'exécution, code lui-même accompagné d'une preuve de correction vis-à-vis de sa spécification. Il s'agit donc de pouvoir écrire des programmes extensibles capables de lire une preuve et donc d'ajouter une notion de sûreté au méchanisme de lien dynamique de code.

Si cette technique est intéressante, elle ressemble à l'idée de vérifier des arbres de typage : le vérificateur récupère un programme sous la forme d'une arbre de typage qu'il est capable de vérifier. En revanche, celui-ci teste la correction de l'inférence de type, le programme ainsi vérifié est ensuite "jeté", il n'y a donc pas de problématique de rendre le programme exécutable directement après vérification : le TypedTree étant de toute manière de trop haut niveau, l'utiliser pour faire des liens dynamiques serait contreproductif. Il faudrait ainsi le compiler vers une forme compréhensible pour être lié dynamiquement, ce qui aurait un impact certain sur les performances du programme source, tout en l'obligeant à embarquer toute une chaîne de compilation.

3. ∧Cette représentation intermédiaire explicite les fermetures, elle est la suite directe de *Lambda*, que l'on peut voir figure 1. Cette représentation intermédiaire a néanmoins été remplacée par *Flambda* depuis, langage dédié à l'optimisation en plus d'avoir des fermetures explicites.

1.4 Programmes en tant que preuves

L'utilisation de langages à types dépendants rend la frontière entre l'écriture de programmes et leur preuve de correction parfois floue. Ces langages permettent d'exprimer un ensemble de propriétés qui peuvent être vérifiées statiquement, au prix d'une complexité élevée quant à l'écriture de programmes. Néanmoins, si l'utilisation de tels langages pour l'écriture de vérificateurs pourrait apporter un gain de sûreté non négligeable, elle est orthogonale à la vérification de correction du système de types. En effet, on cherche ici à s'abstraire de l'implémentation de l'inférence de types pour écrire un vérificateur indépendant et lire des programmes annotés comme des preuves dont on vérifie la cohérence. Cela étant, écrire un moteur d'inférence de types à l'aide d'un tel langage permettrait de s'abstenir de cette étape de vérification de cohérence, puisqu'il serait possible de prouver que l'inférence respecte la spécification du système de types.

CFML : extraction de formules caractéristiques de programmes ML CFML [11] introduit l'idée de l'extraction de formules caractéristiques depuis un programme ML. Une formule caractéristique est une représentation d'un programme sur la forme d'un triplet de Hoare, encodée dans de la logique d'ordre supérieur. Le principe derrière cet encodage est de pouvoir vérifier un ensemble de propriétés sur une fonction donnée. CFML extrait donc des formules caractéristiques en Coq depuis des programmes OCaml. Par exemple, l'expression suivante :

```
if x = true then 0 else 1
```

sera traduite vers le terme Coq :

```
function P : Prop ⇒ forall x : Bool. (x = true → P 0) ∧ (x = false → P 1)
```

soit une fonction prenant en argument une propriété `P` à satisfaire.

CFML met en place un cadre de travail pour formaliser l'extraction de programmes vers des formules caractéristiques. Le cas de ML vers Coq se trouve être relativement simple, étant donné la proximité entre ML et Gallina, le langage de programmation de Coq. La principale difficulté vient de la représentation des fonctions en dans la logique, étant donné qu'il est possible d'écrire des fonctions qui peuvent diverger en ML. La solution est un prédicat qui joue le rôle de preuve inductive : `AppReturns : Func → A → (B → Prop) → Prop`. Par exemple, `AppReturns f x P` est une preuve que l'application de la fonction `f` sur `x` termine et son résultat permet de prouver la propriété `P`.

CakeML CakeML est un compilateur pour un sous-ensemble de SML [48, 49] écrit et prouvé en HOL [52]. En particulier, les programmes CakeML peuvent être traduits vers des formules caractéristiques [34], sous forme de triplets de Hoare, et ainsi permettre de prouver des propriétés autres que celle de sûreté induite du typage. Ces preuves sont bien entendu à la charge du programmeur, le compilateur générant simplement la proposition à prouver.

CakeML est un cas intéressant de compilateur prouvé de bout en bout puisque le compilateur est écrit dans un langage de preuve, sa sémantique ainsi que la correction de son typeur sont prouvées. Le code est ensuite extrait vers du code SML qui peut-être compilé. Ce nouveau compilateur peut ensuite compiler à nouveau ce code, permettant alors la certification de la chaîne de compilation. L'ajout de l'extraction de formules caractéristiques permet un haut niveau de certification des programmes écrit en CakeML, bien plus que ce que le cadre de travail proposé dans cette thèse.

En revanche, CakeML est un projet basé sur SML, un langage dont le système de types et la sémantique sont standardisés, contrairement à OCaml dont il n'existe qu'une implémentation, qui fait alors office de "standard". De plus, il s'agit d'un travail récent écrit dans le but de pouvoir certifier toute une chaîne de compilation. La vérification d'arbres de syntaxe annotés intervient après un certain temps de développement d'OCaml, et permet notamment de proposer une formalisation de son système de types tout en vérifiant la correction de l'inférence à un moment donné. Il s'agit d'un travail de *reconstruction* du formalisme derrière OCaml et son unique compilateur, plutôt que l'écriture de zéro d'un compilateur certifiant respecter le standard. Cela étant, un tel travail de formalisation pourrait amener dans le futur à écrire de la même manière que CakeML ou Compcert un compilateur certifié pour OCaml.

Il est néanmoins intéressant de noter qu'il existe une formalisation du système de types sous forme déclarative [63] et proche de l'algorithme de vérification. Cette formalisation présente une preuve de sûreté et de correction de l'inférence de types vis-à-vis de cette spécification, sur un sous-ensemble restreint à un ML étendu avec du filtrage de motifs. Contrairement à la formalisation présentée dans ce manuscrit, la vérification n'est pas réalisée explicitement sur un langage dont tous les nœuds sont annotés.

CompCert Il est possible finalement d'écrire un compilateur directement dans un langage de preuve tel que Coq, et d'en prouver directement sa correction. Un exemple récent d'un tel travail est CompCert [7, 40, 8], un compilateur C vers assembleur entièrement écrit en Coq et dont la sémantique et la correction sont prouvées directement dans le langage. Cette approche permet d'ajouter une sûreté certaine dans le processus de compilation. Ce travail de preuve de correction se fait sur l'ensemble de la chaîne et non seulement sur la vérification statique de programmes C. Néanmoins, le système de types de C n'est pas aussi fort que d'OCaml, et la partie la plus importante se situe sur la formalisation et la preuve de correction du générateur de code assembleur selon la sémantique du C, bien plus complexe que celle d'OCaml.

ML Il existe plusieurs formalisations de ML (et Standard ML). On pourrait noter par exemple le travail de Catherine Dubois [25, 24] sur la preuve de l'inférence de type de ML en Coq. De même, il existe un travail de formalisation du Standard ML [39], cette fois en Twelf, sous la forme d'un langage intermédiaire dont l'expressivité est équivalente à Standard ML. Il est également intéressant de noter le travail de vérification d'un compilateur pour langage fonctionnel de Z. Dargaye [21]. Contrairement à la formalisation présentée chapitre 2, ces formalismes ne travaillent pas sur des arbres annotés.

OCaml Finalement, quelques travaux de preuve de sûreté ont été conduits autour du système de types d'OCaml, en particulier sur la preuve de ML avec polymorphisme structurel, soit les objets et variants polymorphes, accompagnés de récursion polymorphe [27]. Ceux-ci visent à formaliser un moteur d'inférence en présence de ces constructions, prouver sa correction et sa sûreté.

Chapitre 2

MiniML

Avant de vérifier le TAST d'OCaml, il peut être intéressant de formaliser et valider le concept sur un langage plus simple et plus restreint en termes de constructions. De plus, l'implémentation d'OCaml s'autorise certaines simplifications et optimisations, rendant le processus de vérification plus complexe. Le but de ce chapitre est donc de formaliser la vérification d'arbres de syntaxes annotés de MiniML, en définissant ceux-ci comme contenant les données idéales.

L'utilisation d'un langage dont le nombre de constructions est restreint permet d'en écrire plus facilement une spécification dans un langage à types dépendants tel que Coq, et de prouver la correction d'un vérificateur vis-à-vis de cette spécification, comme nous le verrons au cours de ce chapitre. Une autre spécificité de la vérification telle qu'elle est présentée ici est d'être proche d'une spécification exécutable, et donc de réduire le degré d'abstraction entre le système formel idéal et théorique et l'implémentation. La spécification Coq présentée montrera effectivement la proximité entre la spécification et l'implémentation du vérificateur associé. On présentera d'ailleurs une implémentation en OCaml d'un vérificateur pour MiniML, en montrant la correspondance directe entre les règles de vérification et le code, la différence étant majoritairement une question du choix de la structure de données utilisée pour représenter le TAST, l'environnement et la substitution.

Ce chapitre se concentre sur MiniML[18], autrement dit un lambda-calcul typé avec un let *polymorphe*. Ce langage a l'avantage d'être simple, mais également de présenter du polymorphisme d'ordre supérieur et implicite, comme celui d'OCaml comme nous le verrons au chapitre 4. L'un des enjeux de la vérification de MiniML est de vérifier que les instanciation de schémas de types sont corrects. Dans un langage avec polymorphisme explicite comme Système F[31], les instanciations sont également explicites : il existe une construction pour définir une abstraction quantifiant une variable polymorphe, et une construction pour appliquer un type à cette abstraction. Dans le cadre de ML, ces quantifications sont inférées à l'aide de la généralisation et les applications sont vérifiées à l'aide de la propriété d'instanciation. Ainsi, ce chapitre introduit la notion d'instanciation et sa vérification. On considèrera un MiniML annoté, qui aura été produit par l'inférence. Ce chapitre se concentrera d'abord sur une brève présentation du langage avec sa syntaxe, accompagné des règles de génération des annotations par l'inférence. Ensuite, les différentes propriétés vérifiées par l'unification seront formalisées sous la forme de différentes opérations décrites de manière inductive. En particulier, on prouvera un certain nombre de pro-

Symboles :

$x, y, z...$	*variables*
$\alpha, \beta, \gamma...$	*variables de types*
`int`, `bool`...	*constructeurs de types*

Syntaxe :

$e ::= \langle\ e' : \tau\ \rangle$	*nœud d'expression annoté*
$e' ::= c$	*constante*
$\mid x$	*variable*
$\mid \lambda \langle\ x : \tau\ \rangle. \mathrm{t}$	*abstraction annotée*
$\mid e_1\ e_2$	*application*
$\mid$ **let** $x : \sigma = e_1$ **in** e_2	*let-binding*

FIGURE 2.1 – Syntaxe de MiniML avec annotations

La syntaxe de MiniML reste une présentation classique d'un lambda calcul polymorphe, à l'exception des nœuds annotés à tous les niveaux. Les métavariables τ et σ sont définies en figure 2.2.

priétés sur le calcul d'instanciation,, brique essentielle de notre système de types. Une fois ces opérations de vérification de l'algèbre de types définies, nous définirons un ensemble de règles de vérification de cohérence du langage lui-même. Finalement une sémantique opérationnelle de ce MiniML annoté est présentée en fin de ce chapitre. En particulier, étant donné que l'arbre est complètement annoté il est nécessaire d'évaluer les types eux-mêmes durant la phase d'instanciation. Pour cela, on définira une notion d'instanciation sémantique, un calcul simplifié de l'instance permettant d'effectuer l'évaluation des types. Tout ce chapitre est accompagné d'une représentation de MiniML à la fois en Coq et en OCaml, permettrant de montrer la proximité entre les règles définies pour être exécutables et l'algorithme dérivé.

TAST : Définition Le TAST, pour *Type Annotated Syntax Tree* (ou *Typed Abstract Syntax Tree*) est un arbre de syntaxe résultant de l'inférence : chacun des nœuds est annoté avec des informations issues de celle-ci. Ces informations contiennent en particulier le type de l'expression annotée, mais également d'autres données comme l'environnement utilisé pour inférer ce type ou des annotations utilisateur ou générées par le compilateur pour diriger des optimisations dans la suite du processus. Le Typedtree d'OCaml est un exemple concret de TAST. Cet arbre étant le résultat du typage, on peut le considérer comme une preuve du typage du programme : chaque nœud est cohérent et annoté comme tel puisque ses sous-nœuds sont eux-mêmes cohérents et annotés d'un manière précise.

$\forall\, \overline{\alpha}.\ \tau$	*schémas de types*
$\tau ::= \alpha$	*variables de type*
$\mid \mathtt{int}$	*constructeurs de type (int)*
$\mid \tau_1 \rightarrow \tau_2$	*types fonctionnel*

FIGURE 2.2 – Algèbre de types de MiniML

La définition du schéma de types reste identique à celle de ML sans références.

2.1 MiniML : définition

2.1.1 Expressions et algèbre de types

La syntaxe d'un TAST pour MiniML est décrite dans la figure 2.1. Les constructions sont classiques, mais intègrent quelques spécificitées dues à l'annotation de chaque nœud. En premier lieu, un nœud est lui-même une expression à laquelle est attaché un type. Ces expressions peuvent alors être :

- Une variable ;
- Une constante entière ;
- Une abstraction, dont on remarquera que l'argument est lui aussi annoté. En effet, puisque l'on se place dans un contexte où le programme a déjà été inféré, le type de l'argument est explicite.
- Une application ;
- Un *let-binding*, autrement dit l'association d'une expression à une variable locale, dont le schéma de type est annoté. Tout comme pour l'abstraction, le type ainsi que la quantification sont connus après l'inférence.

L'algèbre de types présentée dans le figure 2.2 est assez classique pour ML. Elle se décompose en types et schémas de types. Les types peuvent prendre 3 formes :

- soit une variable de type ;
- soit un constructeur de type de base, qu'on restreindra à `int` pour représenter les entiers ;
- soit un type fonctionnel, pour représenter les abstractions.

Les schémas de types ajoutent la quantification nécessaire au polymorphisme paramétrique. Ceux-ci n'apparaissent que sur le type annoté aux **`let`** : en effet, la particularité de ML est la généralisation du **`let`**, autrement dit seule cette construction introduit du polymorphisme, contrairement à un système de type comme Système F, où le polymorphisme est explicite et géré par deux constructions dans le langage : une pour introduire une variable polymorphe et une pour instancier un type polymorphe.

2.2 Inférence : génération du TAST

Les règles d'inférence présentées en figure 2.3 sont les règles classiques de Hindley-Milner [20, 58], auxquelles a été rajoutée la génération de nœuds annotés. Cette présentation des règles est classique, et on verra lors de l'analyse des règles de vérification qu'elles ne diffèrent que

$$\text{DM-CONST}\ \Gamma \vdash c : \texttt{int} \rightsquigarrow \langle c : \texttt{int}\rangle \qquad \text{DM-VAR}\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau \rightsquigarrow \langle x : \tau\rangle}$$

$$\text{DM-ABS}\ \frac{\Gamma, x : \tau \vdash e : \tau' \rightsquigarrow e'}{\Gamma \vdash \lambda x.e \ : \tau \to \tau' \rightsquigarrow \langle \lambda\langle x : \tau\rangle.e' \ : \tau \to \tau'\rangle}$$

$$\text{DM-APP}\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : \tau' \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : \tau \rightsquigarrow \langle e_1'\ e_2' : \tau\rangle}$$

$$\text{DM-LET}\ \frac{\Gamma \vdash e_x : \sigma \rightsquigarrow e_x' \qquad \Gamma, x : \sigma \vdash e : \tau \rightsquigarrow e'}{\Gamma \vdash \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau \rightsquigarrow \langle \texttt{let}\ \langle x : \sigma\rangle = e_x'\ \texttt{in}\ e' : \tau\rangle}$$

$$\text{DM-GEN}\ \frac{\Gamma \vdash e : \tau \rightsquigarrow e' \qquad \overline{\alpha} \notin ftv(\Gamma)}{\Gamma \vdash e : \forall\overline{\alpha}.\tau \rightsquigarrow e'} \qquad \text{DM-INST}\ \frac{\Gamma \vdash e : \forall\overline{\alpha}.\tau \rightsquigarrow e'}{\Gamma \vdash e : [\overline{\alpha \mapsto \tau'}]\tau \rightsquigarrow e'}$$

FIGURE 2.3 – Règles d'inférence de MiniML

très peu de celles-ci, exceptées les règles DM-INST et DM-GEN. Pour la première, l'opération d'instanciation est une règle à part entière là où la vérification la considérera comme une opération du calcul de cohérence : les types étant déjà déterminés il n'y a pas lieu de reconstruire son type de cette manière. Pour la seconde, cette vérification est directement introduite dans la règles de vérification du `let` et des variables. Une spécificité des règles de vérification présentées plus loin est qu'elles associent une règle à une construction : les deux règles précédentes sont valables pour toutes les expressions et entraînent un certain degré de complexité supplémentaire quand à leur compréhension et au choix de leur utilisation pendant l'inférence.

2.3 Représentation Coq

Il est possible de représenter cette version de MiniML totalement annotée en Coq. Cette représentation s'inspire fortement et réutilise les travaux de Bryan Aydemir et Arthur Charguéraud [4, 12] sur la représentation des variables quantifiées et les lieurs dans le cas de la preuve de programmes, représentation dite *localement sans nom*. Cette représentation combine l'utilisation d'indices de De Bruijn[9] pour représenter les variables liées d'un programme, avec la quantification dite cofinie pour représenter les variables localement libres durant la traversée d'une abstraction. L'implémentation Coq utilise donc la bibliothèque *LibLN*[13] permettant la manipulation de telles variables, en plus de proposer un ensemble de structures de données utiles à la formalisation de langages.

La figure 2.4 donne la représentation de MiniML annoté présentée précédemment. Le type

```
(** Syntax of types *)

Inductive ty :=
| Int : ty
| Var : nat → ty
| Fvar : var → ty
| Arrow : ty → ty → ty.

Inductive sch :=
| Forall : nat → ty → sch.

Definition sch_arity s :=
  match s with
  | Forall ar _ ⇒ ar
  end.

Definition sch_body s :=
  match s with
  | Forall _ t ⇒ t
  end.

(** Syntax of terms *)

Definition arg := ty.

Inductive annot_term : Set :=
| term_var: nat → ty → annot_term
| term_fvar: var → ty → annot_term
| term_function: arg → annot_term → ty → annot_term
| term_app: annot_term → annot_term → ty → annot_term
| term_let: sch → annot_term → annot_term → ty → annot_term.

Definition typed_term : Set := annot_term.
```

FIGURE 2.4 – Représentation Coq de MiniML annoté

`ty` représente l'algèbre de types, accompagné de `sch` permettant la représentation des schémas de types. Contrairement à l'algèbre donné en 2.2, les variables se distinguent entre `Var` pour les variables liées, et `Fvar` pour les variables libres. Les variables sont représentées par un indice de De Bruijn (un entier). Les variables libres sont représentées par une valeur de type `var`, répondant aux critères énoncés par la représentation *localement sans nom*. Un schéma de type possède une arité et non un ensemble des variables : il s'agit du nombre de variables quantifiées par ce schéma. En effet, la représentation utilisant des indices de De Bruijn, les variables sont indicées par des entiers. Ainsi, toute variable dont l'indice est inférieur à cette arité correspond à une variable quantifiée par ce schéma.

La syntaxe des expressions annotées est classique et suit le schéma précédemment énoncé pour les types : les variables se découpent en variables liées représentées par un indice de De Bruijn et en variables libres représentées par une variable fraîche. Elles sont toutes deux annotées par un argument `ty`. Utilisant la représentation des lieurs de De Bruijn, l'abstraction n'indique pas de nom pour son argument. Néanmoins, l'annotation explicite oblige l'abstraction à déclarer le type de son argument : il s'agit de l'argument `arg`. Son corps est lui-même une expression annotée. L'application est classique, et prend deux expressions annotées. Enfin, la variable locale est similaire à l'abstraction : la variable n'est pas explicitement nommée, mais son schéma de types est annoté.

La représentation *localement sans nom* introduit également la notion d'ouverture des quantifications : durant la "traversée" d'une quantification, la variable quantifiée (liée) est remplacée par une variable dite fraîche (par rapport aux variables libres apparaissant dans le corps du terme quantifié). Une propriété intéressante est de pouvoir automatiquement distinguer les variables indiquées comme liées mais qui ne correspondent à aucune quantification : il s'agit alors d'un type ou d'une expression mal formée, et par conséquent d'une erreur de typage. On définit donc l'ouverture d'une quantification comme la substitution d'une variable libre à sa variable associée. Dans le cas d'un schéma, qui donc peut quantifier plusieurs variables, l'ouverture doit donc générer plusieurs variables fraîches pour se substituer aux variables indicées entre 0 et son arité -1. L'ouverture est donnée en figures 2.5 et 2.6.

L'ouverture est un algorithme de substitution : la fonction prend en argument un type (ou une expression) auquel (à laquelle) appliquer la substitution, un entier qui est l'indice correspondant à la variable à substituer, et un type qui est la variable libre qui doit la remplacer. Dans le cas de `open_type`, puisque l'ouverture d'un schéma se fait sur plusieurs variables, l'argument est par conséquent une liste de variables liées. L'indice de la variable à substituer est la position dans cette liste. La liste de variables utilisées lors de l'ouverture d'un schéma ne doivent pas se substituer aux variables quantifiées par le schéma courant : il faut donc ajouter à la liste les variables du schéma, à la même position que leur indice. L'ajout est relativement simple, puisqu'il suffit de les ajouter en début de liste, opération triviale étant donné la structure de données.

Les expressions étant totalement annotées, il peut arriver que l'ouverture d'un quantificateur de schéma de types doive se propager sur l'ensemble des types annotés dans une expression quantifiée. C'est le rôle de la fonction `open_types_term`. La fonction `open_term` gère le cas de la traversée d'une abstraction ou d'une définition locale : une nouvelle variable libre doit être substituée à la variable liée.

Finalement, on définit des opérateurs pour représenter ces ouvertures, de la même manière

```
(** Types opening *)

Fixpoint open_type (t : ty) (ts : list ty) :=
  match t with
  | Int ⇒ Int
  | Fvar v ⇒ Fvar v
  | Var n ⇒ List.nth n ts (Var n)
  | Arrow t1 t2 ⇒
    Arrow (open_type t1 ts) (open_type t2 ts)
  end.

Definition open_sch_aux (t : ty) (ts : list var) :=
  open_type t (List.map Fvar ts).

Fixpoint make_vars_aux n (acc : list ty) :=
  match n with
  | O ⇒ acc
  | S n' ⇒ make_vars_aux n' (cons (Var n') acc)
  end.

Definition open_sch s ts :=
  let ts' := make_vars_aux (sch_arity s) ts in
  Forall (sch_arity s) (open_type (sch_body s) ts).

Notation "M ^^ Vs" := (open_sch M Vs)
  (at level 67, only parsing): ty_scope.
Notation "M ^ Xs" :=
  (open_sch_aux M Xs) (only parsing): ty_scope.

Fixpoint open_types_term (e : typed_term) (ts : list ty) :=
  match e with
  | term_var n ty ⇒ term_var n (open_type ty ts)
  | term_fvar v ty ⇒ term_fvar v (open_type ty ts)
  | term_function t b ty ⇒
    term_function (open_type t ts) (open_types_term b ts) (open_type ty ts)
  | term_app e1 e2 ty ⇒
    term_app (open_types_term e2 ts) (open_types_term e2 ts) (open_type ty ts)
  | term_let s e1 e2 ty ⇒
    let ts' := make_vars_aux (sch_arity s) ts in
    term_let
      (open_sch s ts)
      (open_types_term e1 ts')
      (open_types_term e2 ts)
      (open_type ty ts)
  end.

Definition open_types_aterm e ts :=
  open_types_term e ts.

Notation "M ^^\t Vs" := (open_types_term M Vs)
  (at level 67, only parsing): ty_scope.

Notation "M ^\t Xs" :=
  (open_types_aterm M Xs) (at level 67, only parsing): ty_scope.
```

FIGURE 2.5 – Ouverture des quantificateurs de types

```
(** Types opening *)

Fixpoint open_term (e : typed_term) (n : nat) (e' : untyped_term) :=
  match e with
  | term_var n' ty ⇒ If n = n' then e' ty else term_var n' ty
  | term_fvar v ty ⇒ term_fvar v ty
  | term_function x b ty ⇒
    term_function x (open_term b (S n) e') ty
  | term_app e1 e2 ty ⇒
    term_app (open_term e1 n e') (open_term e2 n e') ty
  | term_let x e1 e2 ty ⇒
    term_let x (open_term e1 n e') (open_term e2 (S n) e') ty
end.

Definition open_aterm (e : typed_term) n e' :=
  open_term e n e'.

Notation "{ k ~> u } t" := (open_term t k u) (at level 67).
Notation "{ k ~~> u } e" := (open_aterm e k u) (at level 67).
Notation "t ^^ u" := (open_aterm t 0 u) (at level 67).
Notation "t ^ x" := (open_aterm t 0 (term_fvar x)).
```

FIGURE 2.6 – Ouverture des quantificateurs d'expression

que sont représentées les opérations de substitution, et pour aider à la lecture des programmes.

2.4 Règles de vérification de types

Puisque tout est annoté, il ne s'agit pas de faire de l'inférence. Au contraire, on cherchera à vérifier qu'une expression est cohérente vis-à-vis du type auquel elle est associée. Une différence fondamentale avec une présentation classique d'un système de types est qu'ici, toutes les opérations sont aussi explicites que possible. Ainsi, on doit d'abord définir un ensemble de vérifications sur les types eux-mêmes avant de spécifier la cohérence des expressions.

2.4.1 Manipulation et vérification de types

La présentation du système de types de ML introduit classiquement les égalités et instances de manière totalement implicite, via le partage de *métavariables*. Dans notre présentation de ce système de types ces opérations sont explicites. Cela permet alors d'identifier clairement les étapes de la vérification mais également de simplifier la traduction d'une présentation formelle vers une implémentation. Le but est alors d'être capable d'écrire le code d'une règle de typage comme sa transcription littérale. Ces vérifications se décomposent en trois parties que nous décrirons par la suite : l'équivalence de types, l'instanciation de schémas de types et le filtrage de types.

$$\textsc{Equiv-Var}\ \alpha \equiv \alpha \qquad \textsc{Equiv-Arrow}\ \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \textsc{Equiv-Construct}\ \texttt{int} \equiv \texttt{int}$$

$$\textsc{Equiv-Forall}\ \frac{\#(\overline{\alpha}) = \#(\overline{\beta}) \qquad let\ \overline{\gamma} \notin fv(\tau_1) \cup fv(\tau_2) \qquad \tau_1[\overline{\alpha \mapsto \gamma}] \equiv \tau_2[\overline{\beta \mapsto \gamma}]}{\forall\overline{\alpha}.\ \tau_1 \equiv \forall\overline{\beta}.\ \tau_2}$$

Figure 2.7 – Vérification d'équivalence de types de MiniML

Equivalence

L'équivalence de types compare deux types de manière syntaxique et teste leur égalité. Les règles de vérification de l'équivalence sont données figure 2.7.

L'équivalence est purement syntaxique, il s'agit de comparer deux à deux les types et s'assurer de leur égalité. Ainsi, deux variables de types (Equiv-Var) sont équivalentes si et seulement si il s'agit de la même variable. Deux types de fonction (Equiv-Arrow) sont équivalents si leur domaine et leur codomaine sont eux-même équivalents. Deux occurrences du constructeur de types `int` sont toujours équivalentes (Equiv-Construct). Finalement deux schémas de types $\forall\overline{\alpha}.\ \tau_1$ et $\forall\overline{\beta}.\ \tau_2$ sont équivalents si et seulement si leur arité est la même et si τ_1 est équivalent à τ_2 dans lesquels des variables fraîches $\overline{\gamma}$ ont été substituées à toutes les occurrences de $\overline{\alpha}$ dans τ_1 et à $\overline{\beta}$ dans τ_2. L'algorithme de substitution de variables de types est donné dans la figure 2.24.

L'équivalence vérifie plusieurs propriétés, telle que la réflexivité, la transitivé et la symétrie.

Lemme 1 (Réflexivité de l'équivalence)**.** *Pour tout τ, $\tau \equiv \tau$.*

Démonstration. Par induction sur τ. □

Avant de montrer la réflexivité des schémas, on veut d'abord montrer un propriété simple sur la substitution

Lemme 2 (Conservation de la réflexivité après substitution)**.** *Pour tous α, β et τ, $\tau[\alpha \mapsto \beta] \equiv \tau[\alpha \mapsto \beta]$.*

Démonstration. Par induction sur τ. □

Lemme 3 (Réflexivité de l'équivalence des schémas)**.** *Pour tous α et τ, $\forall\overline{\alpha}.\ \tau \equiv \forall\overline{\alpha}.\ \tau$.*

Démonstration. Pour montrer $\forall\overline{\alpha}.\tau \equiv \forall\overline{\alpha}.\tau$, on peut facilement montrer qu'il est possible de substituer n'importe quelles variables fraîches $\overline{\beta}$ à $\overline{\alpha}$, résultat que l'on nommera τ'. Le cas où l'arité est de 0 revient à un type monomorphe, puisqu'il n'y aucune substitution à effectuer. On montre la propriété par induction sur $\overline{\alpha}$.

— Si α n'est pas liée dans τ, alors $\tau' = \tau$. On sait par induction que $\tau \equiv \tau$, et donc par définition de Equiv-Forall et de l'hypothèse d'induction sur le reste des variables que la propriété est vraie.

— Si α est liée, on doit montrer $\tau[\alpha \mapsto \beta] \equiv \tau[\alpha \mapsto \beta]$. Sachant que la substitution conserve la réflexivité, que $\tau \equiv \tau$ par induction et l'hypothèse d'induction sur le reste des variables, la propriété est vraie.

□

Une autre propriété importante pour la suite des preuves est celle d'inversion :

Lemme 4 (Equivalence : inversion)**.** *Selon la forme du type :*
— *pour tous α, τ, si $\alpha \equiv \tau$ alors τ est une variable de type α*
— *si $\tau_1 \to \tau_2 \equiv \tau'$, alors τ' est un type de fonction $\tau'_1 \to \tau'_2$ et $\tau_1 \equiv \tau'_1$ et $\tau_2 \equiv \tau'_2$*
— *si $\mathtt{int} \equiv \tau$, alors τ est un constructeur de type $\mathtt{int}$*
— *si $\forall\overline{\alpha}.\ \tau \equiv \sigma'$, alors σ' est un schéma de type de la forme $\forall\overline{\beta}.\ \tau'$ et il existe $\overline{\gamma}$ tel que $\overline{\gamma} \notin fv(\sigma) \cup fv(\sigma')$ et $\tau[\overline{\alpha \mapsto \gamma}] \equiv \tau'[\overline{\beta \mapsto \gamma}]$.*

Démonstration. Par définition de l'équivalence. □

Il est alors possible de prouver la transitivité de l'équivalence :

Lemme 5 (Equivalence : transitivité)**.** *Pour tous τ_1, τ_2, τ_3, si $\tau_1 \equiv \tau_2$ et $\tau_2 \equiv \tau_3$ alors $\tau_1 \equiv \tau_3$.*

Démonstration. Par induction sur τ_1.

□

Lemme 6 (Equivalence : transitivité des schémas de types)**.** *Pour tout σ_1, σ_2, σ_3, si $\sigma_1 \equiv \sigma_2$ et $\sigma_2 \equiv \sigma_3$ alors $\sigma_1 \equiv \sigma_3$.*

Démonstration. Par inversion, σ_1 est de la forme $\forall\overline{\alpha}.\ \tau_1$. Par inversion sur $\forall\overline{\alpha}.\ \tau_1 \equiv \sigma_2$, alors σ_2 est de la forme $\forall\overline{\beta}.\ \tau_2$ et par inversion toujours σ_3 de la forme $\forall\overline{\gamma}.\ \tau_3$.

On sait qu'il existe $\overline{\delta}$ tels que $\overline{\delta} \notin fv(\tau_1) \cup fv(\tau_2)$ et $\tau_1[\overline{\alpha \mapsto \delta}] \equiv \tau_2[\overline{\beta \mapsto \delta}]$, et $\overline{\epsilon}$ tels que $\overline{\epsilon} \notin fv(\tau_2) \cup fv(\tau_3)$ et $\tau_2[\overline{\beta \mapsto \epsilon}] \equiv \tau_3[\overline{\gamma \mapsto \epsilon}]$.

Sachant un ensemble de variables fraiches $\overline{\zeta} \notin fv(\tau_1) \cup fv(\tau_2) \cup fv(\tau_3)$, on peut donc montrer que $\tau_1[\overline{\alpha \mapsto \zeta}] \equiv \tau_2[\overline{\beta \mapsto \zeta}]$ et $\tau_2[\overline{\beta \mapsto \zeta}] \equiv \tau_3[\overline{\gamma \mapsto \zeta}]$. Et donc, la propriété est vraie sachant la transitivité de l'équivalence sur les types non quantifiés. □

L'équivalence, au final, est une égalité syntaxique modulo alpha-conversion.

Equivalence : représentation Coq La propriété d'équivalence est représentée par une propriété inductive (figure 2.8). L'implémentation est équivalente aux règles formelles énoncées précédemment, à la différence qu'il n'existe pas d'équivalence entre variables liées mais seulement entre variables libres. Étant donné la représentation localement sans nom, dès lors qu'un quantificateur est traversé, toutes les variables qu'il quantifiait sont remplacées par des variables libres fraîches : localement, les variables ne sont plus liées. Ainsi, tout type correctement formé ne doit pas contenir de variable liée après ouverture. L'équivalence des schémas de types introduit donc une ouverture de chacun des schémas : sachant un ensemble de variables fraîches (dont le nombre est l'arité des deux schémas, qui doit évidemment être la même), on vérifie l'équivalence de leur corps dont ces variables libres ont été substituées à leurs variables liées.

```
(** Type equivalence *)

Inductive equiv : ty → ty → Prop :=
| eq_int : equiv Int Int
| eq_fvar : forall v, equiv (Fvar v) (Fvar v)
| eq_arrow : forall t11 t12 t21 t22,
   equiv t11 t21 → equiv t12 t22 →
   equiv (Arrow t11 t12) (Arrow t21 t22).

Inductive equiv_sch : sch → sch → Prop :=
| eq_forall : forall n t1 t2 fvs,
   (forall vs,
      fresh fvs n vs →
      equiv (t1 ^ vs)
            (t2 ^ vs)) →
   equiv_sch (Forall n t1) (Forall n t2).
```

Figure 2.8 – Implémentation de l'équivalence en Coq

$$\textsc{Inst-Var-Unbound}\ \frac{\alpha \notin dom(\theta)}{\theta \vdash_{\tau} \tau \leq\ \alpha \Rightarrow \theta \oplus [\alpha \mapsto \tau]} \qquad \textsc{Inst-Var-Bound}\ \frac{\alpha \in dom(\theta) \qquad th\theta(\alpha) \equiv \tau}{\theta \vdash_{\tau}\ \tau \leq\ \alpha \Rightarrow \theta}$$

$$\textsc{Inst-Fun}\ \frac{\theta \vdash_{\tau} \tau_1 \leq\ \tau_1' \Rightarrow \theta_1 \qquad \theta_1 \vdash_{\tau} \tau_2 \leq\ \tau_2' \Rightarrow \theta_2}{\theta \vdash_{\tau} \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \Rightarrow \theta_2} \qquad \textsc{Inst-Construct}\ \theta \vdash_{\tau} \mathtt{int} \leq \mathtt{int} \Rightarrow \theta$$

Figure 2.9 – Instanciation de schémas de MiniML

Instanciation

Le mécanisme d'instanciation permet de vérifier qu'un type τ est une instance correcte d'un schéma de type $\forall\overline{\alpha}.\tau'$. En d'autres termes, cette opération s'assure qu'il existe une substitution à toute variable quantifiée par ce schéma vers un type donné, et l'application d'une telle substitution sur τ' retourne un type équivalent à τ. Les règles de vérification d'instanciation sont données en figure 2.9.

Le calcul d'instance se fait en comparant structurellement les deux types, retournant alors une nouvelle substitution θ. Celle-ci est une liste associant des types à des variables de types, dont la définition est donnée en figure 2.10. Une substitution est alors soit vide, soit l'extension d'une substitution existante associant une variable de type à un type. Les différentes opérations de manipulations de celles-ci sont :

— le calcul du domaine ($dom(\theta)$) de la substitution, autrement dit l'ensemble des variables de types apparaissant comme clés dans celle-ci ;

— le prédicat de bonne formation ($wf(\theta)$), prouvant que pour toute variable de type, celle-ci n'apparaît jamais plus d'une fois en tant que clé ;

— le prédicat de sous-ensemble $\theta \subseteq \theta'$, autrement dit : $\forall\alpha.\alpha \in dom(\theta) \to \alpha \in dom(\theta') \wedge$

Définitions et propriétés :
— $\theta ::= \emptyset \mid \theta \oplus [\alpha \mapsto \tau]$
— $dom(\theta)$: ensemble des clés (variables) de la substitution.
— $wf(\theta)$: prédicat indiquant l'unicité des clés dans la substitution.
— $\theta \subseteq \theta'$: prédicat de sous-ensemble.
— $\theta_{init}(\forall\overline{\alpha}.\tau) = [\beta \mapsto \beta \mid \beta \in (fv(\tau) - \overline{\alpha})]$
— $apply(\theta, \tau)$: substitution des instances dans θ à leur variable libre de τ.

Domaine :

$$\begin{array}{lcl} dom(\emptyset) & = & \emptyset \\ dom(\theta \oplus [\alpha \mapsto \tau]) & = & dom(\theta) \cup \{\ \alpha\} \end{array}$$

Bonne formation :

$$\textsc{Wf-Subst-Empty}\ wf(\emptyset) \qquad\qquad \textsc{Wf-Subst-Bind}\ \frac{\alpha \notin dom(\theta) \qquad wf(\theta)}{wf(\theta \oplus [\alpha \mapsto \tau])}$$

Substitution de variable :

$$\begin{array}{lcll} apply(\theta, \alpha) & = & \theta(\alpha) & \text{if } \alpha \in dom(\theta) \\ apply(\theta, \alpha) & = & \alpha & \text{if } \alpha \notin dom(\theta) \\ apply(\theta, \texttt{int}) & = & \texttt{int} & \\ apply(\theta, \tau_1 \to \tau_2) & = & apply(\theta, \tau_1) \to apply(\theta, \tau_2) & \end{array}$$

FIGURE 2.10 – Substitutions

$\theta(\alpha) = \theta'(\alpha)$;
— l'application d'une substitution θ à un type τ ($apply(\theta, \tau)$), qui retourne alors un nouveau type dont les variables libres ont été remplacées par le type qui leur était associé dans θ.

Ces opérations sont décrites en figure 2.10.

Il est nécessaire de définir une substitution initiale au moment du calcul de l'instance. En effet, il faut s'assurer que seules les variables quantifiées par le schéma de type sont effectivement instanciées. La substitution initiale θ_{init} d'un schéma $\forall\overline{\alpha}.\tau$ est spécifiée par :

$$[\beta \mapsto \beta \mid \beta \in (fv(\tau) - \overline{\alpha})]$$

En d'autres termes, la substitution initiale associe toute variable de type libre (*i.e.* non quantifiée par le schéma) à elle-même. Ainsi, il devient impossible de lui trouver une instance autre qu'elle-même, si on se conforme à la règle INST-VAR-BOUND donné dans la figure 2.9.

Lemme 7 (Bonne formation de la substition initiale)**.** *Pour tout schéma de type $\forall\overline{\alpha}.\tau$, $wf(\theta_{init}(\forall\overline{\alpha}.\tau))$.*

Démonstration. Selon la définition de $\theta_{init}(\forall\overline{\alpha}.\tau)$, la substitution est construite *au plus* à partir de l'ensemble de variables libres de τ. Sachant qu'un ensemble par définition ne contient pas de doublon, et que la substitution est construite en associant les variables une à une durant la traversée du-dit ensemble, alors la substitution initiale est bien formée. □

Les règles de vérification d'instance sont simples. Les cas importants correspondent à ceux des variables :

— Si on cherche à instancier une variable qui n'est pas déjà liée dans la substitution courante, alors on retourne une nouvelle substitution associant cette variable de type au type qui l'instancie.

— En revanche, si on instancie une variable qui apparaît dans la substitution courante, il faut alors s'assurer que le type qui lui était associé est bien équivalent au type courant.

Lemme 8 (Instanciation retourne une substitution bien formée)**.** *Pour toutes substitutions θ, θ', pour tous types τ, τ', si $wf(\theta)$ et que $\theta \models_{\tau} \tau \leq \tau' \Rightarrow \theta'$, alors $wf(\theta')$.*

Démonstration. Par induction sur τ'.

— si $\tau' = \alpha$, alors par inversion de la propriété d'instanciation, il y a deux cas possibles :
  — INST-VAR-UNBOUND : α n'est pas liée dans la substitution θ, $\theta' = \theta \oplus [\alpha \mapsto \tau]$. Et donc, sachant $wf(\theta)$, on prouve $wf(\theta \oplus [\alpha \mapsto \tau])$.
  — INST-VAR-BOUND : α est liée dans la substitution θ, et $\theta' = \theta$ d'après la règle d'instanciation. Donc, sachant $wf(\theta)$, la preuve est évidente.

— si $\tau' = \tau'_1 \rightarrow \tau'_2$, alors par inversion de la règle d'instanciation $\tau = \tau_1 \rightarrow \tau_2$, $\theta \models_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta''$ (1) et $\theta'' \models_{\tau} \tau_2 \leq \tau'_2 \Rightarrow \theta'$ (2). Par induction, d'après (1) on suppose que θ'' est bien formé. Et donc, sachant que $wf(\theta'')$, on peut prouver par induction que $wf(\theta')$.

— si $\tau' = \texttt{int}$, alors par inversion sur la propriété d'instanciation $\tau = \texttt{int}$ et $\theta = \theta'$. Donc, sachant $wf(\theta)$, on montre $wf(\theta')$.

□

La propriété de bonne formation de la substitution est importante. Si ce n'était pas le cas, alors le calcul d'instanciation serait faux. Si la substitution associait deux fois la même variable avec des types différents, les règles seraient forcément incorrectes. Par exemple, `int` $\rightarrow$ `bool` serait une instance correcte de $\forall \alpha . \alpha \rightarrow \alpha$, si le calcul retournait une substitution de la forme $[\alpha \rightarrow \texttt{int}; \alpha \rightarrow \texttt{bool}]$.

Lemme 9 (Instanciation retourne un sur-ensemble de sa substitution initiale)**.** *Pour toute substitution θ, θ', pour tous types τ, τ', si $\theta \models_{\tau} \tau \leq \tau' \Rightarrow \theta'$ et $wf(\theta)$, alors $\theta \subseteq \theta'$.*

Démonstration. Par induction sur les règles d'instanciation. Seul le cas INST-VAR-UNBOUND est significatif, puisqu'il s'agit de la seule règle retournant explicitement une plus grande substitution, sans changer les associations de sa substitution d'entrée. Celle-ci est donc alors bien un sous-ensemble de la substitution finale. Le cas de l'instanciation entre deux types fonctionnels dérive de la propriété d'induction. □

Cette propriété permet donc de prouver que le calcul de substitution ne risque pas d'oublier des variables durant l'instantiation ni de changer une instance par une autre durant le calcul.

Lemme 10. *Pour tout schéma de type $\forall \overline{\alpha} . \tau'$, pour tout type τ et toute substitution θ, si $\theta_{init}(\forall \overline{\alpha} . \tau) \models_{\tau} \tau \leq \tau' \Rightarrow \theta$, alors $fv(\tau') = dom(\theta)$.*

Démonstration. Par définition de θ_{init}, son domaine ne contient que les variables non quantifiées par le schéma. Par définition de l'instanciation, seules deux règles travaillent sur l'instanciation elle-même :

— INST-VAR-UNBOUND : si une variable n'apparaît déjà pas dans la substitution courante, elle sera rajoutée. Il s'agit donc d'une variable qui n'est pas dans la substitution initiale ni déjà explorée durant l'instanciation.

— INST-VAR-BOUND : si la variable apparaît dans la substitution courante, on ne la rajoute pas une nouvelle fois. Cela signifie que la variable soit n'est pas quantifiée, soit a déjà été traversée.

L'instanciation parcourant tout τ', on sait alors que toutes les variables apparaissant dans celui-ci seront traversées au moins une fois et donc ajoutées à la substitution, et il est impossible que d'autres variables n'appartenant pas au type soient ajoutées. $dom(\theta_{init})$ ne contenant que des variables de τ', il est alors évident que toutes et seulement toutes les variables de τ' constituent $dom(\theta)$. Donc $fv(\tau') = dom(\theta)$. □

Cette propriété est également nécessaire pour s'assurer qu'il n'existe aucune variable qui ne serait pas instanciée durant le calcul, et donc que celui-ci couvre bien tous les cas possibles.

Lemme 11 (Application d'un sur-ensemble bien formé)**.** *Pour toutes substitutions θ, θ', pour tout type τ, si $fv(\tau) \subseteq dom(\theta)$, si $wf(\theta')$ et $\theta \subseteq \theta'$, alors $apply(\theta, \tau) = apply(\theta', \tau)$.*

Démonstration. Par induction sur τ.

— $\tau = \alpha$. Sachant que $fv(\alpha) \subseteq dom(\theta)$, il est évident que $\alpha \in dom(\theta)$. Donc, que $apply(\theta, \alpha) = \theta(\alpha)$.

De plus, sachant que $\theta \subseteq \theta'$, et θ' étant bien formé, on sait également que θ est bien formé par définition. Donc, que si $\alpha \in \theta$, alors $\alpha \in \theta'$ et $\theta'(\alpha) = \theta(\alpha)$. Donc par définition : $apply(\theta', \alpha) = \theta(\alpha)$.

— $\tau =$ `int`. L'application d'une substitution à `int` étant `int`, la propriété est vraie.

— $\tau = \tau_1 \to \tau_2$. Par induction, on a :

1. $fv(\tau_1) \subseteq dom(\theta) \to apply(\theta, \tau_1) = apply(\theta', \tau_1)$
2. $fv(\tau_2) \subseteq dom(\theta) \to apply(\theta, \tau_2) = apply(\theta', \tau_2)$

On a $fv(\tau_1) \cup fv(\tau_2) \subseteq dom(\theta)$, donc que $fv(\tau_1) \subseteq dom(\theta)$ et $fv(\tau_2) \subseteq dom(\theta)$. On veut montrer que $apply(\theta, \tau_1) \to apply(\theta, \tau_2) = apply(\theta', \tau_1) \to apply(\theta', \tau_2)$, ce qui est donc vrai par hypothèses d'induction.

□

La propriété importante est alors la suivante :

Lemme 12 (Equivalence après application de l'instance)**.** *Pour tous types τ, τ', pour un ensemble de variables de types $\overline{\alpha}$ et pour toute substitution θ, alors*

$$\theta_{init}(\forall\overline{\alpha}.\tau') \models_{\tau} \tau \leq \tau' \Rightarrow \theta \to apply(\theta, \tau') \equiv \tau$$

Démonstration. Par induction sur τ'.

— $\tau' = \alpha$. Supposons $\alpha \in \theta$, il y a deux cas possibles :

- α n'est pas quantifiée par $\forall\overline{\alpha}.\tau'$, et donc sa substitution est α d'après θ_{init}. Et donc, par inversion de l'instanciation, $\tau = \alpha$. L'équivalence étant réflexive, on a bien $\alpha \equiv \alpha$.
- α est quantifiée par le schéma de types, donc par inversion de l'instanciation $\theta(\alpha) = \tau$. Et donc, par réflexivité de l'équivalence on a bien $\tau \equiv \tau$.

Le cas $\alpha \notin \theta$ est impossible, puisque toute variable de type apparaissant dans τ', celle-ci est soit dans la substitution initiale θ_{init}, soit elle possède une instance d'après l'hypothèse initiale.

- $\tau' = \tau'_1 \to \tau'_2$. D'après la propriété d'instanciation, on peut supposer que
  1. $\tau = \tau_1 \to \tau_2$
  2. $\theta_{init}(\forall\overline{\alpha}.\tau') \models_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta'$
  3. $\theta' \models_{\tau} \tau_2 \leq \tau'_2 \Rightarrow \theta$

  On veut donc montrer que $apply(\theta, \tau'_1 \to \tau'_2) \equiv \tau_1 \to \tau_2$, soit par définition de l'application de la substition $apply(\theta, \tau'_1) \to apply(\theta, \tau'_2) \equiv \tau_1 \to \tau_2$.
  - On veut montrer que $apply(\theta, \tau'_1) \equiv \tau_1$. Par induction, on suppose que $apply(\theta', \tau'_1) \equiv \tau_1$. Sachant que la substitution initiale est bien formée d'après le lemme 7, alors θ' est bien formée d'après l'hypothèse (2) et le lemme 8. Alors, θ est bien formée d'après l'hypothèse (3). D'après le lemme 9, $\theta' \subseteq \theta$. D'après le lemme 10 et l'hypothèse (2), on sait que seules les variables de τ_1 apparaissent dans le domaine de θ'. Et donc, sachant que θ est bien formée et d'après le lemme 11, on peut montrer que $apply(\theta', \tau'_1) = apply(\theta, \tau_1)$
  - On veut montrer que $apply(\theta, \tau'_2) \equiv \tau_2$. Par induction, on suppose que $apply(\theta, \tau'_2) \equiv \tau_2$, ce qui est donc vrai.
- $\tau' = \texttt{int}$. D'après la propriété d'instanciation, $\tau = \texttt{int}$. Par conséquent $apply(\theta, \texttt{int}) = \texttt{int}$, et donc $\texttt{int} \equiv \texttt{int}$.

$\square$

Cette propriété est essentielle, puisqu'elle montre que l'algorithme du calcul d'instance est correct vis-à-vis de sa spécification.

Instanciation et substitution : représentation Coq L'instantiation est définie en tant que propriété inductive (figure 2.11). Chacune des branches implémente directement les règles présentées précédemment. Avant de pouvoir les détailler, il est important de définir la forme des substitutions et des ensembles, ainsi que les opérations sur ceux-ci. Les substitutions utilisent la structure de données des environnements fournie dans la bibliothèque TLC [14], dont l'interface est donnée figure 2.12.

Les environnements de TLC sont représentés par des listes associatives : à une variable est associée une valeur de type `A`. La valeur `empty` représente une substitution vide. Une valeur de la forme `x ~ t` est un singleton associant à la variable `x` la valeur `t`. La forme `E & F` permet la concaténation de deux environnements. Ainsi, l'ajout d'une liaison à un environnement `E` s'écrit `E & x ~ t`. La notation `x # E` teste que la variable `x` n'est effectivement pas liée dans `E`. Enfin, TLC propose directement les propriétés sur les substitutions énoncées plus tôt : le type inductif `ok E` encode la propriété de bonne formation d'un environnement, et la propriété `extends` celle

```
Definition substs := env ty.

Inductive inst : substs → ty → ty → substs → Prop :=
| inst_fvar_r : forall t v s,
   v # s → inst s t (Fvar v) (s & v  t)
| inst_fvar_in : forall t t' v s,
   binds v t' s → equiv t t' → inst s t (Fvar v) s
| inst_int : forall s, inst s Int Int s
| inst_arrow : forall s s' s'' t11 t12 t21 t22,
   inst s t11 t21 s' → inst s' t12 t22 s'' →
   inst s (Arrow t11 t12) (Arrow t21 t22) s''.

Fixpoint initial_substs (vs : list var) :=
  match vs with
  | nil ⇒ empty
  | cons v vs' ⇒
   initial_substs vs' & v ~ Fvar v
  end.

Inductive inst_sch : ty → sch → substs → Prop :=
| inst_forall : forall t t' ar s vs,
   fresh (fv t \u fv t') ar vs →
   inst (initial_substs (fv_list t')) t (t' ^ vs) s →
   inst_sch t (Forall ar t') s.
```

FIGURE 2.11 – Définition inductive de l'instantiation

```
Definition env (A:Type) := list (var * A).

Definition empty : env A := nil.

Definition dom : env A → vars.

Definition get : env A → var → option A.


(** ** Notations *)

(** [x ~ a] is the notation for a singleton environment mapping x to a. *)

Notation "x ~ a" := (single x a)
  (at level 27, left associativity) : env_scope.

(** [E & F] is the notation for concatenation of E and F. *)

Notation "E & F" := (concat E F)
  (at level 28, left associativity) : env_scope.

(** [x # E] to be read x fresh from E captures the fact that
    x is unbound in E . *)

Notation "x '#' E" := (x ∉ (dom E)) (at level 67) : env_scope.



(** Well-formed environments contains no duplicate bindings. *)

Inductive ok : env A → Prop :=
  | ok_empty :
      ok empty
  | ok_push : forall E x v,
      ok E → x # E → ok (E & x  v).

(** An environment contains a binding from x to a iff the most recent
  binding on x is mapped to a *)

Definition binds x v E :=
  get x E = Some v.

(** Inclusion of an environment in another one. *)

Definition extends E F :=
  forall x v, binds x v E → binds x v F.
```

FIGURE 2.12 – TLC : interface des environnements

de la l'inclusion. La substitution est donc encodée comme un environnement qui associe un type à une variable.

L'instanciation travaille sur variables localement libres : l'ouverture du schéma de types remplace les variables liées par le schéma par des variables fraîches. Cela explique notamment l'intérêt d'utiliser les environnements de TLC qui travaillent justement sur les variables libres. Les conditions d'instanciation sont donc identiques aux règles énoncées en figure 2.9. La génération de la substitution initiale est légèrement différente : plutôt que de retirer les variables quantifiées de l'ensemble des variables libres, il suffit de récupérer les variables libres avant l'ouverture du schéma. De cette manière, puisque les variables libres et liées sont différenciées, seules les variables non quantifiées seront effectivement récupérées.

Filtrage

Le filtrage de type est une opération permettant de raffiner la forme d'un type. Cette opération a alors deux objectifs :

1. Vérifier la forme courante du type de manière explicite.
2. Introduire les éventuels sous-nœuds du type sous la forme de nouvelles métavariables.

Le filtrage est de la forme $\tau \prec \tau'$. On considère alors τ' comme un motif de type. Supposons par exemple $\tau \prec \tau_1 \to \tau_2$: cette opération va alors vérifier que τ est bien un type fonctionnel, et va introduire son domaine et son codomaine dans le contexte courant respectivement sous le nom des métavariables τ_1 et τ_2. L'utilisation du filtrage est nécessaire pour éviter les éventuelles opérations implicites sur les types, par exemple dans le cas de l'introduction d'abréviations et d'équations de types. Le filtrage fait alors office d'opération d'expansion des abréviations et d'extraction de types équivalents, comme montré dans le chapitre dédié à la vérification de TAST OCaml.

2.4.2 Environnement

Pour que la vérification soit possible, il est alors nécessaire d'utiliser un environnement de typage. Celui-ci associe un schéma de type à des variables dans le contexte de la vérification. Un tel environnement est nécessaire pour vérifier que les variables libres dans un contexte donné sont vérifiables et instanciées correctement.

L'environnement est ici représenté sous la forme d'une liste associant des variables à des schéma de types. Sa définition et ses propriétés sont données dans la figure 2.13.

On remarquera que cette définition ainsi que les deux propriétés de bonne formation et de domaine sont isomorphes à celles des substitutions présentées figure 2.10. Ainsi, les différentes propriétés prouvées sur ces seules définitions dans la sous-section 2.4.1 sont compatibles avec celles des environnements.

Environnement : représentation Coq L'environnement utilisé pour la vérification est celui fournit par TLC (voir figure 2.12), et associe des schémas de types à des variables. Les opérations sont donc strictement identiques à celles énoncées pour les substitutions.

Définitions et propriétés :

— $\Gamma ::= \emptyset \mid \Gamma, x : \sigma$

— $dom(\Gamma)$: ensemble des clés (variables) de l'environnement.

— $fv(\Gamma)$: ensemble des variables de types libres apparaissant dans l'environnement.

— $wf(\Gamma)$: prédicat indiquant l'unicité des variables liées dans l'environnement.

— $\Gamma \subseteq \Gamma'$: prédicat de sous-ensemble.

Domaine :

$$\begin{array}{lcl} dom(\emptyset) & \rightsquigarrow & \emptyset \\ dom(\Gamma, x : \sigma) & \rightsquigarrow & dom(\Gamma) \cup \{ x \} \end{array}$$

Bonne formation :

$$\textsc{Wf-Env-Empty}\ wf(\emptyset) \qquad\qquad \textsc{Wf-Env-Bind}\ \frac{x \notin dom(\Gamma) \qquad wf(\Gamma)}{wf(\Gamma, x : \sigma)}$$

FIGURE 2.13 – Environnements

2.4.3 Vérification d'expressions annotées

Maintenant que l'ensemble des opérations de vérification sur les types ont été définies, on peut définir un ensemble de règles pour vérifier des expressions annotées. Celle-ci sont données en figure 2.14.

La vérification des constantes (CONST) est relativement simple, puisqu'il s'agit de s'assurer que le type annoté est équivalent au constructeur de type `int`. La vérification des variables (VAR) nécessite de vérifier d'abord que la variable existe dans l'environnement courant, et que le type annoté est une instance du schéma associé à la variable dans l'environnement. On utilise la substitution initiale générée à partir du schéma pour calculer l'instance. La vérification de l'abstraction (ABS) nécessite d'abord de s'assurer que le type annoté au lambda est un type de fonction. On s'assure ensuite que le type annoté à l'argument est équivalent au domaine de cette fonction, puis que le corps du lambda est cohérent avec l'environnement initial auquel on ajoute la variable associée à un schéma de type *vide*, *i.e.* qui ne quantifie pas de variable, et dont le corps est le type du domaine. Finalement, il suffit de vérifier que le type du corps du lambda est équivalent au codomaine du type de la fonction. La vérification de l'application (APP) est mécanique : on vérifie la fonction, en s'assurant que son type est bien celui d'une fonction dont on extrait le domaine et le codomaine, puis l'argument tout en vérifiant que son type est équivalent au domaine. Finalement, on vérifie l'équivalence du type annoté à l'expression avec celui du codomaine de la fonction. Enfin, la vérification du **`let`** (LET) est une combinaison de l'abstraction et de l'application. On vérifie d'abord l'expression locale du **`let`**, puis que la généralisation n'a pas quantifié des variables déjà libres dans l'environnement, et enfin que son type est équivalent au corps du schéma de type annoté à la variable locale x. Ensuite, on vérifie le corps de l'expression avec l'environnement auquel on ajoute la variable locale associée à son schéma de type. Il suffit ensuite de vérifier que le type du **`let`** est équivalent au type du corps.

$$\textsc{Const}\ \frac{\tau \equiv \texttt{int}}{\Gamma \vdash \langle c : \tau \rangle} \qquad \textsc{Var}\ \frac{x \in dom(\Gamma) \qquad \Gamma(x) \prec \forall\overline{\alpha}.\tau' \qquad \theta_{init}(\forall\overline{\alpha}.\tau') \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta}{\Gamma \vdash \langle x : \tau \rangle}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash_{\tau} \tau \prec \tau_d \rightarrow \tau_{cd} \qquad \tau_x \equiv \tau_d \qquad \Gamma, x : \forall.\tau_x \vdash \langle e : \tau_e \rangle \qquad \tau_e \equiv \tau_{cd}}{\Gamma \vdash \langle \lambda.\langle x : \tau_x \rangle \rightarrow e\ : \tau \rangle}$$

$$\textsc{App}\ \frac{\Gamma \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma \vdash_{\tau} \tau_1 \prec \tau_d \rightarrow \tau_{cd} \qquad \Gamma \vdash_{\tau} \tau_2 \equiv \tau_d \qquad \Gamma \vdash_{\tau} \tau_{cd} \equiv \tau}{\Gamma \vdash \langle e_1\ e_2 : \tau \rangle}$$

$$\textsc{Let}\ \frac{\Gamma \vdash \langle e : \tau_e \rangle \qquad \sigma \prec \forall\overline{\alpha}.\tau_x \qquad \overline{\alpha} \notin fv(\Gamma) \qquad \tau_e \equiv \tau_x \qquad \Gamma, x : \sigma \vdash \langle e' : \tau'_e \rangle \qquad \tau'_e \equiv \tau}{\Gamma \vdash \langle \textbf{let}\ \langle x : \sigma \rangle = e\ \textbf{in}\ e' : \tau \rangle}$$

FIGURE 2.14 – Règles de vérification de types de MiniML

Si on compare ces règles à celles pour l'inférence présentées précédemment (voir section 2.2), on peut remarquer une différence essentielle : toutes les opérations liées aux types sont explicites. En effet, là où les équivalences de types ou les suppositions de formes sont implicites via le partage des métavariables ou les annotations, chacune de ces opérations est ici décrite explicitement. L'intéret est dans notre cas de rendre l'opération de vérification aussi précise que possible, mais aussi d'exprimer le système de manière exécutable. Ainsi, la transcription des règles est mécanique, il est donc possible d'écrire un vérificateur de la même manière que les règles sont lues.

Vérification d'expression annotées : représentation Coq Sans surprise, la représentation des règles de vérification se fait à l'aide d'une propriété inductive (figure 2.15). Chaque branche est une transposition littérale des règles énoncées plus tôt. Il faut néanmoins noter la fonction `type_of` de type `typed_term → ty`, qui permet d'extraire le type annoté à une expression. On retrouve une fois encore le principe d'ouverture des abstractions lors de la traversée des fonctions et des définitions locales.

2.4.4 Implémentation d'un vérificateur de types de MiniML

Supposons la nomenclature suivante :
- `check` désigne l'opération de vérification des expressions ;
- `check_gen` correspond à l'opération de vérification de la généralisation ;
- `match_*` désigne l'opération de filtrage des types, dont * est à remplacer par le type filtré ;
- `equiv` est l'opération d'équivalence ;
- `inst` est l'opération d'instanciation.

Prenons par exemple la règle de vérification du **`let`**. On supposera un langage similaire à

```
(** Type checking *)

Definition chk_env := env sch.

Inductive check : chk_env → typed_term → Prop :=
| chk_fvar : forall env v t sch sub,
    binds v sch env → inst_sch t sch sub →
    check env (term_fvar v t)
| chk_app : forall env e1 e2 td tcd t,
    check env e1 →
    has_arrow_form (type_of e1) td tcd →
    check env e2 → equiv td (·: e2) →
    equiv t tcd →
    check env (term_app e1 e2 t)
| chk_fun : forall env targ body tfun td tcd fvs,
    has_arrow_form tfun td tcd →
    equiv targ td →
    equiv (type_of body) tcd →
    (forall x,
       x \notin fvs →
       check (env & x \sim Forall 0 targ)
             (body ^ x)) →
    check env (term_function targ body tfun)
| chk_let : forall env s e1 e2 t fvs,
    (forall vs,
       fresh fvs (sch_arity s) vs →
       check env (e1 ^\t (List.map Fvar vs))) →
    equiv (type_of e1) (sch_body s) →
    (forall x,
       x \notin fvs →
       check (env & x \sim s)
             (e2 ^ x)) →
    equiv (type_of e2) t →
    check env (term_let s e1 e2 t)
```

FIGURE 2.15 – Implémentation des règles de vérification en Coq

```
let check Γ e =
match e with
...
| ⟨ let ⟨ x, σ ⟩ = ⟨ e, τ_e ⟩ = ⟨ e', τ'_e ⟩, τ ⟩ ->
check_expression Γ ⟨ e, τ_e ⟩;
let ᾱ, τ_σ = match_forall σ in
check_gen Γ alpha;
equiv τ_e τ_σ;
check_expression (Γ,x:σ) ⟨ e', τ'_e ⟩;
equiv τ'_e τ

let check Γ e =
match e with
...
| ⟨ let ⟨ x, σ ⟩ = ⟨ e, τ_e ⟩ = ⟨ e', τ'_e ⟩, τ ⟩ ->
check_expression Γ ⟨ e, τ_e ⟩ >>= fun () ->
match_forall σ >>= fun (ᾱ, τ_σ) ->
check_gen Γ alpha >>= fun () ->
equiv τ_e τ_σ >>= fun () ->
check_expression (Γ,x:σ) ⟨ e', τ'_e ⟩ >>= fun () ->
equiv τ'_e τ
```

FIGURE 2.16 – Implémentation de la règle LET

MiniML, avec du filtrage de motifs. Deux exemples sont donnés en figure 2.16 : le premier dans un style purement impératif lève des exceptions dans le cas où une vérification échouerait, le second étant une version fonctionnelle écrite dans un style monadique. Dans ce cas, chaque opération retourne une valeur de type `('a, error) result`, où `error` est un type décrivant l'erreur.

On peut remarquer que l'ensemble des opérations est identique à celle des règles. Il ne reste alors qu'à déterminer la structure de données pour les types, les schémas de type, les expressions (bien que celles-ci soient représentées ici par une syntaxe concrète) et les environnements.

On définit l'ensemble des structures de données dans la figure 2.17. Toutes les quantifications sont représentées par des indices de De Bruijn, pour éviter les éventuels problèmes liés à l'alpha-conversion. Les types et les expressions sont alors représentés par des types algébriques. Chaque expression est effectivement annotée par un type, et le type (ou le schéma de type) des arguments de fonctions (ou variables locales) est donné explicitement. Pour représenter les ensembles de types, on utilise la représentation des ensembles de la biliothèque standard d'OCaml, instanciée avec la fonction de comparaison polymorphe de celle-ci. Les substitutions sont de simples listes associatives. L'environnement est une liste de schémas de types : l'utilisation d'indices de De Bruijn permet d'utiliser la position dans la liste comme identificateur puisqu'un ajout dans l'environnement concerne toujours la variable locale courante, donc d'indice 0. Les fonctions d'équivalence et d'instanciation peuvent alors être écrites mécaniquement (voir figure 2.18)

Finalement, l'algorithme de vérification est donné en figure 2.19.

```
(** Algèbre de types *)
type ty =
    TVar of int
  | TArrow of ty * ty
  | TInt

type sch = Forall of int * ty

(** Expressions *)
type expr_desc =
    Cst of int
  | Var of int
  | Lam of ty * expr
  | App of expr * expr
  | Let of sch * expr * expr

and expr = expr_desc * ty

(** Ensemble de types *)
module TSet = Set.Make (struct type t = ty let compare = compare end)

(** Substitutions *)
type sub = (int * ty) list

let in_dom_sub nat sub = List.mem_assoc nat sub
let add_sub nat ty sub = (nat, ty) :: sub
let get_sub nat sub = List.assoc nat sub

(** Environnements *)
type env = sch list

let in_dom_env n env = n < List.length env
let get_env n env = List.nth env n
let add_env sch env = sch :: env
```

Figure 2.17 – Structures de données des types, expressions et environnement

```
let rec equiv t1 t2 =
  match t1, t2 with
  | TVar n1, TVar n2 when n1 = n2 -> Ok ()
  | TArrow (t11, t12), TArrow (t21, t22) ->
    equiv t11 t21 >>= fun () ->
    equiv t12 t22
  | TInt, TInt -> Ok ()
  | t1, t2 -> Error (Not_equiv (t1, t2))

let equiv_sch sch1 sch2 =
  match sch1, sch2 with
    Forall (n1, t1), Forall (n2, t2) when n1 = n2 ->
    equiv t1 t2
  | _, _ -> Error (Not_equiv_sch (sch1, sch2))

let rec inst sub t1 t2 =
  match t1, t2 with
  | t1, TVar n when not (in_dom_sub n sub) -> Ok (add_sub n t1 sub)
  | t1, TVar n -> equiv (get_sub n sub) t1 >>= fun () -> Ok sub
  | TArrow (t11, t12), TArrow (t21, t22) ->
    inst sub t11 t21 >>= fun sub ->
    inst sub t12 t22
  | TInt, TInt -> Ok sub
  | t1, t2 -> Error (Not_inst (t1, t2))

let sub_init sch =
  TSet.fold (fun ty acc ->
      match ty with
      | TVar n -> (n, TVar n) :: acc
      | _ -> assert false)
    (fv_sch sch)
    []
```

FIGURE 2.18 – Implémentation de l'équivalence et l'instanciation

```
type error += Not_bound of env * expr

let rec check env e =
  match e with
  | Cst _, t -> equiv t TInt
  | Var n, t ->
    if not (in_dom_env n env) then Error (Not_bound (env, e))
    else
      match_forall (get_env n env) >>= fun (ar, t') ->
      inst (sub_init (Forall (ar, t'))) t t' >>= fun _ -> Ok ()
  | Lam (tx, (b, tb)), t ->
    match_arrow t >>= fun (td, tcd) ->
    equiv tx td >>= fun _ ->
    check (add_env (Forall (0, tx)) env) (b, tb) >>= fun _ ->
    equiv tb tcd
  | App ((e1, t1), (e2, t2)), t ->
    check env (e1, t1) >>= fun _ ->
    check env (e2, t2) >>= fun _ ->
    match_arrow t1 >>= fun (td, tcd) ->
    equiv t2 td >>= fun _ ->
    equiv tcd t
  | Let (sch, (e, te), (e', te')), t ->
    check env (e, te) >>= fun _ ->
    match_forall sch >>= fun (ar, tsch) ->
    equiv te tsch >>= fun _ ->
    check (add_env sch env) (e', te') >>= fun _ ->
    equiv te' t
```

FIGURE 2.19 – Implémentation de la vérification de cohérence de MiniMLannoté

$v ::=$ *valeurs*
$\mid \langle\, c : \tau \,\rangle$ *constante*
$\mid \langle\, \lambda \langle\, x : \tau \,\rangle .\, e : \tau \,\rangle$ *abstraction*

— *value*(e) : teste qu'une expression e est une valeur.

Réduction des expressions annotées

$$\textsc{App-Left}\ \frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad \textsc{App-Right}\ \frac{\mathit{value}(v_1) \qquad e_2 \rightsquigarrow e_2'}{v_1\ e_2 \rightsquigarrow v_1\ e_2'}$$

$$\textsc{App-Fun}\ \frac{\mathit{value}(v_2)}{(\lambda\langle x : \tau\rangle.e_1)\ v_2 \rightsquigarrow v_2[x \mapsto e_1]}$$

$$\textsc{Let-Bind}\ \frac{e_1 \rightsquigarrow e_1'}{\mathbf{let}\ \langle x : \sigma\rangle = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ \langle x : \sigma\rangle = e_1'\ \mathbf{in}\ e_2}$$

$$\textsc{Let-Value}\ \frac{\mathit{value}(v_1)}{\mathbf{let}\ \langle x : \forall\overline{\alpha}.\tau\rangle = v_1\ \mathbf{in}\ e_2 \rightsquigarrow e_2[x \mapsto v_1; \overline{\alpha}]_\sigma}$$

FIGURE 2.20 – Sémantique opérationnelle de MiniML annoté

2.5 Sémantique opérationelle du TAST

La sémantique opérationnelle de MiniML avec annotation est donnée dans la figure 2.20. On distingue alors les valeurs comme étant les expressions irréductibles, en d'autres termes :

— Les variables
— Les constantes entières
— Les abstractions

La sémantique est une sémantique à petits pas, avec substitution des variables liées. La réduction des expressions est alors relativement classique, à un point près :

— Une application se réduit toujours de gauche à droite, d'où les trois cas suivants.
  — Si l'application se fait d'une expression non réduite sur une expression quelconque : la première est réduite ;
  — Si l'application se fait d'une valeur sur une expression, alors cette expression est réduite ;
  — Enfin, si la première expression est une fonction et la seconde une valeur, on substitue la valeur de l'argument à la variable liée dans le corps de la fonction.
— Dans le cas d'une variable locale (LET-*), deux cas peuvent se présenter :

— Tant que l'expression liée n'est pas une valeur, elle se réduit.
— Dès que l'expression liée est une valeur, elle se substitue tout en instanciant les variables de types dans le corps de la seconde expression

Cette dernière condition est en effet importante : lorsque qu'une valeur dont le type est un schéma de type se substitue à sa variable liée dans le corps d'une autre, il est nécessaire pour chaque occurrence de cette variable de calculer une substitution entre ses variables de types quantifiées et le type auquel la variable a été instanciée.

Autrement dit, prenons l'exemple suivant :

```
let ⟨ x : ∀α. α → α ⟩ = λ ⟨ x : α ⟩. ⟨ x : α ⟩ in
⟨ f : int → int ⟩ ⟨ 0: int ⟩
```

Le corps du **let** étant une valeur, la prochaine étape de réduction est donc de substituer à `x` sa valeur dans `⟨ f : int → int ⟩ ⟨ 0: int ⟩`. Avec une substitution classique, le résultat serait :

```
⟨ λ ⟨ x : α ⟩. ⟨ x : α ⟩ : α → α ⟩ ⟨ 0: int ⟩
```

Or, un tel résultat n'est pas un TAST correct : en effet, sachant la règle Abs (figure 2.14), le type de l'argument doit être équivalent au type du domaine de la fonction. Dans ce cas, le type `int` doit être équivalent à α, ce qui n'est trivialement pas le cas. Une telle règle de réduction ne permettrait donc pas de garantir la propriété de préservation, et donc par conséquent la sûreté du système de types vis-à-vis de cette sémantique. La solution est alors de générer une substitution qui prend en compte les instanciations de variables. Celle-ci, ainsi que la substitution simple, est donnée figure 2.21.

La substitution dite classique est simple : il s'agit de remplacer toute occurrence d'une variable par une expression dans le corps d'une expression donnée. Les deux substitutions sont identiques, à une exception près, celle des variables. L'idée est de traverser l'expression pour trouver les occurrences de variables, et si celle-ci correspond : la remplacer. Bien entendu, il ne faut substituer que les variables libres dans le corps de l'expression. Ainsi, si la variable à substituer est x et que l'abstraction à traverser lie une variable x, le corps de celle-ci ne doit pas être substitué. Il en va de même pour les variables locales liées par un **let**. Dans le cas d'une variable, si celle-ci est différente de celle à substituer, il n'y a donc rien à faire. En revanche, dans le cas classique, s'il s'agit de la bonne variable, alors l'algorithme de substitution retourne l'expression qui doit se substituer.

Dans le cas de la substitution avec instanciation, le résultat est plus complexe. Il faut d'abord calculer l'instance entre le type annoté et le type de l'expression : cela permet alors de connaître les instances pour chaque variable quantifiée. Il est nécessaire de commencer ce calcul d'une une substitution initiale (voir page 32) pour éviter d'instancier les variables libres. Ensuite, cette nouvelle substitution obtenue doit être appliquée sur l'expression de substitution.

Reprenons l'exemple précédent. La substition à appliquer est donc :

```
[ f ↦ ⟨ λ ⟨ x : α ⟩. ⟨ x : α ⟩ : α → α ⟩; θinit(∀α. α → α) ]
```

La variable `f` étant annotée avec `int → int`, il faut calculer la substitution permettant une telle

Substitution classique

$$
\begin{array}{lcl}
\langle\, x : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \mathrm{e} \\
\langle\, y : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \langle\, y : \tau \,\rangle \\
\langle\, \mathrm{c} : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \langle\, \mathrm{c} : \tau \,\rangle \\
\langle\, \lambda \langle\, x : \tau' \,\rangle.\, \mathrm{e'} : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \langle\, \lambda \langle\, x : \tau' \,\rangle.\, \mathrm{e'} : \tau \,\rangle \\
\langle\, \lambda \langle\, y : \tau' \,\rangle.\, \mathrm{e'} : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \langle\, \lambda \langle\, y : \tau' \,\rangle.\, (\mathrm{e'}\, [\, x \mapsto \mathrm{e}\,]) : \tau \,\rangle \\
\langle\, \mathrm{e}_1\, \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \langle\, (\mathrm{e}_1\, [\, x \mapsto \mathrm{e}\,])\, (\mathrm{e}_2\, [\, x \mapsto \mathrm{e}\,]) : \tau \,\rangle \\
\langle\, \mathbf{let}\, \langle\, x : \sigma \,\rangle = \mathrm{e}_1\, \mathbf{in}\, \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \langle\, \mathbf{let}\, \langle\, x : \sigma \,\rangle = \mathrm{e}_1\, \mathbf{in}\, \mathrm{e}_2 : \tau \,\rangle \\
\langle\, \mathbf{let}\, \langle\, y : \sigma \,\rangle = \mathrm{e}_1\, \mathbf{in}\, \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e}\,] & \rightsquigarrow & \begin{array}{c}\langle\, \mathbf{let}\, \langle\, y : \sigma \,\rangle = \mathrm{e}_1\, [\, x \mapsto \mathrm{e}\,]\, \mathbf{in} \\ \mathrm{e}_2\, [\, x \mapsto \mathrm{e}\,] : \tau \,\rangle\end{array}
\end{array}
$$

Substitution avec instantiation

$$
\begin{array}{lcl}
\langle\, x : \tau \,\rangle\, [\, x \mapsto \langle\, \mathrm{e} : \tau' \,\rangle; \theta_\alpha\,]_\sigma & \rightsquigarrow & \theta_\alpha \vdash_\tau \tau \leq \tau' \Rightarrow \theta; \langle\, \theta(\mathrm{e}) : \tau \,\rangle \\
\langle\, y : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \langle\, y : \tau \,\rangle \\
\langle\, \mathrm{c} : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \langle\, \mathrm{c} : \tau \,\rangle \\
\langle\, \lambda \langle\, x : \tau' \,\rangle.\, \mathrm{e'} : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \langle\, \lambda \langle\, x : \tau' \,\rangle.\, \mathrm{e'} : \tau \,\rangle \\
\langle\, \lambda \langle\, y : \tau' \,\rangle.\, \mathrm{e'} : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \langle\, \lambda \langle\, y : \tau' \,\rangle.\, (\mathrm{e'}\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma) : \tau \,\rangle \\
\langle\, \mathrm{e}_1\, \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \langle\, (\mathrm{e}_1\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma)\, (\mathrm{e}_2\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma) : \tau \,\rangle \\
\langle\, \mathbf{let}\, \langle\, x : \sigma \,\rangle = \mathrm{e}_1\, \mathbf{in}\, \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \langle\, \mathbf{let}\, \langle\, x : \sigma \,\rangle = \mathrm{e}_1\, \mathbf{in}\, \mathrm{e}_2 : \tau \,\rangle \\
\langle\, \mathbf{let}\, \langle\, y : \sigma \,\rangle = \mathrm{e}_1\, \mathbf{in}\, \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma & \rightsquigarrow & \begin{array}{c}\langle\, \mathbf{let}\, \langle\, y : \sigma \,\rangle = \mathrm{e}_1\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma\, \mathbf{in} \\ \mathrm{e}_2\, [\, x \mapsto \mathrm{e}; \theta_\alpha\,]_\sigma : \tau \,\rangle\end{array}
\end{array}
$$

FIGURE 2.21 – Substitution naïve de variable dans une expression annotée

$$\text{SEMINST-VAR-UNBOUND}\ \frac{\alpha \notin dom(\theta)}{\theta \models_{\tau} \tau \lhd\ \alpha \Rightarrow \theta \oplus [\alpha \to \tau]} \qquad \text{SEMINST-VAR-BOUND}\ \frac{\alpha \in dom(\theta)}{\theta \models_{\tau}\ \tau \lhd\ \alpha \Rightarrow \theta}$$

$$\text{SEMINST-FUN}\ \frac{\theta \models_{\tau} \tau_1 \lhd\ \tau_1' \Rightarrow \theta_1 \qquad \theta_1 \models_{\tau} \tau_2 \lhd\ \tau_2' \Rightarrow \theta_2}{\theta \models_{\tau} \tau_1 \to \tau_2 \lhd\ \tau_1' \to \tau_2' \Rightarrow \theta_2}$$

$$\text{SEMINST-CONSTRUCT}\ \theta \models_{\tau} \texttt{int} \lhd\ \texttt{int} \Rightarrow \theta \qquad \text{SEMINST-WILDCARD}\ \theta \models_{\tau} \_ \lhd \_ \Rightarrow \theta$$

FIGURE 2.22 – Instanciation sémantique

instance (la substitution initiale étant vide) :

$$\dfrac{\dfrac{\alpha \notin \emptyset}{\emptyset \models_{\tau} \texttt{int} \leq \alpha \Rightarrow [\alpha \mapsto \texttt{int}]} \qquad \dfrac{\alpha \in dom([\alpha \mapsto \texttt{int}]) \qquad \texttt{int} \equiv \texttt{int}}{[\alpha \mapsto \texttt{int}] \models_{\tau} \texttt{int} \leq \alpha \Rightarrow [\alpha \mapsto \texttt{int}]}}{\emptyset \models_{\tau} \texttt{int} \to \texttt{int} \leq \alpha \to \alpha \Rightarrow [\alpha \mapsto \texttt{int}]}$$

Si on suit les règles d'instanciation, le test d'équivalence est alors nécessaire lorsqu'une instance pour une variable existe déjà. Néanmoins réutiliser un tel algorithme lié n'est pas forcément nécessaire ni efficace. En effet, dans le cas de MiniML l'équivalence et l'instanciation travaillent sur des équivalences syntaxiques. Comme nous le verrons dans le chapitre suivant, en présence d'abréviations et d'équations de types il devient nécessaire d'utiliser l'environnement de typage. Or, pendant la réduction, l'environnement n'est pas présent. Une solution serait alors de l'annoter dans les nœuds des expressions, mais cela rajoute la contrainte supplémentaire d'être en mesure de vérifier des environnements, puisque le but de la vérification de TAST est de vérifier l'ensemble des informations des nœuds, en particulier si ces utilisations sont utilisées par le reste de l'algorithme. Dans le cas de MiniML, l'environnement est simple, mais il ne l'est plus dans le cas d'OCaml avec déclarations de types, abréviations et équations liées aux types gardés.

Pour la réduction, l'instantiation n'est pas nécessaire : en effet, tester l'équivalence ne change pas la sémantique, puisque ML non annoté serait parfaitement réductible dans tous les cas. L'intérêt de ce calcul d'instance permet de garantir la preuve de préservation. Il suffit alors d'utiliser une version simplifiée de l'instanciation, dite aussi instanciation sémantique donnée en figure 2.22.

Le système calcule une substitution, de manière similaire à l'instanciation. La grande différence réside donc dans l'absence du test d'équivalence pour les variables dont il existe une substitution, mais aussi dans l'ajout d'une règle *par défaut*. La règle SEMINST-WILDCARD est compatible avec toutes les combinaisons de types non exprimables dans les prémisses des autres règles. L'intérêt ici de ne pas rendre le calcul bloquant pour la réduction. Dans le cas d'une expression correctement annotée et donc vérifiable, cette règle ne sera jamais empruntée durant le calcul.

Lemme 13 (Instanciation implique instanciation sémantique)**.** *Pour tous types τ et σ, pour toutes substitutions θ_i et θ, si $\theta_i \models_{\tau} \tau \leq \sigma \Rightarrow \theta$, alors $\theta_i \models_{\tau} \tau \lhd \sigma \Rightarrow \theta$.*

Substitution avec instantiation

$$
\begin{array}{lll}
\langle\, x : \tau \,\rangle\,[\, x \mapsto \langle\, e : \tau' \,\rangle ; \theta_\alpha \,]_\sigma & \leadsto & \theta_\alpha \vdash_\tau \tau \lhd \tau' \Rightarrow \theta ; \langle\, \theta(e) : \tau \,\rangle \\
\langle\, y : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \langle\, y : \tau \,\rangle \\
\langle\, c : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \langle\, c : \tau \,\rangle \\
\langle\, \lambda \langle\, x : \tau' \,\rangle . e' : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \langle\, \lambda \langle\, x : \tau' \,\rangle . e' : \tau \,\rangle \\
\langle\, \lambda \langle\, y : \tau' \,\rangle . e' : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \langle\, \lambda \langle\, y : \tau' \,\rangle . (e'\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma) : \tau \,\rangle \\
\langle\, e_1\ e_2 : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \langle\, (e_1\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma)\ (e_2\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma) : \tau \,\rangle \\
\langle\, \mathbf{let}\, \langle\, x : \sigma \,\rangle = e_1\ \mathbf{in}\ e_2 : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \langle\, \mathbf{let}\, \langle\, x : \sigma \,\rangle = e_1\ \mathbf{in}\ e_2 : \tau \,\rangle \\
\langle\, \mathbf{let}\, \langle\, y : \sigma \,\rangle = e_1\ \mathbf{in}\ e_2 : \tau \,\rangle\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma & \leadsto & \begin{array}{c}\langle\, \mathbf{let}\, \langle\, y : \sigma \,\rangle = e_1\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma\ \mathbf{in} \\ e_2\,[\, x \mapsto e ; \theta_\alpha \,]_\sigma : \tau \,\rangle\end{array}
\end{array}
$$

FIGURE 2.23 – Substitution de variables liée dans une expression annotée

$$
\begin{array}{lll}
\alpha\,[\, \alpha \mapsto \tau \,] & \leadsto & \tau \\
\beta\,[\, \alpha \mapsto \tau \,] & \leadsto & \beta \\
\mathtt{int}\,[\, \alpha \mapsto \tau \,] & \leadsto & \mathtt{int} \\
\tau_1 \to \tau_2\,[\, \alpha \mapsto \tau \,] & \leadsto & (\tau_1\,[\, \alpha \mapsto \tau \,]) \to (\tau_2\,[\, \alpha \mapsto \tau \,])
\end{array}
$$

FIGURE 2.24 – Substitution de variable de type dans un type

Démonstration. Par induction sur les règles de l'instanciation. L'instanciation sémantique étant une version de l'instanciation où seule une règle est changée et appauvrie d'un test, il est simple de montrer que si l'instanciation produit une substitution, l'instanciation produira la même. □

L'application de la substitution sur les expressions et les types sont respectivement données en figures 2.23 et 2.24. La substitution reste très classique, dans le cas des types il s'agit donc de remplacer chaque variable de type qui apparait dans la substitution par le type auquel elle est associée.

2.5.1 Représentation Coq de la sémantique de MiniML

La sémantique du langage est définie de manière inductive en Coq (figure 2.25). Cette définition suit la sémantique formalisée précédemment. Les deux étapes importantes sont celles de la beta-réduction : avant d'effectuer la substitution, il est nécessaire d'ouvrir l'expression avec une variable fraîche sachant un ensemble de variables existantes. Une fois l'ouverture effectuée, la substitution est possible. Dans le cas du **`let`**, son schéma de type doit d'abord être ouvert avant la substitution, et donc toutes les variables généralisées qui apparaissent dans le corps de la valeur liées doivent également être remplacées par des variables libres fraiches. Enfin, il faut générer une substitution initiale pour permettre l'instantiation sémantique lors de la substitution.

2.5.2 Implémentation d'un évaluateur en OCaml

En reprenant la définition de MiniML implémentée précédemment en OCaml, on peut écrire un évaluateur en suivant une sémantique à petits pas. L'implémentation d'un tel évaluateur est

```
Inductive value : typed_term → Prop :=
| value_var : forall n t, value (term_var n t)
| value_fvar : forall v t, value (term_fvar v t)
| value_function : forall t e t', value (term_function t e t').

Inductive step : typed_term → typed_term → Prop :=
| step_app1 : forall e1 e1' e2 t,
   step e1 e1' → step (term_app e1 e2 t) (term_app e1' e2 t)
| step_app2 : forall e1 e2 e2' t,
   value e1 → step e2 e2' → step (term_app e1 e2 t) (term_app e1 e2' t)
| step_app_fun : forall e e' e2 t targ tfun fvs,
   value e2 →
   closed_term e' →
   (forall x,
      x \notin fvs →
      e' = [x ~> e2;
             (initial_substs (fv_list targ))] (e ^ x)) →
   step (term_app (term_function targ e tfun) e2 t) e'
| step_let : forall s e1 e1' e2 t,
   step e1 e1' → step (term_let s e1 e2 t) (term_let s e1' e2 t)
| step_let_val : forall s e1 e2 e' t fvs,
   value e1 →
   (forall x tvs,
      x \notin fvs →
      fresh (fvs \u \{x}) (sch_arity s) tvs →
      e' = [x ~>
             e1 ^\t (List.map Fvar tvs);
             (initial_substs (fv_sch_list s))]
            (e2 ^ x)) →
   step (term_let s e1 e2 t) e'.
```

FIGURE 2.25 – Sémantique à petits pas de MiniML décrite inductivement

```
Inductive sem_inst : substs → ty → ty → substs → Prop :=
| sem_inst_fvar_r : forall t v s,
   v # s → sem_inst s t (Fvar v) (s & v  t)
| sem_inst_fvar_in : forall t t' v s,
   binds v t' s → sem_inst s t (Fvar v) s
| sem_inst_int : forall s, sem_inst s Int Int s
| sem_inst_arrow : forall s s' s'' t11 t12 t21 t22,
   sem_inst s t11 t21 s' → sem_inst s' t12 t22 s'' →
   sem_inst s (Arrow t11 t12) (Arrow t21 t22) s''
| sem_inst_any : forall s s' t1 t2,
   sem_inst s t1 t2 s'.


Inductive sem_inst_sch : ty → sch → substs → Prop :=
| sem_inst_forall : forall t t' ar s vs,
   fresh (fv t \u fv t') ar vs →
   sem_inst (initial_substs (fv_list t')) t (open_sch_aux t' vs) s →
   sem_inst_sch t (Forall ar t') s.
```

FIGURE 2.26 – Instanciation sémantique implémentée en Coq

donné en figure 2.27. La fonction `is_value` permet de déterminer quelles expressions sont des valeurs, autrement dit dans notre cas : les variables, les constantes et les abstractions. La fonction `step` évalue une expression en suivant la sémantique énoncée figure 2.20. La première branche est le cas de la beta-réduction : la partie gauche de l'application est une abstraction et la partie droite une valeur. On doit donc remplacer l'argument d'indice 0 dans `b` par l'expression `e2`. La fonction de substitution `subst` prend également un quatrième argument qui est l'arité de l'argument à substituer : celle-ci est nécessaire pour effectuer l'instanciation sémantique. Dans le cas d'un argument de fonction, l'arité est forcément de 0 puisque seul le **`let`** permet de généraliser. Les deux branches suivantes implémentent la sémantique de l'application. La beta-réduction du **`let`** nécessite seulement que l'expression qu'il lie soit une valeur. Il faut ensuite substituer cette valeur à la variable dans le corps de l'expression principale. Cette fois-ci, le dernier argument est l'arité du schéma de type associé à la variable définie par **`let`**. Finalement, si l'expression est une valeur elle est retournée, et si aucun des cas n'est possible l'évaluateur retourne une erreur. La fonction `reduce` appelle la fonction d'évaluation pas-à-pas jusqu'à obtenir une valeur ou un terme bloqué (elle ne considère donc pas les termes qui se réduiraient indéfiniment).

La substitution et l'instanciation sémantique sont données en figure 2.28, pour référence.

2.6 État des lieux de la vérification de types de TAST

Ce chapitre présente un cadre de travail pour la vérification de types de TASTs, autrement dit d'arbres de syntaxe annotés contenant les informations issues de l'inférence. Le principe de cette vérification est de s'assurer de la cohérence des annotations en les traitant comme des preuves de typage. Écrire un ensemble de règles de vérification de la cohérence d'une preuve de typage peut s'interpréter comme définir une spécification du système de types ayant servi à produire un

```
type error += Stuck of expr

let is_value e =
  match e with
    Var _, t | Cst _, t | Lam (_, _), t -> true
  | _ -> false

let rec step e =
  match e with
  | App ((Lam (tx, b), tl), e2), t when is_value e2 ->
    Ok (subst b 0 e2 0)
  | App (e1, e2), t when is_value e1 ->
    big_step e2 >>= fun e2' ->
    Ok (App (e1, e2'), t)
  | App (e1, e2), t ->
    big_step e1 >>= fun e1' ->
    Ok (App (e1', e2), t)

  | Let (Forall (ar, _), e1, e2), t when is_value e1 ->
    Ok (subst e2 0 e1 ar)
  | Let (sch, e1, e2), t ->
    big_step e1 >>= fun e1' ->
    Ok (Let (sch, e1', e2), t)

  | e when is_value e -> Ok e

  | e -> Error (Stuck e)

let rec reduce e =
  match step e with
  | Ok e when is_value e -> Ok e
  | Ok e -> reduce e
  | Error e -> Error e
```

FIGURE 2.27 – Évaluateur de MiniML annoté, en OCaml

```
(** Substitution of types *)
  let rec apply_sub sub ty =
  match ty with
    TVar n -> if in_dom_sub n sub then get_sub n sub else TVar n
  | TArrow (t1, t2) ->
    TArrow (apply_sub sub t1, apply_sub sub t2)
  | TInt -> TInt

let update_sub_for_sch sub sch =
  match sch with
    Forall (ar, t) -> List.map (fun (n, ty) -> n+ar, ty) sub

let apply_sub_sch sub sch =
  match sch with
    Forall (ar, t) -> Forall (ar, apply_sub (update_sub_for_sch sub
        sch) t)

(** Semantic instantiation *)

let rec seminst sub t1 t2 =
  match t1, t2 with
  | t1, TVar n when not (in_dom_sub n sub) -> (add_sub n t1 sub)
  | t1, TVar n -> sub
  | TArrow (t11, t12), TArrow (t21, t22) ->
    let sub = seminst sub t11 t21 in
    seminst sub t12 t22
  | TInt, TInt -> sub
  | t1, t2 -> sub

(** Substitution with instantiation *)

let rec subst_types sub e =
  match e with
    Var n, t -> Var n, apply_sub sub t
  | Cst c, t -> Cst c, apply_sub sub t
  | Lam (tx, b), t ->
    Lam (apply_sub sub tx, subst_types sub b), apply_sub sub t
  | App (e1, e2), t ->
    App (subst_types sub e1, subst_types sub e2), apply_sub sub t
  | Let (sch, e1, e2), t ->
    Let (apply_sub_sch sub sch,
         subst_types (update_sub_for_sch sub sch) e1,
         subst_types sub e2),
    apply_sub sub t

let rec subst e i e' ar =
  match e with
    Var n, t when n = i ->
    let e', t' = e' in
    let sub = seminst (sub_init (Forall (ar, t'))) t t' in
    subst_types sub (e', t')
  | Var n, t -> Var n, t
  | Cst c, t -> Cst c, t
  | Lam (tx, b), t ->
    Lam (tx, subst b (i+1) e' ar), t
  | Let (sch, e1, e2), t ->
    Let (sch, subst e1 i e' ar, subst e2 (i+1) e' ar), t
  | App (e1, e2), t ->
    App (subst e1 i e' ar, subst e2 i e' ar), t
```

FIGURE 2.28 – Implémentation de la substitution et de l'instanciation sémantique en OCaml

tel TAST. MiniML est un langage simple, contenant peu de constructions mais néanmoins assez expressif pour exprimer un large panel de programmes. Surtout, il s'agit d'un langage qui a été fortement étudié et dont les propriétés sont connues et admises.

Une manière classique de faire de l'inférence est d'utiliser de l'unification : ses propriétés sont connues et son implémentation se veut efficace. Dans cette formalisation, on s'attarde à extraires ces propriétés et les définir de manière inductive. Cela permet en outre de rendre beaucoup moins implicite les égalités de types produites durant le typage. De cette maniière il est possible de définir un ensemble d'opérations rendant ainsi la vérification plus claire au détriment de la concision induite par l'unification. Les propriétés vérifiées par l'unification incluent l'équivalence de types, l'instanciation ou encore la bonne formation. En particulier, l'instanciation nécessite de vérifier que toute monomorphisation est correcte en reconsituant un environnement dit de substitution, puisque l'absence de véritable application de types comme dans un langage avec polymorphisme explicite *à la* Système F pourrait entraîner une monomorphisation de variables de types pourtant libres dans le schéma de types donné, ce qui serait bien entendu une erreur dans l'implémentation de l'inférence de types. Cette propriété d'instantiation est accompagné d'un ensemble de preuves qui permettent de vérifier les propriétés vérifiées par celle-ci, et notamment que l'application de la substitution générée par le calcul d'instance sur le schéma de type retourne effectivement un type équivalent à son type monomorphisé.

A cela on définit une sémantique opérationnelle de MiniML qui s'avère être classique : il s'agit une sémantique avec substitution des variables liées et dont l'évaluation se fait à petits pas. La subtilité de cette sémantique vient de la présence des annotations de types et en particulier la nécessité d'évaluer ceux-ci lors de l'utilisation de valeurs polymorphes dans un contexte monomorphe : il s'agit donc ici du cas de l'instanciation. Pour cela, on définit une notion d'instanciation sémantique, une opération similaire à l'instanciation sans les étapes des vérifications d'égalités de types. Cette instanciation permet donc de calculer une substitution qui servira à monomorphiser tous les noeuds annotés de la valeur, ce qui est nécessaire pour la vérification de types. Cette substitution sera équivalente à celle calculée par l'instanciation si le terme en question est correctement annoté.

L'un des buts derrière cette vérification est de dériver un système de types exécutable, en d'autres termes une spécification aussi proche que possible d'une véritable implémentation. Au final, celle qui est présentée en OCaml au cours de ce chapitre montre que la différence entre la spécification et l'implémentation tient surtout aux structures de données choisies pour représenter les termes, types et environnements. Cette proximité entre la spécification et l'implémentation permet alors de transposer ce formalisme dans un assistant de preuve tel que Coq et de prouver facilement la correction entre la spécification du système de types et le vérificateur permettant de s'assurer du respect de celui-ci. L'annexe A présente notamment une implémentation d'un MiniML comprenant une forme appauvrie de filtrage de motifs. Cette formalisation est accompagnée d'une preuve que l'implémentation est correcte vis-à-vis de la spécification du système de types du langage. Il serait bien entendu intéressant de faire une preuve de sûreté, qui est laissé pour un travail futur. L'instanciation sémantique telle qu'elle est présentée ici est une première étape pour prouver la propriété de préservation du typage.

La partie suivante de ce manuscrit s'attaque à la vérification de types d'un langage plus contemporain : OCaml. Il s'agira alors de vérifier le TAST de ce langage bien plus complexe que MiniML : filtrage, mutabilité, types algébriques gardés. La vérification de cohérence est alors plus difficile, notamment puisque le Typedtree n'est pas une structure de donnée idéale pour effectuer celle-ci. En particulier, certaines informations ne sont présentes que partiellement et doivent être retrouvées “à la main” (on pensera à la quantification des schémas de types qui n'existe pas dans l'algèbre de types interne d'OCaml).

Chapitre 3

Définition du Typedtree, l'arbre annoté d'OCaml

Ce chapitre présente en détail le Typedtree d'OCaml, autrement dit la structure de données issue de l'inférence de types et prenant la forme d'un arbre de syntaxe abstrait dont tous les nœuds sont annotés. Cette présentation introduit également succintement l'unification utilisée durant l'inférence ainsi que le mécanisme de niveaux, principalement conçu pour la généralisation des variables de types, mais également fortement utile pour s'assurer du non-échappement des types existentiels.

Il est important de noter, à la fois dans ce chapitre mais dans les suivants, que le langage traité n'est pas OCaml mais bien le Typedtree représenté avec la syntaxe d'OCaml. Ainsi, certaines constructions peuvent être étonnantes, comme l'apparition des types dont la quantification est explicite, qui n'existent réellement dans l'algèbre de types que dans la représentation interne du compilateur. Ainsi, il est important par la suite de lire les expressions à travers le prisme du compilateur et non du langage. La version du compilateur traitée est la version 4.02.3, version courante au moment de débuter ce travail. Les versions suivantes ne modifieront que peu le langage et le système de types.

3.1 Définitions internes du compilateur

Un programme OCaml jusqu'à sa transformation en un exécutable (ou en "bytecode") passe par un ensemble de transformations en une succession de langages intermédiaires. Le compilateur effectue d'abord une analyse syntaxique du code, dont le résultat sera un arbre de syntaxe abstraite qui représente ce programme. Ce premier langage intermédiaire est nommé Parsetree. Après cette analyse, le compilateur effectue l'inférence de types du programme, et durant celle-ci vérifie que les types inférés ne sont pas incompatibles. Cette phase de la compilation engendre le Typedtree, une version du Parsetree dans lequel chaque nœud du programme contient des informations issues du typage : le type inféré accompagné de l'environnement qui a permis au typeur de déduire celui-ci. Finalement, le Typedtree est transformé en un langage intermédiaire plus bas-niveau et non typé, Lambda, une forme de lambda calcul capable de manipuler des blocs étiquetés d'une

$\tau ::= {}'a$	*free type variable*
$\mid (l : \tau) \rightarrow \tau$	*function with labeled argument*
$\mid \tau_1 * .. * \tau_n$	*tuple*
$\mid (\overline{\tau})\ \mathtt{t}$	*type constructor*
$\mid [(\rho)\ K_1\ \mathtt{of}\ \tau_1\ ; ..\ ; K_n\ \mathtt{of}\ \tau_n > K_i .. K_j\ ]$	*polymorphic variant*
$\mid \alpha$	*universally quantified variable*
$\mid \forall\overline{\alpha}.\tau$	*universally quantified type*
$\mid (\ \mathtt{module}\ \mathtt{P}\ \mathtt{with}\ \overline{\mathtt{t} = \tau})$	*first-class module pack*
$\rho ::= {}'a \mid \epsilon$	*row variable*

FIGURE 3.1 – Algèbre de types

constante. La suite de la chaîne de compilation n'est finalement qu'une suite de transformations et "d'appauvrissement" de ce Lambda vers une forme de plus bas niveau jusqu'à un assembleur donné. Dans le cas d'une compilation non native, le lambda est transformé vers un code objet pour une machine virtuelle dérivée de ZINC.

Dans notre cas, il s'agit d'étudier le résultat de l'inférence de types, le Typedtree, et d'en vérifier la cohérence. En effet, le Typedtree correspond bien à la définition d'un TAST comme décrit précédemment. Ce travail résulte en la formalisation du système de type d'OCaml, sous une forme exécutable et sans unification implicite.

Outre le cœur du langage ML, OCaml possède d'autres constructions de haut niveau. Il peut être divisé en 2 niveaux distincts : celui des expressions, donc ML et ses extensions, et celui des modules, lui-même possédant son propre système de types.

Les expressions comportent plusieurs constructions qui étendent le système de types de ML et le rendent ainsi plus expressif. On trouve notamment les produits qui caractérisent les tuples, les constructeurs de types qui servent à la fois d'abréviations et de déclarations pour les types d'enregistrement ou de types algébriques. À ces derniers s'ajoutent leur équivalent dans l'algèbre de types : les variants polymorphes, sortes de types algébriques munis d'une forme de sous-typage, et les enregistrements polymorphes qui constituent le système d'objets d'OCaml. A ces différents types s'associent une ou plusieurs constructions du langage.

Identifiants Tous les identifiants du Typedtree sont annotés avec un numéro unique, qui est attribué lors de l'inférence. Ce numéro est le résultat de l'alpha-conversion : deux identifiants avec le même numéro correspondent donc au même lieur. Cela permet de différencier deux identifiants du même nom.

3.2 Algèbre de types

L'algèbre de types tel qu'elle est utilisée par le compilateur est donné en figure 3.1.

A chacun de ces différents types est associée une annotation de niveau, générée durant la phase d'inférence. Dans notre cas, le niveau n'aura d'intérêt que sur les variables de types, comme expliqué section 3.6.1.

3.3 Typedtree : expressions

L'ensemble des expressions du langage est donné en figure 3.2.

Puisqu'il s'agit d'un arbre de syntaxe abstraite totalement annoté, chaque nœud est donc bien une expression avec un type. L'environnement utilisé pendant l'inférence, bien que présent dans le Typedtree, n'est pas gardé ici car celui-ci ne sera pas utilisé pendant l'étape de vérification.

3.4 Typedtree : modules, structures et signatures

Le langage est stratifié selon deux niveaux : ML, les expressions, et le langage de modules, *a priori* distinct de celui-ci et permettant d'exprimer l'architecture du programme et de manière générique et composable. On remarquera d'ailleurs que les modules peuvent apparaître comme le résultat d'expressions via l'introduction des modules de premier ordre, rendant la vérification des expressions et celle des modules interdépendantes. La syntaxe du langage de module est donnée en figure 3.3.

Le système de modules peut être découpé en deux couches distinctes : les modules et les structures. On ne détaillera pas ici les subtilités du système de types des modules, leur vérification n'étant pas formalisée dans ce manuscrit. En revanche, ils sont implémentés dans le vérificateur.

Les modules, dans leur syntaxe et leur sémantique, ressemblent à un lambda calcul classique dont les structures seraient des enregistrements.

— `P`, représentant le chemin d'un module dans l'environnement (ou la portée) courant(e).
— La construction `functor (P : Mty) -> M` est l'équivalent de l'abstraction.
— Enfin, l'application de foncteur `M1(M2)` est équivalent à l'application du lambda-calcul.

A celles-ci s'ajoutent alors deux constructions :

— `struct .. end`, dont le corps permet de lier un ensemble d'identifiants du langage *noyau*, ou d'autres modules,
— et `(val m : Mty)`, qui permet de dépaqueter un module empaqueté dans une variable du langage bas-niveau.

La structure est une suite de liaisons d'identifiants, qu'il s'agisse :

— de valeurs du langage noyau : `let x = e`,
— de déclarations de types, qui peuvent être
    — des abréviations : `type t = int`,
    — des types algébriques : `type 'a option = Some of 'a | None`
    — des enregistrements : `type 'a t = { f : 'a -> int) }`
— d'ajout de constructeurs de données à un type extensible, tel que le type des *exceptions* : `type exn += Not_found`
— fonctions externes, par exemple de fonctions C : `external magic : 'a → 'b = "caml_eventually_destroy_everything"`.
— de modules : `module S = String`
— de types de modules : `module type M = Comparable`

A cela, on ajoute deux constructions supplémentaires :

— `open String` qui ajoute à l'environnement courant toutes les valeurs du module donné, permettant ainsi d'y accéder sans préfixer les valeurs par son chemin.

$e ::= \langle e' : \tau \rangle$	*expression node*
$e' ::=$	*expression descriptions*
$x \mid c$	
$\mid$ **function** $\overline{p\ ?\textbf{when}\ c \to e} \mid e_1\ e_2$	
$\mid$ **let** $\overline{p = e}$ **in** $e \mid$ **let rec** $\overline{p = e}$ **in** e	*ML classical constructions*
$\mid$ **function** $\sim l : p\ ?\textbf{when}\ c \to e$	
$\mid e_1\ (\sim l : e_2)$	*labeled functions arguments*
$\mid$ **match** e **with** $\overline{p\ ?\textbf{when}\ c \to e}$	*pattern matching*
$\mid$ **try** e **with** $\overline{p\ ?\textbf{when}\ c \to e}$	*exceptions catching*
$\mid e_1, .., e_n \mid K\ (\overline{e})$	*tuples and variant constructors*
$\mid$ \`$K\ e$	*polymorphic variant constructors*
$\mid \{\overline{l = e;}\} \mid e.l \mid e_1.l \leftarrow e_2$	*records*
$\mid e_1.(e_2) \mid e_1.(e_2) \leftarrow e_3$	*arrays*
$\mid$ **if** e_1 **then** e_2 **else** e_3	*conditional*
$\mid e_1$ **;** e_2	*sequence*
$\mid$ **for** $x = e_1$ **to/downto** e_2 **do** e_3 **done**	
$\mid$ **while** e_1 **do** e_2 **done**	*imperative loops*
$\mid$ **let module** $I = M$ **in** $e \mid$ (**module** M)	*first-class modules*
$\mid$ **lazy** e	*lazyness*
$\mid$ **assert** e	*assertions*
$\mid (e : t_1 :> t_2) \mid (e : t)$	*coercions and constraints*
$\mid$ **fun** (**type** a) $\to e$	*local abstract type introduction*
$\mid$ **let open** P **in** e	*local module opening*
$p ::= \langle p' : \tau \rangle$	*pattern node*
$p' ::=$	*pattern description*
$x \mid c \mid p$ **as** x	*variable, constant and aliases pattern*
$\mid (p_1, .., p_n)$	
$\mid K\ (\overline{p})$	
$\mid$ \`$K\ (\overline{p})$	*tuple, variant and polymorphic variant constructor pattern*
$\mid \{\ \overline{l = e;}\ \} \mid [\ \overline{p}\ ]$	*record and array pattern*
$\mid (\ p_1$ **|** $p_2\ )$	*or pattern*
$\mid$ **lazy** p	*lazy pattern*
$\mid (p : t)$	*coercions and constraints*

FIGURE 3.2 – Syntaxe des expressions

M : := ⟨ M' : $\mathcal{M}$ ⟩	*module node*
M' : :=	*module description*
P	*module path*
| ⟨ **struct** Items **end** : $\mathcal{S}$ ⟩	*structure*
| **functor** (I : M_{ty}) → M	*functor*
| M_1 (M_1)	*functor application*
| M : M_{ty}	*module constraint*
| (**val** e : M_{ty})	*first-class module unpacking*
Items : := I ; ; Items | Σ	
I : :=	*structure item*
let i = e	*value binding*
| **type** ($\alpha_1, .., \alpha_n$) i = T	*type declaration*
| **type** i += K **of** t	*type extension*
| **external** i : t = "prim_name"	*stubs import*
| **module** I = M	*module binding*
| **module type** I = M_{ty}	*module type abbreviation*
| **open** P	*environment extension*
| **include** M	*signature reexport*

FIGURE 3.3 – Syntaxe du système de modules

$\mathcal{M}$ ::=	*module type*
(**module** P)	*module alias*
| P	*module type abbreviation*
| $\mathcal{S}$	*signature*
| **functor** (I : $\mathcal{M}_1$) → $\mathcal{M}_2$	*functor type*

$\mathcal{S}$ ::= $\mathcal{S}_i$; ; $\mathcal{S}$ | Σ

$\mathcal{S}_i$ ::=	*signature item*
val i : τ	*value binding*
| **type** $(\alpha_1, .., \alpha_n)$ i = T	*type declaration*
| **type** i += K **of** τ	*type extension*
| **module** I : $\mathcal{M}$	*module binding*
| **module type** I = $\mathcal{M}$	*module type abbreviation*

FIGURE 3.4 – Algèbre des modules de types

— `include String` qui ajoute au module courant toutes les valeurs du module donné.

3.4.1 Signatures et types de modules

Tout comme ML, le système de modules et de structures possèdent un système de types. La figure 3.4 est une synthèse de son algèbre.

Ainsi, l'algèbre des types de modules comprend :

— L'alias de modules, généralement le type classifiant les expressions de la forme `module M = P`, où P est forcément un chemin de module qui n'est pas un argument de foncteur.
— L'abréviation de module de types, eux-même étant des déclarations de types de modules.
— La signature, soit le type catégorisant les structures
— Le type foncteur classifiant les foncteurs

L'application de foncteur, la contrainte de type et le dépaquetage de module de premier ordre n'ont pas de type les classifiant, celui-ci étant le résultat du “calcul” effectué.

Le type des signatures quand à lui est similaire à la forme de la structure, à quelques exceptions près. Les valeurs externes sont considérées comme des valeurs classiques. Le résultat de l'ouverture (**open**) d'un module n'apparaît pas dans la signature puisque son utilisation n'affecte que l'environnement. Le résultat de l'inclusion de modules est une signature, dont chacun des éléments est rajouté la signature courante.

3.4.2 Syntaxe des types de modules

De la même manière qu'il est possible d'écrire des types dans le langage, il existe une syntaxe concrète pour définir les types de modules.

M_{ty} : :=	*module type*
P	*module type abbreviation*
| S	*signature*
| **functor** $(I : M_{ty_1}) \to M_{ty_2}$	*functor type*
| M_{ty} **with type** $i1 := t$ **and** .. **type** $in = t'$	*module type substitution*
| **module type of** M	*type of module reification*

S : := S_i ; ; S \| Σ

S_i : :=	*signature item*
val i : t	*value binding*
| **type** $('a_1, .., 'a_n)$ i = T	*type declaration*
| **type** i += K **of** t	*type extension*
| **module** I : M	*module binding*
| **module type** I = M_{ty}	*module type abbreviation*
| **open** P	*environment extension*
| **include** M_{ty}	*signature reexport*

FIGURE 3.5 – Syntaxe des types de modules

En comparant la syntaxe des modules de types et signatures (figure 3.5) avec leur algèbre interne, on peut remarquer un certain nombre de différences. Les alias de modules n'ont plus de syntaxe, seule l'inférence peut explicitement en produire. Deux constructions apparaissent :
- La plus simple, **`module type of`**, retourne le type du module donné en paramètre.
- En revanche, `M` **`with type`** `t = u` permet de substituer, dans la signature de `M`, le type abrégé par `t` par le type `u`, et retourne donc la signature résultante. Le type `t` doit être une abréviation déclarée dans `M`. La version `M` **`with type`** `t := u` est dite *destructive*, où toutes les occurrences de `t` sont remplacées et la déclaration de l'abréviation est retirée de la signature.

3.5 Environnements

L'environnement d'OCaml, que l'on nommera Γ, reste relativement simple : il associe des identifiants à un type, au sens général du terme. Cet environnement se découpe en sous-environnements, chacun classifiant une certaine catégorie du langage. On accèdera à ces sous-environnements via une projection sur Γ :
- *Values*, soit les valeurs ;
- *Types*, soit les déclarations de types, *i.e.* les déclarations de types algébriques, d'enregistrements ou d'abréviations ;
- *Class*, soit les déclarations de classes ;
- *Mods*, soit les identifiants associés à des modules ;

$$\text{VAR}\ \frac{\Gamma.Values(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{ABS}\ \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \textbf{fun}\ x \to e\ : \tau_1 \to \tau_2}$$

$$\text{APP}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

FIGURE 3.6 – Règles d'inférence de ML

— *Modtypes*, soit les identifiants associés à des types de modules ;

Ainsi, si v est un identifiant correspondant à une variable, alors $\Gamma.Values(v)$ est son type dans l'environnement, s'il existe. L'opération $dom(\Gamma.Values)$ retourne l'ensemble des identifiants auxquels sont associés un type dans cet environnement (de façon similaire pour les autres sous-environnements). Par la suite, on introduira un ensemble d'opérations permettant de manipuler cet environnement.

3.6 Moteur d'inférence de types

L'inférence de types d'OCaml est effectuée par unification en place des types : autrement dit, elle est destructive. Prenons l'exemple suivant :

```
fun x → incr x
```

avec les règles classiques de ML en figure 3.6

Ici, la fonction `incr` a le type `int → int`. Si on déroule l'algorithme d'inférence tel qu'il est implémenté dans OCaml :

1. L'expression étant une fonction, on lui attribue le type d'une fonction, dont le domaine et le codomaine sont deux nouvelles variables fraîches, soit $'\_a \to\ '\_b$.

$$\Gamma \vdash \textbf{fun}\ x \to incr\ x\ :\ '\_a \to\ '\_b\ \text{ABS}$$

2. On type ensuite `incr x` avec le type $'\_b$ dans l'environnement courant auquel on a associé `x` à $'\_a$.

$$\frac{\Gamma, (x,\ '\_a) \vdash incr\ x :\ '\_b}{\Gamma \vdash \textbf{fun}\ x \to incr\ x\ :\ '\_a \to\ '\_b}\ \text{ABS}$$

3. `incr` attend pour argument une expression de type `int`. Selon la règle `App`, il s'agit alors d'unifier $'\_a$ à `int`, ce qui aura pour conséquence de remplacer toutes les occurrences de $'\_a$ par `int`.

$$\dfrac{\dfrac{\dfrac{\Gamma(incr) = \texttt{int} \to \texttt{int}}{\Gamma \vdash incr : \texttt{int} \to \texttt{int}}\ \text{Var} \qquad \dfrac{\Gamma(x) = \texttt{int}}{\Gamma, (x,\ \texttt{int}) \vdash x\ : \texttt{int}}\ \text{Var}}{\Gamma, (x,\ \texttt{int}) \vdash incr\ x :\ '\_b}\ \text{App}}{\Gamma \vdash \textbf{fun}\ x \to incr\ x\ :\ \texttt{int} \to\ '\_b}\ \text{Abs}$$

4. Le résultat de l'application étant un entier, on unifie $'\_b$ avec `int`, remplaçant alors toute occurrence de cette variable par `int`.

$$\dfrac{\dfrac{\dfrac{\Gamma(incr) = \texttt{int} \to \texttt{int}}{\Gamma \vdash incr : \texttt{int} \to \texttt{int}}\ \text{Var} \qquad \dfrac{\Gamma(x) = \texttt{int}}{\Gamma, (x,\ \texttt{int}) \vdash x\ : \texttt{int}}\ \text{Var}}{\Gamma, (x,\ \texttt{int}) \vdash incr\ x :\ \texttt{int}}\ \text{App}}{\Gamma \vdash \textbf{fun}\ x \to incr\ x\ :\ \texttt{int} \to\ \texttt{int}}\ \text{Abs}$$

L'unification se fait *en place*, autrement dit la variable de type est physiquement remplacée par la représentation d'un autre type. De même, la généralisation des variables se fera en place. Ces effets de bord étant essentiels pour assurer de bonnes performances à la compilation, ils rendent en revanche difficile l'utilisation de cette implémentation en dehors du cadre de l'inférence de types.

3.6.1 Niveaux

Une autre spécificité du typeur d'OCaml est son utilisation des niveaux pendant l'inférence : pour des questions de performance, la généralisation peut être coûteuse. Pour qu'une variable de type puisse être généralisée selon la restriction de valeurs, elle ne doit pas apparaitre dans l'environnement. La solution la plus naïve serait donc vérifier qu'elle n'est référencée dans aucun type de l'environnement, ou plus efficacement de garder dans l'environnement l'ensemble des variables de types libres, ce qui induit néanmoins d'allouer de la mémoire pour stocker cet ensemble. Le généralisation ainsi que la restriction souple de valeurs sont détaillées en sous-section 4.3.2.

La solution proposée par D.Rémy [59] et implémentée dans OCaml [37] est de s'inspirer du principe des régions mémoire. Ainsi, le programme est divisé en *niveaux*. Chaque **let** possède son propre niveau. Par exemple, supposons le squelette suivant :

```
let f x = ..            (1)
  let g y = ..          (2)
  in
  ..
in
...
```

Le premier **let** est créé au niveau 1, par conséquent toutes les variables créées dans son corps auront au moins ce niveau. Le second est créé au niveau 2, de même toutes les variables fraîches créées dans son corps auront au moins ce niveau. L'unification d'une variable de niveau n avec une autre de niveau inférieur k fait "redescendre" le niveau de la première à k. Une fois le corps de `g` typé, toutes les variables de type qui sont restées au niveau 2 ou plus peuvent être généralisées. Si une variable a été créée ou "descendue" à un niveau inférieur, elle apparaît dans l'environnement de généralisation et ne peut donc pas être généralisée.

Cette utilisation des niveaux permet également de s'assurer du non échappement des constructeurs de types. Voyons un exemple simple lié à la restriction de valeurs.

```
type t1 = T1    (* niveau 1 *)

let xs = ref []  (* niveau 2 *)

type t2 = T2    (* niveau 3 *)
```

Selon la restriction de valeurs, `xs` devrait avoir le type `'_a list ref`, puisque le paramètre de type de `ref` est contravariant. Il n'a donc pas pu être généralisé, et ne pourra être unifié qu'avec un type qui ne contient pas de constructeurs de types créés à un niveau supérieur au sien.

```
xs := [ T1 ]
```

Ainsi, cette affectation est parfaitement légale puisque `t1` a été créé au niveau 1 et `xs` au niveau 2, `xs` est désormais fixé au type `t1 list ref`. En revanche :

```
xs := [ T2 ]
```

n'est pas possible, puisque le type `t2` a été créé au niveau 3. Ce mécanisme a également une importance pour vérifier le non échappement des types avec les modules locaux et les modules de premier ordre. Une telle vérification est faite à l'occasion de l'unification.

Ce chapitre détaille le Typedtree d'OCaml dans sa version 4.02.3. L'utilisateur averti du langage remarquera des différences notables avec le langage réel : certaines constructions ont disparu comme la gestion des tableaux, remplacés par des primitives directement dans le langage, là où d'autres sont apparues, comme les types quantifiés universellement comme type de premier ordre. L'utilisateur n'est en réalité pas capable de l'utiliser comme un type classique mais seulement dans des contextes bien précis, mais celui-ci fait partie de l'algèbre de types qui faut gérer pour vérifier le Typedtree. Ce chapitre dense en grammaire permet donc de présenter ces constructions particulières et propres au Typedtree, et est nécessaire au travail de vérification et de formalisation de son système de types effectué au chapitre suivant.

Chapitre 4

Implémentation d'un vérificateur de types pour OCaml

Toute cette machinerie de vérification d'un arbre de syntaxe abstraite annoté peut être mise en œuvre pour le compilateur OCaml, qui génère après sa phase d'inférence de types une telle structure de données. OCaml en tant que langage de programmation n'existe qu'au travers de l'implémentation de son compilateur : il n'existe pas de définition formelle du langage, ni d'un autre compilateur implémentant son système de types et compilant avec la même sémantique (on exclut bien évidemment les compilateurs dérivés directement d'OCaml, tel que MetaOCaml [38]). De ce fait OCaml désignera à la fois le langage, et l'implémentation de son compilateur.

Ce chapitre présente un ensemble de règles de vérification de cohérence du Typedtree (et non du langage OCamldirectement), en se concentrant uniquement sur les expressions du langage de base : le noyau fonctionnel, les extensions impératives, les extensions fonctionnelles dont les types algébriques gardés et les variants polymorphes. Autrement dit, les objets et les modules ne sont pas présentés dans la suite de cette thèse. En revanche, il est à noter que le programme issu de ces règles implémente une vérification des objets et modules, et permet donc de vérifier la quasi-intégralité du langage à l'exception des modules récursifs et des types equi-récursifs [1]. La langage ici présenté n'est donc pas OCaml, mais le Typedtree. Ainsi, certaines constructions n'existent plus comme par exemple les tableaux, puisqu'il s'agit d'un sucre syntaxique au dessus du module `Array` de la bibliothèque standard.

Ce chapitre suit la structure de celui pour la vérification de MiniML. On réintroduit les principes de vérification de l'algèbre de types, dont les propriétés de vérification de l'équivalence, du filtrage et de l'instanciation. A cela, il faut ajouter la vérification du non-échappement de types qui est possible en présence de la généralisation avec des effets de bords, ou à l'utilisation de types existentiels encodés dans des types algébriques gardés. De plus, pour être capable de vérifier ces types algébriques gardés, il est nécessaire d'introduire une notion d'équations locales de types, que l'on gèrera sous la forme d'un algorithme d'*union-find* légèrement modifié pour rejeter les classes d'équivalence qui pourraient être incohérentes. On définira ensuite la vérification de la généralisation suivant les conditions énoncées par la restriciton relâchée de valeur. Comme pour

1. ∧Les types dont la récursivité n'est pas gérée par un constructeur de donnée, comme par exemple `type t = int * t`

MiniML, on s'attellera ensuite à la vérification du TypedTree lui-même, en partant d'abord du noyau fonctionnel, puis en y ajoutant une à une chacune des constructions du langage, en expliquant les spécificités de celles-ci. Finalement, on montrera quelques exemples de programmes acceptés par le typeur d'OCaml et dont il est possible donc de produire un Typedtree, et donc ce système de vérification est capable montrer l'incohérence.

Le vérificateur issu de ces règles de vérification est disponible à l'adresse : `github.com/OCamlPro-Couderc/ocp-typechecker`.

4.1 Environnements

Bien que l'environnement utilisé pendant l'inférence soit annoté dans chaque nœud du Typedtree, cet environnement n'est pas réutilisé durant la vérification. En effet, il s'agit ici de se concentrer sur la vérification d'un arbre annoté, tout en évitant d'utiliser un environnement potentiellement incorrect. Par exemple, il est possible que deux environnements dont l'un est censé être un extension de l'autre lient le même identifiant[2] à un type différent : cela serait considéré comme une incohérence. Faire une telle vérification serait extrêmement coûteuse : il s'agirait de vérifier l'intégralité de l'environnement à chaque nœud. Néanmoins, si celui-ci devait être vérifié, sa condition de bonne formation serait la suivante :

Bonne formation d'un environnement Deux conditions sont nécessaires pour considérer un environnement comme étant bien formé. Comme précisé en page 58, l'alpha-conversion attribue un numéro unique à tout identifiant lorsque celui-ci est lié. Ainsi, puisqu'il s'agit du résultat de l'alpha-conversion, chaque identifiant doit être unique dans l'environnement. Par exemple :

```
let a = .. (* id = 1 *)
let a = .. (* id = 2 *)
print_int a (* a : id = 2 *)
```

Cela permet entre autres de faire de l'analyse de code mort de manière plus fine qu'avec simplement un nom, et est également une l'un des invariants considéré pour l'allocation mémoire. La seconde condition découle de la vérification : supposons que l'environnement soit représenté par une liste d'identifiants comme celui utilisé dans le système de types de MiniML page 38. L'ordre des identifiants dans l'environnement correspond à l'ordre dans lequel les expressions ont été typées, auxquelles on ajoute maintenant les types. Prenons l'exemple suivant :

```
type u = ..
type t = K of u : t
```

Pour que le type `t` puisse être typé, le type `u` doit exister. Autrement dit, il doit exister un identifiant `u` dans l'environnement qui est lié à un type. Si `u` n'est pas présent dans l'environnement alors que `t` l'est, alors l'environnement est mal formé. Si `u` apparaît comme étant plus récent que `t`, l'environnement est mal formé : cette propriété découle de la précédente, puisque la propriété de bonne formation est construite par induction sur la forme de l'environnement.

Pour résumer, on considère qu'un environnement est bien formé, si :

2. ∧Le Typedtree est aussi le résultat de l'alpha-conversion des identifiants liés, il n'est donc pas possible de masquer un identifiant, seule possibilité pour qu'un identifiant ait un type différent dans deux environnements

— Pour tout identifiant, celui-ci n'est lié qu'une fois.
— Si un identifiant apparait dans un type, il doit déjà être lié dans l'environnement.

Le premier cas est assez simple à vérifier : Ainsi, à chaque étape de vérification, il faudrait s'assurer que toutes les variables libres vues dans la dérivation courante sont toujours présentes, et qu'elles sont toujours liées au même type, soit une propriété équivalente à celle du prédicat de sous-ensemble des environnements pour MiniML présentée en figure 2.13 (page 39).

L'environnement utilisé durant la vérification est celui d'OCaml : celui-ci est représenté par un enregistrement dont chaque champ est un arbre équilibré permettant de trier les identifiants par nom et permet donc une recherche rapide dans celui-ci. Il n'existe aucune véritable notion d'ordre chronologique [3] pour l'ajout des nœuds : la structure d'arbre trié empêche ceci, et l'environnement ne garde pas trace de l'ordre des ajouts. La preuve de bonne formation ne peut donc pas être construite de manière inductive sur la forme de l'environnement. Le seul moyen de vérifier cette propriété de bonne formation serait donc de parcourir l'environnement entièrement, tout en naviguant entre ses différents champs en parallèle.

Utilisation des résumés L'environnement possède néanmoins une structure linéaire appelée `summary`, qui enregistre tous les ajouts sous forme de liste, et dont l'ordre est bien chronologique. Un tel résumé pourrait donc permettre ultérieurement de vérifier les ajouts et la bonne formation de celui-ci. Il faudrait donc être capable de comparer deux environnements recréés à partir de ces résumés, ce qui n'est pas trivial aujourd'hui. En effet, la structure de l'environnement n'est pas comparable structurellement : les identifiants sont accompagnés d'une fonction appelée dès lors qu'une valeur est accédée. Celle-ci permet entre autres de noter dans une table donnée que l'identifiant a été accédé au moins une fois, utilisée pour certains avertissements du compilateur.

4.2 Comparaison et vérification de types

Comme décrit dans le chapitre précédent (page 64), l'unification joue un rôle prépondérant dans l'inférence (et en même temps la vérification) de types d'OCaml. On peut la décomposer en plusieurs opérations :
— Le test d'équivalence de types
— Le test d'instanciation, ou de sous-typage
— La vérification de non-échappement des constructeurs de types ou de types existentiels
— Le filtrage de types
— La génération d'équations pour les GADTs

Dans le but de vérifier un arbre de syntaxe totalement annoté après inférence, il pourrait être tentant de réutiliser le mécanisme mis en œuvre par celle-ci et se concentrer sur la vérification elle-même. Néanmoins, et comme expliqué précédemment, l'unification utilise des effets de bord de manière systématique par souci d'efficacité. L'idée ici est donc de remplacer l'unification par différentes opérations, permettant ainsi de formaliser cet aspect du système de types.

On introduit la métavariable Φ qui représente un ensemble de classes d'équivalence liées aux GADTS (on parlera aussi d'équations par la suite), que l'on détaillera en sous-section 4.2.5. Leur

3. ∧Les identifiants sont marqués par un "temps" de création, mais celui-ci peut être remis à zéro ou modifié durant l'exécution du typeur.

utilisation pour le moment n'a pas d'incidence sur la compréhension des différents algorithmes de vérification liés aux types.

4.2.1 Équivalence de types

L'équivalence de types (figure 4.1) permet de tester si deux types sont identiques, tout en expansant les abréviations et les éventuelles équations liées à l'utilisation de GADTs. L'équivalence compare deux à deux les types, et parcourt ceux-ci de manière infixe. Le jugement d'équivalence est :

$$\Gamma, \Phi \vdash_{\tau} \tau \equiv \tau'$$

L'équivalence est réflexive, symétrique et transitive.

Ainsi, la condition d'équivalence de types est relativement simple puisqu'elle s'assure de l'égalité stricte entre deux types. Une variable n'est équivalente qu'à elle-même, deux types fonctionnels sont équivalents si leur domaine et codomaine sont équivalents, etc. Les cas les moins évidents sont ceux faisant intervenir des constructeurs de types : un type est équivalent à un constructeur si son expansion est équivalente à celui-ci (cas de la règle EQUIV-CONSTRUCT-EXP). Le second cas est celui lié aux GADTs (règle EQUIV-RIGID) : ceux-ci engendrent des équivalences de types entre des constructeurs dits *rigides* et des types, et seront détaillés par la suite. On peut également noter la règle de vérification des types de modules de première classe (EQUIV-PACKAGE). La fonction `Sig` génère le type de signature demandé (pour rappel, la construction permettant de substituer un constructeur de type dans une signature calcule une nouvelle signature), puis les deux signatures sont comparées de la même manière que les types. Dans le cas où le type du module de première classe est celui d'un foncteur, la fonction `Sig` échoue s'il y a au moins une substitution (qu'elle soit destructive ou non).

4.2.2 Instanciation de types

Le mécanisme d'instanciation est essentiel pour la vérification : il permet de s'assurer que l'utilisation de variables dont le type est polymorphe est correcte. Le jugement d'instanciation est :

$$\Gamma, \Phi, \theta \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta'$$

Autrement dit, dans un environnement donné (Γ), un ensemble d'équations (Φ) et une substitution (θ), le type τ est une instance de τ', et ce calcul d'instance retourne une nouvelle substitution (θ').

Substitution On définit une substitution θ comme une fonction finie qui associe une variable de types à un type. On suppose que $\theta(\tau)$, où τ est un type, est l'application d'une substitution à ce type. Cette application a pour résultat une instance de τ, où chacune de ses variables de type présente dans le domaine de θ a été remplacée par le type qui lui était associée.

Par conséquent, un type donné τ est une instance d'un type τ' s'il existe une substitution θ telle que pour chaque variable de τ', on peut lui associer un type de τ, et donc que $\theta(\tau') = \tau$.

$$\textsc{Equiv-Var}\ \Gamma, \Phi \vdash_{\tau} {'a} \equiv {'a}$$

$$\textsc{Equiv-Fun}\ \frac{l_1 = l'_1 \qquad \Gamma, \Phi \vdash_{\tau} \tau_1 \equiv \tau'_1 \qquad \Gamma, \Phi \vdash_{\tau} \tau_2 \equiv \tau'_2}{\Gamma, \Phi, \theta \vdash_{\tau} (l_1 : \tau_1) \to \tau_2 \equiv (l'_1 : \tau'_1) \to \tau'_2}$$

$$\textsc{Equiv-Tuple}\ \frac{\forall i_{>1}.\ \Gamma, \Phi \vdash_{\tau} \tau_i \equiv \tau'_i}{\Gamma, \Phi \vdash_{\tau} \tau_1 * .. * \tau_n \equiv \tau'_1 * .. * \tau'_n}$$

$$\textsc{Equiv-Construct}\ \frac{\forall i_{>1}.\ \Gamma, \Phi \vdash_{\tau} \tau_i \equiv \tau'_i}{\Gamma, \Phi \vdash_{\tau} (\overline{\tau})\ \mathtt{p} \equiv (\overline{\tau'})\ \mathtt{p}}$$

$$\textsc{Equiv-Construct-Exp}\ \frac{let\ \tau = expand(\Gamma, \Phi, \mathtt{t}, \overline{\tau}) \qquad \Gamma, \Phi \vdash_{\tau} \tau \equiv \tau'}{\Gamma, \Phi \vdash_{\tau} (\overline{\tau})\ \mathtt{t} \equiv \tau'}$$

$$\textsc{Equiv-Poly}\ \frac{\Gamma, \Phi \vdash_{\tau} \tau \equiv \tau'}{\Gamma, \Phi \vdash_{\tau} \forall \overline{\alpha}.\ \tau \equiv \forall \overline{\alpha'}.\ \tau'}$$

$$\textsc{Equiv-Univar}\ \Gamma, \Phi \vdash_{\tau} \alpha \equiv \alpha'$$

$$\textsc{Equiv-Rigid}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi \vdash_{\tau} \mathtt{t} \equiv \tau}$$

$$\textsc{Equiv-Variant}\ \frac{\begin{array}{c} T_{pres_i} .. T_{pres_j} = T'_{pres_i} .. T'_{pres_j} \\ \Gamma, \Phi \vdash_{\tau} \rho \equiv \rho' \qquad \overline{T} \in \overline{T'} \qquad \forall i_{>1}. \Gamma, \Phi \vdash_{\tau} \tau_{var_i} \equiv \tau'_{var_i} \end{array}}{\Gamma, \Phi \vdash_{\tau} [(\rho)\ \overline{T\ of\ \tau_{var}} > T_{pres_i} .. T_{pres_j}] \equiv [(\rho')\ \overline{T'\ of\ \tau'_{var}} > T'_{pres_i} .. T'_{pres_j}]}$$

$$\textsc{Equiv-Nil}\ \Gamma, \Phi \vdash_{\tau} \epsilon \equiv \epsilon$$

$$\textsc{Equiv-Package}\ \frac{\begin{array}{c} let\ S\ = \mathtt{Sig}(\Gamma, \mathtt{P}\ \overline{\mathbf{with}\ \mathbf{type}\ \mathtt{t} = \tau}) \\ let\ S' = \mathtt{Sig}(\Gamma, \mathtt{P'}\ \overline{\mathbf{with}\ \mathbf{type}\ \mathtt{t'} = \tau'}) \qquad \Gamma, \Phi \vdash_{\tau} S' \equiv S \end{array}}{\Gamma, \Phi \vdash_{\tau} (\mathbf{module}\ \mathtt{P}\ \overline{\mathbf{with}\ \mathbf{type}\ \mathtt{t} = \tau}) \equiv (\mathbf{module}\ \mathtt{P'}\ \overline{\mathbf{with}\ \mathbf{type}\ \mathtt{t'} = \tau'})}$$

FIGURE 4.1 – Règles d'équivalence de types

L'équivalence est réflexive, symétrique et transitive. Celle-ci permet de vérifier que dans un même contexte deux types peuvent être vus comme équivalents, en effectuant d'éventuelles expansions de types.

Le calcul d'instance se fait par comparaison structurelle sur les deux types. Le cas d'instance de variable est le plus important :

— Soit la variable n'est pas liée dans la substitution en cours de construction (cas INST-VAR-UNBOUND). Dans ce cas, le type avec lequel on cherche à l'instancier est *a priori* valide.
— Soit la variable est liée dans la substitution (INST-VAR-BOUND). Pour que cette instanciation soit valide, il faut que le type déjà lié à cette variable soit équivalent à l'instance testée.

Le calcul d'instance génère donc une nouvelle substitution. En revanche, la substitution initiale est importante. Une substitution vide initialement n'est pas satisfaisante. Prenons par exemple le TAST incorrect suivant (seules les annotations utiles sont gardées) :

```
let ⟨ f ⟨ x : 'a ⟩ : 'a -> 'a ⟩ =
  ignore (⟨ x : int ⟩ + 2);
  ⟨ x : 'a ⟩
```

On peut voir que la fonction `f` prend un argument `x` de type `'a` et retourne une valeur de type `'a`. Dans le corps de la fonction, `x` est utilisé avec le type `int`. Si on vérifie maintenant la variable `x` annotée `int` :

$$\textsc{Var}\ \frac{let\,'a = (\Gamma \oplus_{\mathcal{V}} (x,\ 'a)).Values(x) \qquad \Gamma \oplus (x,\ 'a), \Phi, \emptyset \vdash_{\tau} \mathtt{int} \leq\ 'a \Rightarrow [`a \mapsto \mathtt{int}]}{\Gamma \oplus_{\mathcal{V}} (x,\ 'a), \Phi \vdash \langle x : \mathtt{int} \rangle}$$

La variable est à priori correctement annotée, puisque `int` est une instance de `'a`. Prenons ensuite l'exemple suivant, avec ce même `f` :

```
⟨ f : string → string ⟩ ⟨ "Oups" : string ⟩
```

Suivant la sémantique opérationnelle définie plus loin chapitre 5 (page 107), la réduction atteindrait le stade bloquant suivant :

```
ignore (⟨ "Oups" : int ⟩ + 2);
⟨ "Oups" : string ⟩
```

En effet, bien que `int` soit un instance correcte de `'a`, on ne peut la considérer comme sûre, puisqu'à l'intérieur de la fonction ce même `'a` a été généralisé et donc ne peut être instancié qu'avec lui-même.

Comme indiqué plus tôt, ce problème est lié au choix d'une substitution initiale correcte pour le calcul d'instance. La solution est en réalité simple : elle consiste à associer initialement toute variable généralisée à elle-même, ou plus précisément toute variable généralisée étant déjà apparue durant la vérification. En particulier, lors du typage du **`let`** (et **`let rec`**), il est nécessaire d'enregistrer les variables apparaissant généralisées dans le type annoté de l'identifiant avant de vérifier le corps lui-même. Cela justifie l'introduction dans l'environnement d'un ensemble de variables non instanciables, autrement dit des variables qui, au moment de la vérification, ont été généralisées et quantifiées en amont de l'expression courante, auquel on accèdera via une projection de *GenVars* sur Γ. L'utilisation de cet ensemble de variables est nécessaire puisqu'il n'existe pas explicitement de quantification des variables de types : si pour chaque identifiant présent dans l'environnement sa quantification était disponible, alors le calcul d'instance pourrait se faire en partant d'une substitution vide.

Ainsi, on définit la substitution initiale :

$$\theta_{\forall}(\Gamma) = \{\beta \mapsto \beta \mid \beta \in \Gamma.GenVars\}$$

Autrement dit, la substitution initiale associe à elle-même toute variable généralisées dans l'environnement [4].

Finalement, on peut définir l'ensemble des règles du calcul d'instanciation (figure 4.2).

Le calcul d'instance est similaire à l'équivalence, excepté le cas particulier de l'instance de variable comme expliqué en amont. La règle INST-PACKAGE introduit une opération de coercion de signature. Par exemple, l'instanciation

$$\Gamma, \Phi, \theta \vdash (\texttt{module M}) \leq (\texttt{module} \texttt{M}')$$

est valide si `M'` est un sous-type de `M`. Le sous-typage des signatures et foncteurs n'est pas décrit dans ce manuscrit.

4.2.3 Vérification de bonne formation/construction

Un jugement de bonne formation est le suivant :

$$\Gamma, \Phi \vdash_{\tau} \tau\ wf$$

Autrement dit, dans un environnement et un ensemble d'égalités, le type τ est bien formé.

Un type est bien formé si :

— pour un constructeur de type, celui-ci est bien lié dans l'environnement et pour chaque type auquel il est appliqué, ces types forment une instance correcte des paramètres de type de la définition.

— pour un type polymorphe quantifié explicitement, les types apparaissant dans la quantification doivent effectivement être des variables universelles. Chacune de ces variables est ajoutée à l'ensemble des variables généralisées de l'environnement.

— parallèlement, s'il s'agit d'une variable universellement quantifiée, elle doit apparaître dans l'ensemble des variables généralisées courantes.

— pour un type de variant polymorphe, alors l'ensemble des constructeurs acceptés est contenu dans celui des types connus, et chacun de ces constructeurs est distinct des autres. De plus, chacun des types doit lui-même être bien formé et la variable de rangée est soit une variable de types, soit le type ϵ, le type particulier caractérisant la cloture du variant polymorphe.

— pour un type de module de première classe, le type de modules, qui est un chemin, doit être lié dans l'environnement. Pour chaque type à substituer dont la définition n'est pas abstraite, le substitut doit en être une instance.

Les règles de vérification de bonne formation sont données en figure 4.3.

Un tel mécanisme de vérification peut avoir plusieurs utilités. Tout d'abord, il permet de vérifier que les invariants de Typedtree sont respectés. La bonne formation va également permettre

4. ∧Dans le cadre de l'implémentation, *GenVars* est déjà représenté par cette substitution, ce qui évite donc de la recalculer à chaque calcul d'instanciation

$$\textsc{Inst-Var-Unbound}\ \frac{'a \notin \theta}{\Gamma, \Phi, \theta \vdash_{\tau} \tau \leq {}'a \Rightarrow \theta \oplus ['a \to \tau]}$$

$$\textsc{Inst-Var-Bound}\ \frac{'a \in \theta \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau_{'a} \equiv \tau}{\Gamma, \Phi, \theta \oplus ['a \to \tau_{'a}] \vdash_{\tau} \tau \leq {}'a \Rightarrow \theta \oplus ['a \to \tau_{'a}]}$$

$$\textsc{Inst-Fun}\ \frac{l_1 = l'_1 \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta_1 \qquad \Gamma, \Phi, \theta_1 \vdash_{\tau} \tau_2 \leq \tau'_2 \Rightarrow \theta_2}{\Gamma, \Phi, \theta \vdash_{\tau} (l_1 : \tau_1) \to \tau_2 \leq (l'_1 : \tau'_1) \to \tau'_2 \Rightarrow \theta_2}$$

$$\textsc{Inst-Tuple}\ \frac{\Gamma, \Phi, \theta \vdash_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta_1 \qquad \forall i_{>1}.\, \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau'_i \Rightarrow \theta_i}{\Gamma, \Phi, \theta \vdash_{\tau} \tau_1 * .. * \tau_n \leq \tau'_1 * .. * \tau'_n \Rightarrow \theta_n}$$

$$\textsc{Inst-Construct}\ \frac{\Gamma, \Phi, \theta \vdash_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta_1 \qquad \forall i_{>1}.\, \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau'_i \Rightarrow \theta_i}{\Gamma, \Phi, \theta \vdash_{\tau} (\overline{\tau})\ \mathtt{p} \leq (\overline{\tau'})\ \mathtt{p} \Rightarrow \theta_n}$$

$$\textsc{Inst-Construct-Exp-Left}\ \frac{let\ \tau = expand(\Gamma, \Phi, \mathtt{t}, \overline{\tau}) \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta'}{\Gamma, \Phi, \theta \vdash_{\tau} (\overline{\tau})\ \mathtt{t} \leq \tau' \Rightarrow \theta'}$$

$$\textsc{Inst-Construct-Exp-Right}\ \frac{let\ \tau' = expand(\Gamma, \Phi, \mathtt{t}', \overline{\tau'}) \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta'}{\Gamma, \Phi, \theta \vdash_{\tau} \tau \leq (\overline{\tau'})\ \mathtt{t}' \Rightarrow \theta'}$$

$$\textsc{Inst-Poly}\ \frac{let\ \theta' = \forall i. \theta \oplus [\alpha'_i \to \alpha_i] \qquad \Gamma, \Phi, \theta' \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta''}{\Gamma, \Phi, \theta \vdash_{\tau} \forall \overline{\alpha}.\, \tau \leq \forall \overline{\alpha'}.\, \tau' \Rightarrow \theta''}$$

$$\textsc{Inst-Univar}\ \frac{[\alpha' \to \alpha] \in \theta}{\Gamma, \Phi, \theta \vdash_{\tau} \alpha \leq \alpha' \Rightarrow \theta} \qquad \textsc{Inst-Rigid1}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi, \theta \vdash_{\tau} \tau \leq \mathtt{t} \Rightarrow \theta}$$

$$\textsc{Inst-Rigid2}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi, \theta \vdash_{\tau} \mathtt{t} \leq \tau \Rightarrow \theta}$$

$$\textsc{Inst-Variant}\ \frac{\begin{array}{c} T_{pres_i} .. T_{pres_j} \subseteq T'_{pres_i} .. T'_{pres_j} \\ \Gamma, \Phi, \theta \vdash_{\tau} \rho \leq \rho' \Rightarrow \theta_{\rho} \qquad \forall i.\ T_i \in \overline{T'} \implies \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_{var_i} \leq \tau'_{var_i} \Rightarrow \theta_i \end{array}}{\Gamma, \Phi, \theta \vdash_{\tau} [(\rho)\ \overline{T\ of\ \tau_{var}} > T_{pres_i} .. T_{pres_j}] \leq [(\rho')\ \overline{T'\ of\ \tau'_{var}} > T'_{pres_i} .. T'_{pres_j}] \Rightarrow \theta_n}$$

$$\textsc{Inst-Nil}\ \Gamma, \Phi, \theta \vdash_{\tau} \epsilon \leq \epsilon \Rightarrow \theta$$

$$\textsc{Inst-Package}\ \frac{\begin{array}{c} let\ S = \mathtt{Sig}(\Gamma, \mathtt{P}\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t} = \tau}) \\ let\ S' = \mathtt{Sig}(\Gamma, \mathtt{P}'\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t}' = \tau'}) \qquad \Gamma, \Phi, \theta \vdash_{\tau} S' :> S \Rightarrow \theta' \end{array}}{\Gamma, \Phi, \theta \vdash_{\tau} (\mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t} = \tau}) \leq (\mathbf{module}\ \mathtt{P}'\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t}' = \tau'}) \Rightarrow \theta'}$$

FIGURE 4.2 – Règles de vérification d'instanciation

L'instanciation est réflexive, transitive et anti-symétrique. Elle permet de calculer que la monomorphisation d'un type généralisé est sûre.

$$\textsc{Wf-Var}\ \Gamma,\Phi \vdash_{\tau} {'a}\ wf \qquad \textsc{Wf-Fun}\ \frac{\Gamma,\Phi \vdash_{\tau} \tau_1\ wf \qquad \Gamma,\Phi \vdash_{\tau} \tau_2\ wf}{\Gamma,\Phi \vdash_{\tau} (l:\tau_1) \to \tau_2\ wf}$$

$$\textsc{Wf-Tuple}\ \frac{\forall \tau_i.\ \Gamma,\Phi \vdash_{\tau} \tau_i\ wf}{\Gamma,\Phi \vdash_{\tau} \tau_1 * .. * \tau_n\ wf}$$

$$\textsc{Wf-Construct}\ \frac{\begin{array}{c} let\ (\overline{\tau_{param}})\ \mathtt{t} = \Gamma.Types(\mathtt{t}) \\ \forall i.\Gamma,\Phi \vdash_{\tau} \tau_i\ wf \qquad \Gamma,\Phi,\Sigma \vdash_{\tau} (\overline{\tau})\ \mathtt{t} \leq (\overline{\tau_{param}})\ \mathtt{t} \Rightarrow \theta \end{array}}{\Gamma,\Phi \vdash_{\tau} (\overline{\tau})\ \mathtt{t}\ \ wf}$$

$$\textsc{Wf-Poly}\ \frac{\forall i.\ \Gamma,\Phi \vdash_{\tau} \tau_{\alpha_i} < \alpha_i \qquad \Gamma \oplus \overline{\alpha},\Phi \vdash_{\tau} \tau\ wf}{\Gamma,\Phi \vdash_{\tau} \forall \overline{\tau_{\alpha}}.\ \tau\ \ wf} \qquad \textsc{Wf-Univar}\ \frac{\alpha \in \Gamma}{\Gamma,\Phi \vdash_{\tau} \alpha\ wf}$$

$$\textsc{Wf-Variant}\ \frac{\begin{array}{c} \forall i,j.\ i \neq j \implies T_i \neq T_j \qquad \forall i.\Gamma,\Phi \vdash_{\tau} \tau_i\ wf \\ \forall T_{pres_i} \in \{T_{pres_i} .. T_{pres_j}\}.\ T_{pres_i} \in \overline{T} \qquad \Gamma,\Phi \vdash_{\tau} \rho < {'a} \bigvee \Gamma,\Phi \vdash_{\tau} \rho < \epsilon \end{array}}{\Gamma,\Phi \vdash_{\tau} [(\rho)\ \overline{\sim T\ \mathbf{of}\ \tau} > T_{pres_i} .. T_{pres_j}]\ wf}$$

$$\textsc{Wf-Package}\ \frac{\begin{array}{c} \mathtt{P} \in \Gamma.Modtypes \qquad \forall i.\ \mathtt{t_i} \in \Gamma.Modtypes(\mathtt{P}) \\ \forall i.\Gamma,\Phi \vdash_{\tau} \tau_i\ wf \qquad \forall i\ s.t.\ transparent(\mathtt{P.t_i}).\ \Gamma,\Phi \vdash_{\tau} \tau_i \leq \mathtt{P.t_i} \end{array}}{\Gamma,\Phi \vdash_{\tau} (\mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{type\ \mathtt{t} = \tau})\ wf}$$

FIGURE 4.3 – Vérification de bonne formation des types

La propriété de bonne formation permet entre autres de s'assurer du non échappement des variables existentielles et de la construction correcte des types de variants polymorphes et modules de première classe.

de modéliser le principe de non échappement de types, chose possible avec l'utilisation de références, de GADTs ou de modules de première classe. Nous verrons ces cas précis lorsque nous aborderons la vérification de ces expressions.

4.2.4 Filtrage de types

Le filtrage de type permet de s'assurer qu'un type représenté par une métavariable dans le contexte est en réalité de la forme voulue, et d'en extraire les sous-nœuds. Le jugement de filtrage est le suivant :

$$\Gamma, \Phi \models_{\tau} \tau \prec \tau_\rho$$

Soit, dans un environnement (Γ) et un ensemble d'équivalence (Φ), le type τ est de la forme de τ_ρ. Ici, τ_ρ est en réalité un motif (ou *pattern*), dont les sous-nœuds représentés par des métavariables peuvent être utilisés par la suite.

Pour un exemple concret, supposons que le jugement suivant apparaît dans le contexte d'une règle :

$$\Gamma, \Phi \models_{\tau} \tau \prec \tau_1 \rightarrow \tau_2$$

cela doit être compris comme "sachant Γ et Φ, τ est un type de fonction, dont le domaine τ_1 et le codomaine τ_2 sont désormais accessibles dans la règle courante". Il s'agit en d'autres termes du filtrage d'OCaml appliqué aux types, où le langage de motifs est lui-même représenté par la syntaxe des types.

Dans le cas du filtrage, l'environnement (Γ) est nécessaire pour expanser les éventuelles abréviations de types. Dans le cas où le type à filtrer serait une variable de type rigide (voir 78), il est alors nécessaire de chercher s'il existe dans sa classe d'équivalence un type concret de la forme désirée, d'où la nécessité d'avoir l'ensemble courant des classes d'équivalence (Φ).

Dans le cas particulier où le filtrage doit trouver un équivalent dans Φ, celui-ci peut avoir dans sa classe plusieurs types concrets. En revanche, si les classes d'équivalence ont été correctement construites, comme nous le verrons dans la partie suivante, alors tous les types concrets sont de toute manière équivalents et donc le type concret choisi sera dans tous les cas correct. Par exemple, supposons

$$\Gamma, \Phi \models_{\tau} \mathtt{a} \prec \tau_1 * \tau_2$$

où :

- Dans Φ, `a` appartient à la classe d'équivalence {`a`, `int_int`, `int` $*$ `int`, `b * b`}. La variable rigide `b` appartient à la classe d'équivalence {`b`, `int`}.
- Dans Γ, `a` et `b` sont des variables de type rigides, `int_int` est une abréviation pour `int` $*$ `int`.

Peu importe quel sera le type concret choisi pour $\tau_1 * \tau_2$, puisque les trois types concrets sont équivalents, modulo expansions et équivalences. S'assurer que les classes d'équivalences seront toujours construites correctement constitue l'enjeu de la vérification des GADTs.

Implémentation du filtrage

L'une des propriétés de la vérification telle que présentée dans ce manuscrit est qu'il en existe une implémentation concrète. Celle du filtrage se fait à l'aide de GADTs. L'intérêt ici est une meilleure factorisation du code, et par conséquent rendre celui-ci plus robuste aux évolutions du système de types. Dans l'implémentation présentée ici, le filtrage ne se fait que sur un seul niveau de profondeur, puisque son utilisation dans la vérification ne teste que la forme du type "en surface".

```
type 'a repr =
    Rtuple : type_expr list repr
  | Rconstr : (type_expr list * Path.t) repr
  | Rvariant : row_desc repr
  | Rarrow : (string * type_expr * type_expr) repr
  | Rpackage : (Path.t * Longident.t list * type_expr list) repr
  | Rpoly : (type_expr list * type_expr) repr
  | Robject : type_expr repr
```

A chaque type correspond une représentation. Chacun des constructeurs est indexé par le type de ses composants, qui sont extraits à l'issue de ce filtrage. Ainsi, un type de fonction est représenté par un triplet : une chaîne de caractères et deux types, respectivement le label, le domaine et le codomaine. On peut donc facilement écrire une fonction qui, pour un type et un représentant, retourne ses composants :

```
let extract : type a. type_expr * a repr -> (a, error) result =
  function
    ({desc = Ttuple tys}, Rtuple) -> Ok tys
  | ({desc = Tconstr (tys, p, _)}, Rconstr) -> Ok (tys, p)
  | ({desc = Tvariant rd}, Rvariant) -> Ok rd
  | ({desc = Tpoly (ty, params)}, Rpoly) -> Ok (params, ty)
  | ({desc = Tarrow (l, ty, ty', _)}, Rarrow) -> Ok (l, ty, ty')
  | ({desc = Tpackage (p, lids, tys)}, Rpackage) -> Ok (p, lids, tys
      )
  | ({desc = Tobject (f, _)}, Robject) -> Ok f
  | ty, r -> Error (Expected_type_mismatch (r, ty))
```

Cette fonction d'extraction est ensuite utiisée par la fonction `extract_type_info` :

```
let rec extract_type_info :
  type a. ?expand_poly -> context -> a repr -> type_expr
  -> (a, error) result =
    fun ctx r ty ->
```

Celle-ci prend alors un contexte, un témoin de type et le type lui-même, et retourne alors un résultat d'extraction. On remarque également un argument optionnel `expand_poly`, vrai par défaut, qui indique à la fonction que dans le cas où le type est de la forme $\forall\overline{\alpha}.\tau$, alors l'extraction est effectuée sur τ directement. La fonction d'extraction est alors appelée :

— L'extraction a réussi, le résultat peut donc être retourné :

```
match extract ty r, ty with
  Ok _ as res, _ -> res
```

— L'extraction a échoué, et le type dont il faut extraire les informations est un constructeur de type local utilisé pour représenter un paramètre de type de GADT :

```
| Error _, { desc = Tconstr (p, [], _); } when gadt_mode ctx p
    ->
```

Pour ce second cas, la solution est de naviguer dans la classe d'équivalence du type pour trouver le premier type concret possible, que l'on récupère via la fonction `find_equiv_ambi`.

```
let ty_repr, ambi = find_equiv_ambi ctx ty [] in
```

Il s'agit alors de trouver au moins un type concret correspondant à la représentation demandée, et donc d'écarter les types locaux abstraits de cette classe.

```
TySet.fold (fun ty acc -> match acc, ty with
    Ok _, _ -> acc
    | _, { desc = Tconstr (p, [], _); _ } when is_newtype p ->
       acc
    | _, { desc = Tconstr _; _} when not (is_rconstr r) ->
      extract (safe_expand_abbrev_max ctx ty, r)
    | _, ty ->
      extract (ty, r))
  ambi.value (Error (Expected_type_mismatch (r, ty)))
```

— L'extraction a échoué, le type est un constructeur de type. Le constructeur est alors expansé puisqu'il pourrait s'agir d'une abréviation de type, et la fonction est rappelée sur le résultat.

4.2.5 Vérification d'équations de types

fold* et *fold2 Les opérations *fold* et *fold2* qui seront utilisées par la suite sont deux combinateurs qui, pour le premier, traverse la structure d'un type en ordre préfixe et prend en argument une fonction, un accumulateur et un type. A chaque nœud, la fonction prend en argument l'accumulateur et le nœud courant et retourne un nouvel accumulateur. Il s'agit en d'autres termes d'une fonction *fold* classique telle qu'on peut la trouver pour toutes les structures de données de la bibliothèque standard d'OCaml. La seule exception est que la fonction est appliquée sur tous les nœuds, et pas seulement les feuilles du graphe que représente un type. Par exemple,

$$\texttt{fold}\ f\ acc\ (\texttt{int} \rightarrow \texttt{bool}) \equiv f\ (f\ (f\ acc\ \texttt{int})\ \texttt{bool})\ \texttt{int} \rightarrow \texttt{bool}$$

Le combinateur *fold2* traverse deux types de manière préfixe, tant que les deux types ont la même structure. Par exemple,

$$\texttt{fold2}\ f\ acc\ (\texttt{int} \rightarrow \texttt{bool})\ (\texttt{a} \rightarrow \texttt{b})$$

$$\equiv$$

$$f\ (f\ (f\ acc\ \texttt{int}\ \texttt{a})\ \texttt{bool}\ \texttt{b})\ (\texttt{int} \rightarrow \texttt{bool})\ (\texttt{a} \rightarrow \texttt{b})$$

Contrairement au *fold2* des structures de données d'OCaml, on considère deux types n'ayant pas la même structure comme étant acceptable pour le parcours (les sous-nœuds ne sont tout simplement pas traversés), on et laisse le soin à la fonction passée en paramètre de gérer ce cas particulier. Il est par exemple possible de réécrire l'ensemble des opérations de comparaison de types, laissant à *fold2* la problématique de traverser les deux types à comparer, et laisser à la fonction d'ordre supérieur le soin de comparer deux à deux directement et d'effectuer par exemple les expansions de types.

L'introduction des types algébriques gardés (dits aussi “GADT”, pour *Generalized Algebraic Datatypes*) permet d'exprimer un ensemble de programmes qui jusqu'à présent (avant la version 4.0 du compilateur) n'étaient pas acceptés par le système de types d'OCaml. Leur utilisation, combinée avec types locaux abstraits (que l'on considérera comme des variables de types rigides par la suite), génère un certain nombre d'équations de types locales. Reprenons un exemple classique généralement donné pour illustrer cette construction, celui des témoins de types permettant d'écrire des fonctions retournant des valeurs de types différents dépendant de la valeur donnée en paramètre :

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

Ici, `a` est un type local abstrait, une variable de type rigide qui servira à représenter le paramètre de type de `t`, dont les deux constructeurs sont des GADTs.

Dans chaque branche du filtrage, la variable rigide est associée au type du paramètre. Dans la branche (1), une équation est créée dans Φ : `a` et `int` sont désormais équivalents pour le reste de la branche. Comme indiqué précédemment, Φ est une structure de données dédiée à stocker des classes d'équivalence de types, notées $\mathcal{A}$. On peut distinguer deux cas pour rajouter une équation, dont l'algorithme en *pseudo-ML* est donné dans la figure 4.2.5. Il s'agit basiquement d'un mécanisme d'*union-find* :

— Au moins l'un des deux types n'apparaît pas lié dans Φ, il s'agit donc de le rajouter et le faire pointer sur la classe de l'autre type (ou de créer une nouvelle classe d'équivalence dans le cas où le second n'existe pas encore dans Φ). On s'assure alors que le nouveau type ajouté à la classe est bien équivalent avec les types concrets de celle-ci. Ainsi s'il n'existe pas de classe pour l'un des types (prenons τ), une classe singleton (ne contenant que lui-même) est retournée dans le cas de l'opération $\Phi(\tau_1)$.

— Les deux types font partie de deux classes distinctes : on unit les deux classes. Il faut alors vérifier que chaque type concret de la première est bien équivalent avec ceux de la seconde. Il s'agit de l'opération `is_inconsistent` : un type τ est incompatible avec une classe $\mathcal{A}$ s'il existe un type concret $\tau_{\mathcal{A}}$ dans cette classe tel que ces deux types ne sont pas équivalents, ou que τ apparaît dans la structure de $\tau_{\mathcal{A}}$, auquel cas le type serait récursif[5]. Cette vérification à chaque union de classes permet de garantir la sûreté de l'équivalence de types. Dans le cas présent, `int` et `a` n'apparaissent pas dans Φ, l'un étant une variable rigide les unir dans une classe est donc évident.

4.3 Algorithme de vérification des expressions

Une fois défini l'ensemble des opérations de manipulation et vérification de types, il devient alors possible de formaliser l'ensemble des règles de vérification d'un Typedtree.

5. ∧Les types récursifs ne sont pas gérés par le vérificateur actuel.

```
let is_inconsistent (Γ, Φ) τ 𝒜 =
  ∃τ𝒜 ∈ 𝒜. not equiv (Γ, Φ) τ𝒜 τ ⋀ not rigid τ𝒜 ⋁ occurs τ τ𝒜

let add_equation (Γ, Φ, generalized) τ₁ τ₂ =
  if generalized ⋀ τ₁ is rigid then
      let 𝒜₁ = Φ(τ₁) in
      let 𝒜₂ = Φ(τ₂) in
      if 𝒜₁ = 𝒜₂ then (Γ, Φ)
      else if is_inconsistent (Γ, Φ, τ₁, 𝒜₂)
        ⋁ is_inconsistent (Γ, Φ, τ₂, 𝒜₁) then Error
      else
        let 𝒜 = 𝒜₁ ⋃ 𝒜₂ in
        let Φ′ = ∀τ ∈ 𝒜. Φ(τ) ← 𝒜 in
        (Γ, Φ′, generalized)
  else
    (Γ, Φ, generalized)

let add_equations (Γ, Φ) generalized τ₁ τ₂ =
  fold2 add_equation (Γ, Φ, generalized) τ₁ τ₂
```

FIGURE 4.4 – Algorithme de vérification d'équations

L'algorithme correspond dans les faits à un algorithme d'*union-find*. La vérification des types algébriques gardés nécessite de vérifier que les équations de types qu'ils introduisent conservent la cohérence de l'environnement.

4.3.1 ML

On définit les règles nécessaires à la vérification des constructions minimales de ML en figure 4.5.

La notation $\overline{e}$ désigne une liste finie d'éléments, il peut s'agir d'expressions annotées, de motifs, de variables, etc. Ainsi, l'accès à un élément en particulier se traduit par e_i, où i est sa position dans la liste. Par conséquent, l'application d'une opération ou d'une condition sur tous les éléments de la liste s'écrit $\forall i ... e_i ...$

La vérification d'une constante (CONST) vérifie simplement que la constante appartient bien au type avec lequel elle est annotée modulo équivalence. Le cas de la variable (VAR) dont nous avons vu un exemple concret plus tôt est lui aussi simple : la variable doit être liée dans l'environnement, et le type annoté doit être une instance du type enregistré pour celle-ci. La substitution initiale est générée à partir de l'environnement courant, garantissant ainsi que les variables de type ne seront pas instanciées dans un contexte où elles seraient quantifiées dans le contexte courant.

Le cas de la fonction (ABS) est en revanche plus complexe. La construction **function** permet, en plus de créer une abstraction, d'effectuer directement un filtrage sur la forme de l'argument. La notation $\overline{|p \rightarrow e}$ désigne les différentes branches possibles du filtrage, où p désigne un motif (*pattern*) contenant en particulier les variables qui seront liées dans e, le corps de la branche [6]. La notation **fun** est un sucre syntaxique de **function**, où une seule branche est possible et surtout l'argument peut être labellisé. Par exemple, la fonction **fun** `x y → x y` sera traduite en

6. ∧On remarquera que la condition **when** n'apparaît pas, par soucis de concision.

$$\textsc{Const}\ \frac{c : \tau}{\Gamma, \Phi \vdash \langle c : \tau \rangle} \qquad\qquad \textsc{Var}\ \frac{\Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \Gamma.Values(x) \Rightarrow \theta}{\Gamma, \Phi \vdash \langle x : \tau \rangle}$$

$$\textsc{Abs}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash_{\tau} \tau < \tau_d \to \tau_{cd} \qquad \forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ \forall i.\ \Gamma, \Phi \vdash_{\tau} \tau_d \equiv \tau_i \qquad \forall i.\ let\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}}) \\ \forall i.\ \Gamma_i, \Phi_i \vdash \langle e_i : \tau_i' \rangle \qquad \forall i.\ \Gamma, \Phi \vdash_{\tau} \tau_i'\ wf \qquad \forall i.\ \Gamma_I, \Phi_i \vdash_{\tau} \tau_{cd} \equiv \tau_i' \end{array}}{\Gamma, \Phi \vdash \langle \texttt{function}\ \overline{\mid p \to e}\ : \tau \rangle}$$

$$\textsc{App}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \\ \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_1 < \tau_d \to \tau_{cd} \qquad \Gamma, \Phi \vdash_{\tau} \tau_2 \equiv \tau_d \qquad \Gamma, \Phi \vdash_{\tau} \tau_{cd} \equiv \tau \end{array}}{\Gamma, \Phi \vdash \langle e_1\ e_2 : \tau \rangle}$$

$$\textsc{Let}\ \frac{\begin{array}{c} \forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_{\exists} = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_1}}) \\ \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \forall i.\ \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \qquad \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i) \\ let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_{\exists} \qquad \Gamma', \Phi_n \vdash \langle e' : \tau' \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau'\ wf \qquad \Gamma', \Phi_n \vdash_{\tau} \tau \equiv \tau' \end{array}}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \texttt{in}\ e' : \tau \rangle}$$

FIGURE 4.5 – Règles de vérifications du noyau d'OCaml
Ces règles concernent le même ensemble de constructions présentées chapitre 2 pour MiniML.

```
⟨ function ⟨ x : 'a → 'b ⟩ →
  ⟨ function ⟨ y : 'a ⟩ →
    ⟨ ⟨ x : 'a → 'b ⟩ ⟨ y: 'a ⟩ : 'b ⟩ : 'a → 'b ⟩ :
  ('a → 'b) → 'a → 'b ⟩
```

où x est un motif de variable. La construction n'a alors qu'une branche, dont le filtrage ne peut échouer puisqu'il englobe tous les cas. La vérification de l'abstraction doit effectuer plusieurs opérations avant de pouvoir rendre son verdict. Avant tout, il faut s'assurer que le type annoté pour l'abstraction (τ) est bien un type de fonction, via l'opération de filtrage décrite en 4.2.4 :

$$\Gamma, \Phi \vdash_{\tau} \tau \prec \tau_d \to \tau_{cd}$$

Ainsi, le type est décomposé et il est possible de manipuler le type de son domaine (τ_d) et celui de son codomaine (τ_{cd}). Ensuite, chaque branche doit être vérifiée, d'abord le motif puis l'expression elle-même. La vérification d'un motif, lui-même annoté, retournera l'ensemble des variables liées dans celui-ci, les types existentiels potentiellement dépaquetés ainsi qu'un nouvel ensemble d'équivalences. Il faut ensuite s'assurer que le type annoté à chaque branche est équivalent au domaine, puis vérifier chaque corps avec les variables nouvellement liées, les types existentiels créés et les nouvelles équivalences. Il faut ensuite s'assurer que le type du corps est bien équivalent à celui du codomaine dans ce même nouvel environnement. Finalement, on vérifie que chaque type de résultat est bien formé dans l'environnement initial, *i.e.* qu'aucun type existentiel créé par le motif ne s'échappe de cette branche. En effet, ce type n'étant valable que dans la branche courante, il ne peut exister en dehors de celle-ci.

Le cas de l'application (App) est plus simple : il s'agit d'abord de vérifier la fonction et l'argument, puis de s'assurer que la fonction est bien annotée avec un type de fonction, pour finalement vérifier que le type de l'argument est bien équivalent à celui du domaine et le type de l'expression entière est bien équivalent au type du codomaine. Particulièrement dans cette règle, on retrouve l'idée de rendre explicite chacune des opérations et égalités de types, et ainsi ne laisser aucune ambiguité.

Finalement, la vérification des variables locales (Let) combine plusieurs conditions énoncées précédemment. D'abord, l'ensemble des variables locales sont en réalités liées par des motifs : cela permet entre autres de destructurer certaines valeurs comme les n-uplets, ou les types avec un unique constructeur. La vérification de ces motifs retourne un ensemble de variables liées mais également la liste des nouveaux types existentiels à ajouter à l'environnement, liste accompagnée des nouvelles équations de types générées par les constructeurs gardés. Les expressions associées sont ensuite vérifiées dans l'environnement initial. Le type de l'expression doit être une instance du motif auquel on cherche à le lier. L'étape importante est alors la vérification de généralisation : le but est ici de tester, selon les conditions énoncées pour la restriction souple de valeurs, si pour toute variable généralisée qui apparait dans le type du motif celle-ci peut effectivement être généralisée. On teste également qu'une variable non généralisée ne peut effectivement pas être généralisée, bien que cela n'affecte pas la sûreté du programme. L'algorithme de généralisation est présenté en sous-section 4.3.2. Finalement, la vérification de cohérence de l'expression générale est effectuée dans un nouvel environnement contenant toutes les informations récupérées des motifs. On doit bien entendu s'assurer que son type est bien formé, sous peine de voir des types existentiels locaux s'échapper de leur contexte. Le type du nœud doit être équivalent à ce type.

$$\textsc{Pat-Const}\ \frac{c : \tau}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle c : \tau \rangle \Rightarrow \mathcal{V}, \mathcal{T}, \Phi} \qquad \textsc{Pat-Wildcard}\ \Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle \_ : \tau \rangle \Rightarrow \mathcal{V}, \mathcal{T}, \Phi$$

$$\textsc{Pat-Var}\ \frac{v \notin dom(\mathcal{V}) \qquad let\ \mathcal{V}' = \mathcal{V} \oplus (v, \tau)}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle v : \tau \rangle \Rightarrow \mathcal{V}', \mathcal{T}, \Phi}$$

$$\textsc{Pat-Or}\ \frac{\begin{array}{c}\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle p_1 : \tau_1 \rangle \Rightarrow \mathcal{V}_1, \mathcal{T}_1, \Phi_1 \qquad \Gamma, \Phi_1, \mathcal{V}, \mathcal{T}_1 \vdash_{\rho} \langle p_2 : \tau_2 \rangle \Rightarrow \mathcal{V}_2, \mathcal{T}_2, \Phi_2 \\ \Gamma, \Phi_2 \vdash_{\tau} \tau_1 \equiv \tau_2 \qquad \Gamma, \Phi_2 \vdash_{\tau} \tau_2 \equiv \tau \qquad \Gamma, \Phi_2 \vdash_{\tau} \mathcal{V}_1 \equiv \mathcal{V}_2\end{array}}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle p_1 \mid p_2 : \tau \rangle \Rightarrow \mathcal{V}_2, \mathcal{T}_2, \Phi_2}$$

FIGURE 4.6 – Vérification des motifs

La vérification des motifs retourne l'ensemble des variables liées et donc des sous-expressions de la valeur filtrée. Avec l'introduction dans le langage des types algébriques gardés, le filtrage doit aussi retourner un nouvel ensemble d'égalités de types, ainsi que de nouveaux constructeurs de types abstraits représentants des types existentiels générés par l'inférence (voir section 4.3.3). Comme pour la vérification d'expressions, la vérification de motifs nécessite l'environnement Γ pour les types construits, ainsi que Φ pour les égalités de types. $\mathcal{V}$ associe des variables à des types et correspond à l'ensemble des variables déjà liées par le motif courant. $\mathcal{T}$ correspond à l'ensemble des types existentiels extraits du motif.

Un motif de constante (PAT-CONST) est correct si la constante est bien du type auquel on l'a annotée. Il ne génère pas de nouvelle variable, ni de types existentiels ou d'égalités.

Le motif _ (PAT-WILDCARD) représente un motif irréfutable : il peut correspondre à n'importe quelle valeur. Par conséquent, il n'y a aucune vérification supplémentaire et il ne génère aucune nouvelle donnée.

Le motif de variable (PAT-VAR) peut représenter n'importe quelle valeur. En revanche, cette variable ne doit pas déjà apparaître dans l'environnement courant ni dans le motif courant : en effet, le filtrage d'OCaml étant linéaire une variable ne peut apparaître qu'une seule fois par motif. Dans le cas où une variable pourrait apparaître plusieurs fois dans un motif, cela impliquerait une équivalence structurelle entre chacune des occurrences de celle-ci. Supposons par exemple le motif `(x, x)`, si le filtrage n'était pas linéaire, celui-ci correspondrait à des couples où les deux éléments sont structurellement identiques, par exemple `(0, 0)` ou `(4, 4)`. En revanche, les fonctions étant des valeurs de première classe, cela implique d'être capable de tester l'équivalence structurelle entre deux fonctions, ce qui est impossible. La comparaison polymorphe de la bibliothèque standard (`compare`, ou l'opérateur =) ne fonctionne d'ailleurs pas sur les fonctions et lève une exception à l'exécution. Par conséquent, la vérification de motifs de variable doit s'assurer que celle-ci n'apparaît pas dans l'ensemble des variables déjà liées par le motif.

Enfin, le motif `p | q` filtre une valeur `v` si celle-ci est de la forme de `p` ou `q`. Cela implique de s'assurer que les variables liées par chacun sont les mêmes, en s'assurant bien sûr que le type annoté pour les deux motifs sont équivalents, soit

$$\Gamma, \Phi_2 \vdash_{\mathcal{T}} \mathcal{V}_1 \equiv \mathcal{V}_2$$

$$\Leftrightarrow$$

$$dom(\mathcal{V}_1) = dom(\mathcal{V}_2) \wedge \forall v. v \in \mathcal{V}_1 \rightarrow \Gamma, \Phi \vdash_{\mathcal{T}} \mathcal{V}_1(v) \equiv \mathcal{V}_2(v)$$

Les deux tests sont nécessaires. Pour le premier, il est nécessaire de s'assurer que les mêmes variables sont extraites et avec le même type, auquel cas il serait possible d'accéder à une variable dont il n'existe pas de valeur, ou dont le type ne concorde pas à son utilisation. Les éventuels types existentiels sont en revanche accumulés entre les deux branches du motif, sans vérifier que les types existentiels nouvellement extraits soient les mêmes. Un exemple simple permet de montrer ce phénomène (voir la section 4.3.3 pour plus d'explications sur les types algébriques gardés et l'utilisation des types existentiels) :

```
type t =
  Ex : int * 'a -> t

let f = function
    Ex (0, ⟨ _ : τ∃ ⟩) | Ex (1, ⟨ _ : τ'∃ ⟩) -> 0
  | _ -> 1
```

Le premier motif irréfutable étant existentiellement quantifié, un constructeur de type frais est généré. Pour celui de la branche droite, un autre constructeur frais est généré, mais celui-ci est différent. Si on les remplaçait par la même variable `x`, le programme serait faux : le type de la valeur étant existentiel, on ne peut considérer que leur type "réel" puisse être le même dans chacun des motifs. Par exemple, la fonction suivante est correctement typée grâce à l'utilisation des types existentiels :

```
let make i =
  if i = 0 then Ex (i, "zero")
  else if i = 1 then Ex (i, 0.0)
  else Ex (i, 0)
```

Dans les deux cas du filtrage, la représentation de la valeur existentielle est différente : si l'entier est 0, alors il s'agit d'une chaîne de caractères, et si l'entier vaut 1, c'est une valeur flottante. On ne pourrait donc pas leur attribuer le même type existentiel, quand bien même celui-ci est complètement abstrait et ne permet donc pas d'inférer la véritable représentation.

Exemple d'implémentation de vérification de fonctions

On peut mettre en parallèle chacune de ces règles avec le code correspondant, et ainsi constater que cette formulation permet de dériver une implémentation en OCaml relativement facilement. Par exemple, la transcription de la règle ABS (les fonctions OCaml sont ordonnées pour simplifier la lecture, qui n'est évidemment pas l'ordre réel) :

```
| Texp_function (_, cases, _), ty_annot ->
```

On traite ici le cas de la function, où `cases` est une liste de cas, dont le type est

```
type case = {
  c_lhs: pattern;
  c_guard: expression option;
  c_rhs: expression;
}
```

dont chacun des champs est respectivement : le motif, la garde si elle existe et le corps.

```
    let ty_arg_exp, ty_res_exp =
      match Extract.extract_arrow_info ctx ty_annot with
        Ok (_, ty, ty') -> ty, ty'
      | Error e -> raise (Typecheck_Error e)
    in
```

Il s'agit d'abord de s'assurer que le type annoté `ty_annot`, est une fonction. Les opérations sur les types retournent généralement une valeur de type `('a, error) result`, notamment pour améliorer la propagation et la traçabilité des erreurs. Le résultat consiste ici en trois valeurs : un label (masqué ici mais utilisé dans l'implémentation) et deux types : le domaine et le codomaine de la fonction. Cette partie correspond alors à

$$\Gamma, \Phi \vdash_{\tau} \tau \prec \tau_d \rightarrow \tau_{cd}$$

```
    type_cases ctx ty_arg_annot ty_res_annot cases
```

L'appel à la fonction `type_cases` se fera alors sur l'environnement courant (`ctx`), le domaine (`ty_arg_annot`), le codomaine (`ty_res_annot`) et les branches du filtrage.

```
and type_cases ctx ty_arg ty_res cases =
  List.fold_left (fun ty case ->
```

et, pour chacune des branches,

```
        let typat, binded, ctx =
          type_pattern ctx SMap.empty case.c_lhs ty_arg in
```

on vérifie d'abord le motif de l'argument. La fonction `type_pattern` prend également le type attendu, et s'occupe alors d'effectuer le test d'équivalence. Il s'agit donc de la transcription de

$$\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i$$

$$\forall i.\ \Gamma, \Phi \vdash_{\tau} \tau_d \equiv \tau_i$$

où `ty_pat` est le type annoté au motif, `binded` l'ensemble des variables apparaissant dans le motif avec leur type, et `ctx` l'environnement initial contenant les nouvelles équivalences et types existentiels.

```
        let vars = SMap.fold (fun _ (ty, _, _) vars ->
            let tys =
              Typecheck_types.extract_vars_list [] ty in
            List.fold_left
              (fun vars ty -> TySet.add ty vars) vars tys)
            binded ctx.generalized_vars in
        let ctx' =
```

```
      { ctx with
          env = pattern_bindings true ctx.env binded;
          generalized_vars = vars
      } in
```

L'environnement pour la branche courante est ensuite mis à jour avec les nouvelles variables

$$\forall i.\ let\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}})$$

```
    begin
      match case.c_guard with
        None -> ()
      | Some e ->
        ignore @@ type_expr ctx' e Predef.type_bool
    end;
```

La garde est vérifiée, en s'assurant que son type est bien équivalent au type `bool`.

$$\forall i.\ \Gamma_i, \Phi_i \vdash \langle c_i : \tau'_{c_i} \rangle$$

$$\forall i.\ \Gamma_I, \Phi_i \vdash_{\mathcal{T}} \tau_{c_i} \equiv \texttt{bool}$$

```
    type_expr ctx' case.c_rhs ty
```

Finalement, le corps de chaque branche est vérifié avec le nouvel environnement généré. La fonction `type_expr` va effectuer les trois opérations restantes suivantes :

$$\forall i.\ \Gamma_i, \Phi_i \vdash \langle e_i : \tau'_i \rangle$$

$$\forall i.\ \Gamma, \Phi \vdash_{\mathcal{T}} \tau'_i\ wf$$

$$\forall i.\ \Gamma_I, \Phi_i \vdash_{\mathcal{T}} \tau_{cd} \equiv \tau'_i$$

La fonction retourne alors le type annoté à l'expression.

```
      (* fin de l'appel à `fold_left` *)
      )
    ty_res cases
```

4.3.2 Généralisation et restriction souple de valeurs

Dans un langage typé implicitement, il peut être difficile d'inférer correctement la quantification des variables de types polymorphes. Le choix fait dans ML, et par extension OCaml, est de toujours quantifier de manière prénexe, il n'y a donc pas de polymorphisme d'ordre supérieur [7].

Ainsi, il n'est pas possible d'écrire la fonction (annotée) suivante :

```
let ⟨ f : (∀α. α → α). → int ∗ bool ⟩ = fun id -> (id 0, id true)
```

7. ∧Les enregistrements permettent une forme explicite de types polymorphes d'ordre supérieur, comme nous le verrons plus loin.

autrement dit une fonction qui prendrait en argument une valeur polymorphe. L'inférence n'est pas capable de générer une telle quantification, et `id` ne pourrait alors avoir que le type `int` → `int` ou `bool` → `bool`. La fonction n'est donc pas admise par le système de types d'OCaml. Ce choix de quantification prénexe est nécessaire pour la décidabilité de l'inférence de types.

La généralisation est le processus par lequel l'inférence de type infère la quantification des variables de types. Ainsi, pour la fonction précédente, on dira que les variables α et β ont été généralisées, et donc pourront par la suite être instanciées vers un type monomorphe. Pour OCaml, le choix de généraliser ou non s'effectue dans tous les cas lors du typage de l'expression **`let`**, et s'appuie sur le principe de la restriction de valeur[69]. Selon la restriction de valeur "classique", on considère que seules les expressions étant syntaxiquement des valeurs peuvent être généralisées. Ainsi, une application de fonction n'est pas généralisable. Par exemple, considérons l'application partielle de la fonction `f` précédente

```
let ⟨ x : int → int * '_b ⟩ = f 2
```

On cherche à inférer le type de `x`. Le type de `f 2` est `int → int * '?b`, où `'?b` est une variable de type fraîche permettant d'instancier β. Cette expression n'étant pas une valeur, `f 2` se réduisant en `fun y -> (2, y)`, on ne peut généraliser la variable de type `'?b`. Par conséquent, celle-ci est considérée comme étant monomorphe et donc sera unifiée avec un type monomorphe dès la prochaine utilisation de `x`. Le polymorphisme peut être retrouvé par eta-expansion dans le cas des applications partielles, soit dans le cas présent :

```
let ⟨ x' : int → 'b ⟩ i = f 2i
```

Cette restriction de valeur permet principalement de traiter le cas du polymorphisme en présence d'effets de bords, particulièrement des références. Ainsi, supposons la valeur suivante :

```
let ⟨ f : (? → ?) ref ⟩ = ref (fun x -> x)
```

Dans le cas sans restriction de valeur, `f` aurait pour type `('a → 'a) ref`. Par conséquent, une telle affectation serait valide :

```
⟨ f : (int → int) ref := (fun x -> x + 1)
```

En effet, si on considère la règle VAR définie précédement, `f` doit être une instance valide de son type contenu dans l'environnement courant. Trivialement,

$$\Gamma, \Phi, \emptyset \mathrel{|}\!\!\frac{}{\tau} (\texttt{int} \rightarrow \texttt{int})\,\texttt{ref} \leq ('a \rightarrow\ 'a)\,\texttt{ref} \Rightarrow ['a \mapsto \texttt{int}]$$

L'affectation est alors correcte. Le type de `f` n'a lui pas été modifié dans l'environnement. Finalement, il est donc possible d'appliquer `f` sur n'importe quelle valeur, étant donné son type :

```
let _ = ⟨ !f : string → string ⟩ "Bad idea"
```

Son exécution résultant sur le terme `"Bad idea"+ 1`, qui est une erreur. Le système de types n'est alors plus sûr en présence de références. La restriction de valeur permet d'éliminer ce cas, car l'application de **`ref`** n'est pas une valeur syntaxique.

La restriction de valeur originale considère donc le cas de la généralisation des valeurs lors du typage d'un **`let`**. Si l'expression liée est une valeur syntaxique (voir figure 4.7) [8], alors le type de

8. ∧On parlera également d'expression non expansive, *i.e.* qui ne se réduit pas lors de l'exécution.

$nonexp\,(\mathtt{x}) ::= \top$
$nonexp\,(\mathtt{c}) ::= \top$
$nonexp\,(\mathbf{let}\ \mathtt{x} = \mathtt{e}_1\ \mathbf{in}\ \mathtt{e}_2) ::=$
$\quad nonexp\,(\mathtt{e}_1) \wedge nonexp\,(\mathtt{e}_2)$
$nonexp\,(\mathbf{fun}\ \mathtt{x}\ \text{->}\ \mathtt{e}) ::= \top$
$nonexp\,(\mathbf{match}\ \mathtt{e}\ \mathbf{with}\ \overline{\mid\ \mathtt{p}\ \mathbf{when}\ \mathtt{g}\ \text{->}\ \mathtt{b}}) ::=$
$\quad nonexp\,(\mathtt{e}) \wedge \bigwedge_{i=1}^{n} (nonexp\,(\mathtt{g}_i) \wedge nonexp\,(\mathtt{b}_i))$
$nonexp\,(\mathtt{e}_1\,,\ \mathtt{e}_2) ::= nonexp\,(\mathtt{e}_1) \wedge nonexp\,(\mathtt{e}_2)$
$nonexp\,(K) ::= \top$
$nonexp\,(K\,e) ::= nonexp\,(\mathtt{e})$
$nonexp\,(`K) ::= \top$
$nonexp\,(`K\,e) ::= nonexp\,(\mathtt{e})$
$nonexp\,(\{\ \overline{\mathtt{l} = \mathtt{e};}\ \}) ::= \bigwedge_{i=1}^{n} (immutable\,(\mathtt{l}_i) \wedge nonexp\,(\mathtt{e}_i))$
$nonexp\,(\mathtt{e.l}) ::= nonexp\,(\mathtt{e})$
$nonexp\,(\mathbf{if}\ \mathtt{c}\ \mathbf{then}\ \mathtt{e}_1\ \mathbf{else}\ \mathtt{e}_2) ::=$
$\quad nonexp\,(\mathtt{c}) \wedge nonexp\,(\mathtt{e}_1) \wedge nonexp\,(\mathtt{e}_2)$
$nonexp\,(\mathtt{e}_1\,;\ \mathtt{e}_2) ::= nonexp\,(\mathtt{e}_2)$
$nonexp\,(\mathbf{lazy}\ \mathtt{e}) ::= nonexp\,(\mathtt{e})$
$nonexp\,(\mathbf{let}\ \mathbf{module}\ \mathtt{I} = \mathtt{M}\ \mathbf{in}\ \mathtt{e}) ::=$
$\quad nonexp\_mod\,(\mathtt{M}) \wedge nonexp\,(\mathtt{e})$
$nonexp\,((\mathbf{module}\ \mathtt{M})) ::= nonexp\_mod\,(\mathtt{M})$
$nonexp\,(\_) ::= \bot$

FIGURE 4.7 – *Valeurs syntaxiques, ou non expansives Critère de la restriction souple de valeur.*

celle-ci peut être généralisé. Dans le cas contraire, toutes les variables de types fraîches, *i.e.* qui n'apparaissaient pas déjà dans l'environnement courant, ne peuvent être généralisées et devront être monomorphisées durant le typage de la suite du programme. Ainsi, on peut considérer que toute apparition de variable de type non généralisée dans un programme OCaml comme étant un type mal formé.

OCaml utilise une version moins stricte : la restriction souple de valeur[29]. Ici, on considère qu'il est possible de généraliser les variables de types qui n'apparaissent pas de manière contravariante. Par exemple, dans le type $\alpha \to \beta$ classifiant une expression expansive, la variable α n'est pas généralisable car elle est directement contravariante. Dans le cas $(\alpha \to \mathtt{int}) \to \mathtt{int}$, bien que α soit covariant, cette variable apparaît dans un type qui apparaît lui-même de manière contravariant. Finalement, dans le cas de l'identité $\alpha \to \alpha$, α apparaît à la fois covariante et contravariante, et ne peut donc être généralisée. En d'autres termes, une variable qui n'apparaît pas à gauche d'une flèche peut être généralisée, que la valeur soit expansive ou non.

Pour gérer le cas des références, on considère que le paramètre de types `'a` du constructeur **`ref`** est toujours contravariant. La représentation des références étant liées à celle des enregistrements, ce point sera donc détaillé en section 4.3.4.

Généralisation et niveaux Pour gérer la généralisation, l'inférence d'OCaml utilise les niveaux. Ceux-ci ne sont pas utilisés de la même manière dans l'inférence et la vérification : la vérification vérifie simplement si le niveau correspond à une variable généralisée ou non. Leur fonctionnement est expliqué section 3.6.1. Leur utilisation nécessitant de reproduire le comportement du moteur d'inférence, notre approche est beaucoup plus académique : l'environnement contient, à chaque expression, l'ensemble des variables déjà généralisées, puisqu'elles sont également nécessaires pour l'instanciation. Ainsi, il est relativement simple de reproduire un algorithme de vérification de la généralisation.

L'algorithme décrit figure 4.8 est relativement simple. Il s'agit de considérer deux cas : l'expression dont le type doit être généralisé peut être considéré comme expansif ou non. Dans le cas non expansif, il s'agit tout d'abord de collecter l'ensemble des variables apparaissant dans ce type mais qui n'apparaissent pas déjà dans l'environnement, via la fonction `isvar_nonexpans` : il s'agit donc *a priori* des variables quantifiées par ce type. Le parcours du type se fait à l'aide du combinateur *fold* décrit en section 4.2.5. Enfin, pour chacune de ces variables, si celle-ci n'est pas généralisée on retourne alors une erreur.

Si l'expression est expansive, *i.e.* si elle peut se réduire, l'algorithme doit collecter toutes les variables de types qui n'apparaissent pas dans l'environnement, et leur associer une position : covariante ou contravariante. Il y a trois cas particuliers à distinguer pendant le parcours du type à généraliser :

— Si le type à analyser est une variable de type, alors, si celle-ci a déjà été collectée et que la position courante n'est pas contravariante, il n'est pas nécessaire de la rajouter dans l'environnement. Dans le cas contraire, elle doit être ajoutée. Par conséquent, si la variable est déjà présente mais que la position courante est covariante, l'ajout signifie que la position qui lui est déjà associée sera mise à jour pour indiquer qu'elle apparaît de manière contravariante.

— Si le type est un type de fonction, les variables du domaine seront récupérées avec pour position `Contravariant`. Celles du codomaine seront collectées avec la position courante.

— Si le type est un constructeur de type, alors ses paramètres de types peuvent chacun posséder une variance différente. Celle-ci est récupérée depuis l'environnement, et si elle est contravariante, transférée au sous-nœud correspondant.

Le reste de l'algorithme étant un simple parcours du type en ordre préfixe, il n'est pas nécessaire de rajouter du bruit à un pseudo-code déjà encombré en le décrivant. Une fois ces variables et leur position récupérées, il suffit de s'assurer que toutes les variables apparaissant contravariantes ne sont pas généralisées, et que les variables qui sont à une position covariante sont au contraire généralisées.

Cet algorithme néanmoins est plus général que la vérification le nécessite : en effet, il considère que des variables non généralisées peuvent apparaître dans un programme, là où la vérification de bonne formation considère que toute variable de types non généralisée est une erreur.

4.3.3 Types algébriques gardés

Comme expliqué précédemment, les types algébriques gardés sont une forme s'inspirant des systèmes de types dépendants, permettant une plus grande expressivité ainsi que d'exprimer des

```
let isvar_nonexpans (Γ, Φ) 𝒯 τ =
  if Γ,Φ ⊢_τ 'a ≺ τ ∧ τ ∉ Γ.𝒱ars then
    𝒯 ⊕ τ
  else
    𝒯

let aggregate_nonexpans (Γ, Φ) τ =
  fold (isvar_nonexpans (Γ, Φ)) ∅ τ

let aggregate_expans (Γ, Φ) 𝒯 pos τ =
  if Γ,Φ ⊢_τ 'a ≺ τ then
    if τ ∈ Γ.𝒱ars ∨ (τ ∈ 𝒯 ∧ pos ≠ Contravariant) then
      𝒯
    else
      𝒯 ⊕ pos
  else if Γ,Φ ⊢_τ τ1 → τ2 ≺ τ then
    let 𝒯' =
      aggregate_expans (Γ, Φ) 𝒯 Contravariant τ in
    aggregate_expans (Γ, Φ) 𝒯' pos τ
  else if Γ,Φ ⊢_τ τ̅_arg t ≺ τ then
    list_fold_indexed_left
      (fun 𝒯 index τ_arg →
        let pos =
          if get_variance (Γ, Φ) index t = Contravariant
          then Contravariant
          else pos in
        aggregate_expans (Γ, Φ) 𝒯 pos τ_arg)
      𝒯
      τ̅_arg
  else
    [...]

let check_gen (Γ, Φ) τ e =
  if nonexp e then
    let 𝒯 = aggregate_nonexpans (Γ, Φ) τ in
    TypeSet.iter (fun τ →
      if not (generalized τ) then Error) 𝒯
  else
    let 𝒯 = aggregate_expans (Γ, Φ) ∅ Covariant τ in
    TypeMap.iter (fun τ pos ->
      if generalized τ ∧ pos = Contravariant then Error
      else if not (generalized τ) ∧ pos = Covariant
      then Error) 𝒯
```

FIGURE 4.8 – Algorithme de vérification de généralisation

invariants sur certaines valeurs qui seront vérifiés par le système de types. On peut alors reprendre le type donné en section 4.2.5, et écrire une fonction qui n'acceptera que des valeurs de type `int t` :

```
let ⟨ f : int t → int ⟩ = function Int -> 0
```

Ici, le type de la fonction a été inféré comme étant `int t → int`. On remarque d'ailleurs que bien qu'il n'y ait qu'une seule branche, le typeur d'OCaml considère que le filtrage est complet : en effet, le constructeur `Int` est la seule valeur de type `int t`. Rajouter une branche `Bool` avec le type inféré ici ne serait d'ailleurs pas correct, puisque ce constructeur serait de type `bool t`. Cet exemple montre l'un des problèmes de l'utilisation de GADTs en présence d'inférence de types : le type inféré ne sera pas toujours le plus général. C'est cette restriction de l'inférence de types qui oblige le programmeur à contraindre explicitement des types à être polymorphes. Ainsi, pour qu'une fonction contenant les deux branches soit acceptée, il faut l'annoter avec le type suivant :

```
let f (type a) : a t -> int = function
  Int -> 0
| Bool -> 1
```

D'abord, on introduit un type abstrait local `a` : celui-ci est utilisé comme paramètre de type pour l'argument de type `a t`. Durant le filtrage, `a` est raffiné par le type du constructeur gardé, autrement dit, dans le cas de la première branche il sera strictement équivalent à `int`, et `bool` dans la seconde. Cette équivalence est locale à la branche dans laquelle la variable a été raffinée.

Cette particularité permet aux constructeurs gardés d'exprimer des équivalences de types au sein de leur définition. Dans le type précédent, le type de chaque constructeur impliquait une type concret pour le paramètre de type. Ainsi, un constructeur `Int` ne pourra avoir d'autre type que `int t`, et une valeur de type `int t` ne pourra être que le constructeur `Int`. Il est tout à fait possible d'écrire une fonction de type `string t → int` :

```
let g : string t -> int = fun x -> 2
```

mais qui ne pourrait être appelée, puisqu'il est impossible de produire une valeur de type `string t`. Le type ne doit d'ailleurs pas être considéré comme mal formé, puisqu'il n'existe aucune contrainte sur la variable de type.

Un exemple intéressant de cette particularité est celui du constructeur `Eq`, qui permet de manipuler des témoins d'égalité de types :

```
type (_, _) eq = Eq : ('a, 'a) eq
```

Ainsi, pour tous types `t` et `t'` et pour toute valeur de type `(t, t') eq`, cette valeur est un témoin que les type `t` et `t'` sont équivalents. On peut alors écrire la fonction suivante :

```
let cast (type a) (type b) : (a, b) eq -> a -> b =
  fun eq v ->
    match eq with Eq -> v
```

Étant donné le type de `eq`, le filtrage sur celui-ci ne peut avoir pour forme que celui du constructeur `Eq`. Son type suppose que les deux paramètres de type sont équivalents : `a` et `b` sont donc équivalents. Ainsi, sachant que `v` est de type `a`, on peut le retourner avec le type `b` de manière sûre.

Quantification existentielle Une autre particularité des constructeurs gardés est que chaque constructeur possède sa propre quantification. Ainsi, la déclaration de constructeur

```
Eq : ('a, 'a) eq
```

pourrait être lue comme `Eq : ∀ 'a. ('a, 'a) eq`. Par conséquent, il est tout à fait possible de quantifier plus de variables que le nombre de paramètres de types déclarés. Il est alors possible de représenter des types existentiels :

```
type ex =
  Ex : 'a -> ex
```

Ainsi, une fois qu'une valeur est *empaquetée* dans un constructeur `Ex`, son type est oublié. En effet, le type de l'argument n'apparaît pas dans le type du constructeur. La valeur peut ensuite être *dépaquetée* à l'aide d'un filtrage :

```
match ex with Ex v -> ...
```

Dans ce cas, puisqu'il n'existe aucun type pour représenter celui de `v`, le typeur doit lui générer un type qu'on appellera "existentiel", représenté par un constructeur de types. Ce type n'apparaît pas au programmeur, il est généré par le compilateur pour lui-même, pour pouvoir classifier et annoter cette valeur. Cette représentation de valeur existentielle n'est bien évidemment pas utilisable en pratique, puisqu'il est impossible de retirer une quelconque information de la valeur empaquetée. Une version utilisable, par exemple, serait un constructeur de type `Ex : 'a * ('a -> view) -> ex`, où `view` pourrait être une représentation sous une forme quelconque de la valeur de l'argument empaqueté. Le constructeur `Ex` embarquerait alors une valeur et une fonction associée capable de "lire" une valeur de ce type. Il s'agit exactement du fonctionnement des types existentiels tels que décrits dans la littérature des systèmes de types. Une telle représentation des valeurs de type existentiel pourrait, par exemple, être utilisée pour implémenter des listes hétérogènes.

Finalement, comme évoqué dans la section sur la vérification d'équivalences, l'utilisation de GADTs permet d'exprimer des fonctions dont le type de retour pourra être dépendant de la valeur du constructeur. Par exemple, une telle fonction est valide :

```
let h (type a) : a t -> a = function
  Int -> 0
| Bool -> true
```

Ici, le type de retour de la fonction est lié au type du paramètre de type. Dans le premier cas, `a` est équivalent à `int`, il est donc possible de retourner un entier. Dans le second cas, `a` est équivalent à `bool`, on peut donc retourner une expression booléenne. Cette fonction est sûre, puisque le constructeur donné en argument donnera le type attendu pour retour de la fonction. La contrainte `a t` → `a` est nécessaire pour l'inférence, pour *forcer* le type de chaque branche à être de type `a`, si on se réfère à la règle de vérification de l'abstraction ABS (ou plus globalement celle du filtrage MATCH). Du point de vue interne à la branche, le type `a` est raffiné pour devenir équivalent à un type concret. D'un point de vue externe, sans information sur les équivalences générées, `a` est abstrait et donc, si toutes les branches sont annotées avec ce type, alors le filtrage est correctement typé.

Vérification de types algébriques gardés

Comme on peut le supposer d'après la définition des types gardés donnée précédemment, leur vérification va engendrer un certain nombre de difficultés. On peut distinguer plusieurs étapes de vérification :

— lors de la déclaration du constructeur, principalement pour extraire les variables quantifiées existentiellement;
— lors du filtrage, pour générer les équations de types;
— lors du filtrage, pour extraire les constructeurs de types créés par le compilateur pour représenter la quantification existentielle.

La vérification des déclarations de types doit s'assurer de deux choses. D'abord, le type de retour du constructeur, puisqu'il est explicite, doit être une instance du constructeur de type déclaré. Prenons l'exemple suivant :

```
type 'a t = L : 'b list t
  constraint 'a = _ option
```

Ici le type de `L` est `'a list t`. Or, l'annotation `constraint` raffine le paramètre de type `'a` à être une instance du type `option`. Par conséquent, `'b list t` ne peut être une application correcte du constructeur, et doit être rejetée. Ensuite, comme expliqué section 4.3.3, certaines variables peuvent être quantifiées existentiellement. Dans la structure de données du compilateur qui représente les déclarations de constructeurs de données, les variables existentielles sont explicitement indiquées. Il faut donc pouvoir les vérifier, en d'autres termes extraire les variables qui apparaissent dans la déclaration d'un constructeur gardé mais pas dans son "type de retour", et s'assurer qu'il s'agit bien des variables indiquées par le compilateur. L'algorithme est décrit dans la figure 4.9

La génération et vérification des équations est décrite section 4.2.5.

L'algorithme d'extraction des constructeurs dits *existentiels* durant le filtrage est donné en figure 4.9. L'algorithme est le suivant : sachant le type des arguments, le type du constructeur dans la définition et celui de ses arguments, on peut extraire les constructeurs existentiels générés par le typeur de la manière suivante :

1. Les variables quantifiées universellement sont extraites, c'est-à-dire les variables qui apparaissent dans le type du constructeur.

2. Sachant celles-ci et le type des arguments dans la définition, on peut retrouver les variables quantifiées existentiellement, c'est-à-dire l'ensemble des variables qui n'apparaissent pas dans le type de retour du constructeur.

3. Finalement, il suffit de comparer le type défini pour les arguments avec celui en lequel il est instancié : si la variable a été considérée comme étant existentielle, alors son instance est un constructeur de type généré pour la représenter.

Enfin, la règle de vérification des motifs de constructeur est donnée dans la figure 4.10. On commence d'abord par vérifier les motifs correspondant aux arguments, en récupérant l'ensemble des nouvelles classes d'équivalences, types existentiels et variables extraits. Il faut ensuite s'assurer que le constructeur donné correspond bien au type auquel il est annoté dans l'environnement, et récupérer le type de ses arguments, son type ainsi qu'un booléen indiquant si ce constructeur

```
let is_universal Univ τ =
  if τ ≺ 'a then
    Univ ⊕ τ
  else
    Univ

let find_universals τ_ret =
  fold is_universal ∅ τ_ret

let find_existential (Exs, Univ) τ_arg =
  if τ_arg ≺ 'a ⋀ τ_arg ∉ Univ then
    (Exs ⊕ τ_arg, Univ)
    else (Exs, Univ)

let find_existentials (Exs, Univ) τ_arg =
  fold find_existential (Exs, Univ) τ_arg


let add_existential (𝒯, Exs) τ_1 τ_2 =
  if τ_2 ≺ 'a ⋀ τ_2 ∈ Exs then
    let τ_∃ = τ_1 ≺ t in
    (𝒯 ⊕ τ_∃, Exs)
  else
    (𝒯, Exs)

let add_existential (𝒯, Exs) τ_1 τ_2 =
  fold2 add_existential (𝒯, Exs) τ_1 τ_2


let existential_types τ_args τ_ret τ_def generalized =
  if generalized then
    let Univ = find_universals τ_ret in
    let Exs, _ = find_existentials (∅, Univ) τ_def in
    let 𝒯, _ = add_existentials (∅, Exs) τ_args τ_def in
    𝒯
  else
    ∅
```

FIGURE 4.9 – Algorithme d'extraction des types existentiels

$$\text{PAT-CONSTRUCT}\ \frac{\begin{array}{c}\Gamma, \Phi, \mathcal{V}, \mathcal{T} \models_{\rho} \langle p_1 : \tau_1 \rangle \Rightarrow \mathcal{V}_1, \mathcal{T}_1, \Phi_1 \\ \forall i_{>1}.\ \Gamma, \Phi_{i-1}, \mathcal{V}_{i-1}, \mathcal{T}_{i-1} \models_{\rho} \langle p_i : \tau_i \rangle \Rightarrow \mathcal{V}_i, \mathcal{T}_i, \Phi_i \\ let\ (\tau_{arg_1}, \ ..\ , \tau_{arg_n}, \tau_{constr}, generalized) = find\_constructor(\Gamma, \Phi, T, \tau) \\ let\ \mathcal{T}_{args} = existential\_types((\tau_1 * \ ..\ * \tau_n), \tau_{constr}, (\tau_{arg_1} * \ ..\ * \tau_{arg_n}), generalized) \\ \Gamma, \Phi_n, \Sigma \models_{\tau} \tau_1 \leq \tau_{arg_1} \Rightarrow \theta_1 \qquad \forall i_{>1}.\ \Gamma, \Phi_n, \theta_{i-1} \models_{\tau} \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \\ \forall i.\ let\ \Phi_{p_i} = add\_equations((\Gamma, \Phi_{p_{i-1}}), \tau_i, \tau_{arg_i}) \\ let\ \Phi_{ret} = add\_equations((\Gamma, \Phi_{p_n}), \tau, \tau_{constr}) \qquad \Gamma, \Phi_{p_n}, \theta_{\forall}(\Gamma) \models_{\tau} \tau \leq \tau_{constr}\end{array}}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \models_{\rho} \langle T(p_1, \ ..\ , p_n) : \tau \rangle \Rightarrow \mathcal{V}_n, \mathcal{T}_n, \Phi_{ret}}$$

FIGURE 4.10 – Vérification des motifs de constructeurs

$$\text{TUPLE}\ \frac{\forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau'_i \rangle \qquad \Gamma, \Phi \models_{\tau} \tau \prec \tau_1 \ * \ ..\ * \ \tau_n \qquad \forall i. \Gamma, \Phi \models_{\tau} \tau_i \equiv \tau'_i}{\Gamma, \Phi \vdash \langle (e_1, \ ..\ ,\ e_n) : \tau \rangle}$$

$$\text{PAT-TUPLE}\ \frac{\begin{array}{c}\Gamma, \Phi \models_{\tau} \tau \prec \tau_{p_1} * \ ..\ * \tau_{p_n} \qquad \Gamma, \Phi, \mathcal{V}, \mathcal{T} \models_{\rho} \langle p_1 : \tau_1 \rangle \Rightarrow \mathcal{V}_1, \mathcal{T}_1 \\ \forall i_{>1}.\ \Gamma, \Phi_{i-1}, \mathcal{V}_{i-1}, \mathcal{T}_{i-1} \models_{\rho} p_i : \tau_i \Rightarrow \mathcal{V}_i, \mathcal{T}_i, \Phi_i \qquad \forall i.\ \Gamma, \Phi_i \models_{\tau} \tau_{p_i} \equiv \tau_i\end{array}}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \models_{\rho} \langle p_1, \ ..\ , p_n : \tau \rangle \Rightarrow \mathcal{V}_n, \mathcal{T}_n, \Phi_n}$$

FIGURE 4.11 – Vérification des tuples

est gardé. Pour chacun des sous-motifs, on vérifie que le type annoté est une instance du type de la définition, puis on s'assure que le type annoté est une instance du type défini pour le constructeur. Finalement, on extrait l'ensemble des nouvelles équations générées pour les arguments.

4.3.4 Extensions

ML constitue le coeur d'OCaml, on peut néanmoins compter un certain nombre d'extensions, qu'il s'agisse d'extensions impératives ou fonctionnelles.

Extensions fonctionnelles

On note comme extensions fonctionnelles les plus évidentes les n-uplets et les types algébriques.

La vérification de n-uplets (figure 4.11) se fait de manière simple. Il s'agit de vérifier chacune des sous-expressions qui constituent le n-uplet, de s'assurer que le type annoté est bien un type de tuple de la même arité que l'expression et que chaque type le composant est bien équivalent à celui annoté à la sous-expression correspondante.

$$
\textsc{Construct}\;
\dfrac{\begin{array}{c}
\forall i.\, \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \mathit{let}\ (\tau_{arg_1}, \;..\;, \tau_{arg_n}, \tau_{constr}) = \mathit{find\_constructor}(\Gamma, \Phi, T, \tau) \\
\Gamma, \Phi, \emptyset \vdash_{\tau} \tau_1 \leq \tau_{arg_1} \Rightarrow \theta_1 \\
\forall i_{>1}.\, \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \qquad \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{constr}
\end{array}}{\Gamma, \Phi \vdash \langle T(e_1, \;..\;, e_n) : \tau \rangle}
$$

$$
\textsc{Match}\;
\dfrac{\begin{array}{c}
\Gamma, \Phi \vdash \langle e : \tau_{scrut} \rangle \qquad \forall i.\, \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\
\forall i.\, \Gamma, \Phi \vdash_{\tau} \tau_{scrut} \equiv \tau_i \qquad \forall i.\, \mathit{let}\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}}) \\
\forall i.\, \Gamma_i, \Phi_i \vdash \langle e_i : \tau_{e_i} \rangle \qquad \forall i.\, \Gamma, \Phi \vdash_{\tau} \tau_{e_i}\ \mathit{wf} \qquad \forall i.\, \Gamma_i, \Phi_i \vdash_{\tau} \tau \equiv \tau_{e_i}
\end{array}}{\Gamma, \Phi \vdash \langle \mathtt{match}\ e\ \mathtt{with}\ \overline{\mid p \to e} : \tau \rangle}
$$

FIGURE 4.12 – Vérification des constructeurs de données et du filtrage

$$
\textsc{Record}\;
\dfrac{\begin{array}{c}
\mathit{let}\ \tau_{rec}, \mathcal{L} = \mathit{find\_record}(\Gamma, \Phi, \tau) \qquad \mathit{check\_unicity}(\mathcal{L}, \{l_1, .., l_n\}) \qquad \forall i.\, \mathit{let}\ \tau_{l_i} = \mathcal{L}(i) \\
\forall i.\, \Gamma, \Phi \vdash \langle e : \tau_i \rangle \qquad \Gamma, \Phi, \emptyset \vdash_{\tau} \tau_1 \leq \tau_{l_1} \Rightarrow \theta_1 \qquad \forall i_{>1}.\, \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \\
\forall i_{\geq 1}.\, \mathit{check\_gen\_record}(\Gamma, \Phi, \tau_{l_i}, \tau_i, e_i) \qquad \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{rec}
\end{array}}{\Gamma, \Phi \vdash \langle \{l_1 = e_1;\; .. \;; l_n = e_n\} : \tau \rangle}
$$

$$
\textsc{Field}\;
\dfrac{\begin{array}{c}
\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \mathit{let}\ \tau_{rec}, \mathcal{L} = \mathit{find\_record}(\Gamma, \Phi, \tau_e) \\
l \in \mathit{dom}(\mathcal{L}) \qquad \mathit{let}\ \tau_l = \mathcal{L}(l) \qquad \Gamma, \Phi, \Sigma \vdash_{\tau} \tau \leq \tau_l \Rightarrow \theta \qquad \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau_e \leq \tau_{rec}
\end{array}}{\Gamma, \Phi \vdash \langle e.l : \tau \rangle}
$$

FIGURE 4.13 – Vérification des enregistrements

La vérification du filtrage (figure 4.12) est identique à un détail près à la règle ABS : il faut s'assurer de la validité de la sous-expression sur laquelle le filtrage est effectué.

Pour qu'un constructeur de donnée soit valide (règle CONSTRUCT), il faut d'abord s'assurer que chacune de ses sous-expressions l'est. Ensuite, le constructeur doit exister avec le type auquel il est annoté dans l'environnement. L'appel à la fonction *find_constructor* expanse le type et cherche dans les équations courantes si nécessaire pour trouver un constructeur de type dont la définition est un type algébrique, et retourne le type de retour du constructeur ainsi que le type de ses arguments. Les types retournés étant potentiellement polymorphes, on vérifie que l'ensemble des arguments du constructeur constituent bien une instance des types attendus. Finalement, on vérifie que le type annoté est une instance du type attendu.

Type enregistrement

Les enregistrements, au même titre que les types algébriques, doivent avoir été préalablement déclarés. Dans le cas contraire, on ne peut leur donner de type. Le type des objets peut être

```
let rec check_unicity (𝒟, 𝒫) =
match 𝒟, 𝒫 with
  | ∅, ∅ -> ⊤
  | ∅, _ | _, ∅ -> ⊥
  | _, _ ->
    let l, 𝒫′ = take 𝒫 in
    if l ∉ dom(𝒟) then ⊥
    else
      let 𝒟′ = remove (l, 𝒟) in
      check_unicity (𝒟′, 𝒫′)

let rec check_atmost (𝒟, 𝒫) =
  match 𝒟, 𝒫 with
  | ∅, ∅ | _, ∅ -> ⊤
  | ∅, _ -> ⊥
  | _, _ ->
    let l, 𝒫′ = take 𝒫 in
    if l ∉ dom(𝒟) then ⊥
    else
      let 𝒟′ = remove (l, 𝒟′) in
      check_atmost (𝒟′, 𝒫′)
```

FIGURE 4.14 – Vérification de la présence et de l'unicité de l'ensemble des champs d'un enregistrement

vu comme une version polymorphe de ceux-ci. Ainsi, la vérification de la création d'un enregistrement (figure 4.13) va d'abord s'assurer que le type annoté apparait dans l'environnement, et récupérer l'ensemble des champs associés à leur type. L'algorithme de vérification doit ensuite vérifier que les champs sont tous définis et n'apparaissent jamais en double : il s'agit de la fonction `check_unicity` (figure 4.14). Cette vérification est simple : on parcourt les champs récupérés depuis la définition ($\mathcal{D}$) et les champs présents ($\mathcal{P}$) simultanément :

— Si les deux sont vides, alors la propriété est vraie.
— S'il reste des champs dans $\mathcal{D}$, *i.e.* si des champs n'ont pas été déclarés, la propriété est fausse.
— S'il reste des champs dans $\mathcal{P}$, la propriété est fausse. Il y a alors deux raisons possibles :
  — Un champ a été déclaré mais n'existe pas dans la définition.
  — Un champ a été déclaré deux fois dans le même enregistrement.
— Sinon, on récupère un des champs `l` de $\mathcal{P}$, tel que $\mathtt{l} + \mathcal{P}' = \mathcal{P}$. Si le champ n'apparaît pas dans $\mathcal{D}$, cela signifie qu'il n'est pas défini pour cet enregistrement (ou qu'il a déjà été déclaré) et donc que la propriété n'est pas vérifiée. Sinon, on l'enlève de $\mathcal{D}$ (on obtient $\mathcal{D}'$), pour vérifier qu'il n'est pas déclaré une nouvelle fois, et on rappelle la propriété sur $\mathcal{D}'$ et $\mathcal{P}'$.

La vérification est ensuite équivalente à celle des types algébriques non gardés. L'accès aux champs est similaire : il s'agit d'abord de vérifier l'expression qui doit être l'enregistrement, puis de récupérer le type du champ. Il faut ensuite vérifier pour chaque déclaration de champ que le type de l'expression est bien une instance de sa définition, et que le type de l'enregistrement est

$$
\textsc{Record-Copy}\ \frac{\begin{array}{c}
\mathit{let}\ \tau_{rec}, \mathcal{L} = \mathit{find\_record}(\Gamma, \Phi, \tau) \qquad \Gamma, \Phi \vdash \langle o : \tau_o \rangle \\
\Gamma, \Phi, \emptyset \vdash_{\tau} \tau_o \leq \tau_{rec} \Rightarrow \theta_o \qquad \mathit{let}\ \mathcal{L}_o = \mathit{extract\_labels}(\Gamma, \Phi, \tau_o) \\
\mathit{let}\ \mathit{Copied} = \mathcal{L}_o - \{l_1;\ ..\ ; l_k\} \qquad \mathit{check\_unicity}(\mathcal{L}, \mathit{Copied} + \{l_1;\ ..\ ; l_k\}) \\
\forall i_{i \in [0..k]}.\ \mathit{let}\ \tau_{l_i} = \mathcal{L}(l_i) \qquad \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \Gamma, \Phi, \emptyset \vdash_{\tau} \tau_1 \leq \tau_{l_1} \Rightarrow \theta_1 \\
\forall i_{>1}.\ \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \qquad \forall i_{\geq 1}.\ \mathit{check\_gen\_record}(\Gamma, \Phi, \tau_{l_i}, \tau_i, e_i) \\
\forall i_{l_i \in \mathit{Copied}}.\ \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \mathit{Copied}(l_i) \leq \text{Ł}(l_i) \Rightarrow \theta_i \qquad \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{rec}
\end{array}}{\Gamma, \Phi \vdash \langle \{o\ \mathbf{with}\ l_1 = e_1;\ ..\ ; l_k = e_k\} : \tau \rangle}
$$

FIGURE 4.15 – Vérification des copies d'enregistrements

$$
\textsc{Set-Field}\ \frac{\begin{array}{c}
\Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \qquad \mathit{let}\ \tau_{rec}, \mathcal{L} = \mathit{find\_record}(\Gamma, \Phi, \tau_e) \\
l \in \mathit{dom}(\mathcal{L}) \qquad \mathit{let}\ \tau_l = \mathcal{L}(l) \qquad \mathit{label\_kind}(\Gamma, \Phi, l, \tau_{rec}) = \mathit{Mutable} \\
\Gamma, \Phi, \emptyset \vdash_{\tau} \tau_2 \leq \tau_l \Rightarrow \theta \qquad \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau_1 \leq \tau_{rec} \qquad \Gamma, \Phi \vdash_{\tau} \tau \equiv \mathtt{unit}
\end{array}}{\Gamma, \Phi \vdash \langle e_1.l \leftarrow e_2 : \tau \rangle}
$$

FIGURE 4.16 – Vérification de l'assignation d'un champ d'enregistrement

correctement instancié. En revanche, il faut s'assurer que les champs polymorphes sont correctement généralisés. La fonction $\mathit{check\_gen\_record}(\Gamma, \Phi, \tau_l, \tau, e)$ va vérifier la généralisation de τ en fonction de e si et seulement si τ_l est un type de la forme $\forall \overline{\alpha}.\ \tau'$.

Il existe un cas plus général de la déclaration d'enregistrements, que l'on nommera la copie : il s'agit de déclarer partiellement un enregistrement, et laisser le compilateur remplir les champs manquants depuis un enregistrement existant. La règle de vérification est donnée figure 4.15. Si celle-ci est similaire à RECORD (figure 4.13), la principale différence réside dans la vérification des types des champs hérités depuis l'enregistrement original. Il faut alors vérifier l'enregistrement hérité, vérifier qu'il s'agisse bien d'un enregistrement et récupérer l'ensemble de ses champs avec leurs types instanciés. Ensuite, pour chacun des champs redéfinis, le type de sa valeur est vérifiée, et une instance entre la définition et le type annoté est calculée. Finalement, on vérifie que chacun des champs hérités est correctement instancié.

Les enregistrements présentés précédemment permettent d'introduire de la mutabilité dans le langage. Ainsi, il est possible de déclarer certains champs comme étant mutables. Le type des références `'a ref` est d'ailleurs un enregistrement ne contenant qu'un seul champ `contents` mutable. Tout paramètre de type apparaissant dans un champ mutable est considérée comme contravariant.

La vérification de l'assignation d'un champ (figure 4.16) est similaire à celle décrite pour l'accès. La principale différence est de s'assurer que le nœud de la valeur assignée est cohérente, son type équivalent au champ, mais aussi que celui-ci est bien décrit comme mutable. Finalement, le type résultant d'une assignation doit être équivalent à `unit`.

$$\textsc{Pat-Record}\ \frac{\begin{array}{c} let\ \tau_{rec}, \mathcal{L} = find\_record(\Gamma, \Phi, \tau) \qquad check\_atmost(\mathcal{L}, \{l_1, .., l_n\}) \\ l_i \in dom(\mathcal{L}) \qquad \forall i.\ let\ \tau_{l_i} = \mathcal{L}(l_i) \qquad \Gamma, \Phi, \emptyset, \emptyset \vdash_{\rho} \langle p_1 : \tau_1 \rangle \rightarrow \mathcal{V}_1, \mathcal{T}_1, \Phi_1 \\ \forall i_{>1}.\ \Gamma, \Phi_{i-1}, \mathcal{V}_{i-1}, \mathcal{T}_{i-1} \vdash_{\rho} \langle p_i : \tau_i \rangle \Rightarrow \mathcal{V}_i, \mathcal{T}_i, \Phi_i \qquad \Gamma, \Phi, \emptyset \vdash_{\tau} \tau_1 \leq \tau_{l_1} \Rightarrow \theta_1 \\ \forall i_{>1}.\ \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \qquad \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{rec} \end{array}}{\Gamma, \Phi \vdash_{\rho} \langle \{l_1 = p_1;\ ..\ ; l_n = p_n\} : \tau \rangle}$$

FIGURE 4.17 – Vérification de motifs d'enregistrement

$$\textsc{Sequence}\ \frac{\Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_1 \equiv \mathtt{unit} \qquad \Gamma, \Phi \vdash_{\tau} \tau \equiv \tau_2}{\Gamma, \Phi \vdash \langle e_1;\ e_2 : \tau \rangle}$$

$$\textsc{While}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \\ \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_1 \equiv \mathtt{bool} \qquad \Gamma, \Phi \vdash_{\tau} \tau_2 \equiv \mathtt{unit} \qquad \Gamma, \Phi \vdash_{\tau} \tau \equiv \mathtt{unit} \end{array}}{\Gamma, \Phi \vdash \langle \mathbf{while}\ e_1\ \mathbf{do}\ e_2\ \mathbf{done} : \tau \rangle}$$

$$\textsc{For}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_1 \equiv \mathtt{int} \qquad \Gamma, \Phi \vdash_{\tau} \tau_2 \equiv \mathtt{int} \\ \Gamma \oplus_{\mathcal{V}} (x, \tau_1), C.\Phi \vdash \langle e_3 : \tau_3 \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_3 \equiv \mathtt{unit} \qquad \Gamma, \Phi \vdash_{\tau} \tau \equiv \mathtt{unit} \end{array}}{\Gamma, \Phi \vdash \langle \mathbf{for}\ x = e_1\ \mathbf{to}\ e_2\ \mathbf{do}\ e_3\ \mathbf{done} : \tau \rangle}$$

FIGURE 4.18 – Vérification des instructions impératives

Finalement, il est possible de filtrer sur des enregistrements (PAT-RECORD, figure 4.17). La vérification d'un motif d'enregistrement est similaire à celle d'une déclaration d'enregistrement. La principale différence étant que tous les champs n'ont pas à apparaître : il faut simplement s'assurer que pour chaque champ filtré celui-ci appartient bien à l'enregistrement et n'est pas déclaré plus d'une fois. L'algorithme de `check_atmost` qui vérifie cette propriété est donné figure 4.14.

Extensions impératives

OCaml étant un langage multi-paradigme, il est possible d'écrire un programme de manière impérative. Ainsi, le langage emprunte certaines constructions aux langages impératifs, tels que la séquence, permettant de représenter une suite d'instructions, ou les boucles conditionnelles.

Ces trois règles en figure 4.18 sont relativement simples et supposent simplement des équivalences de types. Dans le cas de la séquence, le premier nœud doit être annoté avec un type équivalent à `unit`[9] et le second à celui de la séquence. Le type de l'expression **while** doit tou-

9. ∧Il s'agit d'une restriction plus forte que le typeur d'OCaml, qui n'envoie qu'un avertissement dans le cas

$$\textsc{Raise}\ \frac{\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau_e \equiv \texttt{exn}}{\Gamma, \Phi \vdash \langle \textbf{raise}\ e : \tau \rangle}$$

$$\textsc{TryWith}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau \equiv \tau_e \qquad \forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ \forall i.\ \Gamma, \Phi \vdash_{\mathcal{T}} \texttt{exn} \equiv \tau_i \qquad \forall i.\ let\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}}) \\ \forall i.\ \Gamma_i, \Phi_i \vdash \langle e_i : \tau_{e_i} \rangle \qquad \forall i.\ \Gamma, \Phi \vdash_{\mathcal{T}} \tau_{e_i}\ wf \qquad \forall i.\ \Gamma_i, \Phi_i \vdash_{\mathcal{T}} \tau \equiv \tau_{e_i}\end{array}}{\Gamma, \Phi \vdash \langle \textbf{try}\ e\ \textbf{with}\ \overline{\mid p \to e} : \tau \rangle}$$

$$\textsc{Assert}\ \frac{\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau_e \equiv \texttt{bool} \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau \equiv \texttt{unit}}{\Gamma, \Phi \vdash \langle \textbf{assert}\ e : \tau \rangle}$$

$$\textsc{Assert-False}\ \frac{\Gamma, \Phi \vdash \langle false : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau_e \equiv \texttt{bool}}{\Gamma, \Phi \vdash \langle \textbf{assert}\ false : \tau \rangle}$$

FIGURE 4.19 – Vérification des constructions liées aux exceptions

jours être équivalent à `unit`, tout comme le type annoté de son corps. La condition quand à elle doit être booléenne, et donc annotée avec un type équivalent à `bool`. Finalement, l'expression **`for`** diffère de **`while`** dans le sens où la condition permet d'introduire un identifiant dont le type sera équivalent à `int`.

Exceptions

Les exceptions permettent d'interrompre le cours normal d'un programme. Une fois l'exception levée (`raise ...`), l'exécution courante s'arrête jusqu'à être rattrapée par un gestionnaire qui pourra récupérer la valeur contenue par celle-ci (`try .. with Exn v -> ..`).

La levée d'exception peut être annotée avec n'importe quel type. Effectivement, cette construction a pour but de se substituer à un calcul où il n'est pas possible de donner une valeur, il faut donc que l'instruction d'échappement possède le type de cette valeur pour être vérifiable. La vérification est alors simple : il faut vérifier que l'expression est cohérente, et que son type est équivalent à `exn`, le type représentant les exceptions.

La garde permettant de rattraper des exceptions, `try e with E -> ..`, permet de calculer une expression `e`, et si celle-ci lève une exception filtrée par l'une des branches `E`, de retourner l'expression associée à la branche correspondante. La vérification est quasiment identique à celle du filtrage classique. La seule différence étant que les motifs doivent être annotés avec un type équivalent à celui de `exn`. Le corps et les expressions de chaque branche doivent également être de types équivalents.

La construction **`assert`** `e` pourrait n'être qu'un sucre syntaxique et se traduire par

contraire.

$$\textsc{Lazy}\ \frac{\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau < \tau_{arg}\ \texttt{lazy\_t} \qquad \Gamma, \Phi \vdash_{\tau} \tau_e \equiv \tau_{arg}}{\Gamma, \Phi \vdash \langle \textbf{lazy}\ e : \tau \rangle}$$

FIGURE 4.20 – Vérification des expressions paresseuses

```
(fun x -> if not x then raise Assert_failure) e}
```

mais `assert false` est un cas particulier strictement équivalent à `raise Assert_failure`, puisqu'il peut prendre n'importe quel type. Il n'y a donc aucune difficulté à vérifier cette construction.

Evaluation paresseuse

OCaml est un langage dont l'appel se fait par valeur. Ainsi, dans le cas d'un appel de fonction, les arguments sont évalués avant d'être substitués aux paramètres pour l'exécution du corps de la fonction. En revanche, il existe des cas où il peut être intéressant de n'évaluer l'argument que lorsque celui-ci doit être effectivement utilisé : il s'agit de la stratégie dite d'*appel par nom*. Dans le cas d'un langage *pur*, *i.e.* sans effets de bord, cette stratégie d'évaluation n'a pas d'incidence sur le résultat à l'exception de programmes qui pourraient ne pas terminer en appel par valeur, mais dans le cas d'OCaml cela peut drastiquement modifier le résultat de la réduction. Il existe des moyens d'exprimer de l'appel par nom dans un langage implémentant l'appel par valeur [10], mais celui-ci n'est pas connu pour son efficacité car l'argument est recalculé chaque fois qu'il est utilisé.

OCaml implémente une version dérivée de cette stratégie : l'appel par nécessité, appelée aussi évaluation paresseuse. La construction `lazy e` permet donc d'exprimer une valeur `e` dont le résultat sera calculé explicitement la première fois qu'elle sera accédée via `Lazy.force` ou à l'aide du filtrage, grâce au motif

```
| lazy e  -> .. e ..
```

La vérification d'une expression `lazy_t` est donc relativement simple : il s'agit de s'assurer que l'expression à calculer est correcte, que le type du nœud est bien de la forme de τ_{arg} `lazy_t`, et finalement que le type de l'expression suspendue est équivalent à celui de l'argument. La règle est donnée en figure 4.20.

Variants polymorphes

Les variants polymorphes permettent d'utiliser des types algébriques sans pour autant les définir au préalable. L'inférence se charge alors de donner le type le plus général à celui-ci. Ainsi, l'expression

```
let ⟨ x : [> ‘K ] ⟩ = ‘K
```

10. ∧On utilisera alors un *thunk*. L'expression est encapsulée dans une fonction attendant unit, et appelée explicitement lorsque la valeur est nécessaire.

sera inférée avec le type `[> 'K ]`, signifiant que la valeur peut être au minimum de la forme `'K`. Le type variant est alors représenté par 4 composantes :

— L'ensemble $\mathcal{K}$ des constructeurs qui peuvent (*Either*) ou doivent (*Present*) apparaître ;
— L'ensemble $\mathcal{T}$ des constructeurs connus avec leur type ;
— Une variable de rangée, indiquant si le type est fixé ou extensible ;
— Un booléen, indiquant si le variant est clos, *i.e.* qu'il n'accepte pas d'autres constructeurs.

Les deux derniers composants peuvent être décrits par la métavariable de rangée ρ. Cette dernière peut prendre trois formes :

— > : signifiant que le variant peut accepter d'autres constructeurs que ceux énoncés. La variable de rangée est alors une variable de type et le variant n'est pas clos. Par abus de langage, un type dont la métavariable de rangée est > s'écrira [> $\mathcal{T}$], puisque par définition tous les constructeurs $\mathcal{K}$ sont forcément présents dans $\mathcal{T}$.
— < : le variant n'est compatible qu'avec les constructeurs apparaissant dans $\mathcal{K}$. La variable de rangée est alors une variable de type, et le variant est clos.
— La métavariable peut aussi être absente, signifiant alors que l'ensemble des constructeurs est fixé. Ainsi, la variable de rangée sera le cas particulier ϵ, et le variant sera effectivement clos.

Ainsi, le cas où le variant n'est pas clos et la variable de rangée est ϵ est considéré comme étant erroné, et donc le type mal formé.

Cette idée de type "ouvert" ou fermé, acceptant d'autres constructeurs est essentielle pour le polymorphisme. Prenons l'exemple suivant :

```
let ⟨ x : [> 'K ] ⟩ = 'K
let ⟨ y : [> 'L ] ⟩ = 'L
let ⟨ l : [> 'K | 'L ] list ⟩ = x :: y :: []
```

Si on veut pouvoir donner un type à la liste `l`, il faut que `x` et `y` soient de types équivalents. En inférant qu'ils puissent avoir pour valeurs d'autres constructeurs qu'eux-mêmes, il est possible d'instancier `x` par la suite avec un type moins général qui est `[> 'K | 'L ]`, de même pour `y`. On garde alors la possibilité a posteriori de rajouter de nouveaux constructeurs à cette liste. De cette manière, les variants polymorphes permettent d'ajouter une forme de sous-typage au langage, à l'aide du mécanisme des variables de rangées et de l'unification. OCaml permet une forme plus générale du sous-typage (sans variables de rangées), à l'aide de l'opérateur de coercion `:>`, mais celui-ci n'est pas géré dans ce formalisme.

La vérification d'un constructeur de variant polymorphe est alors simple (voir figure 4.21). D'abord, le type τ du nœud doit être un variant. Le constructeur doit apparaître dans l'ensemble des constructeurs possibles, et indiqué comme explicitement présent. Si le constructeur n'a pas d'argument, il doit apparaître dans l'ensemble des types comme n'ayant pas d'argument. Dans le cas contraire, il doit apparaître comme ayant un type, et ce type doit être équivalent à celui annoté à son argument.

La seconde forme des variants polymorphes restreint l'ensemble des constructeurs possibles : [< $\mathcal{T}$ > $\mathcal{K}$]. Par exemple, un variant de type [< 'K | 'L] indique que la valeur associée est compatible avec tout variant qui contient au maximum les constructeurs `'K` et `'L`. Ainsi, en reprenant l'exemple précédent (sans donner la valeur de `x` et `y`) :

```
let ⟨ x : [> 'K ] ⟩ = ..
```

$$\textsc{Variant-Const}\ \frac{\Gamma, \Phi \vdash_{\mathcal{T}} \tau < [(\rho)\, \mathcal{T} > \mathcal{K}] \qquad T \in \mathcal{T} \qquad \mathcal{K}(T) = Present}{\Gamma, \Phi \vdash \langle `T : \tau \rangle}$$

$$\textsc{Variant}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash_{\mathcal{T}} \tau < [(\rho)\, \mathcal{T} > \mathcal{K}] \\ T\ of\ \tau_{arg} \in \mathcal{T} \qquad \mathcal{K}(T) = Present \qquad \Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau_{arg} \equiv \tau_e\end{array}}{\Gamma, \Phi \vdash \langle `T\ e : \tau \rangle}$$

Figure 4.21 – Vérification des constructeurs de variants polymorphes

$$\textsc{Pat-Variant-Const}\ \frac{\Gamma, \Phi_p \vdash_{\mathcal{T}} \tau < [(\rho)\, \mathcal{T} > \mathcal{K}\,] \qquad T \in \mathcal{T} \qquad \mathcal{K}(T) = Either}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle `T : \tau \rangle \Rightarrow \mathcal{V}, \mathcal{T}, \Phi}$$

$$\textsc{Pat-Variant}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash_{\mathcal{T}} \tau < [(\rho)\, \mathcal{T} > \mathcal{K}] \qquad T\ of\ \tau_{arg} \in \mathcal{T} \\ \mathcal{K}(T) = Either \qquad \Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle p : \tau_p \rangle \Rightarrow \mathcal{V}', \mathcal{T}', \Phi' \qquad \Gamma, \Phi \vdash_{\mathcal{T}} \tau_{arg} \equiv \tau_p\end{array}}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash_{\rho} \langle `T\ p : \tau \rangle \Rightarrow \mathcal{V}', \mathcal{T}', \Phi'}$$

Figure 4.22 – Vérification des motifs de variants polymorphes

```
let ⟨ y : [< ‘K | ‘L ] ⟩ = ..
let ⟨ l : [< ‘K | ‘L > ‘K] list ⟩ = x :: y :: []
```

Ainsi, pour que `x` et `y` soient compatibles, il devient nécessaire de leur trouver une instance commune. Sachant le premier type, il faut au moins le constructeur `‘K`, et pour le second au plus `‘K` et `‘L`. Par conséquent, l'instance commune est le variant qui ne connaît pas d'autre constructeur que `‘K` et `‘L`, et nécessite `‘K`, ce qui se traduit naturellement par `[< ‘K | ‘L > ‘K]`, type qui est une instance correcte pour les deux valeurs.

Cette forme de variant est "naturellement" inférée pendant le filtrage sur des constructeurs de variants. Puisqu'il existe un choix entre plusieurs constructeurs, l'inférence doit indiquer qu'ils n'ont aucune raison d'être présents, mais seulement un cas acceptable.

4.4 Résultats et limitations

Un tel processus de vérification a été implémenté pour pour OCaml 4.02. Celui-ci travaille donc sur des Typedtree produits par le compilateur, qu'ils soient enregistrés sous la forme de *cmt*s, autrement dit des fichiers binaires, ou en demandant directement au vérificateur d'effectuer l'inférence de types du programme. Celui-ci a permis en particulier de vérifier plusieurs erreurs de l'inférence, celles-ci étant généralement le résultat de l'utilisation et le mélange de fonctionnalités a priori orthogonales. Celles-ci ont bien entendues éte corrigées dans les versions ultérieures du

```
type (_, _) eq = Eq : ('a, 'a) eq
let cast : type a b . (a, b) eq -> a -> b = fun Eq x -> x

module Fix (F : sig type 'a f end) = struct
  type 'a fix = ('a, 'a F.f) eq
  let uniq (type a) (type b) (Eq : a fix) (Eq : b fix) : (a, b) eq =
      Eq
end

module FixId = Fix (struct type 'a f = 'a end)
let bad : (int, string) eq = FixId.uniq Eq Eq
let _ = Printf.printf "Oh dear: %s\n" (cast bad 42)
```

FIGURE 4.23 – Un programme OCaml non sûr

compilateur.

4.4.1 Exemples d'erreurs de l'inférence

Prenons l'exemple de programme en figure 4.23 [11]. Celui-ci provoque une erreur de segmentation à l'exécution, qui est due à une contorsion du système de types pour les types algébriques gardés en présence de foncteur permettant de *caster* (*i.e.* considérer une valeur avec un type donné comme étant d'un autre type) une valeur de n'importe quel type vers n'importe quel autre type arbitraire.

On commence par déclarer un type gardé permettant de représenter des équivalences de types : toute valeur de type `(a, b) eq` est une preuve que `a` et `b` sont équivalents. On peut donc écrire une fonction `cast` qui, sachant une valeur d'un type `a`, d'une preuve `(a, b) eq`, retourne cette même valeur avec le type `b`. On définit ensuite un foncteur dont l'argument est un module `F` ne contenant qu'un type `f` paramétré par une unique variable. Dans ce foncteur, on définit deux entités :

— `'a fix`, une abréviation de la preuve que pour tout `'a`, `'a` est équivalent à `'a F.f`. Autrement dit, une preuve que le type `F.f` est équivalent à son paramètre de type.

— `uniq`, qui sachant deux preuves de type `a fix` et `b fix`, retourne un preuve que `a` et `b` sont équivalents.

Si on applique ce foncteur à une structure dont le type `f` est égal à son paramètre de type `'a`, il devient alors possible de créer des preuves d'équivalence entre n'importe quels types.

Une telle erreur est en réalité rattrapable durant la vérification de la fonction `uniq`, lorsqu'il s'agira de vérifier le premier motif `Eq` annoté avec le type `(a, a F.f) eq`, en particulier l'ajout de l'équation entre ce type et celui de `Eq : ('a, 'a) eq` :

1. D'abord, l'équivalence `a = 'a` est ajoutée à l'ensemble des équations connues. Celle-ci est valide par défaut, puisqu'il n'existe aucune équation pour `a` ou `'a`.

2. Ensuite, il s'agit d'ajouter l'équivalence `a F.f = 'a`. Puisqu'il existe déjà une classe d'équivalence pour `'a`, il faut donc s'assurer que chacun des types qui lui est associé est

11. ∧Voir le rapport de bug 6992 [22] sur le *bug tracker* d'OCaml.

```
type 'a n = N
type nil = private Nil_type
type (_,_) elt = Elt: 'nat n -> ('l,'nat -> 'l) elt
type _ t = Nil : nil t | Cons : ('x, 'fx) elt * 'x t -> 'fx t

let undetected: ('a -> 'b -> nil) t -> 'a n -> 'b n -> unit = fun sh i
    j ->
  let Cons(Elt dim, _) = sh
  in ()

undetected (Cons(Elt N,Cons(Elt N,Nil))) N N
```

FIGURE 4.24 – Un autre programme OCaml non sûr.

compatible à `a F.f`, autrement dit dans notre cas : `a = a F.f`. Puisque le constructeur de type `F.f` n'est pas une variable rigide mais un type abstrait, il est impossible de le rendre équivalent à `a`. Par conséquent, le type est rejeté.

Une autre défaillance de l'inférence [3] permettait à des types existentiels de s'échapper. Le programme donné en figure 4.24 en est une version légèrement modifiée, permettant de mettre en exèrgue l'échappement de types.

Le type inféré pour `undetected` contient un constructeur de type généré par le compilateur `b#0`, autrement dit un type existentiel normalement valable que dans la portée du filtrage qui l'a généré. Une telle erreur est en réalité simple à détecter : le vérificateur s'assure à chaque instant de la bonne formation des types. Au moment de vérifier le type annoté pour `undetected`, son type le constructeur `b#0` sera rejeté puisqu'il n'apparaît pas lié dans l'environnement (règle WF-CONSTRUCT).

Limitations L'une des limitations de la vérification du Typedtree est que celui-ci est généré par le compilateur après inférence. Ainsi, il est difficile de produire des arbres dont les annotations sont incohérentes, à moins de connaître les éventuelles régressions du compilateur pouvant entraîner des programmes dont le typage est incorrect, tels que ceux présentés précédemment. Une solution pourrait être d'accepter une syntaxe où tous les nœuds sont annotés, ou plus raisonnablement les arguments de fonctions et les variables liées par des **`let`**. Il s'agirait alors d'effectuer une propagation locale de ces types pour annoter chaque sous-expression.

Conclusion

Ce chapitre présente un système de vérification formel des TAST d'OCaml. Ce formalisme prend en compte un large sous-ensemble du langage dit "noyau", en particulier les types algébriques gardés qui rajoutent à eux seuls une certaine complexité en terme de vérification (et bien entendu d'inférence, ce qui ne concerne pas ce travail de formalisation). Un tel système de vérification de cohérence, vu sous un angle de vérification d'arbres de preuves de typage, peut alors être considéré comme un présentation formelle du système de types permettant de produire de tels TAST.

Il s'agit d'appliquer la méthode employée au chapitre 2 sur MiniML, mais au Typedtree. La vérification du Typedtree d'OCaml est bien entendu bien plus complexe que celle de MiniML : le langage contient bien plus de constructions, et introduit des concepts de mutabilité rendant la vérification plus dense. Il faut dont redéfinir les concepts de généralisation et de restriction relâchée de valeur. MiniML était un langage simple et idéal, là où le Typedtreemanque de certaines informations. En particulier, l'absence de quantification sur les schémas de types rend la vérification d'instance plus complexe et nécessite de retrouver ces informations *a posteriori*, là où l'inférence aurait pu les annoter. La prise en charge des GADTs implique d'être capable de gérer de nouveaux cas, tels que les équations locales de types ou encore l'échappement de types existentiels. Il faut alors rajouter de nouvelles opérations capables de vérifier ces propriétés. Ce chapitre présente les différentes opérations nécessaires à la vérification de ces nouvelles contraintes par rapport à MiniML, ainsi que l'ensemble des règles de vérification pour le langage de base d'OCaml, menant donc à une specification du système de types de ce sous-ensemble du langage.

Le chapitre suivant s'attelle à la tâche de définir une sémantique opérationnelle de ce Typedtree, étape importante pour une future preuve de sûreté de ce système de types.

Chapitre 5

Sémantique opérationnelle du Typedtree

Le Typedtree n'est pas prévu pour être exécuté, il peut néanmoins être intéressant d'en définir une sémantique opérationnelle. Ce chapitre suit le principe énoncé en section 2.5. On notera que les expressions dans les règles apparaissent généralement comme non annotées : on travaille bien sur des termes du Typedtree, mais on omet les annotations lorsque celles-ci ne sont pas utiles. Chacune des constructions ayant une sémantique distincte et orthogonale aux autres,la description de cette sémantique est décomposée en différentes catégories : noyau, mutabilité, extensions fonctionnelles, extensions impératives, etc. Une description complète comprenant chacun des ajouts nécessaires de cette sémantique est donnée en annexe B.

Ce bref chapitre essaie donc d'apporter une base à une future preuve de sûreté du système de types défini au chapitre précédent, en résolvant le problème de l'évaluation des types au moment de la substitution des variables à l'aide de l'instanciation sémantique. Cette sémantique opérationnelle reste néanmoins théorique et n'est pas implémentée dans un évaluateur. Cette sémantique serait également une base intéressante dans le cadre de la formalisation de techniques d'optimisations du langage, comme par exemple pour la defonctionnalisation des foncteurs ou plus simplement de la beta-réduction de valeurs durant la compilation (comme par exemple celle effectuée entre autre par la passe de compilation travaillant sur la représentation intermédiaire *Flambda*).

5.1 Sémantique du noyau ML d'OCaml

La sémantique du noyau est décrite en figure 5.1. La principale différence avec celle donnée pour MiniML annoté est l'ajout du filtrage de motifs. Dans le cas de l'application, si les deux membres sont des valeurs, deux cas de réductions sont possibles :

— Si la valeur correspond au motif de la première branche, on récupère les sous-valeurs correspondant aux variables présentes dans celui-ci, et on les substitue dans le corps de cette même branche.
— En revanche, si la valeur n'est pas filtrée par le motif de la branche courante, alors on

$$
\begin{array}{llr}
v ::= & & \textit{valeurs} \\
 & \mid \langle\, \mathrm{c} : \tau \,\rangle & \textit{constante} \\
 & \mid \langle\, \mathbf{function}\ \overline{\mid p \to e} : \tau \,\rangle & \textit{abstraction}
\end{array}
$$

— $value(e)$: prédicat indiquant si l'expression donnée est une valeur.

Réduction des expressions annotées

$$\textsc{App-Left}\ \frac{e_1 \rightsquigarrow v_1}{e_1\ e_2 \rightsquigarrow v_1\ e_2} \qquad \textsc{App-Right}\ \frac{value(v_1) \qquad e_2 \rightsquigarrow v_2}{v_1\ e_2 \rightsquigarrow v_1\ v_2}$$

$$\textsc{App-Function-Match}\ \frac{value(v_2) \qquad match(p, v) = \overline{(x, v)}}{\left\langle\, \mathbf{function}\ \begin{array}{l} \mid p \to e \\ \mid \overline{p' \to e'} : \tau \end{array} \right\rangle v_2 \rightsquigarrow e[x_1 \mapsto v_1, .., x_n \mapsto v_n]}$$

$$\textsc{App-Function-Bot}\ \frac{value(v_2) \qquad match(p, v) = \bot}{\left\langle\, \mathbf{function}\ \begin{array}{l} \mid p \to e \\ \mid \overline{p' \to e'} : \tau \end{array} \right\rangle v_2 \rightsquigarrow \langle\, \mathbf{function}\ \overline{\mid\ p' \to e'} : \tau \rangle\ v_2}$$

$$\textsc{Let-Bind}\ \frac{\forall i. e_i \rightsquigarrow v_i}{\mathbf{let}\ \overline{\langle p : \sigma \rangle = e}\ \mathbf{in}\ e' \rightsquigarrow \mathbf{let}\ \overline{\langle p : \sigma \rangle = v}\ \mathbf{in}\ e'}$$

$$\textsc{Let-Value}\ \frac{\forall i. value(v_i) \qquad \forall i. \overline{(p_i, v_i)} = match(p_i, v_i)}{\mathbf{let}\ \overline{\langle p : \forall \overline{\alpha}. \tau \rangle = v}\ \mathbf{in}\ e' \rightsquigarrow e'[x_{11} \mapsto v_{11}, .., x_{nk} \mapsto v_{nk}]_{\triangleleft}}$$

Filtrage de motifs

$$
\begin{array}{lcl}
\mathrm{match}\ (x, v) & = & (x, v) \\
\mathrm{match}\ (\_, v) & = & \emptyset \\
\mathrm{match}\ (\mathrm{c}, \mathrm{c}) & = & \emptyset \\
\mathrm{match}\ (\mathrm{p}_1 \mid \mathrm{p}_2, v) & = & \begin{cases} \mathrm{match}\ (\mathrm{p}_1, v) & \\ \mathrm{match}\ (\mathrm{p}_2, v), & \text{si match } (\mathrm{p}_1, v) = \bot \end{cases} \\
\mathrm{match}\ (\_, \_) & = & \bot
\end{array}
$$

FIGURE 5.1 – Sémantique opérationnelle du noyau d'OCaml

La sémantique est fortement similaire à celle donnée pour MiniML en section 2.5, avec l'ajout du filtrage de motifs pour les arguments de fonctions.

continue à tester les cas suivants.

Le test de filtrage repose sur la fonction `match(`p`, `v`)`, qui retourne l'ensemble des variables de p associées aux sous-valeurs correspondantes de v si le motif p correspond à la valeur v. Si cet appel à la fonction est une erreur, alors cette valeur n'est pas filtrable par le motif donné. On peut voir le résultat de cette fonction comme étant une instance du type `option`, où `Some ..` représente une correspondance entre le motif et la valeur, et `None` son contraire. Le cas de la réduction de **`let`** est similaire : on collecte l'ensemble des variables qui apparaissent dans les définitions locales, pour ensuite les substituer dans l'expression du corps du **`let`**. L'ordre de réduction des expressions locales d'un même **`let`** n'est en revanche pas spécifié.

5.2 Filtrage, n-uplets et constructeurs de données

La sémantique opérationnelle des n-uplets, constructeurs et du filtrage de motifs est décrite en figure 5.2. Un n-uplet (ou une valeur construite) est une valeur si ses composantes sont elles-mêmes des valeurs. L'expression **`match`** n'est pas une valeur, puisqu'elle peut se réduire. De la même manière que **`function`** (figure 5.1), sa réduction considère d'abord l'expression à filtrer qui doit être une valeur, puis teste une à une les branches et se réduit en la première dont le motif correspond à la valeur filtrée, dans l'ordre de déclaration.

5.3 Enregistrements

La réduction des enregistrements est donnée en figure 5.4. Une première remarque importante est l'ajout d'une notion de mémoire $\mathcal{M}$ introduite figure 5.3, utilisée comme un nouvel environnement durant l'évaluation des expressions, et plus particulièrement des enregistrements, dont certains champs peuvent être mutables. La réduction d'un enregistrement produit une valeur localisée à une adresse m dans la mémoire $\mathcal{M}$. La mémoire $\mathcal{M}$ peut être vue comme une liste de triplets, chacun constitué de :

- m, l'identifiant représentant l'emplacement mémoire alloué ;
- τ, le type de l'enregistrement qu'il représente ;
- $[v_1 ; .. ; v_n]$, un bloc mémoire (autrement dit un tableau) contenant chacun des champs de l'enregistrement de type τ.

Chaque accès à un champ dans le Typedtree est en réalité décoré d'informations supplémentaires, en particulier son indice dans le bloc mémoire alloué pour son enregistrement. Ainsi, l'allocation d'un enregistrement n'est rien d'autre qu'un tableau dont la taille est son nombre de champs. On peut donc définir $\mathcal{M}\ (m, l)$ comme étant l'accès à la valeur du champ l dans le bloc mémoire à l'emplacement m.

Ainsi, la réduction d'une expression se fait désormais toujours en accord avec une mémoire $\mathcal{M}$. S'il s'agit de réduire un enregistrement dont tous les champs sont des valeurs, sa réduction ajoute un nouvel emplacement mémoire m frais dans $\mathcal{M}$, auquel on associe son type τ et un tableau contenant toutes les valeurs pour ses champs, dans l'ordre indiqué par le type. De même, l'accès à un champ d'enregistrement se fera toujours sur un emplacement mémoire, et le résultat est la valeur associée au-dit champ. Finalement, l'assignation d'un champ modifie le bloc mémoire

$$
\begin{array}{ll}
v ::= & \textit{valeurs} \\
\quad .. & \\
\quad \mid \langle\, (v_1, .., v_n) : \tau \,\rangle & \textit{n-uplets} \\
\quad \mid \langle\, K\ (v_1, .., v_n) : \tau \,\rangle & \textit{constructeurs}
\end{array}
$$

Réduction des expressions annotées

La réduction des expressions de n-uplets et des arguments de constructeurs se fait de la gauche vers la droite. Sont présentés ici les cas particuliers des couples et constructeurs à deux arguments.

$$
\textsc{Tuple-First}\ \frac{e_1 \rightsquigarrow v_1}{(e_1,\ e_2) \rightsquigarrow (v_1,\ e_2)}
\qquad
\textsc{Tuple-Second}\ \frac{value(v_1) \qquad e_2 \rightsquigarrow v_2}{(v_1,\ e_2) \rightsquigarrow (v_1,\ v_2)}
$$

$$
\textsc{Constr-First}\ \frac{e_1 \rightsquigarrow v_1}{K(e_1,\ e_2) \rightsquigarrow K(v_1,\ e_2)}
\qquad
\textsc{Constr-Second}\ \frac{value(v_1) \qquad e_2 \rightsquigarrow v_2}{K(v_1,\ e_2) \rightsquigarrow K(v_1,\ v_2)}
$$

$$
\textsc{Match-Scrut}\ \frac{e \rightsquigarrow v}{\textbf{match}\ e\ \textbf{with} \overline{\mid p \rightarrow e} \rightsquigarrow \textbf{match}\ v\ \textbf{with} \overline{\mid p \rightarrow e}}
$$

$$
\textsc{Match-Ok}\ \frac{value(v) \qquad match(p, v) = \overline{(x, v)}}{\begin{array}{ll}\textbf{match}\ v\ \textbf{with} & \mid p \rightarrow e \\ & \mid \overline{p' \rightarrow e'}\end{array} \rightsquigarrow e[x_1 \mapsto v_1, .., x_n \mapsto v_n]}
$$

$$
\textsc{Match-Bot}\ \frac{value(v_2) \qquad match(p, v) = \bot}{\begin{array}{ll}\textbf{match}\ v\ \textbf{with} & \mid p \rightarrow e \\ & \mid \overline{p' \rightarrow e'}\end{array} \rightsquigarrow \textbf{match}\ v\ \textbf{with} \overline{\mid\ p' \rightarrow e'}}
$$

Filtrage de motifs

$$
\begin{array}{lcl}
\text{match } ((p_1, .., p_n), (e_1, .., e_n)) & = & \text{match } (p_1, e_1) \cup .. \cup \text{match } (p_n, e_n) \\
\text{match } (K\ (p_1, .., p_n), K\ (e_1, .., e_n)) & = & \text{match } (p_1, e_1) \cup .. \cup \text{match } (p_n, e_n)
\end{array}
$$

FIGURE 5.2 – Sémantique opérationnelle du filtrage et des constructeurs

La sémantique opérationnelle du filtrage est identique à celle du filtrage des arguments de fonctions.

e' : :=	*expressions*
..	
$\mid m$	*emplacement mémoire*
v : :=	*valeurs*
..	
$\mid \langle\, m : \tau \,\rangle$	*emplacement mémoire*

Mémoire

$$\mathcal{M} ::= \emptyset \mid \mathcal{M}, (m : \tau = [v_1 ; .. ; v_n])$$

— dom ($\mathcal{M}$) : ensemble des emplacements mémoire alloués
— $\mathcal{M}_\tau(m)$: type associé à l'emplacement mémoire
— $\mathcal{M}\,(m, l)$: valeur enregistrée dans le champ l du bloc mémoire alloué à m.
— $\mathcal{M}\,(m, l) \leftarrow v$: assignation de la valeur v au champ l dans le bloc associé à m.

FIGURE 5.3 – Mémoire et emplacements mémoire

associé, et se réduit en la valeur `()`. On notera que la fonction de filtrage doit désormais prendre un argument la mémoire, puisqu'elle doit accéder à celle-ci pour récupérer la valeur des champs. Elle ne la modifie pas, il n'est donc pas nécessaire de la retourner (contrairement au cas du filtrage des valeurs paresseuses, comme nous le verrons plus loin).

Dans l'éventualité d'une preuve de préservation, la mémoire doit être typée : en effet, l'enregistrement étant transformé en "bloc", il est nécessaire de pouvoir retrouver son type initial pour s'assurer que l'enregistrement enregistré est bien du type attendu. Autrement, il devient impossible de prouver que le bloc mémoire est bien du type de l'enregistrement : il pourrait s'agir d'un bloc dont les champs ont le même type, mais pas issus de la même déclaration.

Les emplacements de blocs sont indexés par des champs, dont l'ordre de déclaration du type de l'enregistrement. La mémoire ne supporte pas le sous-typage, puisqu'OCaml gère le sous-typage sous la forme d'un calcul d'instanciation. Ainsi, supposons la référence suivante :

```
let ⟨ r : [ ‘A | ‘B ] ref ⟩ = ref ‘A
```

Toute assignation de `r` doit être d'une valeur de type `[ ‘A | ‘B ]`. Dans le cas de l'affectation d'une valeur d'un de ses sous-types, deux cas sont possibles :

— la valeur n'est pas une variable. Il n'est donc pas question d'instanciation, et celle-ci doit déjà être inférée avec le type `[ ‘A | ‘B ]`.
— la valeur est une variable, son type sera alors une instance de `[ ‘A | ‘B ]`, par exemple `[> ‘B]` (pour rappel, `[ ‘B ]` ne serait par une instance correcte, puisque le variant est fermé). Son type devra dans tous les cas être annoté avec le type `[ ‘A | ‘B ]`, puisque la règle de vérification des variables s'assure de vérifier l'instanciation.

Le sous-typage des variants (sans l'utilisation de variables de rangées) dans OCaml est explicite, à l'aide de coercions. Celles-ci ne sont pas gérées dans notre formalisme.

La réduction des champs d'enregistrements se fait de la gauche vers la droite. Le cas présenté ici correspond à un enregistrement contenant uniquement deux champs.

$$\textsc{Record-First}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash \{l_1 = e_1;\ l_2 = e_2\} \rightsquigarrow \{l_1 = v_1.;\ l_2 = e_2\}, \mathcal{M}_1}$$

$$\textsc{Record-Second}\ \frac{value(v_1) \qquad M \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash \{l_1 = v_1;\ l_2 = e_2\} \rightsquigarrow \{l_1 = v_1.;\ l_2 = v_2\}, \mathcal{M}_2}$$

$$\textsc{Record-Location}\ \frac{value(v_1) \qquad value(v_2) \qquad let\ m \notin dom(\mathcal{M})}{\mathcal{M} \vdash \langle\{l_1 = v_1;\ l_2 = e_2\} : \tau\rangle \rightsquigarrow \langle m : \tau\rangle, (\mathcal{M}, (m : \tau = [v_1; v_2]))}$$

$$\textsc{Record-Access}\ \frac{\mathcal{M} \vdash e \rightsquigarrow v, \mathcal{M}'}{\mathcal{M} \vdash e.l \rightsquigarrow v.l, \mathcal{M}'} \qquad \textsc{Location-Access}\ \frac{}{\mathcal{M} \vdash \langle m : \tau\rangle.l \rightsquigarrow \mathcal{M}(m, l)}$$

$$\textsc{Record-Set}\ \frac{\mathcal{M} \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash e_1.l \leftarrow e_2 \rightsquigarrow e_1.l \leftarrow v_2, \mathcal{M}_2}$$

$$\textsc{Record-Set-Value}\ \frac{value(v_2) \qquad \mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1.l \leftarrow v_2 \rightsquigarrow v_1.l \leftarrow v_2, \mathcal{M}_1}$$

$$\textsc{Location-Set-Value}\ \frac{value(v_2)}{\mathcal{M} \vdash \langle m : \tau\rangle.l \leftarrow v_2 \rightsquigarrow \mathcal{M}(m, l) \leftarrow v_2}$$

Filtrage de motifs

$$\text{match}\ (\{\mathrm{l}_i = \mathrm{p}_i\,;\ ..,;\ l_j = \mathrm{l}_j\}),\ \langle\ m : \tau\ \rangle,\ \mathcal{M}) \quad = \quad \bigcup_{k \in [i..j]} \text{match}\ (\mathrm{p}_k,\ \mathcal{M}\ (m,\ \mathrm{l}_k))$$

FIGURE 5.4 – Sémantique opérationnelle des enregistrements

Les enregistrements permettent d'introduire la mutabilité dans le langage. Il est alors nécessaire de considérer la mémoire comme un environnement nécessaire à la réduction.

$$\textsc{Location}\ \frac{\Gamma, \Phi \mid_{\overline{\tau}}\ \tau \equiv \mathcal{M}_\tau(m)}{\Gamma, \mathcal{M}, \Phi \mid\!\!- \langle m : \tau \rangle}$$

$$\textsc{App-Mem}\ \frac{\begin{array}{c}\Gamma, \mathcal{M}, \Phi \mid\!\!- \langle e_1 : \tau_1 \rangle \qquad \Gamma, \mathcal{M}, \Phi \mid\!\!- \langle e_2 : \tau_2 \rangle \\ \Gamma, \Phi \mid_{\overline{\tau}}\ \tau_1 \prec \tau_d \rightarrow \tau_{cd} \qquad \Gamma, \Phi \mid_{\overline{\tau}}\ \tau_2 \equiv \tau_d \qquad \Gamma, \Phi \mid_{\overline{\tau}}\ \tau_{cd} \equiv \tau\end{array}}{\Gamma, \mathcal{M}, \Phi \mid\!\!- \langle e_1\ e_2 : \tau \rangle}$$

FIGURE 5.5 – Règles de vérification des emplacements mémoire

e' : := *expressions*

..

| **fix** e *combinateur de point fixe*

Vérification

$$\textsc{Fix}\ \frac{\Gamma, \Phi \mid\!\!- \langle e : \tau_e \rangle \qquad \Gamma, \Phi \mid_{\overline{\tau}}\ \tau_e \prec \tau_d \rightarrow \tau_{cd} \qquad \Gamma, \Phi \mid_{\overline{\tau}}\ \tau_d \equiv \tau \qquad \Gamma, \Phi \mid_{\overline{\tau}}\ \tau_{cd} \equiv \tau}{\Gamma, \Phi \mid\!\!- \langle \mathbf{fix}\ e : \tau \rangle}$$

FIGURE 5.6 – Combinateur de point fixe

La mémoire étant typée, il n'est pas possible d'effectuer des conversions de types à l'aide du module `Obj` en sachant que le modèle mémoire est identique. Ainsi, bien que les enregistrements et les tableaux aient la même forme, l'utilisation de `Obj.magic` ne sera pas possible. De même, le passage de `string` à `bytes` d'une valeur contenue dans un emplacement mémoire n'est pas possible, bien que la représentation sous-jacente soit la même.

Finalement, pour pouvoir vérifier les emplacements mémoire, il est nécessaire d'ajouter à toutes les règles de vérification celles concernant cette mémoire annotée, pour ainsi pouvoir récupérer le type associé à l'emplacement mémoire m (figure 5.5). Cette mémoire ne sera nécessaire que dans le cas de la vérification de la forme réduite d'un terme de Typedtree : en effet, il n'est pas possible pour l'utilisateur d'écrire des identifiants mémoire. Initialement, cet argument $\mathcal{M}$ sera donc vide. L'ajout étant trivial dans les règles de vérification, on ne donne ici que celui de l'application comme exemple : il s'agit simplement de propager la mémoire courante dans toute la dérivation dès le moment où intervient la vérification d'une expression.

5.4 Récursion

La récursion peut être encodée [56] via l'ajout d'un combinateur de point fixe **fix** (figure 5.6). La sémantique de celui-ci est décrite dans la figure 5.7. Sachant une fonction récursive `f`, Expansion de **let rec**

```
let rec ⟨ f₁ : τ₁ ⟩ = ⟨ e₁ : τ₁ ⟩
..
and ⟨ fₙ : τₙ ⟩ = ⟨ eₙ : τₙ ⟩
```

$\rightsquigarrow$

(Sachant $\tau_{fs} = \tau_1 * .. * \tau_n$)

```
let ⟨ (f₁, .., fₙ) : τ_fs ⟩ =
  fix (function ⟨ f₁', .., fₙ' : τ_fs ⟩ →
    ( e₁ [ ⟨ f₁ : τ_f1 ⟩ ↦ ⟨ f₁' : τ_f1 ⟩;
           ..
           ⟨ fₙ : τ_fn ⟩ ↦ ⟨ fₙ' : τ_fn ⟩],
      ..,

      eₙ [ ⟨ f₁ : τ_f1 ⟩ ↦ ⟨ f₁' : τ_f1 ⟩; ..
           ..
           ⟨ fₙ : τ_fn ⟩ ↦ ⟨ fₙ' : τ_fn ⟩]))
```

Réduction des expressions annotées

— $child_i$: retourne le *i-ème* fils d'un n-uplet.

$$\textsc{Fix}\ \frac{e \rightsquigarrow v}{\mathbf{fix}\ e \rightsquigarrow \mathbf{fix}\ v}$$

$$\textsc{Fix-Value}\ \frac{match(p,e) = \overline{(p,e)} \qquad let\ \theta = \begin{array}{l} [\quad p_1 \mapsto child_1(\mathbf{fix}\ (\mathbf{function}\langle x : \tau_x\rangle \rightarrow e)); \\ \qquad\qquad .. \\ \quad p_n \mapsto child_n(\mathbf{fix}\ (\mathbf{function}\langle x : \tau_x\rangle \rightarrow e))] \end{array}}{\mathbf{fix}\ (\mathbf{function}\ \langle p : \tau\rangle \rightarrow e) \rightsquigarrow (e_1\ \theta, .., e_n\ \theta)}$$

FIGURE 5.7 – Sémantique opérationnelle de la récursion mutuelle
Le récursion nécessite l'ajout d'un combinateur de point-fixe et d'une transformation en n-uplet de fonctions pour gérer la rećursion mutuelle.

son encodage correspond à une nouvelle fonction f' telle que `fix f' f'` et `f` sont de même type. S'agissant d'une nouvelle construction syntaxique, celle-ci doit également être typée : son argument doit lui-même être correct, et son type de la forme d'une fonction. Son domaine et son codomaine doivent être équivalents, et le type du nœud **`fix`** doit lui aussi être équivalent au domaine (et par transitivité au codomaine) de la fonction.

La forme

```
let rec f = e in ..
```

n'est alors qu'un sucre syntaxique de

```
let f = fix (function f -> e) in ..
```

Néanmoins, OCaml permettant de définir des fonctions mutuellement récursives, la transformation doit être adaptée. Prenons par exemple le cas de deux fonctions mutuellement récursives :

```
let rec f x = e₁ and g y = e₂ in f ()
```

Une première étape de réécriture serait de transformer les deux valeurs récursives en un couple :

```
let rec fg = (fun x -> e1', fun y -> e2') in (fst fg) ()
```

Ainsi, dans le corps de `f` et `g`, les appels récursifs seront remplacés respectivement par `fst fg` et `snd fg`, soit :

$e'_1 = e_1$ [`f` $\mapsto$ `(fst fg); g` $\mapsto$ `(snd fg)`] et
$e'_2 = e_2$ [`f` $\mapsto$ `(fst fg); g` $\mapsto$ `(snd fg)`].

On note cependant qu'il n'est pas possible d'écrire **`let rec`** `(f, g) = ..` [70], puisqu'il n'est pas possible d'avoir un autre motif que celui d'une variable pour définir une valeur récursive. Maintenant, il est possible d'appliquer la transformation vers le combinateur de point fixe :

```
let fg = fix (fun fg -> (fun x -> e1', fun y -> e2')) in (fst fg) ()
```

Finalement, la construction **`let`** permettant de destructurer une valeur (qui n'est en l'occurrence plus récursive), on peut récupérer les noms de fonctions pour la suite du programme et également comme argument de la fonction donnée à **`fix`** :

```
let (f, g) = fix (fun (f', g') -> (fun x -> e1', fun y -> e2')) in f
  ()
```

Ici, `f'` et `g'` sont des identifiants frais, ce qui est nécessaire pour conserver l'invariant issu de l'alpha-conversion comme expliqué page 58, et donc :

$e'_1 = e_1$ [`f` $\mapsto$ `f'; g` $\mapsto$ `g'`] et
$e'_2 = e_2$ [`f` $\mapsto$ `f'; g` $\mapsto$ `g'`].

On peut alors généraliser cette réécriture pour une nombre quelconque de valeurs récursives, la différence étant alors que les appels récursifs sont remplacés par d'abord une déconstruction du n-uplet pour récupérer la variable correspondant à la valeur précise, puis à un appel de celle-ci.

Néanmoins, cette représentation ne permet pas de représenter les valeurs cycliques qu'il est possible de définir, comme par exemple la liste infinie suivante :

```
let rec l = 0 :: l
```

L'exécution d'un tel programme OCaml termine[35] : le compilateur détecte le cycle et empêche donc l'évaluation de `l` en boucle. Dans le cas de la sémantique présentée ici, le programme ne terminerait pas (on omet les annotations de types, celles-ci n'étant pas utiles) :

```
let l = fix (function l -> 0:: l) (* n'est pas une valeur *)
⇝
let l = 0 :: (fix (function l -> 0:: l)) (* n'est pas une valeur *)
⇝
let l = 0 :: (0 :: (fix (function l -> 0:: l))) (* n'est pas une
    valeur *)
⇝
..
```

L'évaluation d'OCaml se faisant par appel par valeur, le programme ne peut pas terminer. Une solution à ce problème néanmoins serait de rendre paresseux l'appel récursif, par exemple :

```
let l = lazy (fix (function l -> 0 :: (lazy l)))
```

et de remplacer tous les appels à `l` par `Lazy.force l`. Néanmoins, une telle transformation entraîne un changement de type pour `l`, mais aussi pour celui de tous les appels récursifs. On fait donc le choix de ne pas gérer ce type de valeurs dans cette sémantique opérationnelle du Typedtree.

5.5 Extensions impératives

La figure 5.8 décrit la sémantique des constructions impératives. La séquence réduit d'abord la première expression, et, si celle-ci est une valeur, se réduit alors en la seconde. Il existe une sémantique plus stricte (voir l'option `-strict-sequence` du compilateur) pour la séquence dans le cas où la première expression est une valeur :

$$\texttt{()}; e_2 \quad \rightsquigarrow \quad e_2$$

Celle-ci oblige alors la première valeur à se réduire forcément en `()` (unit), néanmoins OCaml ne force pas cette condition dans son comportement normal [1].

Les boucles se réduisent dans l'ordre suivant :

1. Dans le cas de **`for`**, les conditions de départ et d'arrêt sont réduites avant toute chose.
2. Si la condition pour accéder au corps de la boucle est satisfaite, on réduit la boucle en une séquence qui correspond à son corps, puis la boucle à nouveau. Dans le cas de **`for`**, la condition initiale étant une valeur entière, celle-ci est incrémentée (ou décrémementée) de 1.
3. Si la condition n'est pas vérifiée, la boucle se réduit en `()`.

5.6 Exceptions

La figure 5.9 présente les règles de réduction propres aux exceptions, ainsi que les règles additionnelles nécessaires dans le noyau pour la propagation des exceptions qui ont été levées, jusqu'au gestionnaire englobant (**`try .. with`**) le plus proche. Une exception est une valeur de type `exn`, levée à l'aide de l'instruction **`raise`**. Dans le cas de l'exécution d'une instruction

1. ∧Le système de types présenté ici force néanmoins cette condition.

$$\textsc{Seq-Left}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1;\ e_2 \rightsquigarrow v_1;\ e_2, \mathcal{M}_1} \qquad \textsc{Seq-Right}\ \frac{value(v_1) \qquad M \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash v_1;\ e_2 \rightsquigarrow v_1;\ v_2, \mathcal{M}_2}$$

$$\textsc{While-True}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \texttt{true}, \mathcal{M}_1}{\mathcal{M} \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2\ \textbf{done} \rightsquigarrow e_2;\ \textbf{while}\ e_1\ \textbf{do}\ e_2\ \textbf{done}, \mathcal{M}_1}$$

$$\textsc{While-False}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \texttt{false}, \mathcal{M}_1}{\mathcal{M} \vdash \textbf{while}\ e_1\ \textbf{do}\ e_2\ \textbf{done} \rightsquigarrow \textbf{unit}, \mathcal{M}_1}$$

$$\textsc{For-Init}\ \frac{\mathcal{M} \vdash e_j \rightsquigarrow v_j, \mathcal{M}_j}{\mathcal{M} \vdash \textbf{for}\ i = e_i\ \textbf{to}\ e_j\ \textbf{do}\ e\ \textbf{done} \rightsquigarrow \textbf{for}\ i = e_i\ \textbf{to}\ v_j\ \textbf{do}\ e\ \textbf{done}, \mathcal{M}_j}$$

$$\textsc{For-Continue}\ \frac{\mathcal{M} \vdash e_i \rightsquigarrow v_i, \mathcal{M}_i \qquad v_i \leq v_j}{\mathcal{M} \vdash \textbf{for}\ i = e_i\ \textbf{to}\ v_j\ \textbf{do}\ e\ \textbf{done} \rightsquigarrow e[i \mapsto v_i]; \textbf{for}\ i = v_i + 1\ \textbf{to}\ v_j\ \textbf{do}\ e\ \textbf{done}, \mathcal{M}_i}$$

$$\textsc{For-Stop}\ \frac{\mathcal{M} \vdash e_i \rightsquigarrow v_i, \mathcal{M}_i \qquad v_i > v_j}{\mathcal{M} \vdash \textbf{for}\ i = e_i\ \textbf{to}\ v_j\ \textbf{do}\ e\ \textbf{done} \rightsquigarrow \textbf{()}, \mathcal{M}_i}$$

FIGURE 5.8 – Sémantique opérationnelle des constructions impératives
La sémantique prend en compte le modèle mémoire. Les constructions impératives prennent tout leur sens en présence d’effets de bords, d’où leur prise en compte.

$v ::=$ *valeurs*

$\quad \vdots$

$\quad \mid \langle\, \mathtt{raise}\ v : \tau\, \rangle$ *levée d'exception*

Réduction des expressions annotées

$$\textsc{Raise}\ \frac{e \leadsto v}{\mathtt{raise}\ e \leadsto \mathtt{raise}\ v} \qquad \textsc{Raise-Raise}\ \frac{value(v)}{\mathtt{raise}\ (\mathtt{raise}\ v) \leadsto \mathtt{raise}\ v}$$

$$\textsc{Try}\ \frac{e \leadsto v}{\mathtt{try}\ e\ \mathtt{with}\ \overline{p \to e} \leadsto \mathtt{try}\ v\ \mathtt{with}\ \overline{p \to e}} \qquad \textsc{Try-Value}\ \frac{value(v)}{\mathtt{try}\ v\ \mathtt{with}\ \overline{p \to e} \leadsto v}$$

$$\textsc{Try-Raise}\ \frac{value(v)}{\mathtt{try}\ \langle \mathtt{raise}\ v : \tau\rangle\ \mathtt{with}\ \overline{p \to e} \ \leadsto \begin{array}{l} \mathtt{match}\ v\ \mathtt{with} \\ \mid \overline{p \to e} \\ \mid \langle\, \_ : \mathtt{exn}\rangle \to \langle \mathtt{raise}\ v : \tau\rangle \end{array}}$$

Sémantique du noyau en présence d'exceptions

$$\textsc{App-Left-Raise}\ \frac{}{(\mathtt{raise}\ v_1)\ e_2 \leadsto \mathtt{raise}\ v_1} \qquad \textsc{App-Right-Raise}\ \frac{value(v_1)}{v_1\ (\mathtt{raise}\ v_2) \leadsto \mathtt{raise}\ v_2}$$

$$\textsc{Let-Raise}\ \frac{\forall i.value(v_i) \qquad \exists i.v_i \leadsto \mathtt{raise}\ v_i'}{\mathtt{let}\ \overline{\langle p : \sigma\rangle = v}\ \mathtt{in}\ e' \leadsto \mathtt{raise}\ v_i'}$$

FIGURE 5.9 – Sémantique opérationnelle en présence d'exceptions

`try .. with`, il est d'abord nécessaire de réduire son corps. S'il s'agit d'une valeur, l'instruction se réduit en cette valeur. En revanche, si l'évaluation du corps produit une levée d'exception (`raise E`), l'instruction se réduit en un filtrage de l'exception avec un ensemble de motifs correspondant aux exceptions gérées par le gestionnaire courant. Ces motifs sont complétés, si nécessaire, par un motif joker, dont le but est simplement de propager l'exception si le filtrage a échoué à comparer l'exception avec les différents cas prévus.

La propagation des exceptions nécessite d'ajouter de nouvelles règles pour toutes les constructions où il y a une réduction.Dans le cas de l'application, si la fonction ou l'argument produit une levée d'exception, alors le résultat de cette application est la propagation de cette exception. Pour celui des expressions locales, si l'une d'entre elles se réduit en une exception, la réduction va la propager. L'ajout des exceptions dans la sémantique est systématique : pour toute expression, si l'une de ses sous-expressions se réduit en une levée d'exception, alors l'expression se réduit en cette levée d'exception.

5.7 Evaluation paresseuse

Si la construction `lazy` n'est pas exprimable directement en OCaml comme un sucre syntaxique (la généralisation rend le paramètre de type de `lazy` monomorphe, ce qui ne correspond pas à sa définition selon la restriction souple de valeur), on peut toutefois exprimer sa sémantique à l'aide d'expressions du langage, sans ajouter de nouvelle construction. Pour cela, on se basera sur la sémantique des enregistrements et notamment des emplacements mémoire. La sémantique est donnée en figure 5.10.

Une valeur de type `lazy` est représentée par une cellule mémoire, laquelle contient une valeur de type `thunk`. Une valeur est donc soit en suspension (`Thunk`), soit déjà calculée (`Res`). On peut alors implémenter la fonction `force` qui calcule la valeur suspendue ou récupère celle-ci si le calcul a déjà eu lieu. Pour écrire cette fonction dans le langage du Typedtree, il est nécessaire d'annoter tous les nœuds. Heureusement, tous les types nécessaires aux annotations sont présents dans la mémoire courante ainsi que sur le nœud d'application `Lazy.force e`. L'implémentation est donc donnée figure 5.10 [2].

On notera que le motif `lazy p` permet de filtrer sur la valeur d'un calcul suspendu. Par conséquent, l'opération de filtrage est capable de modifier la mémoire, et donc d'effectuer des effets de bord, il devient donc nécessaire de lui donner en argument la mémoire et son résultat contient également la nouvelle mémoire.

Conclusion et perspectives

Ce bref chapitre présente une sémantique opérationnelle du Typedtree. Bien que celui-ci n'a pas vocation à être executé, il est néanmoins intéressant d'étudier sa sémantique dynamique et d'y

2. ∧Les annotations pouvant être particulièrement lourdes à la lecture, notamment les imbrications, toutes ne sont donc pas notées. En pratique, les annotations sont nécessaires seulement sur les feuilles de l'arbre, les nœuds composés (applications, `let .. in ..` ou séquences) étant généralement le résultat des expressions qui les composent.

Implémentation du type **lazy** à l'aide d'un calcul suspendu :

```
type 'a thunk = Res of 'a | Thunk of (unit -> 'a)
type 'a lazy = {
  mutable thunk: 'a thunk;
}
```

Implémentation de `Lazy.force` :

$$
\begin{array}{l}
\langle\ \texttt{Lazy.force}\ \langle\ m\ :\ \tau\ \rangle\ :\ \tau_{arg}\ \rangle\ \texttt{::=}\\[1em]
\quad \textbf{match}\ \langle\ \mathcal{M}\ (m,\ \mathit{thunk})\ :\ \mathcal{M}_\tau\ (m)\ \rangle\ \textbf{with}\\
\quad |\ \langle\ \texttt{Thunk}\ \langle\ \texttt{f}\ :\ \texttt{unit}\ \rightarrow\ \tau_{arg}\ \rangle\ :\ \mathcal{M}_\tau\ (m)\ \rangle\ \rightarrow\\
\qquad \langle\ \textbf{let}\ \langle\ \texttt{x}\ :\ \tau_{arg}\ \rangle\ =\ \langle\ \texttt{f}\ :\ \texttt{unit}\ \rightarrow\ \tau_{arg}\ \rangle\ \langle\ \texttt{()}\ :\ \texttt{unit}\ \rangle\ \textbf{in}\\
\qquad \langle\ \mathcal{M}\ (m,\ \mathit{thunk})\ :\ \tau\ \rangle\ \leftarrow\ \langle\ \texttt{Res x}\ :\ \mathcal{M}_\tau\ (m)\ \rangle;\\
\qquad \langle\ \texttt{x}\ :\ \tau_{arg}\ \rangle\\[1em]
\quad |\ \langle\ \texttt{Res}\ \langle\ \texttt{x}\ :\ \tau_{arg}\ \rangle\ :\ \mathcal{M}_\tau\ (m)\ \rangle\ \rightarrow\\
\qquad \langle\ \texttt{x}\ :\ \tau_{arg}\ \rangle
\end{array}
$$

Réduction des expressions annotées

$$
\textsc{Lazy}\ \frac{\mathit{let}\ m \notin \mathit{dom}(\mathcal{M})}{\mathcal{M} \vdash \textbf{lazy}\ e \rightsquigarrow \langle m : \tau \rangle, (\mathcal{M}, (m : \tau = [\mathit{thunk} = \mathit{Thunk}\ e]))}
$$

Filtrage de motifs

$$
\text{match}\ (\textbf{lazy}\ \text{p}, \langle\ m : \tau\ \rangle, \mathcal{M}) \quad = \quad \text{let}\ v, \mathcal{M}' = \text{Lazy.force}\ m\ \mathcal{M}\ \text{in match}\ (\text{p}, v, \mathcal{M}')
$$

FIGURE 5.10 – Sémantique opérationnelle des expressions paresseuses

Une valeur *lazy* est encodée dans un enregistrement mutable, dont le champ est soit un calcul suspendu (un *thunk*) soit le résultat d'un tel calcul.

appliquer l'instanciation sémantique présentée section 2.5 (page 46). Cette définition reste nécessaire dans le cadre d'une preuve future de sûreté, il s'agit donc d'une première étape vers celle-ci. Il n'existe pas non plus d'implémentation d'un évaluateur, ce qui est laissé pour un travail futur. On remarque donc que chacune de ces constructions introduit un nouveau concept nécessaire à sa réduction : gestion d'un environnement de mémoire dans le cas de la mutabilité, réductions alternatives pour la propagation d'exceptions, mais également l'apparition de constructions intermédiaires nécessaires à l'évaluation. Ainsi, la récusivité mutuelle nécessite l'ajout d'un combinateur de point fixe et l'utilisation de la mémoire des opérations d'écriture et lecture de celle-ci. Pour que le langage puisse être toujours typable et vérifiable selon l'algorithme du chapitre précédent, on définit les règles de vérification nécessaires.

Finalement, ce chapitre consistue une première étape dans le cadre d'uen preuve de sûreté et de formalisation éventuelle du langage.

Conclusion et perspectives

L'implémentation de l'inférence de types dans un compilateur tel qu'OCaml n'est pas infaillible. Les différents rapports d'erreurs présentant diverses erreurs de segmentations et autres défaillances qui auraient dûes être impossible en présence du système de types le prouvent très bien. L'utilisation d'un système de types fort pour l'implémentation du compilateur ne garantit que l'absence d'erreurs de types *pendant la compilation du compilateur*, mais pas dans le résultat produit par ce même compilateur. L'utilisation de types dépendants permettrait de prouver que l'algorithme du programme à compiler respecte bien une spécification donnée, celle d'un algorithme d'inférence de types dont on aurait prouvé la sûreté [63]. Dans le cadre d'un compilateur déjà riche et dont l'implémentation est suffisament complexe pour être difficile à prouver, cela se révèle pratiquement impossible.

En revanche, il est possible d'utiliser les données issues de ce compilateur pour en vérifier un certain nombre de propriétés. Dans le cas de l'inférence, il est possible de récupérer un arbre de syntaxe abstraite dont chaque nœud est annoté par des informations de types. Vérifier la cohérence de cet arbre revient alors à s'assurer que le résultat de l'inférence est correct pour le programme correspondant sachant la spécification du système de type.

Néanmoins, on en revient au problème initial : quelle assurance avons-nous que ce vérificateur est lui-même correct vis-à-vis de la spécification du langage ? Après tout, comme l'inférence, un tel programme de vérification n'est pas exempt d'erreurs dans son implémentation. Cela étant, il est possible de réduire autant que possible ces erreurs humaines de plusieurs manières :

- La spécification elle-même peut être écrite de manière exécutable, de telle sorte que son implémentation soit pratiquement dérivable mécaniquement de celle-ci.
- L'implémentation n'a pas pour vocation à être rapide contrairement à celle de l'inférence qui est elle-même intégrée à la chaîne de compilation. Ainsi, le code est écrit de manière simple, sans optimisation, sans rechercher les performances. En effet, l'outil dérivé de cet algorithme n'a pas pour vocation à être utilisé durant le développement, mais plutôt comme une étape de certification au moment où le programme à vérifier est complet, il ne s'agit donc pas de le rendre rapide pour améliorer les temps de développement. Pour rendre celui-ci simple à comprendre et surtout à comparer à la spécification, le delta entre les règles et l'algorithme est rendu minime, ce qui explique également l'absence d'optimisations.

L'objectif de cette thèse était double : formaliser un large sous-ensemble du système de types d'OCaml et écrire un vérificateur d'arbres annotés suivant cette spécification. Dans le premier cas, l'intérêt était de produire un document capable de résumer ce système de types et d'expliquer de manière moins abstraite et plus algorithmique le typage du langage. En particulier, il s'agissait de remplacer l'unification, véritable couteau suisse de l'inférence, par un ensemble d'opérations distinctes et dont l'usage est spécifique à une étape de la vérification. Une telle formalisation du langage dans son ensemble n'existait pas notre connaissance. Le second était [3] une conséquence de cette formalisation : si celle-ci est pensée pour être exécutable alors il doit être simple d'en dériver un algorithme. En considérant cet arbre annoté comme une preuve de typage, alors vérifier la cohérence de ces preuves est équivalent à écrire un vérificateur qui implémente le système de types sous-jacent.

Cette thèse décrit la vérification d'arbres annotés (TAST) d'abord pour MiniML, puis OCaml. L'intérêt de cette première partie sur un langage restreint est de pouvoir définir le cadre de travail pour la vérification d'un tel type d'arbres, mais également de pouvoir se baser sur un langage simple contenant les données idéales. En particulier, on peut remarquer que l'absence de la quantification sur les types généralisés lors de la vérification de programmes OCaml complexifie certains algorithmes. De plus, le langage étant restreint, il est également plus simple à formaliser dans un assistant de preuves tel que Coq. Finalement, en suivant l'idée de pouvoir définir un système de types exécutable, celui-ci étant relativement simple, *i.e.* possède un nombre restreint de constructions et de vérifications, la correspondance entre l'implémentation et le système formel est directe.

La seconde partie décrit un sous-ensemble relativement important du langage OCaml. Le second chapitre formalise plusieurs aspects du langage : chaque construction est étudiée en considérant sa représentation concrète et non un objet théorique permettant de la représenter. Par exemple, les références sont étudiées sous la forme d'enregistrements mutables, puisqu'il s'agit de leur implémentation, et les locations (emplacements mémoire) ne sont introduites qu'au moment de définir la sémantique opérationnelle du TAST d'OCaml, le Typedtree. Contrairement à l'implémentation de l'inférence qui réutilise autant que possible ses mécanismes pour typer des constructions complexes telles que les types algébriques gardés, ou l'unification qui vérifie plusieurs propriétés de manière implicite, la vérification est définie de telle sorte que les mécanismes soient indépendants. Ainsi, l'unification propre à l'inférence qui permet entre autre de garantir un ensemble de propriétés du système de types est remplacée par plusieurs opérations vérifiant chaque fois une unique propriété. Dans le cas des types algébriques gardés, la vérification utilise toute une mécanique d'*union-find* pour gérer les équations de types (ou classes d'équivalence), là où l'inférence réutilise l'implémentation des abréviations. Ces deux approches se justifient : l'inférence est implémentée dans un compilateur, écrite de manière efficace, et on sait exprimer ces constructions au moins en partie à l'aide du système en place ; dans le cas de la vérification, que l'on voudrait plus proche d'un système formel, il est plus intéressant de rendre ces mécaniques aussi indépendantes que possible pour en faciliter la compréhension. Finalement, cette partie s'accompagne d'une sémantique opérationnelle à petits pas sur les TASTs, sémantique parfois assez particulière, notamment parce qu'elle nécessite des calculs sur les types pour garantir une éventuelle propriété de préservation, et donc la sûreté du langage vis-à-vis du sytème étudié.

À cette seconde partie est adossé un vérificateur de Typedtree écrit en OCaml, dont l'implémentation est le reflet du système formel, et qui couvre OCaml entièrement, à l'exception de quelques constructions du langage, telles que les types équi-récursifs et les modules récursifs. Cet

3. $\wedge$et l'est toujours

outil est donc capable de vérifier des programmes après typage et d'en extraire potentiellement des erreurs liées à une inférence défaillante. Par exemple, certains *bugs* connus de l'inférence ont effectivement pu être rattrapés par ce vérificateur.

La perspective de formaliser le système de types d'un langage tel qu'OCaml est importante. Il s'agit d'un langage fonctionnel connu pour son typage statique fort et utilisé industriellement et non plus seulement réservé au milieu de la recherche académique. Être capable d'ajouter une passe de correction après l'inférence permet d'assurer plus de garanties de sûreté sur les programmes compilés avec OCaml. Cela étant, il reste certains détails du système de types à formaliser, tels que les objets ou le système de modules, bien qu'ils soient intégrés dans le vérificateur. De plus, il n'y a pas de preuve de sûreté sur le système de types ainsi formalisé, qui est laissée pour un travail ultérieur. L'espoir de ce travail est d'apporter une meilleure compréhension du sytème de types de ce langage particulièrement riche, et d'apporter en quelques sorte un rapport technique pour maintenir le typeur de celui-ci. Le système formel présenté introduit parfois quelques différences vis-à-vis de l'implémentation actuelle de l'inférence, mais reste néanmoins capable de vérifier un très grand nombre des programmes OCaml. Le vérificateur associé n'est pas exempt de défaut, et est écrit de manière à ne pouvoir vérifier que des programmes acceptés par l'inférence, mais cela est inhérent au défi technique initial de travailler sur le résultat de celle-ci.

Une telle méthode de vérification pourrait être utilisée de plusieurs manières, par exemple comme un moyen d'ajouter une forme de confiance supplémentaire en envers l'implémentation de l'inférence de types. Un tel vérificateur a toute sa place dans un contexte où la sécurité et la sûreté peuvent être critiques. Une autre utilisation possible serait dans le cas d'une utilisation conjointe avec l'inférence, permettant alors à l'avenir de distinguer les deux mécanismes dans une chaîne de compilation : en effet, le moteur d'inférence s'occupe très généralement de la vérification de manière implicite, rendant sa maintenance et son évolution parfois complexe.

Il peut être intéressant de comparer cette méthode de vérification de l'inférence de type à celles existantes. On peut distinguer deux cas : celui où l'implémentation de l'inférence est prouvée correcte, et celui où les transformations conservent les types tout au long de la compilation. Dans le premier cas, l'inférence peut être prouvée correcte vis-à-vis du système de type qu'elle implémente, soit à l'aide d'une preuve manuelle, soit d'une preuve mécanique : le moteur d'inférence est par exemple écrit à l'aide d'un assistant de preuve, et l'algorithme résultant extrait dans le langage visé. De cette manière, la correction est prouvée et les erreurs de types correspondent effectivement à des éléments du programme source. Néanmoins, cette technique requiert de réécrire l'inférence de zéro, et peut poser problème pour effectuer certaines optimisations sur l'inférence elle-même, celles-ci n'étant pas toujours simples à prouver. Le second cas prend le parti de propager les types en compilant le programme inféré vers un ou plusieurs langages intermédiaires euxmêmes typés, et donc potentiellement vérifiables. Cette méthode permet notamment d'utiliser des types pour effectuer certaines optimisations, mais aussi de garantir la sûreté des programmes après optimisation. En revanche, le système de types de ces langages intermédiaires n'est jamais identique à celui du langage source. En effet, ces langages intermédiaires simplifient les constructions du langage dit *de surface* vers des constructions plus bas niveau, et perdent donc toute information sur la construction du langage source. De ce fait, si une défaillance de l'inférence intervient et est rattrapée par la vérification du langage intermédiaire suivant, le rapport d'erreur produit ne sera probablement pas d'une précision qui permettrait de rapporter celle-ci à la manière dont a été inféré le programme source. Il deviendrait donc difficile de vérifier efficacement l'inférence de types du langage.

La vérification de l'inférence telle que présentée dans cette thèse est située entre ces deux écoles. De l'inférence prouvée, elle garde la proximité du langage vérifié avec le système de types qui a permis l'inférence. Les defauts de cohérence peuvent donc être directement rapportés au programme source et plus facilement identifiés dans l'inférence. Des représentations intermédiaires typées, elle garde la vérification directe de l'inférence, sans avoir besoin de réécrire celle-ci, et donc comme une étape potentiellement supplémentaire de la compilation du programme. De cette manière, la vérification des arbres de syntaxe annotés est un compromis intéressant s'il s'agit d'ajouter une forme de confiance supplémentaire aux programmes, mais aussi au développement du compilateur, dans le cas particulier d'un compilateur déjà existant. Bien entendu, cela n'apporte ni la garantie d'un compilateur certifié et prouvé mécaniquement, comme cela serait le cas avec un typeur prouvé correct, ni la garantie que les optimisations conservent les types dans le cas d'un langage intermédiaire typé.

Cette thèse est une base pour de futurs travaux, qui s'orienteraient vers deux axes principaux :

— Dans ce manuscrit, seul le noyau d'OCaml est formalisé. Il serait donc intéressant d'y ajouter le système de modules, ainsi que celui des objets, qui pour le moment n'existent que dans le vérificateur. De plus, les ajouts récents au langage (à partir de la version 4.03) ne sont pas gérés.

— Bien que le noyau soit formalisé, il n'existe pour le moment pas de preuve de sûreté du système de types tel qu'il est décrit chapitre 4. Une telle preuve permettrait donc de s'assurer que tout programme accepté par le vérificateur n'aurait aucune erreur de types à l'exécution, et ce peu importe l'implémentation de l'inférence.

Annexe A

Implémentation d'un MiniML en Coq avec `let` destructurant

Finalement, cette courte annexe présente une implémentation légèrement différente d'un MiniML dans lequel ont été ajoutés des couples et dont les `let` sont dits destructurants : le motif qu'ils lient à une valeur locale n'est plus restreint à une variable mais peut contenir plusieurs variables. Les paramètres de fonctions peuvent également être des motifs. Cette implémentation rajoute alors une forme très simple de filtrage de motifs, et sa structure de données (figure A.1) est plus proche de celle du Typedtree d'OCaml, présentée chapitre 3. Cette implémentation se base une fois de plus sur la représentation dite localement sans nom [4].

Cette structure de données diffère principalement sur les annotations de types : celles-ci ne sont plus propres à chaque construction, le TAST est construit comme sa formalisation : un nœud de TAST contient un type et une expression. Ces expressions contiennent elles-même des sous-nœuds annotés. Cette représentation nécessite de redéfinir les principes d'induction normalement générés par Coq, ceux-ci n'étant pas suffisants. La technique utilisée est empruntée à Adam Chlipala [17] pour la définition de principes d'induction de types inductifs imbriqués. Le principe d'induction est donné en figure A.2. L'exemple donné ne concerne que les expressions, mais il en va de même pour les motifs dont la structure est similaire.

Les différentes opérations sont écrites de manière inductive, de la même manière que montrée précédemment. Pour chacune de celles-ci, il existe un algorithme qui, sachant un type (ou une expression) retourne une valeur booléenne ou une valeur optionnelle contenant le résultat de l'opération. Le but étant de prouver que l'algorithme est correct vis-à-vis de sa spécification inductive, autrement dit que pour toute propriété formelle vraie pour un terme, ce terme est également accepté par l'algorithme correspondant, mais également que pour tout terme accepté par l'algorithme, alors la propriété inductive est prouvable sur le-dit terme. La finalité étant ici d'écrire un vérificateur exécutable pour un MiniML annoté dont l'algorithme respecte les spécifications du système de types.

Un exemple simple est donné en figure A.3. La fonction `equiv_impl` retourne un booléen dont la valeur est `true` si les deux types sont équivalents, et `false` dans le cas contraire. La propriété de correction est simple : pour tous types `t1` et `t2`, si on peut construire une valeur `equiv t1 t2` alors `equiv_impl t1 t2` se réduit en `true`, et inversement. La preuve se fait alors

```
Module Tree.
  Definition ident := Ident.t.

  Inductive variance : Set :=
  | pos_var: variance | neg_var: variance | invar: variance.

  Inductive type : Set :=
  | Tvar: nat → type
  | Tfvar: var → type
  | Tarrow: type → type → type
  | Ttuple: type → type → type
  | Tconstr: nat → type.

  Inductive sch : Set :=
  | Tforall: nat → type → sch.

  Definition annot (desc : Set) : Set := desc * type.

  Definition annot_sch (desc : Set) : Set := desc * sch.

  Inductive pat : Set :=
  | pat_var: nat → pat
  | pat_tuple: annot pat → annot pat → pat.

  Definition typed_pat : Set := annot_sch pat.
  Definition typed_subpat : Set := annot pat.

  Inductive term : Set :=
  | term_var: nat → term
  | term_fvar: var → term
  | term_function: typed_subpat → annot term → term
  | term_app: annot term → annot term → term
  | term_let: typed_pat → annot term → annot term → term
  | term_tuple: annot term → annot term → term.

  Definition typed_term : Set := annot term.

End Tree.
```

FIGURE A.1 – Structure de données représentant un TAST pour MiniML destructurant

```
Module Principles.
  Import Tree.

  Section term_ind_stronger.
    Variable P : term → Prop.

    Hypothesis term_var_case: forall n, P (term_var n).

    Hypothesis term_fvar_case: forall v, P (term_fvar v).

    Hypothesis term_function_case: forall arg targ body tbody,
        P body → P (term_function (arg, targ) (body, tbody)).

    Hypothesis term_app_case: forall e1 t1 e2 t2,
        P e1 → P e2 → P (term_app (e1, t1) (e2, t2)).

    Hypothesis term_let_case: forall p sch e1 t1 e2 t2,
        P e1 → P e2 → P (term_let (p, sch) (e1, t1) (e2, t2)).

    Hypothesis term_tuple_case: forall e1 t1 e2 t2,
        P e1 → P e2 → P (term_tuple (e1, t1) (e2, t2)).

    Fixpoint term_ind_stronger (t: term) : P t :=
      match t with
      | term_var n ⇒ term_var_case n
      | term_fvar v ⇒ term_fvar_case v
      | term_function (arg, targ) (body, tbody) ⇒
        term_function_case arg targ body tbody (term_ind_stronger body)
      | term_app (e1, t1) (e2, t2) ⇒
        term_app_case e1 t1 e2 t2 (term_ind_stronger e1) (term_ind_stronger e2)
      | term_let (p, sch) (e1, t1) (e2, t2) ⇒
        term_let_case p sch e1 t1 e2 t2 (term_ind_stronger e1) (term_ind_stronger
             e2)
      | term_tuple (e1, t1) (e2, t2) ⇒
        term_tuple_case e1 t1 e2 t2 (term_ind_stronger e1) (term_ind_stronger e2)
      end.
  End term_ind_stronger.

End Principles.
```

FIGURE A.2 – Principe d'induction d'un TAST pour MiniML destructurant

```
(** Specification inductive *)
Inductive equiv : context → type → type → Prop :=
| eq_fvar : forall ctx v, equiv ctx (Tfvar v) (Tfvar v)
| eq_arrow : forall ctx t1 t2 t1' t2',
    equiv ctx t1 t1' → equiv ctx t2 t2' →
    equiv ctx (Tarrow t1 t2) (Tarrow t1' t2')
| eq_tuple : forall ctx t1 t2 t1' t2',
    equiv ctx t1 t1' → equiv ctx t2 t2' →
    equiv ctx (Ttuple t1 t2) (Ttuple t1' t2')
| eq_constr : forall ctx v,
    equiv ctx (Tconstr v) (Tconstr v).

(** Verificateur *)
Fixpoint equiv_impl (ctx:context) t1 t2 :=
  match t1, t2 with
  | Tfvar v1, Tfvar v2 ⇒ isTrue (v1 = v2)
  | Tarrow t11 t12, Tarrow t21 t22 ⇒
    andb (equiv_impl ctx t11 t21) (equiv_impl ctx t12 t22)
  | Ttuple t11 t12, Ttuple t21 t22 ⇒
    andb (equiv_impl ctx t11 t21) (equiv_impl ctx t12 t22)
  | Tconstr v1, Tconstr v2 ⇒ isTrue (v1 = v2)
  (* | Ttuple ts1, Ttuple ts2 ⇒ false *)
  (*   (* equiv_list_aux equiv_impl true ts1 ts2 *) *)
  | _, _ ⇒ false
  end.

(* Preuve de correction *)
Theorem equiv_impl_if_equiv: forall ctx t1 t2,
    equiv ctx t1 t2 → equiv_impl ctx t1 t2 = true.
Proof. (* ... *) Qed.

Theorem equiv_if_equiv_impl: forall ctx t1 t2,
    equiv_impl ctx t1 t2 = true → equiv ctx t1 t2.
Proof. (* ... *) Qed.

Theorem equiv_impl_correct: forall ctx t1 t2,
    equiv ctx t1 t2 ↔ equiv_impl ctx t1 t2 = true.
Proof.
  intros; split. apply equiv_impl_if_equiv. apply equiv_if_equiv_impl.
Qed.
```

FIGURE A.3 – Spécification et implémentation de l'équivalence

```
Inductive check_term : context → typed_term → Prop :=
| chk_fvar : (* ... *)

| chk_fun : (* ... *)

| chk_app : (* ... *)

| chk_let: forall ctx p sch e1 t1 e2 t2 tres vs S,
   check_pat ctx (Map.empty type) (p, sch) vs →
   check_term ctx (e1, t1) →
   inst_sch ctx empty (fv t1 \u fv_term e1 \u fv t2 \u fv_term e2)
          (Tforall (sch_arity sch) t1) sch S →
   (~ nonexpansive e1 → check_gen_expans_sch ctx sch pos_var) →
   (forall fvs' pvs itbl,
      intro_vars (fv t1 \u fv_term e1 \u fv t2 \u fv_term e2) vs =
      (fvs', pvs, itbl) →
      check_term (union_values ctx pvs) (open_aterm_list itbl (e2, t2))) →
   wf ctx tres →
   equiv ctx t2 tres →
   check_term ctx (term_let (p, sch) (e1, t1) (e2, t2), tres)
| chk_tuple: (* ... *).
```

FIGURE A.4 – Spécification du système de types

par induction sur le terme `equiv t1 t2` dans le premier cas, et sur `t1` et `t2` dans le second.

Au final, on définit la spécification du système de types de notre MiniML en figure A.4 et le vérificateur implémentant celle-ci en figure A.5. Dans un souci de lisibilité mais aussi de présentation des différentes propriétés nécessaires à la vérification de MiniML, seule la règle du **`let`** est gardée, bien qu'il s'agisse de la plus complexe et verbeuse. La règle de vérification est la suivante :

$$\textsc{Let}\ \frac{\Gamma, \emptyset \vdash^{\langle}_{p} p : \sigma\rangle \Rightarrow (\overline{v : \sigma_v}) \qquad \Gamma \vdash \langle e : \tau\rangle \qquad \Gamma, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \sigma \Rightarrow \theta \qquad check\_gen(\Gamma, \sigma, e) \\ let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} (\overline{v : \sigma_v}) \qquad \Gamma' \vdash \langle e' : \tau'\rangle \qquad \Gamma \vdash_{\tau} \tau\ wf \qquad \Gamma' \vdash_{\tau} \tau \equiv \tau'}{\Gamma \vdash \langle \mathbf{let}\ p = e\ \mathbf{in}\ e' : \tau\rangle}$$

Prenons la spécification : le type de l'expression globale (τ) est représenté par `tres`, `t1` correspond au type de l'expression liée localement (τ), `sch` celui de son schéma de type (σ) et `t2` le type de l'expression courante du **`let`** (τ'). Pour que l'expression annotée soit cohérente, on vérifie que :

— Le motif `p` est cohérent et on en extrait un ensemble de variables `vs`.
— L'expression locale `e` est cohérente.
— Le type de l'expression locale est une instance du schéma de types annoté au motif. On remarque d'ailleurs que la définition inductive de la propriété d'instanciation prend en argument un ensemble de variables liées, qui lui seront utiles pour générer des variables fraîches au moment d'ouvrir le quantificateur.
— Si l'expression locale est expansive, autrement dit si elle peut présenter des effets de bord obversables, il faut s'assurer que les variables généralisées l'ont été à juste titre.

```
Function check_term_impl (ctx : context) (t: typed_term)
  { measure term_nodes t } :=
  let (exp, ty) := t in
  match exp with
  | term_var _ ⇒ false

  | term_fvar v ⇒ (* ... *)

  | term_function (arg, targ) (body, tbody) ⇒ (* ... *)

  | term_app (e1, t1) (e2, t2) ⇒ (* ... *)

  | term_let (p, sch) (e1, t1) (e2, t2) ⇒

    match check_pat_impl ctx (Map.empty type) (p, sch) with
    | Some vs ⇒
      let fvs := (fv t1 \u fv_term e1 \u fv t2 \u fv_term e2) in
      let '(_, pvs, itbl) := intro_vars fvs vs in
      let t' := open_aterm_list itbl (e2, t2) in
      check_inst_aux ctx fvs sch t1 &&
      check_term_impl ctx (e1, t1) &&
      check_gen_aux ctx sch e1 &&
      check_term_impl (union_values ctx pvs) t' &&
      equiv_impl ctx t2 ty &&
      wf_impl ctx ty
    | None ⇒ false
    end

  | term_tuple (e1, t1) (e2, t2) ⇒ (* ... *)
  end.
```

FIGURE A.5 – Implémentation du vérificateur

```
Lemma check_term_if_impl: forall ctx exp ty,
   check_term_impl ctx (exp, ty) = true → check_term ctx (exp, ty).
Proof with eauto. (* ... *) Qed.

Lemma impl_if_check_term: forall ctx exp ty,
   check_term ctx (exp, ty) → check_term_impl ctx (exp, ty) = true.
Proof with eauto. (* ... *) Qed.

Lemma check_term_impl_correct: forall ctx exp ty,
   check_term ctx (exp, ty) ↔ check_term_impl ctx (exp, ty) = true.
Proof with eauto.
  intros. split. apply check_term_if_impl. apply impl_if_check_term.
Qed.
```

FIGURE A.6 – Preuve de correction du vérificateur vis-à-vis de la spécification

— L'expression courante est cohérente dans l'environnement initial auquel on a ajouté les variables liées dans le motif.
— Le type de l'expression globale est bien formé dans l'environnement courant.
— Le type de l'expression courante et celui du **`let`** sont équivalents.

L'implémentation du vérificateur suit exactement ces étapes de vérification en appelant chaque fois l'implémentation équivalente de la propriété demandée. La fonction `intro_vars` permet de générer des variables fraîches pour un ensemble de variables à ouvrir, et retourne l'ensemble des variables liées après l'appel (`fvs'`), les variables ouvertes (`pvs`) et une table associant les indices de De Bruijn aux variables auxquelles ils se substituent.

L'implémentation est définie à l'aide du mot clé `Function`, une construction de Coq permettant d'écrire des fonctions récursives dont la terminaison n'est pas triviale d'après la structure des appels récursifs. La terminaison est montrée par la décroissance du nombre de nœuds au fur et à mesure des appels récursifs successifs, via la condition en en-tête `measure term_nodes t`. Cette définition génère alors des obligations de preuve demandant de montrer cette décroissance. Une fois définie, il est possible de prouver la correction de ce vérificateur (figure A.6), c'est-à-dire que pour toute expression dont il est possible de construire une preuve de cohérence, alors l'implantation retourne **`true`**, et inversement.

Annexe B

Spécification complète du noyau d'OCaml

Cette dernière annexe condense l'ensemble des règles de vérification et la sémantique opérationnelle du Typedtree, dont les explications et commentaires ont été donnés dans les chapitres 3.1, 4 et 5.

B.1 Définition du Typedtree

B.1.1 Algèbre de types

$\tau ::= {}'a$	*free type variable*
$\mid (l : \tau) \rightarrow \tau$	*function with labeled argument*
$\mid \tau_1 * .. * \tau_n$	*tuple*
$\mid (\overline{\tau})\ \mathtt{t}$	*type constructor*
$\mid [(\rho)\ K_1\ \mathbf{of}\ \tau_1\ ;\ ..\ ;\ K_n\ \mathbf{of}\ \tau_n > K_i\ ..\ K_j\ ]$	*polymorphic variant*
$\mid \alpha$	*universally quantified variable*
$\mid \forall\overline{\alpha}.\tau$	*universally quantified type*
$\mid (\ \mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{\mathtt{t} = \tau})$	*first-class module pack*
$\rho ::= {}'a \mid \epsilon$	*row variable*

B.1.2 Expressions

$e ::= \langle\ e' : \tau\ \rangle$	*expression node*
$e' ::=$	*expression descriptions*
$x \mid c$	
$\mid \mathbf{function}\ \overline{p\ ?\mathbf{when}\ c \rightarrow e} \mid e_1\ e_2$	
$\mid \mathbf{let}\ \overline{p = e}\ \mathbf{in}\ e \mid \mathbf{let\ rec}\ \overline{p = e}\ \mathbf{in}\ e$	*ML classical constructions*
$\mid \mathbf{function}\ {\sim}l : p\ ?\mathbf{when}\ c \rightarrow e$	
$\mid e_1\ ({\sim}l : e_2)$	*labeled functions arguments*

$\mid$ **match** e **with** $\overline{p\ ?\textbf{when}\ c \rightarrow e}$	*pattern matching*
$\mid$ **try** e **with** $\overline{p\ ?\textbf{when}\ c \rightarrow e}$	*exceptions catching*
$\mid e_1, .., e_n \mid K\ (\overline{e})$	*tuples and variant constructors*
$\mid$ \`$K\ e$	*polymorphic variant constructors*
$\mid \{\overline{l = e;}\} \mid e.l \mid e_1.l \leftarrow e_2$	*records*
$\mid e_1.(e_2) \mid e_1.(e_2) \leftarrow e_3$	*arrays*
$\mid$ **if** e_1 **then** e_2 **else** e_3	*conditional*
$\mid e_1$ **;** e_2	*sequence*
$\mid$ **for** $x = e_1$ **to/downto** e_2 **do** e_3 **done**	
$\mid$ **while** e_1 **do** e_2 **done**	*imperative loops*
$\mid$ **let module** $I = M$ **in** $e \mid$ (**module** M)	*first-class modules*
$\mid$ **lazy** e	*lazyness*
$\mid$ **assert** e	*assertions*
$\mid (e : t_1 :> t_2) \mid (e : t)$	*coercions and constraints*
$\mid$ **fun** (**type** a) $\rightarrow e$	*local abstract type introduction*
$\mid$ **let open** P **in** e	*local module opening*
$p ::= \langle\ p' : \tau\ \rangle$	*pattern node*
$p' ::=$	*pattern description*
$x \mid c \mid p$ **as** x	*variable, constant and aliases pattern*
$\mid (p_1, .., p_n)$	
$\mid K\ (\overline{p})$	
$\mid$ \`$K\ (\overline{p})$	*tuple, variant and polymorphic variant constructor pattern*
$\mid \{\ \overline{l = e;}\ \} \mid [\ \overline{p}\ ]$	*record and array pattern*
$\mid (\ p_1$ **|** $p_2\ )$	*or pattern*
$\mid$ **lazy** p	*lazy pattern*
$\mid (p : t)$	*coercions and constraints*

B.2 Règles de vérification

B.2.1 Vérification et manipulation de types

Équivalence

$$\text{EQUIV-VAR}\ \Gamma, \Phi \models_{\tau} {}'a \equiv {}'a \qquad \text{EQUIV-FUN}\ \frac{l_1 = l'_1 \qquad \Gamma, \Phi \models_{\tau} \tau_1 \equiv \tau'_1 \qquad \Gamma, \Phi \models_{\tau} \tau_2 \equiv \tau'_2}{\Gamma, \Phi, \theta \models_{\tau} (l_1 : \tau_1) \to \tau_2 \equiv (l'_1 : \tau'_1) \to \tau'_2}$$

$$\text{EQUIV-TUPLE}\ \frac{\forall i_{>1}.\ \Gamma, \Phi \models_{\tau} \tau_i \equiv \tau'_i}{\Gamma, \Phi \models_{\tau} \tau_1 * .. * \tau_n \equiv \tau'_1 * .. * \tau'_n} \qquad \text{EQUIV-CONSTRUCT}\ \frac{\forall i_{>1}.\ \Gamma, \Phi \models_{\tau} \tau_i \equiv \tau'_i}{\Gamma, \Phi \models_{\tau} (\overline{\tau})\ \mathtt{p} \equiv (\overline{\tau'})\ \mathtt{p}}$$

$$\text{EQUIV-CONSTRUCT-EXP}\ \frac{let\ \tau = expand(\Gamma, \Phi, \mathtt{t}, \overline{\tau}) \qquad \Gamma, \Phi \models_{\tau} \tau \equiv \tau'}{\Gamma, \Phi \models_{\tau} (\overline{\tau})\ \mathtt{t} \equiv \tau'}$$

$$\text{EQUIV-POLY}\ \frac{\Gamma, \Phi \models_{\tau} \tau \equiv \tau'}{\Gamma, \Phi \models_{\tau} \forall \overline{\alpha}.\ \tau \equiv \forall \overline{\alpha'}.\ \tau'} \qquad \text{EQUIV-UNIVAR}\ \Gamma, \Phi \models_{\tau} \alpha \equiv \alpha'$$

$$\text{EQUIV-RIGID}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi \models_{\tau} \mathtt{t} \equiv \tau}$$

$$\text{EQUIV-VARIANT}\ \frac{\begin{array}{c} T_{pres_i}\ ..\ T_{pres_j} = T'_{pres_i}\ ..\ T'_{pres_j} \\ \Gamma, \Phi \models_{\tau} \rho \equiv \rho' \qquad \overline{T} \in \overline{T'} \qquad \forall i_{>1}.\Gamma, \Phi \models_{\tau} \tau_{var_i} \equiv \tau'_{var_i} \end{array}}{\Gamma, \Phi \models_{\tau} [(\rho)\ \overline{T\ of\ \tau_{var}} > T_{pres_i}\ ..\ T_{pres_j}] \equiv [(\rho')\ \overline{T'\ of\ \tau'_{var}} > T'_{pres_i}\ ..\ T'_{pres_j}]}$$

$$\text{EQUIV-NIL}\ \Gamma, \Phi \models_{\tau} \epsilon \equiv \epsilon$$

$$\text{EQUIV-PACKAGE}\ \frac{\begin{array}{c} let\ S = Sig(\Gamma, \mathtt{P}\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t} = \tau}) \\ let\ S' = Sig(\Gamma, \mathtt{P}'\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t}' = \tau'}) \qquad \Gamma, \Phi \models_{\tau} S' \equiv S \end{array}}{\Gamma, \Phi \models_{\tau} (\mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t} = \tau}) \equiv (\mathbf{module}\ \mathtt{P}'\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t}' = \tau'})}$$

Instanciation

Substitution Fonction finie associant une variable de type à un type.
La substitution initiale est définie de la manière suivante :

$$\theta_\forall(\Gamma) = \{\beta \mapsto \beta \mid \beta \in \Gamma.GenVars\}$$

- $\theta ::= \emptyset \mid \theta \oplus [\alpha \mapsto \tau]$
- $dom(\theta)$: ensemble des clés (variables) de la substitution.
- $wf(\theta)$: prédicat indiquant l'unicité des clés dans la substitution.
- $\theta \subseteq \theta'$: prédicat de sous-ensemble.
- $\theta_{init}(\forall\overline{\alpha}.\tau) = [\alpha \mapsto \alpha | \forall \alpha. \alpha \in fv(\tau) - \overline{\alpha}]$
- $apply(\theta, \tau)$: substitution des variables libres de τ par leur instance dans θ.

Domaine :

$$\begin{array}{lcl} dom(\emptyset) & \rightsquigarrow & \emptyset \\ dom(\theta \oplus [\alpha \mapsto \tau]) & \rightsquigarrow & dom(\theta) \cup \{\ \alpha\} \end{array}$$

Bonne formation :

$$\textsc{Wf-Subst-Empty}\ wf(\emptyset) \qquad \textsc{Wf-Subst-Bind}\ \frac{\alpha \notin dom(\theta) \qquad wf(\theta)}{wf(\theta \oplus [\alpha \mapsto \tau])}$$

Substitution de variable :

$$\begin{array}{lcll} apply(\theta, \alpha) & \rightsquigarrow & \theta(\alpha) & \text{if } \alpha \in dom(\theta) \\ apply(\theta, \alpha) & \rightsquigarrow & \alpha & \text{if } \alpha \notin dom(\theta) \\ apply(\theta, \mathtt{int}) & \rightsquigarrow & \mathtt{int} & \\ apply(\theta, \tau_1 \rightarrow \tau_2) & \rightsquigarrow & apply(\theta, \tau_1) \rightarrow apply(\theta, \tau_2) & \end{array}$$

$$\textsc{Inst-Var-Unbound}\ \frac{'a \notin \theta}{\Gamma, \Phi, \theta \vdash_{\tau}\ \tau \leq\ 'a \Rightarrow \theta \oplus ['a \rightarrow \tau]}$$

$$\textsc{Inst-Var-Bound}\ \frac{'a \in \theta \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau_{'a} \equiv \tau}{\Gamma, \Phi, \theta \oplus ['a \rightarrow \tau_{'a}] \vdash_{\tau}\ \tau \leq\ 'a \Rightarrow \theta \oplus ['a \rightarrow \tau_{'a}]}$$

$$\textsc{Inst-Fun}\ \frac{l_1 = l'_1 \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta_1 \qquad \Gamma, \Phi, \theta_1 \vdash_{\tau} \tau_2 \leq \tau'_2 \Rightarrow \theta_2}{\Gamma, \Phi, \theta \vdash_{\tau} (l_1 : \tau_1) \rightarrow \tau_2 \leq (l'_1 : \tau'_1) \rightarrow \tau'_2 \Rightarrow \theta_2}$$

$$\textsc{Inst-Tuple}\ \frac{\Gamma, \Phi, \theta \vdash_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta_1 \qquad \forall i_{>1}.\ \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau'_i \Rightarrow \theta_i}{\Gamma, \Phi, \theta \vdash_{\tau} \tau_1 * .. * \tau_n \leq \tau'_1 * .. * \tau'_n \Rightarrow \theta_n}$$

$$\textsc{Inst-Construct}\ \frac{\Gamma, \Phi, \theta \vdash_{\tau} \tau_1 \leq \tau'_1 \Rightarrow \theta_1 \qquad \forall i_{>1}.\ \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau'_i \Rightarrow \theta_i}{\Gamma, \Phi, \theta \vdash_{\tau} (\overline{\tau})\ \mathtt{p} \leq (\overline{\tau'})\ \mathtt{p} \Rightarrow \theta_n}$$

$$\textsc{Inst-Construct-Exp-Left}\ \frac{let\ \tau = expand(\Gamma, \Phi, \mathtt{t}, \overline{\tau}) \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta'}{\Gamma, \Phi, \theta \vdash_{\tau} (\overline{\tau})\ \mathtt{t} \leq \tau' \Rightarrow \theta'}$$

$$\textsc{Inst-Construct-Exp-Right}\ \frac{let\ \tau' = expand(\Gamma, \Phi, \mathtt{t}', \overline{\tau'}) \qquad \Gamma, \Phi, \theta \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta'}{\Gamma, \Phi, \theta \vdash_{\tau} \tau \leq (\overline{\tau'})\ \mathtt{t}' \Rightarrow \theta'}$$

$$\textsc{Inst-Poly}\ \frac{let\ \theta' = \forall i.\theta \oplus [\alpha'_i \rightarrow \alpha_i] \qquad \Gamma, \Phi, \theta' \vdash_{\tau} \tau \leq \tau' \Rightarrow \theta''}{\Gamma, \Phi, \theta \vdash_{\tau} \forall \overline{\alpha}.\ \tau \leq \forall \overline{\alpha'}.\ \tau' \Rightarrow \theta''}$$

$$\textsc{Inst-Univar}\ \frac{[\alpha' \rightarrow \alpha] \in \theta}{\Gamma, \Phi, \theta \vdash_{\tau} \alpha \leq \alpha' \Rightarrow \theta} \qquad \textsc{Inst-Rigid1}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi, \theta \vdash_{\tau} \tau \leq \mathtt{t} \Rightarrow \theta}$$

$$\textsc{Inst-Rigid2}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi, \theta \vdash_{\tau} \mathtt{t} \leq \tau \Rightarrow \theta}$$

$$\textsc{Inst-Variant}\ \frac{\begin{array}{c} T_{pres_i} .. T_{pres_j} \subseteq T'_{pres_i} .. T'_{pres_j} \\ \Gamma, \Phi, \theta \vdash_{\tau} \rho \leq \rho' \Rightarrow \theta_\rho \qquad \forall i.\ T_i \in \overline{T'} \implies \Gamma, \Phi, \theta_{i-1} \vdash_{\tau} \tau_{var_i} \leq \tau'_{var_i} \Rightarrow \theta_i \end{array}}{\Gamma, \Phi, \theta \vdash_{\tau} [(\rho)\ \overline{T\ of\ \tau_{var}} > T_{pres_i} .. T_{pres_j}] \leq [(\rho')\ \overline{T'\ of\ \tau'_{var}} > T'_{pres_i} .. T'_{pres_j}] \Rightarrow \theta_n}$$

$$\textsc{Inst-Nil}\ \Gamma, \Phi, \theta \vdash_{\tau} \epsilon \leq \epsilon \Rightarrow \theta$$

$$\textsc{Inst-Package}\ \frac{\begin{array}{c} let\ S\ = Sig(\Gamma, \mathtt{P}\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t} = \tau}) \\ let\ S' = Sig(\Gamma, \mathtt{P}'\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t}' = \tau'}) \qquad \Gamma, \Phi, \theta \vdash_{\tau} S' :> S \Rightarrow \theta' \end{array}}{\Gamma, \Phi, \theta \vdash_{\tau} (\mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t} = \tau}) \leq (\mathbf{module}\ \mathtt{P}'\ \mathbf{with}\ \overline{\mathbf{type}\ \mathtt{t}' = \tau'}) \Rightarrow \theta'}$$

Bonne formation

$$\textsc{Wf-Var}\ \Gamma, \Phi \vdash_{\tau} {'a}\ wf \qquad \textsc{Wf-Fun}\ \frac{\Gamma, \Phi \vdash_{\tau} \tau_1\ wf \qquad \Gamma, \Phi \vdash_{\tau} \tau_2\ wf}{\Gamma, \Phi \vdash_{\tau} (l : \tau_1) \to \tau_2\ wf}$$

$$\textsc{Wf-Tuple}\ \frac{\forall \tau_i.\ \Gamma, \Phi \vdash_{\tau} \tau_i\ wf}{\Gamma, \Phi \vdash_{\tau} \tau_1 * .. * \tau_n\ wf}$$

$$\textsc{Wf-Construct}\ \frac{\begin{array}{c} let\ (\overline{\tau_{param}})\ \mathtt{t} = \Gamma.Types(\mathtt{t}) \\ \forall i. \Gamma, \Phi \vdash_{\tau} \tau_i\ wf \qquad \Gamma, \Phi, \Sigma \vdash_{\tau} (\overline{\tau})\ \mathtt{t} \leq (\overline{\tau_{param}})\ \mathtt{t} \Rightarrow \theta \end{array}}{\Gamma, \Phi \vdash_{\tau} (\overline{\tau})\ \mathtt{t}\ \ wf}$$

$$\textsc{Wf-Poly}\ \frac{\forall i.\ \Gamma, \Phi \vdash_{\tau} \tau_{\alpha_i} \prec \alpha_i \qquad \Gamma \oplus \overline{\alpha}, \Phi \vdash_{\tau} \tau\ wf}{\Gamma, \Phi \vdash_{\tau} \forall \overline{\tau_{\alpha}}.\ \tau\ \ wf} \qquad \textsc{Wf-Univar}\ \frac{\alpha \in \Gamma}{\Gamma, \Phi \vdash_{\tau} \alpha\ wf}$$

$$\textsc{Wf-Variant}\ \frac{\begin{array}{c} \forall i, j.\ i \neq j \implies T_i \neq T_j \qquad \forall i. \Gamma, \Phi \vdash_{\tau} \tau_i\ wf \\ \forall T_{pres_i} \in \{T_{pres_i} .. T_{pres_j}\}.\ T_{pres_i} \in \overline{T} \qquad \Gamma, \Phi \vdash_{\tau} \rho \prec {'a} \bigvee \Gamma, \Phi \vdash_{\tau} \rho \prec \epsilon \end{array}}{\Gamma, \Phi \vdash_{\tau} [(\rho)\ \overline{\sim T\ \mathbf{of}\ \tau} > T_{pres_i} .. T_{pres_j}]\ wf}$$

$$\textsc{Wf-Package}\ \frac{\begin{array}{c} \mathtt{P} \in \Gamma.Modtypes \qquad \forall i.\ \mathtt{t_i} \in \Gamma.Modtypes(\mathtt{P}) \\ \forall i. \Gamma, \Phi \vdash_{\tau} \tau_i\ wf \qquad \forall i\ s.t.\ transparent(\mathtt{P.t_i}).\ \Gamma, \Phi \vdash_{\tau} \tau_i \leq \mathtt{P.t_i} \end{array}}{\Gamma, \Phi \vdash_{\tau} (\mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{type\ \mathtt{t} = \tau})\ wf}$$

Filtrage

Le jugement de bonne formation est de la forme suivante :

$$\Gamma, \Phi \vdash_{\tau} \tau \prec \tau_{\rho}$$

Il permet de vérifier que le type τ est de la forme de τ_{ρ}, et introduit ses métavariables dans le contexte courant.

Equations de types

Pour la vérification des types algébriques gardés, il est nécessaire d'enregistrer des formes d'équations locales de types.

```
let is_inconsistent (Γ, Φ) τ 𝒜 =
  ∃τ_𝒜 ∈ 𝒜. not equiv (Γ, Φ) τ_𝒜 τ ∧ not rigid τ_𝒜 ∨ occurs τ τ_𝒜

let add_equation (Γ, Φ, generalized) τ1 τ2 =
  if generalized ∧ τ1 is rigid then
      let 𝒜1 = Φ(τ1) in
      let 𝒜2 = Φ(τ2) in
      if 𝒜1 = 𝒜2 then (Γ, Φ)
      else if is_inconsistent (Γ, Φ, τ1, 𝒜2)
        ∨ is_inconsistent (Γ, Φ, τ2, 𝒜1) then Error
      else
        let 𝒜 = 𝒜1 ∪ 𝒜2 in
        let Φ′ = ∀τ ∈ 𝒜. Φ(τ) ← 𝒜 in
        (Γ, Φ′, generalized)
  else
    (Γ, Φ, generalized)

let add_equations (Γ, Φ) generalized τ1 τ2 =
  fold2 add_equation (Γ, Φ, generalized) τ1 τ2
```

Extraction des types existentiels

```
let is_universal Univ τ =
  if τ ≺ 'a then
    Univ ⊕ τ
  else
    Univ

let find_universals τ_ret =
  fold is_universal ∅ τ_ret

let find_existential (Exs, Univ) τ_arg =
  if τ_arg ≺ 'a ∧ τ_arg ∉ Univ then
    (Exs ⊕ τ_arg, Univ)
  else (Exs, Univ)

let find_existentials (Exs, Univ) τ_arg =
  fold find_existential (Exs, Univ) τ_arg


let add_existential (𝒯, Exs) τ1 τ2 =
  if τ2 ≺ 'a ∧ τ2 ∈ Exs then
    let τ∃ = τ1 ≺ t in
    (𝒯 ⊕ τ∃, Exs)
  else
    (𝒯, Exs)

let add_existential (𝒯, Exs) τ1 τ2 =
  fold2 add_existential (𝒯, Exs) τ1 τ2


let existential_types τ_args τ_ret τ_def generalized =
  if generalized then
    let Univ = find_universals τ_ret in
    let Exs, _ = find_existentials (∅, Univ) τ_def in
    let 𝒯, _ = add_existentials (∅, Exs) τ_args τ_def in
    𝒯
  else
    ∅
```

Restriction souple de valeur

Condition d'expansivité des expressions

$nonexp\,(\texttt{x})\ ::=\ \top$
$nonexp\,(c)\ ::=\ \top$
$nonexp\,(\texttt{let x = e}_1\ \texttt{in e}_2)\ ::=$
$\quad nonexp\,(\texttt{e}_1) \wedge nonexp\,(\texttt{e}_2)$
$nonexp\,(\texttt{fun x -> e})\ ::=\ \top$
$nonexp\,(\texttt{match e with}\ \overline{\texttt{| p when g -> b}})\ ::=$
$\quad nonexp\,(\texttt{e}) \wedge \bigwedge_{i=1}^{n}\ (nonexp\,(\texttt{g}_i) \wedge nonexp\,(\texttt{b}_i))$
$nonexp\,(\texttt{e}_1\texttt{, e}_2)\ ::=\ nonexp\,(\texttt{e}_1) \wedge nonexp\,(\texttt{e}_2)$
$nonexp\,(K)\ ::=\ \top$
$nonexp\,(K\ e)\ ::=\ nonexp\,(\texttt{e})$
$nonexp\,(\text{`}K)\ ::=\ \top$
$nonexp\,(\text{`}K\ e)\ ::=\ nonexp\,(\texttt{e})$
$nonexp\,(\{\ \overline{\texttt{l = e;}}\ \})\ ::=\ \bigwedge_{i=1}^{n}\ (immutable\,(\texttt{l}_i) \wedge nonexp\,(\texttt{e}_i))$
$nonexp\,(\texttt{e.l})\ ::=\ nonexp\,(\texttt{e})$
$nonexp\,(\texttt{if c then e}_1\ \texttt{else e}_2)\ ::=$
$\quad nonexp\,(\texttt{c}) \wedge nonexp\,(\texttt{e}_1) \wedge nonexp\,(\texttt{e}_2)$
$nonexp\,(\texttt{e}_1\texttt{; e}_2)\ ::=\ nonexp\,(\texttt{e}_2)$
$nonexp\,(\texttt{lazy e})\ ::=\ nonexp\,(\texttt{e})$
$nonexp\,(\texttt{let module I = M in e})\ ::=$
$\quad nonexp\_mod\,(\texttt{M}) \wedge nonexp\,(\texttt{e})$
$nonexp\,(\texttt{(module M)})\ ::=\ nonexp\_mod\,(\texttt{M})$
$nonexp\,(\_)\ ::=\ \bot$

Calcul et vérification de généralisation

```
let isvar_nonexpans (Γ, Φ) 𝒯 τ =
  if Γ,Φ ⊨_τ 'a < τ ∧ τ ∉ Γ.𝒱ars then
    𝒯 ⊕ τ
  else
    𝒯

let aggregate_nonexpans (Γ, Φ) τ =
  fold (isvar_nonexpans (Γ, Φ)) ∅ τ

let aggregate_expans (Γ, Φ) 𝒯 pos τ =
  if Γ,Φ ⊨_τ 'a < τ then
    if τ ∈ Γ.𝒱ars ∨ (τ ∈ 𝒯 ∧ pos ≠ Contravariant) then
      𝒯
    else
      𝒯 ⊕ pos
  else if Γ,Φ ⊨_τ τ₁ → τ₂ < τ then
    let 𝒯' =
      aggregate_expans (Γ, Φ) 𝒯 Contravariant τ in
```

```
      aggregate_expans (Γ, Φ) 𝒯' pos τ
    else if Γ,Φ ⊢_τ τ̄_arg t ≺ τ then
      list_fold_indexed_left
        (fun 𝒯 index τ_arg →
          let pos =
            if get_variance (Γ, Φ) index t = Contravariant
            then Contravariant
            else pos in
          aggregate_expans (Γ, Φ) 𝒯 pos τ_arg)
        𝒯
        τ̄_arg
    else
      [...]

  let check_gen (Γ, Φ) τ e =
    if nonexp e then
      let 𝒯 = aggregate_nonexpans (Γ, Φ) τ in
      TypeSet.iter (fun τ →
        if not (generalized τ) then Error) 𝒯
    else
      let 𝒯 = aggregate_expans (Γ, Φ) ∅ Covariant τ in
      TypeMap.iter (fun τ pos ->
        if generalized τ ∧ pos = Contravariant then Error
        else if not (generalized τ) ∧ pos = Covariant
        then Error) 𝒯
```

B.2.2 Système de types des expressions

$$\textsc{Const}\ \frac{c : \tau}{\Gamma, \Phi \vdash \langle c : \tau \rangle} \qquad \textsc{Var}\ \frac{\Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \Gamma.Values(x) \Rightarrow \theta}{\Gamma, \Phi \vdash \langle x : \tau \rangle}$$

$$\textsc{Abs}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash_{\tau} \tau \prec \tau_d \rightarrow \tau_{cd} \qquad \forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho}^{\langle} p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ \forall i.\ \Gamma, \Phi \vdash_{\tau} \tau_d \equiv \tau_i \qquad \forall i.\ let\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}}) \\ \forall i.\ \Gamma_i, \Phi_i \vdash \langle e_i : \tau_i' \rangle \qquad \forall i.\ \Gamma, \Phi \vdash_{\tau} \tau_i'\ wf \qquad \forall i.\ \Gamma_I, \Phi_i \vdash_{\tau} \tau_{cd} \equiv \tau_i' \end{array}}{\Gamma, \Phi \vdash \langle \mathbf{function}\ \overline{\mid p \rightarrow e}\ : \tau \rangle}$$

$$\textsc{App}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \\ \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_1 \prec \tau_d \rightarrow \tau_{cd} \qquad \Gamma, \Phi \vdash_{\tau} \tau_2 \equiv \tau_d \qquad \Gamma, \Phi \vdash_{\tau} \tau_{cd} \equiv \tau \end{array}}{\Gamma, \Phi \vdash \langle e_1\ e_2 : \tau \rangle}$$

$$\textsc{Let}\ \frac{\begin{array}{c} \forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho}^{\langle} p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_{\exists} = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_1}}) \\ \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \forall i.\ \Gamma, \Phi, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \qquad \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i) \\ let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_{\exists} \qquad \Gamma', \Phi_n \vdash \langle e' : \tau' \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau'\ wf \qquad \Gamma', \Phi_n \vdash_{\tau} \tau \equiv \tau' \end{array}}{\Gamma, \Phi \vdash \langle \mathbf{let}\ \overline{p = e}\ \mathbf{in}\ e' : \tau \rangle}$$

$$\textsc{Tuple}\ \dfrac{\forall i.\ \Gamma,\Phi \vdash \langle e_i : \tau'_i\rangle \qquad \Gamma,\Phi \vdash_{\tau} \tau < \tau_1\ *\ ..\ *\ \tau_n \qquad \forall i. \Gamma,\Phi \vdash_{\tau} \tau_i \equiv \tau'_i}{\Gamma,\Phi \vdash \langle (e_1,\ ..,\ e_n) : \tau\rangle}$$

$$\textsc{Construct}\ \dfrac{\begin{array}{c}\forall i.\ \Gamma,\Phi \vdash \langle e_i : \tau_i\rangle \qquad let\ (\tau_{arg_1},\ ..,\tau_{arg_n},\tau_{constr}) = find\_constructor(\Gamma,\Phi,T,\tau)\\ \Gamma,\Phi,\emptyset \vdash_{\tau} \tau_1 \leq \tau_{arg_1} \Rightarrow \theta_1\\ \forall i_{>1}.\ \Gamma,\Phi,\theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \qquad \Gamma,\Phi,\theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{constr}\end{array}}{\Gamma,\Phi \vdash \langle T(e_1,\ ..,e_n) : \tau\rangle}$$

$$\textsc{Match}\ \dfrac{\begin{array}{c}\Gamma,\Phi \vdash \langle e : \tau_{scrut}\rangle \qquad \forall i.\ \Gamma,\Phi,\emptyset \vdash_{\rho} \langle p_i : \tau_i\rangle \Rightarrow (\overline{v_i : \tau_{v_i}}),(\overline{\tau_{\exists_i}}),\Phi_i\\ \forall i.\ \Gamma,\Phi \vdash_{\tau} \tau_{scrut} \equiv \tau_i \qquad \forall i.\ let\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}})\\ \forall i.\ \Gamma_i,\Phi_i \vdash \langle e_i : \tau_{e_i}\rangle \qquad \forall i.\ \Gamma,\Phi \vdash_{\tau} \tau_{e_i}\ wf \qquad \forall i.\ \Gamma_i,\Phi_i \vdash_{\tau} \tau \equiv \tau_{e_i}\end{array}}{\Gamma,\Phi \vdash \langle \mathbf{match}\ e\ \mathbf{with}\ \overline{\mid p \rightarrow e} : \tau\rangle}$$

$$\textsc{Record}\ \dfrac{\begin{array}{c}let\ \tau_{rec},\mathcal{L} = find\_record(\Gamma,\Phi,\tau) \qquad check\_unicity(\mathcal{L},\{l_1,..,l_n\}) \qquad \forall i.\ let\ \tau_{l_i} = \mathcal{L}(i)\\ \forall i.\ \Gamma,\Phi \vdash \langle e : \tau_i\rangle \qquad \Gamma,\Phi,\emptyset \vdash_{\tau} \tau_1 \leq \tau_{l_1} \Rightarrow \theta_1 \qquad \forall i_{>1}.\ \Gamma,\Phi,\theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{l_i} \Rightarrow \theta_i\\ \forall i_{\geq 1}.\ check\_gen\_record(\Gamma,\Phi,\tau_{l_i},\tau_i,e_i) \qquad \Gamma,\Phi,\theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{rec}\end{array}}{\Gamma,\Phi \vdash \langle \{l_1 = e_1;\ ..\ ; l_n = e_n\} : \tau\rangle}$$

$$\textsc{Field}\ \dfrac{\begin{array}{c}\Gamma,\Phi \vdash \langle e : \tau_e\rangle \qquad let\ \tau_{rec},\mathcal{L} = find\_record(\Gamma,\Phi,\tau_e)\\ l \in dom(\mathcal{L}) \qquad let\ \tau_l = \mathcal{L}(l) \qquad \Gamma,\Phi,\Sigma \vdash_{\tau} \tau \leq \tau_l \Rightarrow \theta \qquad \Gamma,\Phi,\theta_{\forall}(\Gamma) \vdash_{\tau} \tau_e \leq \tau_{rec}\end{array}}{\Gamma,\Phi \vdash \langle e.l : \tau\rangle}$$

$$\textsc{Record-Copy}\ \dfrac{\begin{array}{c}let\ \tau_{rec},\mathcal{L} = find\_record(\Gamma,\Phi,\tau) \qquad \Gamma,\Phi \vdash \langle o : \tau_o\rangle\\ \Gamma,\Phi,\emptyset \vdash_{\tau} \tau_o \leq \tau_{rec} \Rightarrow \theta_o \qquad let\ \mathcal{L}_o = extract\_labels(\Gamma,\Phi,\tau_o)\\ let\ Copied = \mathcal{L}_o - \{l_1;\ ..\ ; l_k\} \qquad check\_unicity(\mathcal{L}, Copied + \{l_1;\ ..\ ; l_k\})\\ \forall i_{i\in[0..k]}.\ let\ \tau_{l_i} = \mathcal{L}(l_i) \qquad \forall i.\ \Gamma,\Phi \vdash \langle e_i : \tau_i\rangle \qquad \Gamma,\Phi,\emptyset \vdash_{\tau} \tau_1 \leq \tau_{l_1} \Rightarrow \theta_1\\ \forall i_{>1}.\ \Gamma,\Phi,\theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \qquad \forall i_{\geq 1}.\ check\_gen\_record(\Gamma,\Phi,\tau_{l_i},\tau_i,e_i)\\ \forall i_{l_i \in Copied}.\ \Gamma,\Phi,\theta_{i-1} \vdash_{\tau} Copied(l_i) \leq \text{Ł}(l_i) \Rightarrow \theta_i \qquad \Gamma,\Phi,\theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{rec}\end{array}}{\Gamma,\Phi \vdash \langle \{o\ \mathbf{with}\ l_1 = e_1;\ ..\ ; l_k = e_k\} : \tau\rangle}$$

$$\textsc{Set-Field}\ \dfrac{\begin{array}{c}\Gamma,\Phi \vdash \langle e_1 : \tau_1\rangle \qquad \Gamma,\Phi \vdash \langle e_2 : \tau_2\rangle \qquad let\ \tau_{rec},\mathcal{L} = find\_record(\Gamma,\Phi,\tau_e)\\ l \in dom(\mathcal{L}) \qquad let\ \tau_l = \mathcal{L}(l) \qquad label\_kind(\Gamma,\Phi,l,\tau_{rec})\ =\ Mutable\\ \Gamma,\Phi,\emptyset \vdash_{\tau} \tau_2 \leq \tau_l \Rightarrow \theta \qquad \Gamma,\Phi,\theta_{\forall}(\Gamma) \vdash_{\tau} \tau_1 \leq \tau_{rec} \qquad \Gamma,\Phi \vdash_{\tau} \tau \equiv \mathtt{unit}\end{array}}{\Gamma,\Phi \vdash \langle e_1.l \leftarrow e_2 : \tau\rangle}$$

Condition d'unicité des champs d'enregistrement

```
let rec check_unicity (𝒟, 𝒫) =
match 𝒟, 𝒫 with
  | ∅, ∅ -> ⊤
  | ∅, _ | _, ∅ -> ⊥
  | _, _ ->
    let l, 𝒫' = take 𝒫 in
    if l ∉ dom(𝒟) then ⊥
    else
      let 𝒟' = remove (l, 𝒟) in
      check_unicity (𝒟', 𝒫')

let rec check_atmost (𝒟, 𝒫) =
  match 𝒟, 𝒫 with
  | ∅, ∅ | _, ∅ -> ⊤
  | ∅, _ -> ⊥
  | _, _ ->
    let l, 𝒫' = take 𝒫 in
    if l ∉ dom(𝒟) then ⊥
    else
      let 𝒟' = remove (l, 𝒟') in
      check_atmost (𝒟', 𝒫')
```

$$\textsc{Sequence}\ \frac{\Gamma,\Phi \vdash \langle e_1 : \tau_1\rangle \qquad \Gamma,\Phi \vdash \langle e_2 : \tau_2\rangle \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_1 \equiv \mathtt{unit} \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau \equiv \tau_2}{\Gamma,\Phi \vdash \langle e_1;\ e_2 : \tau\rangle}$$

$$\textsc{While}\ \frac{\begin{array}{c}\Gamma,\Phi \vdash \langle e_1 : \tau_1\rangle \\ \Gamma,\Phi \vdash \langle e_2 : \tau_2\rangle \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_1 \equiv \mathtt{bool} \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_2 \equiv \mathtt{unit} \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau \equiv \mathtt{unit}\end{array}}{\Gamma,\Phi \vdash \langle \mathbf{while}\ e_1\ \mathbf{do}\ e_2\ \mathbf{done} : \tau\rangle}$$

$$\textsc{For}\ \frac{\begin{array}{c}\Gamma,\Phi \vdash \langle e_1 : \tau_1\rangle \qquad \Gamma,\Phi \vdash \langle e_2 : \tau_2\rangle \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_1 \equiv \mathtt{int} \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_2 \equiv \mathtt{int} \\ \Gamma \oplus_{\mathcal{V}} (x, \tau_1), C.\Phi \vdash \langle e_3 : \tau_3\rangle \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_3 \equiv \mathtt{unit} \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau \equiv \mathtt{unit}\end{array}}{\Gamma,\Phi \vdash \langle \mathbf{for}\ x = e_1\ \mathbf{to}\ e_2\ \mathbf{do}\ e_3\ \mathbf{done} : \tau\rangle}$$

$$\textsc{Raise}\ \frac{\Gamma,\Phi \vdash \langle e : \tau_e\rangle \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau_e \equiv \mathtt{exn}}{\Gamma,\Phi \vdash \langle \mathbf{raise}\ e : \tau\rangle}$$

$$\textsc{TryWith}\ \frac{\begin{array}{c}\Gamma,\Phi \vdash \langle e : \tau_e\rangle \qquad \Gamma,\Phi \models_{\mathcal{T}} \tau \equiv \tau_e \qquad \forall i.\ \Gamma,\Phi,\emptyset \models^{\langle}_{\rho} p_i : \tau_i\rangle \Rightarrow \overline{(v_i : \tau_{v_i})}, \overline{(\tau_{\exists_i})}, \Phi_i \\ \forall i.\ \Gamma,\Phi \models_{\mathcal{T}} \mathtt{exn} \equiv \tau_i \qquad \forall i.\ let\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} \overline{(v_i : \tau_{v_i})} \oplus_{\mathcal{T}} \overline{(\tau_{\exists_i})} \\ \forall i.\ \Gamma_i,\Phi_i \vdash \langle e_i : \tau_{e_i}\rangle \qquad \forall i.\ \Gamma,\Phi \models_{\mathcal{T}} \tau_{e_i}\ wf \qquad \forall i.\ \Gamma_i,\Phi_i \models_{\mathcal{T}} \tau \equiv \tau_{e_i}\end{array}}{\Gamma,\Phi \vdash \langle \mathbf{try}\ e\ \mathbf{with}\ \overline{\mid p \to e} : \tau\rangle}$$

$$\textsc{Assert}\ \frac{\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_e \equiv \texttt{bool} \qquad \Gamma, \Phi \vdash_{\tau} \tau \equiv \texttt{unit}}{\Gamma, \Phi \vdash \langle \textbf{assert}\ e : \tau \rangle}$$

$$\textsc{Assert-False}\ \frac{\Gamma, \Phi \vdash \langle false : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_e \equiv \texttt{bool}}{\Gamma, \Phi \vdash \langle \textbf{assert}\ false : \tau \rangle}$$

$$\textsc{Lazy}\ \frac{\Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau < \tau_{arg}\ \texttt{lazy\_t} \qquad \Gamma, \Phi \vdash_{\tau} \tau_e \equiv \tau_{arg}}{\Gamma, \Phi \vdash \langle \textbf{lazy}\ e : \tau \rangle}$$

$$\textsc{Variant-Const}\ \frac{\Gamma, \Phi \vdash_{\tau} \tau < [(\rho)\ \mathcal{T} > \mathcal{K}] \qquad T \in \mathcal{T} \qquad \mathcal{K}(T) = Present}{\Gamma, \Phi \vdash \langle `T : \tau \rangle}$$

$$\textsc{Variant}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash_{\tau} \tau < [(\rho)\ \mathcal{T} > \mathcal{K}] \\ T\ of\ \tau_{arg} \in \mathcal{T} \qquad \mathcal{K}(T) = Present \qquad \Gamma, \Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau_{arg} \equiv \tau_e\end{array}}{\Gamma, \Phi \vdash \langle `T\ e : \tau \rangle}$$

Système de types : motifs

$$\textsc{Pat-Const}\ \frac{c : \tau}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} c : \tau\rangle \Rightarrow \mathcal{V}, \mathcal{T}, \Phi} \qquad \textsc{Pat-Wildcard}\ \Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} \_ : \tau\rangle \Rightarrow \mathcal{V}, \mathcal{T}, \Phi$$

$$\textsc{Pat-Var}\ \frac{v \notin dom(\mathcal{V}) \qquad let\ \mathcal{V}' = \mathcal{V} \oplus (v, \tau)}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} v : \tau\rangle \Rightarrow \mathcal{V}', \mathcal{T}, \Phi}$$

$$\textsc{Pat-Or}\ \frac{\begin{array}{c}\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} p_1 : \tau_1\rangle \Rightarrow \mathcal{V}_1, \mathcal{T}_1, \Phi_1 \qquad \Gamma, \Phi_1, \mathcal{V}, \mathcal{T}_1 \vdash^{\langle}_{\rho} p_2 : \tau_2\rangle \Rightarrow \mathcal{V}_2, \mathcal{T}_2, \Phi_2 \\ \Gamma, \Phi_2 \vdash_{\tau} \tau_1 \equiv \tau_2 \qquad \Gamma, \Phi_2 \vdash_{\tau} \tau_2 \equiv \tau \qquad \Gamma, \Phi_2 \vdash_{\tau} \mathcal{V}_1 \equiv \mathcal{V}_2\end{array}}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} p_1 \mid p_2 : \tau\rangle \Rightarrow \mathcal{V}_2, \mathcal{T}_2, \Phi_2}$$

$$\textsc{Pat-Construct}\ \frac{\begin{array}{c}\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} p_1 : \tau_1\rangle \Rightarrow \mathcal{V}_1, \mathcal{T}_1, \Phi_1 \\ \forall i_{>1}.\ \Gamma, \Phi_{i-1}, \mathcal{V}_{i-1}, \mathcal{T}_{i-1} \vdash^{\langle}_{\rho} p_i : \tau_i\rangle \Rightarrow \mathcal{V}_i, \mathcal{T}_i, \Phi_i \\ let\ (\tau_{arg_1},\ ..\ , \tau_{arg_n}, \tau_{constr}, generalized) = find\_constructor(\Gamma, \Phi, T, \tau) \\ let\ \mathcal{T}_{args} = existential\_types((\tau_1 *\ ..\ * \tau_n), \tau_{constr}, (\tau_{arg_1} *\ ..\ * \tau_{arg_n}), generalized) \\ \Gamma, \Phi_n, \Sigma \vdash_{\tau} \tau_1 \leq \tau_{arg_1} \Rightarrow \theta_1 \qquad \forall i_{>1}.\ \Gamma, \Phi_n, \theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \\ \forall i.\ let\ \Phi_{p_i} = add\_equations((\Gamma, \Phi_{p_{i-1}}), \tau_i, \tau_{arg_i}) \\ let\ \Phi_{ret} = add\_equations((\Gamma, \Phi_{p_n}), \tau, \tau_{constr}) \qquad \Gamma, \Phi_{p_n}, \theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{constr}\end{array}}{\Gamma, \Phi, \mathcal{V}, \mathcal{T} \vdash^{\langle}_{\rho} T(p_1,\ ..\ , p_n) : \tau\rangle \Rightarrow \mathcal{V}_n, \mathcal{T}_n, \Phi_{ret}}$$

$$\textsc{Pat-Tuple}\ \frac{\begin{array}{c}\Gamma,\Phi \vdash_{\tau} \tau < \tau_{p_1} * \ ..\ * \tau_{p_n} \qquad \Gamma,\Phi,\mathcal{V},\mathcal{T} \vdash^{\langle}_{\rho} p_1 : \tau_1\rangle \Rightarrow \mathcal{V}_1,\mathcal{T}_1 \\ \forall i_{>1}.\ \Gamma,\Phi_{i-1},\mathcal{V}_{i-1},\mathcal{T}_{i-1} \vdash^{p}_{\rho\, i}: \tau_i \Rightarrow \mathcal{V}_i,\mathcal{T}_i,\Phi_i \qquad \forall i.\ \Gamma,\Phi_i \vdash_{\tau} \tau_{p_i} \equiv \tau_i\end{array}}{\Gamma,\Phi,\mathcal{V},\mathcal{T} \vdash^{\langle}_{\rho} p_1,\ ..\ ,p_n : \tau\rangle \Rightarrow \mathcal{V}_n,\mathcal{T}_n,\Phi_n}$$

$$\textsc{Pat-Record}\ \frac{\begin{array}{c} let\ \tau_{rec},\mathcal{L} = find\_record(\Gamma,\Phi,\tau) \qquad check\_atmost(\mathcal{L},\{l_1,..,l_n\}) \\ l_i \in dom(\mathcal{L}) \qquad \forall i.\ let\ \tau_{l_i} = \mathcal{L}(l_i) \qquad \Gamma,\Phi,\emptyset,\emptyset \vdash^{\langle}_{\rho} p_1 : \tau_1\rangle \rightarrow \mathcal{V}_1,\mathcal{T}_1,\Phi_1 \\ \forall i_{>1}.\ \Gamma,\Phi_{i-1},\mathcal{V}_{i-1},\mathcal{T}_{i-1} \vdash^{\langle}_{\rho} p_i : \tau_i\rangle \Rightarrow \mathcal{V}_i,\mathcal{T}_i,\Phi_i \qquad \Gamma,\Phi,\emptyset \vdash_{\tau} \tau_1 \leq \tau_{l_1} \Rightarrow \theta_1 \\ \forall i_{>1}.\ \Gamma,\Phi,\theta_{i-1} \vdash_{\tau} \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \qquad \Gamma,\Phi,\theta_{\forall}(\Gamma) \vdash_{\tau} \tau \leq \tau_{rec}\end{array}}{\Gamma,\Phi \vdash^{\langle}_{\rho} \{l_1 = p_1;\ ..\ ; l_n = p_n\} : \tau\rangle}$$

$$\textsc{Pat-Variant-Const}\ \frac{\Gamma,\Phi_p \vdash_{\tau} \tau < [(\rho)\,\mathcal{T} > \mathcal{K}\,] \qquad T \in \mathcal{T} \qquad \mathcal{K}(T) = Either}{\Gamma,\Phi,\mathcal{V},\mathcal{T} \vdash^{\langle}_{\rho} `T : \tau\rangle \Rightarrow \mathcal{V},\mathcal{T},\Phi}$$

$$\textsc{Pat-Variant}\ \frac{\begin{array}{c}\Gamma,\Phi \vdash_{\tau} \tau < [(\rho)\,\mathcal{T} > \mathcal{K}] \qquad T\ of\ \tau_{arg} \in \mathcal{T} \\ \mathcal{K}(T) = Either \qquad \Gamma,\Phi,\mathcal{V},\mathcal{T} \vdash^{\langle}_{\rho} p : \tau_p\rangle \Rightarrow \mathcal{V}',\mathcal{T}',\Phi' \qquad \Gamma,\Phi \vdash_{\tau} \tau_{arg} \equiv \tau_p\end{array}}{\Gamma,\Phi,\mathcal{V},\mathcal{T} \vdash^{\langle}_{\rho} `T\ p : \tau\rangle \Rightarrow \mathcal{V}',\mathcal{T}',\Phi'}$$

B.3 Sémantique opérationnelle

Mutabilité

La mutabilité nécessite l'ajout d'un modèle de mémoire, d'une valeur représentant des pointeurs dans cette mémoire et d'une règle de typage associée à celle-ci.

$e' ::= ..$	*expressions*
$\mid m$	*emplacement mémoire*

Mémoire

$$\begin{array}{l}\mathcal{M} ::= \emptyset \\ \qquad \mid \mathcal{M},\ (m : \tau = [v_1 ; .. ; v_n])\end{array}$$

— dom ($\mathcal{M}$) : ensemble des emplacements mémoire alloués
— $\mathcal{M}_\tau(m)$: type associé à l'emplacement mémoire
— $\mathcal{M}\,(m, \mathrm{l})$: valeur enregistrée dans le champ l du bloc mémoire alloué à m.
— $\mathcal{M}\,(m, \mathrm{l}) \leftarrow v$: assignation de la valeur v au champ l dans le bloc associé à m.

Récursion

La récursion est basée sur un combinateur de point fixe : il faut donc ajouter celui-ci aux expressions du langage et donc une règle de vérification associée. Il faut également définir une règle d'expansion des **let** récursifs vers leur équivalent utiliant le combinateur de point fixe.

e' : := ..	*expressions*
| **fix** e	*combinateur de point fixe*

Vérification

$$\textsc{Fix}\ \frac{\Gamma,\Phi \vdash \langle e : \tau_e\rangle \qquad \Gamma,\Phi \vdash_{\tau} \tau_e \prec \tau_d \rightarrow \tau_{cd} \qquad \Gamma,\Phi \vdash_{\tau} \tau_d \equiv \tau \qquad \Gamma,\Phi \vdash_{\tau} \tau_{cd} \equiv \tau}{\Gamma,\Phi \vdash \langle \mathbf{fix}\ e : \tau\rangle}$$

Expansion de **let rec**

```
let rec ⟨ f₁ : τ₁ ⟩ = ⟨ e₁ : τ₁ ⟩
..
and ⟨ fₙ : τₙ ⟩ = ⟨ eₙ : τₙ ⟩
```

$\rightsquigarrow$

(Sachant $\tau_{fs} = \tau_1 * .. * \tau_n$)

```
let ⟨ (f₁, .., fₙ) : τfs ⟩ =
  fix (function ⟨ f₁', .., fₙ' : τfs ⟩ →
    ( e₁ [ ⟨ f₁ : τf₁ ⟩ ↦ ⟨ f₁' : τf₁ ⟩;
           ..
           ⟨ fₙ : τfn ⟩ ↦ ⟨ fₙ' : τfₙ ⟩],
      ..,

      eₙ [ ⟨ f₁ : τf₁ ⟩ ↦ ⟨ f₁' : τf₁ ⟩; ..
           ..
           ⟨ fₙ : τfn ⟩ ↦ ⟨ fₙ' : τfₙ ⟩]))
```

Évaluation paresseuse

L'évaluation paresseuse est implémentée à l'aide d'enregistrements.

$$\langle\ \texttt{Lazy.force}\ \langle\ m : \tau\ \rangle : \tau_{arg}\ \rangle ::=$$

$$\begin{array}{l}
\textbf{match}\ \langle\ \mathcal{M}\ (m,\ thunk) : \mathcal{M}_\tau\ (m)\ \rangle\ \textbf{with} \\
\mid\ \langle\ \texttt{Thunk}\ \langle\ \texttt{f} : \texttt{unit} \rightarrow \tau_{arg}\ \rangle : \mathcal{M}_\tau\ (m)\ \rangle \rightarrow \\
\quad \langle\ \textbf{let}\ \langle\ \texttt{x} : \tau_{arg}\ \rangle = \langle\ \texttt{f} : \texttt{unit} \rightarrow \tau_{arg}\ \rangle\ \langle\ \texttt{()} : \texttt{unit}\ \rangle\ \textbf{in} \\
\quad \langle\ \mathcal{M}\ (m,\ thunk) : \tau\ \rangle \leftarrow \langle\ \texttt{Res x} : \mathcal{M}_\tau\ (m)\ \rangle; \\
\quad \langle\ \texttt{x} : \tau_{arg}\ \rangle \\
\\
\mid\ \langle\ \texttt{Res}\ \langle\ \texttt{x} : \tau_{arg}\ \rangle : \mathcal{M}_\tau\ (m)\ \rangle \rightarrow \\
\quad \langle\ \texttt{x} : \tau_{arg}\ \rangle
\end{array}$$

B.3.1 Valeurs

$v ::=$		*valeurs*
	$\mid \langle\ c : \tau\ \rangle$	*constante*
	$\mid \langle\ \textbf{function}\ \overline{\mid p \rightarrow e} : \tau\ \rangle$	*abstraction*
	$\mid \langle\ m : \tau\ \rangle$	*emplacement mémoire*
	$\mid \langle\ (v_1, .., v_n) : \tau\ \rangle$	*n-uplets*
	$\mid \langle\ K\ (v_1, .., v_n) : \tau\ \rangle$	*constructeurs*
	$\mid \langle\ m : \tau\ \rangle$	*emplacement mémoire*
	$\mid \langle\ \textbf{raise}\ v : \tau\ \rangle$	*levée d'exception*

B.3.2 Réduction

Réduction classique

$$\textsc{App-Left}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1\ e_2 \rightsquigarrow v_1\ e_2, \mathcal{M}_1} \qquad \textsc{App-Right}\ \frac{value(v_1) \qquad \mathcal{M} \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash v_1\ e_2 \rightsquigarrow v_1\ v_2, \mathcal{M}_2}$$

$$\textsc{App-Function-Match}\ \frac{value(v_2) \qquad match(p, v, \mathcal{M}) = \overline{(x, v)}, \mathcal{M}'}{\mathcal{M} \vdash \begin{array}{ll} \langle\ \mathtt{function} & \mid p \to e \\ & \mid \overline{p' \to e'} : \tau\rangle \end{array} v_2 \rightsquigarrow e[x_1 \mapsto v_1, .., x_n \mapsto v_n], \mathcal{M}'}$$

$$\textsc{App-Function-Bot}\ \frac{value(v_2) \qquad match(p, v, \mathcal{M}) = \bot, \mathcal{M}'}{\mathcal{M} \vdash \begin{array}{ll} \langle\ \mathtt{function} & \mid p \to e \\ & \mid \overline{p' \to e'} : \tau\rangle \end{array} v_2 \rightsquigarrow (\langle\ \mathtt{function} \mid \overline{p' \to e'} : \tau\rangle\ v_2), \mathcal{M}'}$$

$$\textsc{Let-Bind}\ \frac{let \mathcal{M}_0 = \mathcal{M} \qquad \forall i. \mathcal{M}_{i-1} e_i \rightsquigarrow v_i, \mathcal{M}_i}{\mathcal{M} \vdash \mathtt{let}\ \overline{\langle p : \sigma\rangle = e}\ \mathtt{in}\ e' \rightsquigarrow \mathtt{let}\ \overline{\langle p : \sigma\rangle = v}\ \mathtt{in}\ e', \mathcal{M}_n}$$

$$\textsc{Let-Value}\ \frac{\forall i. value(v_i) \qquad let \mathcal{M}_0 = \mathcal{M} \qquad \forall i. \overline{(p_i, v_i)}, \mathcal{M}_i = match(p_i, v_i, \mathcal{M}_{i-1})}{\mathcal{M} \vdash \mathtt{let}\ \overline{\langle p : \forall\overline{\alpha}.\tau\rangle = v}\ \mathtt{in}\ e' \rightsquigarrow e'[x_{11} \mapsto v_{11}, .., x_{nk} \mapsto v_{nk}]_{\triangleleft}, \mathcal{M}_n}$$

$$\textsc{Tuple-First}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash (e_1,\ e_2) \rightsquigarrow (v_1,\ e_2), \mathcal{M}_1} \qquad \textsc{Tuple-Second}\ \frac{value(v_1) \qquad \mathcal{M} \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash (v_1,\ e_2) \rightsquigarrow (v_1,\ v_2), \mathcal{M}_2}$$

$$\textsc{Constr-First}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash K(e_1,\ e_2) \rightsquigarrow K(v_1,\ e_2), \mathcal{M}_1}$$

$$\textsc{Constr-Second}\ \frac{value(v_1) \qquad \mathcal{M} \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash K(v_1,\ e_2) \rightsquigarrow K(v_1,\ v_2), \mathcal{M}_2}$$

$$\textsc{Match-Scrut}\ \frac{\mathcal{M} \vdash e \rightsquigarrow v, \mathcal{M}'}{\mathcal{M} \vdash \texttt{match}\ e\ \texttt{with}\overline{|\ p \rightarrow e} \rightsquigarrow \texttt{match}\ v\ \texttt{with}\overline{|\ p \rightarrow e}, \mathcal{M}'}$$

$$\textsc{Match-Ok}\ \frac{value(v) \qquad match(p, v, \mathcal{M}) = \overline{(x, v)}, \mathcal{M}'}{\texttt{match}\ v\ \texttt{with} \begin{array}{l} |\ p \rightarrow e \\ |\ \overline{p' \rightarrow e'} \end{array} \rightsquigarrow e[x_1 \mapsto v_1, .., x_n \mapsto v_n]}$$

$$\textsc{Match-Bot}\ \frac{value(v_2) \qquad match(p, v, \mathcal{M}) = \bot, \mathcal{M}'}{\texttt{match}\ v\ \texttt{with} \begin{array}{l} |\ p \rightarrow e \\ |\ \overline{p' \rightarrow e'} \end{array} \rightsquigarrow \texttt{match}\ v\ \texttt{with}\overline{|\ p' \rightarrow e', \mathcal{M}'}}$$

$$\textsc{Record-First}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash \{l_1 = e_1;\ l_2 = e_2\} \rightsquigarrow \{l_1 = v_1.;\ l_2 = e_2\}, \mathcal{M}_1}$$

$$\textsc{Record-Second}\ \frac{value(v_1) \qquad M \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash \{l_1 = v_1;\ l_2 = e_2\} \rightsquigarrow \{l_1 = v_1.;\ l_2 = v_2\}, \mathcal{M}_2}$$

$$\textsc{Record-Location}\ \frac{value(v_1) \qquad value(v_2) \qquad let\ m \notin dom(\mathcal{M})}{\mathcal{M} \vdash \langle\{l_1 = v_1;\ l_2 = e_2\} : \tau\rangle \rightsquigarrow \langle m : \tau\rangle, (\mathcal{M}, (m : \tau = [v_1; v_2]))}$$

$$\textsc{Record-Access}\ \frac{\mathcal{M} \vdash e \rightsquigarrow v, \mathcal{M}'}{\mathcal{M} \vdash e.l \rightsquigarrow v.l, \mathcal{M}'} \qquad \textsc{Location-Access}\ \frac{}{\mathcal{M} \vdash \langle m : \tau\rangle.l \rightsquigarrow \mathcal{M}(m, l)}$$

$$\textsc{Record-Set}\ \frac{\mathcal{M} \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash e_1.l \leftarrow e_2 \rightsquigarrow e_1.l \leftarrow v_2, \mathcal{M}_2}$$

$$\textsc{Record-Set-Value}\ \frac{value(v_2) \qquad \mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1.l \leftarrow v_2 \rightsquigarrow v_1.l \leftarrow v_2, \mathcal{M}_1}$$

$$\textsc{Location-Set-Value}\ \frac{value(v_2)}{\mathcal{M} \vdash \langle m : \tau\rangle.l \leftarrow v_2 \rightsquigarrow \mathcal{M}(m, l) \leftarrow v_2}$$

$$\textsc{Fix}\ \frac{e \rightsquigarrow v}{\mathbf{fix}\ e \rightsquigarrow \mathbf{fix}\ v}$$

$$\textsc{Fix-Value}\ \frac{\begin{array}{c} match(p, e, \mathcal{M}) = \overline{(p, e)}, \mathcal{M}' \\ let\ \theta = \begin{array}{l} [\quad p_1 \mapsto child_1(\mathbf{fix}\ (\mathbf{function}\langle x : \tau_x\rangle \rightarrow e)); \\ \qquad .. \\ \quad p_n \mapsto child_n(\mathbf{fix}\ (\mathbf{function}\langle x : \tau_x\rangle \rightarrow e))] \end{array} \end{array}}{\mathcal{M}, \mathbf{fix}\ (\mathbf{function}\ \langle p : \tau\rangle \rightarrow e) \rightsquigarrow (\theta\ e_1, .., \theta\ e_n), \mathcal{M}'}$$

$$\textsc{Seq-Left}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1;\ e_2 \rightsquigarrow v_1;\ e_2, \mathcal{M}_1}$$

$$\textsc{Seq-Right}\ \frac{value(v_1) \qquad M \vdash e_2 \rightsquigarrow v_2, \mathcal{M}_2}{\mathcal{M} \vdash v_1;\ e_2 \rightsquigarrow v_1;\ v_2, \mathcal{M}_2}$$

$$\textsc{While-True}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \mathtt{true}, \mathcal{M}_1}{\mathcal{M} \vdash \mathbf{while}\ e_1\ \mathbf{do}\ e_2\ \mathbf{done} \rightsquigarrow e_2;\ \mathbf{while}\ e_1\ \mathbf{do}\ e_2\ \mathbf{done}, \mathcal{M}_1}$$

$$\textsc{While-False}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \mathtt{false}, \mathcal{M}_1}{\mathcal{M} \vdash \mathbf{while}\ e_1\ \mathbf{do}\ e_2\ \mathbf{done} \rightsquigarrow \mathbf{unit}, \mathcal{M}_1}$$

$$\textsc{For-Init}\ \frac{\mathcal{M} \vdash e_j \rightsquigarrow v_j, \mathcal{M}_j}{\mathcal{M} \vdash \mathbf{for}\ i = e_i\ \mathbf{to}\ e_j\ \mathbf{do}\ e\ \mathbf{done} \rightsquigarrow \mathbf{for}\ i = e_i\ \mathbf{to}\ v_j\ \mathbf{do}\ e\ \mathbf{done}, \mathcal{M}_j}$$

$$\textsc{For-Continue}\ \frac{\mathcal{M} \vdash e_i \rightsquigarrow v_i, \mathcal{M}_i \qquad v_i \leq v_j}{\mathcal{M} \vdash \mathbf{for}\ i = e_i\ \mathbf{to}\ v_j\ \mathbf{do}\ e\ \mathbf{done} \rightsquigarrow e[i \mapsto v_i]; \mathbf{for}\ i = v_i + 1\ \mathbf{to}\ v_j\ \mathbf{do}\ e\ \mathbf{done}, \mathcal{M}_i}$$

$$\textsc{For-Stop}\ \frac{\mathcal{M} \vdash e_i \rightsquigarrow v_i, \mathcal{M}_i \qquad v_i > v_j}{\mathcal{M} \vdash \mathbf{for}\ i = e_i\ \mathbf{to}\ v_j\ \mathbf{do}\ e\ \mathbf{done} \rightsquigarrow \mathbf{()}, \mathcal{M}_i}$$

$$\textsc{Lazy}\ \frac{let\ m \notin dom(\mathcal{M})}{\mathcal{M} \vdash \mathbf{lazy}\ e \rightsquigarrow \langle m : \tau\rangle, (\mathcal{M}, (m : \tau = [thunk = Thunk\ e]))}$$

Réduction en présence d'exceptions

$$\textsc{Raise}\ \frac{e \leadsto v}{\texttt{raise}\ e \leadsto \texttt{raise}\ v} \qquad \textsc{Raise-Raise}\ \frac{\mathit{value}(v)}{\texttt{raise}\ (\texttt{raise}\ v) \leadsto \texttt{raise}\ v}$$

$$\textsc{Try}\ \frac{e \leadsto v}{\texttt{try}\ e\ \texttt{with}\ \overline{p \to e} \leadsto \texttt{try}\ v\ \texttt{with}\ \overline{p \to e}} \qquad \textsc{Try-Value}\ \frac{\mathit{value}(v)}{\texttt{try}\ v\ \texttt{with}\ \overline{p \to e} \leadsto v}$$

$$\textsc{Try-Raise}\ \frac{\mathit{value}(v)}{\texttt{try}\ \langle \texttt{raise}\ v : \tau \rangle\ \texttt{with}\ \overline{p \to e} \ \leadsto \begin{array}{l} \texttt{match}\ v\ \texttt{with} \\ \mid \overline{p \to e} \\ \mid \langle\ \_ : \mathtt{exn} \rangle \to \langle \texttt{raise}\ v : \tau \rangle \end{array}}$$

$$\textsc{App-Left-Raise}\ \frac{}{(\texttt{raise}\ v_1)\ e_2 \leadsto \texttt{raise}\ v_1} \qquad \textsc{App-Right-Raise}\ \frac{\mathit{value}(v_1)}{v_1\ (\texttt{raise}\ v_2) \leadsto \texttt{raise}\ v_2}$$

$$\textsc{Let-Raise}\ \frac{\forall i.\mathit{value}(v_i) \qquad \exists i.v_i \leadsto \texttt{raise}\ v_i'}{\texttt{let}\ \overline{\langle p : \sigma \rangle = v}\ \texttt{in}\ e' \leadsto \texttt{raise}\ v_i'}$$

$$\textsc{Tuple-First-Raise}\ \frac{\mathcal{M} \vdash e_1 \leadsto \texttt{raise}\ v_1, \mathcal{M}_1}{\mathcal{M} \vdash (e_1,\ e_2) \leadsto \texttt{raise}\ v_1, \mathcal{M}_1}$$

$$\textsc{Tuple-Second-Raise}\ \frac{\mathit{value}(v_1) \qquad \mathcal{M} \vdash e_2 \leadsto \texttt{raise}\ v_2, \mathcal{M}_2}{\mathcal{M} \vdash (v_1,\ e_2) \leadsto \texttt{raise}\ v_2, \mathcal{M}_2}$$

$$\textsc{Constr-First-Raise}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}{\mathcal{M} \vdash K(e_1,\ e_2) \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}$$

$$\textsc{Constr-Second}\ \frac{value(v_1) \qquad \mathcal{M} \vdash e_2 \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}{\mathcal{M} \vdash K(v_1,\ e_2) \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}$$

$$\textsc{Match-Scrut-Raise}\ \frac{\mathcal{M} \vdash e \rightsquigarrow \texttt{raise}\ v, \mathcal{M}'}{\mathcal{M} \vdash \texttt{match}\ e\ \texttt{with}\overline{|\ p \to e} \rightsquigarrow \texttt{raise}\ v, \mathcal{M}'}$$

$$\textsc{Record-First-Raise}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}{\mathcal{M} \vdash \{l_1 = e_1;\ l_2 = e_2\} \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}$$

$$\textsc{Record-Second-Raise}\ \frac{value(v_1) \qquad M \vdash e_2 \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}{\mathcal{M} \vdash \{l_1 = v_1;\ l_2 = e_2\} \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2} \textsc{Record-Access-Raise}\ \frac{\mathcal{M} \vdash e \rightsquigarrow \texttt{raise}\ v, \mathcal{M}'}{\mathcal{M} \vdash e.l \rightsquigarrow \texttt{raise}\ v, \mathcal{M}'}$$

$$\textsc{Record-Set-Raise}\ \frac{\mathcal{M} \vdash e_2 \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}{\mathcal{M} \vdash e_1.l \leftarrow e_2 \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}$$

$$\textsc{Record-Set-Value-Raise}\ \frac{value(v_2) \qquad \mathcal{M} \vdash e_1 \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1.l \leftarrow v_2 \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1} \qquad \textsc{Fix-Raise}\ \frac{e \rightsquigarrow \texttt{raise}\ v}{\texttt{fix}\ e \rightsquigarrow \texttt{raise}\ v}$$

$$\textsc{Seq-Left-Raise}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}{\mathcal{M} \vdash e_1;\ e_2 \rightsquigarrow \texttt{raise}\ v_1, \mathcal{M}_1}$$

$$\textsc{Seq-Right-Raise}\ \frac{value(v_1) \qquad M \vdash e_2 \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}{\mathcal{M} \vdash v_1;\ e_2 \rightsquigarrow \texttt{raise}\ v_2, \mathcal{M}_2}$$

$$\textsc{While-Raise}\ \frac{\mathcal{M} \vdash e_1 \rightsquigarrow \texttt{raise}\ v, \mathcal{M}_1}{\mathcal{M} \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} \rightsquigarrow \texttt{raise}\ v, \mathcal{M}_1}$$

$$\textsc{For-Init-Raise}\ \frac{\mathcal{M} \vdash e_j \rightsquigarrow \texttt{raise}\ v_j, \mathcal{M}_j}{\mathcal{M} \vdash \texttt{for}\ i = e_i\ \texttt{to}\ e_j\ \texttt{do}\ e\ \texttt{done} \rightsquigarrow \texttt{raise}\ v_j, \mathcal{M}_j}$$

$$\textsc{For-Continue-Raise}\ \frac{\mathcal{M} \vdash e_i \rightsquigarrow \texttt{raise}\ v_i, \mathcal{M}_i \qquad v_i \leq v_j}{\mathcal{M} \vdash \texttt{for}\ i = e_i\ \texttt{to}\ v_j\ \texttt{do}\ e\ \texttt{done} \rightsquigarrow \texttt{raise}\ v_i, \mathcal{M}_i}$$

B.3.3 Filtrage de motifs

Le motif **lazy** permettant de calculer la valeur suspendue, la fonction de filtrage prend désormais la mémoire en argument, et retourne une nouvelle mémoire dans laquelle l'éventuelle valeur suspendue a été calculée.

$$
\begin{array}{lcl}
\text{match } (x, \text{e}, \mathcal{M}) & = & (\text{x}, \text{e}), \mathcal{M} \\
\text{match } (\_, \text{e}, \mathcal{M}) & = & \emptyset, \mathcal{M} \\
\text{match } (\text{c}, \text{c}, \mathcal{M}) & = & \emptyset, \mathcal{M} \\
\text{match } (\text{p}_1 \mid \text{p}_2, \text{e}, \mathcal{M}) & = & \begin{cases} \text{match } (\text{p}_1, \text{e}, \mathcal{M}) \\ \text{match } (\text{p}_2, \text{e}, \mathcal{M}'), \quad \text{si match } (\text{p}_1, \text{e}) = \bot, \mathcal{M}' \end{cases} \\
\text{match } ((\text{p}_1, .., \text{p}_n), (\text{e}_1, .., \text{e}_n), \mathcal{M}) & = & \text{let } \mathcal{M}_0 = \mathcal{M} \text{ in } \bigcup_{k \in [i..j]} \text{match } (\text{p}_k, \text{e}_k, \mathcal{M}_{k-1}) \\
\text{match } (\text{K } (\text{p}_1, .., \text{p}_n), \text{K } (\text{e}_1, .., \text{e}_n), \mathcal{M}) & = & \text{let } \mathcal{M}_0 = \mathcal{M} \text{ in } \bigcup_{k \in [i..j]} \text{match } (\text{p}_k, \text{e}_k, \mathcal{M}_{k-1}) \\
\text{match } (\{\text{l}_i = \text{p}_i \,; ..;\, l_j = \text{l}_j\}), \langle\, m : \tau \,\rangle, \mathcal{M}) & = & \text{let } \mathcal{M}_0 = \mathcal{M} \text{ in } \bigcup_{k \in [i..j]} \text{match } (\text{p}_k, \mathcal{M}_{k-1} \ (m, \text{l}_k), \mathcal{M}_{k-1} k) \\
\text{match } (\textbf{lazy } \text{p}, \langle\, m : \tau \,\rangle, \mathcal{M}) & = & \text{let } v, \mathcal{M}' = \text{Lazy.force } m \ \mathcal{M} \text{ in match } (\text{p}, v, \mathcal{M}') \\
\text{match } (\_, \_, \mathcal{M}) & = & \bot, \mathcal{M}
\end{array}
$$

Bibliographie

[1] Sébastien AILLERET. Typage du bytecode Caml. In *Journées Françaises des Langages Applicatifs, JFLA02*, pages 43-60, 2002 (cf. page 18).

[2] Hassan AÏT-KACI et Jacques GARRIGUE. Label-Selective Lambda-Calculus Syntax and Confluence. In tome 151 de numéro 2, pages 353-383, 1995 (cf. page 7).

[3] Florian <octachron> ANGELETTI. Escaped existential type. `https://caml.inria.fr/mantis/view.php?id=7222` (cf. page 105).

[4] Brian E. AYDEMIR, Arthur CHARGUÉRAUD, Benjamin C. PIERCE, Randy POLLACK et Stephanie WEIRICH. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3-15, 2008 (cf. pages 24, 127).

[5] Nick BENTON, Andrew KENNEDY et George RUSSELL. Compiling standard ML to java bytecodes. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.* Pages 129-140, 1998 (cf. page 17).

[6] Nick BENTON, Andrew KENNEDY et Claudio V. RUSSO. Adventures in interoperability : the SML.NET experience. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 24-26 August 2004, Verona, Italy*, pages 215-226, 2004 (cf. page 18).

[7] Sandrine BLAZY, Zaynah DARGAYE et Xavier LEROY. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, tome 4085 de *Lecture Notes in Computer Science*, pages 460-475. Springer, 2006 (cf. page 20).

[8] Sandrine BLAZY et Xavier LEROY. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3) :263-288, 2009 (cf. page 20).

[9] Nicolaas Govert De BRUIJN. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *INDAG. MATH*, 34 :381-392, 1972 (cf. page 24).

[10] Emmanuel CHAILLOUX, Grégoire HENRY et Raphaël MONTELATICI. Mixing the Objective Caml and C# programming models in the .NET framework. *CoRR*, abs/0705.1458, 2007 (cf. page 18).

[11] Arthur CHARGUÉRAUD. Program verification through characteristic formulae. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 321-332, 2010 (cf. page 19).

[12] Arthur CHARGUÉRAUD. The locally nameless representation. *J. Autom. Reasoning*, 49(3) :363-408, 2012 (cf. pages 10, 24).

[13] Arthur CHARGUÉRAUD et François POTTIER. Locally nameless representation with cofinite quantification. `http://chargueraud.org/softs/ln/` (cf. pages 10, 24).

[14] Arthur CHARGUÉRAUD et François POTTIER. TLC : a non-constructive library for Coq. `http://chargueraud.org/softs/tlc/` (cf. page 35).

[15] Perry CHENG, Robert HARPER, J. Gregory MORRISETT, Leaf PETERSEN et Christopher A. STONE. The Design of the TILT Compiler for SML. Rapport technique (cf. page 14).

[16] Perry CHENG, Robert HARPER, J. Gregory MORRISETT, Leaf PETERSEN et Christopher A. STONE. Tilt : standard ml compiler based on typed intermediate languages. `https://github.com/RobertHarper/TILT-Compiler` (cf. page 14).

[17] Adam CHLIPALA. *Certified programming with dependent types - A pragmatic introduction to the Coq proof assistant*. In MIT Press, 2013. Chapitre 3, pages 64-69 (cf. page 127).

[18] Dominique CLÉMENT, Joëlle DESPEYROUX, Th. DESPEYROUX et Gilles KAHN. A simple applicative language : mini-ml. In *LISP and Functional Programming*, pages 13-27, 1986 (cf. page 21).

[19] Guy COUSINEAU, Pierre-Louis CURIEN et Michel MAUNY. The Categorical Abstract Machine. *Science of Computer Programming*, 8 :173-202, 1987 (cf. page 7).

[20] Luís DAMAS et Robin MILNER. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207-212, 1982 (cf. page 23).

[21] Zaynah DARGAYE. *Vérification formelle d'un compilateur pour langages fonctionnels*. Thèse de doctorat, Université Paris 7 Diderot, juillet 2009 (cf. page 20).

[22] Stephen <stdolan> DOLAN. Segfault from bug in gadt/module typing. `https://caml.inria.fr/mantis/view.php?id=6992` (cf. page 104).

[23] Damien DOLIGEZ et Xavier LEROY. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 113-123, 1993 (cf. page 7).

[24] Catherine DUBOIS. Proving ML type soundness within Coq. In *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, pages 126-144, 2000 (cf. page 20).

[25] Catherine DUBOIS et Valérie MÉNISSIER-MORAIN. Certification of a type inference tool for ML : Damas-Milner within Coq. *J. Autom. Reasoning*, 23(3-4) :319-346, 1999 (cf. page 20).

[26] Stephen N. FREUND et John C. MITCHELL. A type system for the Java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4) :271-321, 2003 (cf. page 17).

[27] Jacques GARRIGUE. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25(4) :867-891, 2015 (cf. page 20).

[28] Jacques GARRIGUE. Programming with polymorphic variants. In *ML Workshop*, tome 13. Baltimore, 1998 (cf. page 7).

[29] Jacques GARRIGUE. Relaxing the value restriction. In *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*, pages 31-45, 2002 (cf. page 88).

[30] Jacques GARRIGUE et Didier RÉMY. Ambivalent types for principal type inference with gadts. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 257-272, 2013 (cf. page 7).

[31] Jean-Yves GIRARD. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat, Paris Diderot University, France, 1972 (cf. page 21).

[32] Michael J. C. GORDON, Robin MILNER, L. MORRIS, Malcolm C. NEWEY et Christopher P. WADSWORTH. A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 119-130, 1978 (cf. page 7).

[33] James GOSLING, Bill JOY, Guy STEELE, Gilad BRACHA, Alex BUCKLEY et Daniel SMITH. *The Java® Language Specification, Java SE 10 Edition*. Oracle. Février 2018. URL : `https://docs.oracle.com/javase/specs/jls/se10/html/index.html` (cf. page 17).

[34] Armaël GUÉNEAU, Magnus O. MYREEN, Ramana KUMAR et Michael NORRISH. Verified characteristic formulae for cakeml. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 584-610, 2017 (cf. page 19).

[35] Tom HIRSCHOWITZ, Xavier LEROY et J. B. WELLS. Compilation of extended recursion in call-by-value functional languages. *CoRR*, abs/0902.1257, 2009 (cf. page 115).

[36] Stefan KAES. Parametric overloading in polymorphic programming languages. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, pages 131-144, 1988 (cf. page 14).

[37] Oleg KISELYOV. How OCaml type checker works – or what polymorphism and garbage collection have in common. `http://okmij.org/ftp/ML/generalization.html`, 2013 (cf. page 65).

[38] Oleg KISELYOV. The design and implementation of BER metaocaml - system description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 86-102, 2014 (cf. page 67).

[39] Daniel K. LEE, Karl CRARY et Robert HARPER. Towards a mechanized metatheory of standard ML. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173-184, 2007 (cf. page 20).

[40] Xavier LEROY. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363-446, 2009 (cf. page 20).

[41] Xavier LEROY. A modular module system. *J. Funct. Program.*, 10(3) :269-303, 2000 (cf. page 7).

[42] Xavier LEROY. The ZINC experiment : an economical implementation of the ML language. Technical report 117, INRIA, 1990 (cf. page 18).

[43] Tim LINDHOLM, Frank YELLIN, Gilad BRACHA et Alex BUCKLEY. *The Java® Virtual Machine Specification, Java SE 10 Edition*. Oracle. Février 2018. URL : `https://docs.oracle.com/javase/specs/jvms/se10/html/index.html` (cf. page 17).

[44] Michel MAUNY. *Compilation des Langages Fonctionnels dans les Combinateurs Catégoriques – Application au langage ML*. Thèse de doctorat, Université Paris 7, 1985 (cf. page 7).

[45] Michel MAUNY et Ascander SUÁREZ. Implementing functional languages in the Categorical Abstract Machine. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, pages 266-278, 1986 (cf. page 7).

[46] Arie MIDDELKOOP, Atze DIJKSTRA et S. Doaitse SWIERSTRA. A lean specification for gadts : system f with first-class equality proofs. *Higher Order Symbol. Comput.*, 23(2) :145-166, juin 2010. ISSN : 1388-3690 (cf. page 14).

[47] Robin MILNER. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3) :348-375, 1978 (cf. page 7).

[48] Robin MILNER, Mads TOFTE et Robert HARPER. *Definition of standard ML*. MIT Press, 1990. ISBN : 978-0-262-63132-7 (cf. page 19).

[49] Robin MILNER, Mads TOFTE et Robert HARPER. *Definition of standard ML (Revised)*. MIT Press, 1997. ISBN : 978-0-262-63181-5 (cf. page 19).

[50] Raphaël MONTELATICI. *Langages fonctionnels, typage et interopérabilité : Objective Caml sur .NET. (Functional languages, typing and interoperability : Objective Caml on .NET)*. Thèse de doctorat, Paris Diderot University, France, 2007 (cf. page 18).

[51] J. Gregory MORRISETT, David WALKER, Karl CRARY et Neal GLEW. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3) :527-568, 1999 (cf. page 14).

[52] Magnus O. MYREEN et Scott OWENS. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3) :284-315, 2014 (cf. page 19).

[53] George C. NECULA. Proof-carrying code. In *Conference Record of POPL'97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106-119, 1997 (cf. page 18).

[54] Leaf PETERSEN, Perry CHENG, Robert HARPER et Chris STONE. Implementing the TILT Internal Language. Rapport technique CMU-CS-00-180, School of Computer Science, Carnegie Mellon University, décembre 2000 (cf. page 14).

[55] Simon PEYTON JONES. Haskell 98 Language and Libraries : the Revised Report, janvier 2003 (cf. page 14).

[56] Benjamin C. PIERCE. *Types and programming languages*. MIT Press, 2002. ISBN : 978-0-262-16209-8 (cf. page 113).

[57] Richard PORTER. *Top Gear : Ambitious but Rubbish : The Secrets Behind Top Gear's Craziest Creations*. BBC Books, 2012. ISBN : 978-1849905039 (cf. page 7).

[58] François POTTIER et Didier RÉMY. The essence of ML type inference. In Benjamin C. PIERCE, éditeur, *Advanced Topics in Types and Programming Languages*, chapitre 10, pages 389-489. MIT Press, 2005 (cf. page 23).

[59] Didier REMY. Extension of ML Type System with a Sorted Equational Theory on Types. Rapport technique, 1992 (cf. page 65).

[60] Didier RÉMY et Jerome VOUILLON. Objective ML : an effective object-oriented extension to ML. *TAPOS*, 4(1) :27-50, 1998 (cf. page 7).

[61] Microsoft RESEARCH. *.NET Standard*. Microsoft. Août 2018. URL : `https://github.com/dotnet/standard` (cf. page 18).

[62] Martin SULZMANN, Manuel M. T. CHAKRAVARTY, Simon Peyton JONES et Kevin DONNELLY. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53-66, Nice, Nice, France. ACM, 2007. ISBN : 1-59593-393-X (cf. page 14).

[63] Yong Kiam TAN, Scott OWENS et Ramana KUMAR. A verified type system for cakeml. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, Koblenz, Germany, September 14-16, 2015*, 7 :1-7 :12, 2015 (cf. pages 20, 123).

[64] David TARDITI, Greg MORRISETT, Perry CHENG, Chris STONE, Robert HARPER et Peter LEE. Til : a type-directed, optimizing compiler for ml. *SIGPLAN Not.*, 39(4) :554-567, avril 2004. ISSN : 0362-1340 (cf. page 13).

[65] David TARDITI, J. Gregory MORRISETT, Perry CHENG, Christopher A. STONE, Robert HARPER et Peter LEE. TIL : A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, pages 181-192, 1996 (cf. page 13).

[66] Stephanie WEIRICH, Justin HSU et Richard A. EISENBERG. System FC with Explicit Kind Equality. *SIGPLAN Not.*, 48(9) :275-286, septembre 2013. ISSN : 0362-1340 (cf. page 14).

[67] Stephanie WEIRICH, Dimitrios VYTINIOTIS, Simon PEYTON JONES et Steve ZDANCEWIC. Generative type abstraction and type-level computation. *SIGPLAN Not.*, 46(1) :227-240, janvier 2011. ISSN : 0362-1340 (cf. page 14).

[68] Pierre WEIS et Xavier LEROY. *Le langage Caml*. InterEditions, 1993. ISBN : 978-2-7296-0493-6 (cf. page 7).

[69] Andrew K. WRIGHT. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4) :343-355, 1995 (cf. page 87).

[70] Jeremy YALLOP. A new check that 'let rec' bindings are well formed. https://github.com/ocaml/ocaml/pull/556 (cf. page 115).

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Vérification des résultats d'inférence de types du langage OCaml

Mots clés : systèmes de types, OCaml, langage fonctionnel, compilation

Résumé : OCaml est un langage fonctionnel statiquement typé, qui génère après inférence de types un arbre de syntaxe abstraite dans lequel chacun des noeuds est annoté avec un ensemble d'informations issues de cette inférence. Ces informations, en particulier les types inférés, constituent une preuve de typage de l'expression annotée.
Ce manuscrit de thèse s'intéresse à la vérification de ces arbres annotés en les considérant comme des preuves de typages du programme, et décrit un ensemble de règles permettant d'en vérifier la cohérence. La formalisation de ces règles de vérification de preuves de types peut être vue comme une représensation du système de types du langage étudié.
Cette thèse présente plusieurs aspects de la vérification d'arbres de syntaxe annotés. Le premier cas étudié est la formalisation d'un dérivé de MiniML où toutes les expressions sont annotées de manière théoriquement parfaite, et montre qu'il est possible d'écrire des règles de vérification de manière algorithmique, rendant directe la preuve de correction vis-à-vis de la spécification. La seconde partie s'intéresse à la formalisation de règles de vérification pour un sous-ensemble du premier langage intermédiaire d'OCaml, le TypedTree, accompagné d'un vérificateur implémentant ces règles. Ces règles constituent alors une représentation du système de types d' OCaml, document jusqu'alors inexistant, au mieux disséminé dans diverses publications.

Title : Checking type inference results of the OCaml language

Keywords : type systems, OCaml, functional language, compilation

Abstract : OCaml is a statically typed programming language that generates typed annotated abstract syntax trees after type inference. Each of their nodes contains information derived from the inference like the inferred type and the environment used to find this information. These annotated trees can then be seen as typing proofs of the program.
In this thesis, we introduce a consistency checking of type-annotated trees, considering them as typing proof, and we describe a set of rules that defines the consistency property. Such consistency checking rules can then be seen as a formalized representation of the type system, since consistency ensures the typing invariants of the language.
This thesis introduces multiple aspects of checking type-annotated trees. First of all, it considers a simplified and ideal version of MiniML and formalizes a set of rules to check consistency. In this formalism, we consider ideally type-annotated trees, which might not be the case for OCaml typed trees. Such type checking rules are presented in an algorithmic form, reducing as much as possible the gap from formalism to implementation. As such, they ease the correction proof between the implementation of the type checker and the specification of the type system. The second part of this thesis is dedicated to the formalization of a set of rules for a subset of the OCaml annotated trees: the TypedTree. The formalism described in these chapters is implemented as a type checker working on large subset of the language, leaving the formalization of some aspects for a further work. These rules constitute a formalized representation of the OCaml type system in a single document.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France