université PARIS-SACLAY

ENSTA ParisTech université PARIS-SACLAY

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 Ville de soutenance, le Date, par

Pierrick Couderc

Composition du Jury :

XXX XXX Statut, Établissement (– Unité de recherche)	Président
M. Emmanuel Chailloux Professeur, Université Pierre et Marie Curie, Paris 6 (– APR)	Rapporteur
M. Jacques Garrigues Professeur, Université de Nagoya (– Graduate School of Mathematics)	Rapporteur
XXX XXX Statut, Établissement (– Unité de recherche)	Examinateur
XXX XXX Statut, Établissement (– Unité de recherche)	Examinateur
M. Michel Mauny Professeur, Inria Paris (– Gallium)	Directeur de thèse
M. Fabrice Le Fessant Directeur technique, OcamlPro	Co-Directeur de thèse
XXX XXX Statut, Établissement (– Unité de recherche)	Invité

Thèse de doctorat

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 :

Title : Checking type inference results of the OCaml language

Keywords : type systems, OCaml, functional language, compilation

Abstract :

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Table des matières

Chapitre 1

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 [33].

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 [18, 28]. Il s'agit à l'origine de CAML [26], une première version écrite au milieu des années 80 dont l'intérêt était la compilation de lan-

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

gages fonctionnels, et en particulier ML, vers la Machine Abstraite Catégorique [25, 10]. Une nouvelle implémentation sera proposée cinq ans plus tard, sous le nom de *Caml-Light* [41], une version plus rapide compilant vers une machine virtuelle et possédant notamment un glaneur de cellules générationnel [12]. OCaml en est le successeur : il y ajoute un système de modules avec foncteurs applicatifs [24] et une couche d'objets avec de l'héritage et du sous-typage [36]. Le langage connaît de nombreux ajouts de fonctionnalités par la suite : variants polymorphes [16], modules de première classe, types algébriques gardés [17] ou encore arguments de fonctions labellisés [1]. 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 le tout intégré par divers développeurs. L'implementation 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 :

— 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 implementée 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.

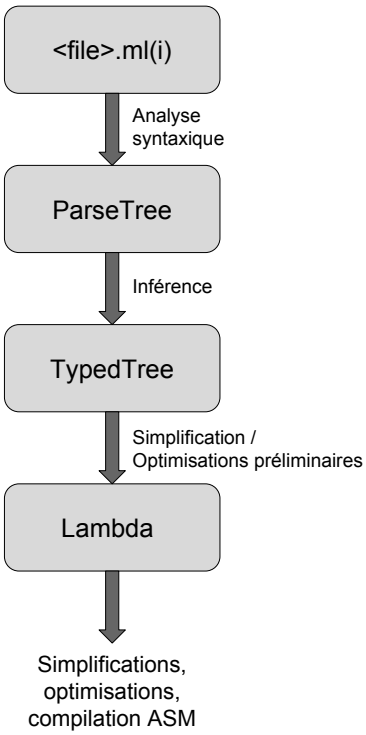

FIGURE 1.1 – Représentation de la chaîne de compilation d'OCaml (jusqu'au Lambda)

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, la travail de reconstruction se base sur l'un des langages intermédiaires du compilateur : les arbre de syntaxe annotés, ou *Typedtree* (voir figure 1.1 pour 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 : le nœud représentant cette expression peut être annoté par son type si ses sous-nœuds sont eux-même cohérents et 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 type. 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), rendant tout le processus de retour en arrière fastidieux. L'une des conséquences est l'extraction des erreurs de types qui peuvent parfois être difficiles à interpréter. 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

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.

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 celle-ci 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 d'un système exécutable que possible : 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. Le second chapitre présente une version de MiniML (sans références) dont chaque nœud est annoté, avec les prémisses de la décomposition de l'unification en différentes opérations. Ce chapitre est accompagné d'une présentation en OCaml d'un vérificateur pour MiniML, et d'une formalisation Coq de ces règles de vérification. Le troisième chapitre aborde la formalisation de la vérification de cohérence du Typedtreed'OCaml pour le cœur du langage, autrement dit tout ce qui ne concerne ni modules ni objets. Ce chapitre présente notamment des extraits du vérificateur accompagnés de la règle associée, pour montrer la correspondance directe entre le code et la règle. On y donne également une sémantique opérationnelle à petit pas pour le langage. 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.

Chapitre 2

Etat de l'art

Ce bref chapitre présente un tour des différentes techniques de vérification de cohérence de programmes étant donnés une spécification d'un système de type.

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 typée, 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 de prouver son programme, mais celui-ci n'a pas besoin de l'écrire dans le langage de l'assistant de preuve en question.

2.1 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 implanté pour des langages tels que SML dans les compilateurs TIL ([38]) et CakeML, mais aussi pour la vérification de programmes Haskell dans le compilateur ghc.

2.1.1 Haskell : Core et Système FC

Haskell [31] 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* [20] (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 [37] [27] [40] [39] 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, lui-même 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` sont alors des témoins de types que ce paramètre `a` est égal 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 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 exprimable 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 langa 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-même 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.

2.1.2 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 pour les foncteurs, avant d'être effectivement éliminés comme des modules classiques. Les abbréviations 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 appe-

lée "XML" [1], une représentation du programme où tous les noeuds 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 verront plus tard, ont lieu exclusivement au niveau des occurences de variables. La transformation de CoreML vers XML est principalement utilisée pour la compilation du filtrage de motifs et 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 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 noeud, et la vérification s'assure de la cohérence d'un noeud sachant les annotations de ses noeuds fils. Néanmoins, le langage est plus restreint que le TypedTree d'OCaml. Au dela des constructions supplémentaires que compte OCaml, les modules et foncteurs ont déjà été éliminés, les abbréviations 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 considéré comme une implémentation du système de types de SML, comme l'est un vérificateur de types pour le TypedTree

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

d'OCaml.

2.2 Programmes avec preuve embarquée

La technique de programmes avec preuves embarquées [30] 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.

2.3 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'implantation 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 [7] 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 = 0 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 formalizer 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 écrit et prouvé en HOL [29]. En particulier, les programmes CakeML peuvent être traduits vers des formules caractéristiques [19], 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 standardisé, contrairement à OCaml dont il n'existe qu'une implantation, 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'arbre de syntaxe annoté 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.

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 [5] [23] [6], un compilateur C vers assembleur entièrement écrit en Coq et dont la sémantique et la correction est prouvée 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 formalization 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 [14] [13] sur la preuve de l'inférence de type de ML en Coq. De même, il existe un travail de formalization du Standard ML [22], cette fois en Twelf, sous la forme d'un langage intermédaire 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 [11].

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 [15]. Ceux-ci visent à formaliser un moteur d'inférence en présence de ces constructions, prouver sa correction et sa sûreté.

Chapitre 3

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 construction. 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 restreinte 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 spécifité 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, le différence étant majoritairement une question du choix de la structure de donnée utilisée pour représenter le TAST, l'environnement et la substitution.

Symboles :

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

Syntaxe :

$e ::= \langle\, e' : \tau \,\rangle$	*noeud d'expression annoté*
$e' ::= c$	*constante*
$\mid x$	*variable*
$\mid \lambda \langle\, x : \tau \,\rangle .\ t$	*abstraction annotée*
$\mid t1\ t2$	*application*
$\mid$ **let** $x : \sigma = t1$ **in** $t2$	*let-binding*

FIGURE *3.1 – Syntaxe de MiniML avec annotations*

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 noeuds 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 Typedtreed'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 noeud est cohérent et annoté de telle manière car ses sous-noeuds sont eux-mêmes cohérents et annotés d'un manière précise.

3.1 MiniML : définition

3.1.1 Expressions et algèbre de types

La syntaxe d'un TAST pour MiniML est décrite dans la figure 3.1. Les constructions sont classiques, mais intègrent quelques spécificitées dues à l'annotation de chaque noeud. En premier lieu, un noeud est lui-même une expression à laquelle est attaché un type. Ces expressions peuvent

$$
\begin{array}{lll}
\sigma ::= \tau & & \\
\quad \mid \forall\, \alpha.\, \sigma & & \\
 & & \\
\tau ::= \alpha & & \textit{variable de type} \\
\quad \mid \texttt{int} & & \textit{constructeur de type (int)} \\
\quad \mid \tau_1 \rightarrow \tau_2 & & \textit{type de fonction}
\end{array}
$$

FIGURE 3.2 – *Algèbre de types de MiniML*

alors être :

— Une variable ;

— Un entier constant ;

— 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 connues après l'inférence.

L'algèbre de types présentée dans le figure 3.2 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, qu'on restreindra à `int` pour représenter les entiers ;

— soit un type de fonction, 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 let polymorphisme du **`let`**, autrement dit seule cette construction permet d'introduire du polymorphisme, contrairement à un système de type comme Système F, où le polymorphisme est géré par deux constructions dans le langage pour introduire une variable polymorphe et instancier un type polymorphe.

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

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

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

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

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

FIGURE 3.3 – Règles d'inférence de MiniML

3.2 Inférence : génération du TAST

Les règles d'inférence présentées en figure 3.3 sont extraites de [34], auxquelles a été rajoutée la génération de noeuds annotés. Cette présentation des règles est classique, et on verra lors de l'analyse des règles de vérification qu'elles ne diffèrent que 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 considèrera 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 vérifications présentées plus loin est qu'elle associent une règle à une construction : les deux règles précédentes sont valables pour toutes les expressions et entraînent un dertain degré de complexité supplémentaire quand à leur compréhension et au choix de leur utilisation pendant l'inférence.

3.3 Représentation Coq

Il est possible de représenter cette version de MiniML totalement annotée en Coq. Celle-ci se base notamment sur les travaux de Bryan Aydemir et Arthur Chargueraud ([4], [8]) 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 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*[2] permettant la manipulation de telles variables, en plus de proposer un ensemble de structures de données utiles à la formalisation de langages.

La figure 3.4 donne la représentation de MiniML annoté présenté précédemment. Le type `ty` représente l'algèbre de types, accompagné de `sch` permettant la représentation des schémas de types. Contrairement à l'algèbre donnée en 3.2, 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`, une variable répondant aux critères énoncés par la représentation *localement sans nom*. Les schémas de types, contrairement à l'algèbre donné précédemment, ne sont pas imbriqués les uns dans les autres : un schéma possède une arité, et les variables contenues dans celui-ci dont l'indice est inférieur à cette arité sont quantifiées 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

```
(** 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 3.4 – Représentation Coq de MiniML annoté

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 de sa variable associée par une nouvelle variable libre. Dans le cas d'un schéma, qui donc peut quantifier plusieurs variables, l'ouverture doit donc générer plusieurs variables fraîches et substituer les variables indicées entre 0 et son arité -1. L'ouverture est donnée en figures 3.5 et 3.6.

L'ouverture est un algorithme de substitution : la fonction prend en argument un type (ou une expression) sur lequel (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. L'ouverture d'un schéma avec une liste de variables ne doit pas substituer les 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 : la variable liée doit être substituée par une nouvelle variable libre.

Finalement, on définit des opérateurs pour représenter ces ouvertures, de la même manière que sont représentées les opérations de substitution, et pour aider à la lecture des programmes.

```
(** 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 :=
  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 3.5 – 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 3.6 – Ouverture des quantificateurs d'expression

3.4 Règles de vérification de types

Puisque tout est annoté,il ne s'agit donc 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 type 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.

3.4.1 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

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

$$\textsc{Equiv-Forall}\ \frac{\mathit{let}\ \gamma \notin fv(\sigma_1) \cup fv(\sigma_2) \qquad \sigma_1[\alpha \mapsto \gamma] \equiv \sigma_2[\beta \mapsto \gamma]}{\forall\alpha.\ \sigma_1 \equiv \forall\beta.\ \sigma_2}$$

FIGURE 3.7 – Vérification d'équivalence de types de MiniML

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.

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 3.7.

L'équivalence est purement syntaxique, il s'agit de comparer deux à deux les types et s'assurer de leur égalité, dont les règles sont données en figure 3.7. 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 constructeurs de types `int` sont toujours équivalents (EQUIV-CONSTRUCT). Finalement deux schémas de types $\forall\alpha.\ \sigma_1$ et $\forall\beta.\ \sigma_2$ sont équivalents si et seulement si σ_1 est équivalent à σ_2 dans lequel toutes les occurences de α et β ont été substituées par une variable fraîche γ n'étant pas liées dans ces deux schémas. L'algorithme de substitution de variables de types est donné dans la figure 3.24.

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.

Les cas de la variable de type et du constructeur de type sont triviaux, puisqu'ils découlent de la définition de l'équivalence. Dans le cas $\tau_1 \to \tau_2 \equiv \tau_1 \to \tau_2$, on sait par hypothèse d'induction

que $\tau_1 \equiv \tau_1$ et $\tau_2 \equiv \tau_2$. Alors, la propriété est vraie d'après Equiv-Arrow. □

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 substitutiom)**.** *Pour tous α, β et τ, $\tau[\alpha \mapsto \beta] \equiv \tau[\alpha \mapsto \beta]$.*

Démonstration. Par induction sur τ.

Le seul cas affecté est celui de la variable. Si τ est α il y a substitutiob, alors on doit prouver $\beta \equiv \beta$, ce qui est vrai d'après le lemme de réflexivité. Si τ n'est pas α, il n'y a pas de substitution et $\alpha \equiv \alpha$ d'après le lemme de réflexivité. □

Lemme 3 (Réflexivité de l'équivalence des schémas)**.** *Pour tous α et σ, $\forall\alpha.\ \sigma \equiv \forall\alpha.\ \sigma$.*

Démonstration. Pour montrer $\forall\alpha.\sigma \equiv \forall\alpha.\sigma$, on peut facilement montrer que α est substituable par n'importe quel variable fraîche β, donnant alors σ'.

— Si α n'est pas liée dans σ, alors $\sigma' = \sigma$. On sait par induction que $\sigma \equiv \sigma$, et donc par définition de Equiv-Forall que la propriété est vraie.

— Si α est liée, on doit montrer $\sigma[\alpha \mapsto \beta] \equiv \sigma[\alpha \mapsto \beta]$. Sachant que la substitution conserve la réflexivité et que $\sigma \equiv \sigma$ par induction, la propriété est vraie.

□

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

Lemme 4 (Equivalence : inversion)**.** *Sachant 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* `int` $\equiv \tau$, *alors τ est un constructeur de type* `int`

— *si $\forall\alpha.\ \sigma \equiv \sigma'$, alors σ' est un schéma de type de la forme $\forall\beta.\ \sigma''$ et il existe γ tel que $\gamma \notin fv(\sigma) \cup fv(\sigma')$ et $\sigma[\alpha \mapsto \gamma] \equiv \sigma''[\beta \mapsto \gamma]$.*

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.

— si $\tau_1 = \alpha$, alors par inversion de $\tau_1 \equiv \tau_2$, $\tau_2 = \alpha$. Donc, par inversion de $\tau_2 \equiv \tau_3$, $\tau_3 = \alpha$. Ainsi il suffit de prouver $\alpha \equiv \alpha$, ce qui est vrai par définition de Equiv-Var.

— si $\tau_1 = \tau_{11} \to \tau_{12}$, alors par inversion de $\tau_1 \equiv \tau_2$, $\tau_2 = \tau_{21} \to \tau_{22}$, $\tau_{11} \equiv \tau_{21}$ et $\tau_{12} \equiv \tau_{22}$. De même, par inversion de $\tau_2 \equiv \tau_3$, on a $\tau_3 = \tau_{31} \to \tau_{32}$, $\tau_{21} \equiv \tau_{31}$ et $\tau_{22} \equiv \tau_{32}$. D'après l'hypothèse d'induction, on peut montrer que $\tau_{11} \equiv \tau_{31}$ et $\tau_{12} \equiv \tau_{32}$. Donc, d'après la définition de Equiv-Arrow, la propriété est vraie pour les types de fonction.

— la propriété pour les constructeurs de types est vraie par inversion, de la même manière que la preuve pour les variables de types.

□

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 induction sur σ_1.

Si le schéma n'est pas quantifié, i.e. σ_1 est un type classique, et par inversion de l'équivalence σ_2 n'est pas quantifié, et par conséquent σ_3 non plus. Alors la propriété la propriété est vraie d'après le lemme 5.

Si le schéma est quantifié, alors σ_1 est de la forme $\forall\alpha.\ \sigma'_1$. Par inversion sur $\forall\alpha.\ \sigma'_1 \equiv \sigma_2$, alors σ_2 est de la forme $\forall\beta.\ \sigma'_2$ et par inversion toujours σ_3 de la forme $\forall\gamma.\ \sigma'_3$.

On sait qu'il existe δ tel que $\delta \notin fv(\sigma_1) \cup fv(\sigma_2)$ et $\sigma_1[\alpha \mapsto \delta] \equiv \sigma_2[\beta \mapsto \delta]$, et ϵ tel que $\epsilon \notin fv(\sigma_2) \cup fv(\sigma_3)$ et $\sigma_2[\beta \mapsto \epsilon] \equiv \sigma_3[\gamma \mapsto \epsilon]$.

...

□

```
(** 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 3.8 – Implémentation de l'équivalence en Coq

Equivalence : représentation Coq La propriété d'équivalence est représentée par une propriété inductive (figure 3.8). L'implémentation est équivalente aux règles formelles énoncées précédement, à la différence qu'il n'existe pas d'équivalence entre variables liées mais seulement les 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 les variables liées ont été substituées par ces variables libres.

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 pour 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

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

$$\textsc{Inst-Fun}\ \frac{\theta \vdash \tau_1 \leqslant\ \tau'_1 \Rightarrow \theta_1 \qquad \theta_1 \vdash \tau_2 \leqslant\ \tau'_2 \Rightarrow \theta_2}{\theta \vdash \tau_1 \to \tau_2 \leqslant\ \tau'_1 \to \tau'_2 \Rightarrow \theta_2} \qquad \textsc{Inst-Construct}\ \theta \vdash \texttt{int} \leqslant \texttt{int} \Rightarrow \theta$$

FIGURE 3.9 – Instanciation de schémas de MiniML

données en figure 3.9.

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 3.10. Une substitution est alors soit vide, soit l'ajout à substitution existante d'un lien entre une variable de type et un type. A cela, on peut considérer plusieurs opérations :

— 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 : $\forall \alpha.\alpha \in dom(\theta) \to \alpha \in dom(\theta') \wedge \theta(\alpha) = \theta'(\alpha)$;

— 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 substituées par le type qui leur était associé dans θ.

Ces opeŕations sont décrites en figure 3.10.

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 donc spécifiée par :

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

Définition et propriétés :

— $\theta ::= \varnothing \mid \theta \oplus [\alpha \mapsto \tau]$
— $dom(\theta)$: 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 :

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

Bonne formation :

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

Substitution de variable :

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

FIGURE 3.10 – Substitutions

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 3.9.

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. Sachant la définition de $\theta_{init}(\forall\overline{\alpha}.\tau)$, alors l'ensemble de départ est construit *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 de l'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* $\theta \vdash \tau \leqslant \tau' \Rightarrow \theta'$, *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 $\tau' = \tau'_1 \to \tau'_2$, alors par inversion de la règle d'instanciation $\tau = \tau_1 \to \tau_2$, $\theta \vdash \tau_1 \leqslant \tau'_1 \Rightarrow \theta''$ (1) et $\theta'' \vdash \tau_2 \leqslant \tau'_2 \Rightarrow \theta'$ (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 $wf(\theta')$.

□

Lemme 9 (Instanciation retourne un sur-ensemble de sa substitution initiale)**.** *Pour tout substitution θ, θ', pour tous types τ, τ', si $\theta \vdash \tau \leqslant \tau' \Rightarrow \theta'$ 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 de fonction dérive de la propriété d'induction. □

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

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 sur `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*

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

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. $\theta_{init}(\forall\overline{\alpha}.\tau') \vdash \tau_1 \leqslant \tau'_1 \Rightarrow \theta'$
3. $\theta' \vdash \tau_2 \leqslant \tau'_2 \Rightarrow \theta$

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.

A revoir, semble tr

simple

— $\tau' = \texttt{int}$. D'après la propriété d'instanciation, $\tau = \texttt{int}$. Par conséquent $apply(\theta, \texttt{int}) = \texttt{int}$, et donc $\texttt{int} \equiv \texttt{int}$.

□

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

```
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 3.11 – 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 3.12 – TLC : interface des environnements

Instanciation et substitution : représentation Coq L'instantiation est définie en tant que propriété inductive (figure 3.11). 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 [3], dont l'interface est donnée figure 3.12.

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 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'instanciations sont donc identiques aux règles énoncées en figure 3.9. 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-noeuds 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 $\tau \prec \tau_1 \rightarrow \tau_2$: cette opération va alors vérifier que τ est bien une fonction, et va introduire son domaine et son codomaine dans le contexte courant respectivement sous la forme des metavariables 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'abbréviations et d'équations de types. Le filtrage fait alors office d'opération d'expansion des abbréviations et d'extraction de types équivalents, comme montré dans le chapitre dédié à la vérification de TAST OCaml.

3.4.2 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 données 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 proriétés sont donnés dans la figure 3.13.

On remarquera que cette définition ainsi que les deux propriétés de bonne formation et de domaine sont isomorphes à celles des substitutions présentées figure 3.10. Ainsi, les différentes propriétés prouvées sur ces seules définitions dans la sous section 3.4.1 sont compatibles avec celles des environnements.

Environnement : représentation Coq L'environnement utilisé pour la vérification est celui fournit par TLC (voir figure 3.12), et associe des schémas de types à des variables. Les opérations sont donc strictement identiques à celles énoncées pour les substitutions.

Définition et propriétés :
- $\Gamma ::= \varnothing \mid \Gamma, x : \sigma$
- $dom(\Gamma)$: ensemble des clés (variables) de l'environnement.
- $wf(\Gamma)$: prédicat indiquant l'unicité des variables liées dans l'environnement.
- $\Gamma \subseteq \Gamma'$: prédicat de sous-ensemble.

Domaine :

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

Bonne formation :

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

FIGURE 3.13 – Environnements

3.4.3 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 3.14.

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

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

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

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

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

FIGURE 3.14 – Règles de vérification de types de MiniML

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**, et on vérifie 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.

Si on compare ces règles à celle présentée pour l'inférence présentées précédemment (voir section 3.2), 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 méchanique, il est donc possible d'écrire un vérificateur de la même manière que les règles sont lues.

```
(** 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 (·: 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 (·: body) tcd →
    (forall x,
       x \notin fvs →
       check (env & x  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 (·: e1) (sch_body s) →
    (forall x,
       x \notin fvs →
       check (env & x  s)
             (e2 ^ x)) →
    equiv (·: e2) t →
    check env (term_let s e1 e2 t)
```

FIGURE 3.15 – Implémentation des règles de vérification en Coq

Vérification d'expression annotées : représentation Coq Sans surprise, la représentation des règles de vérification se fait à l'aide d'une propriété inductive (figure 3.15). Chaque branche est une transposition littérale des règles énoncées plus tôt. Il faut néanmoins noter l'opérateur `@:` 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.

3.4.4 Implémentation d'un vérificateur de types de MiniML

Supposons la nomenclature suivante :

— `check` désigne l'opération de vérification des expressions ;

— `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 MiniML, avec du filtrage de motifs. Deux exemples sont donnés en figure 3.16 : 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 3.17. 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, instan-

```
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
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) ->
equiv τ_e τ_σ >>= fun () ->
check_expression (Γ, x : σ) ⟨ e', τ'_e ⟩ >>= fun () ->
equiv τ'_e τ
```

FIGURE 3.16 – Implémentation de la règle LET

ciée avec la fonction de comparaison polymorphe ce 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 premet 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 3.18)

Finalement, l'algorithme de vérification est donné en figure 3.19.

3.5 Sémantique opérationelle du TAST

La sémantique opérationnelle de MiniML avec annotation est donnée dans la figure 3.20. 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 réduc-

```
(** 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 3.17 – 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 3.18 – 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 3.19 – Implémentation de la vérification de cohérence de MiniMLannoté

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

Réduction des expressions annotées

$\mathrm{e}_1\ \mathrm{e}_2$	$\rightsquigarrow$	$\mathrm{e}'_1\ \mathrm{e}_2$
$v_1\ \mathrm{e}_2$	$\rightsquigarrow$	$v_1\ \mathrm{e}_2$’
$(\lambda \langle\, x : \tau \,\rangle . \mathrm{e}_1)\ v_2$	$\rightsquigarrow$	$v_2[x \mapsto \mathrm{e}_1]$
$\mathbf{let} \langle\, x : \sigma \,\rangle = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2$	$\rightsquigarrow$	$\mathbf{let} \langle\, x : \sigma \,\rangle = \mathrm{e}'_1\ \mathbf{in}\ \mathrm{e}_2$
$\mathbf{let} \langle\, x : \forall \overline{\alpha}.\tau \,\rangle = v_1\ \mathbf{in}\ \mathrm{e}_2$	$\rightsquigarrow$	$\mathrm{e}_2[x \mapsto v_1 ; \overline{\alpha}]_\sigma$

FIGURE 3.20 – Sémantique opérationnelle de MiniML annoté

tion des expressions est alors relativement classique, à un point près :

- Une application se réduit toujours de gauche à droite, autrement dit il existe 3 cas
  - 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 variable liée dans le corps de la fonction par la valeur de l'argument.
- Dans le cas d'une variable locale, il y a alors deux cas :
  - Si l'expression liée n'est pas une valeur, elle est alors elle se réduit.
  - En revanche, si l'expression liée est une valeur, elle est substituée 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 est substituée dans le corps d'une autre, il est nécessaire pour chaque occurence de la variable associée 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 :

$$\texttt{let} \langle\ \texttt{x} : \forall\alpha.\ \alpha \to \alpha\ \rangle = \lambda \langle\ \texttt{x} : \alpha\ \rangle.\ \langle\ \texttt{x} : \alpha\ \rangle\ \texttt{in}$$
$$\langle\ \texttt{f} : \texttt{int} \to \texttt{int}\ \rangle \langle\ \texttt{0}\texttt{: int}\ \rangle$$

Le corps du `let` étant une valeur, la prochaine étape de réduction est donc de substituer `x` par sa valeur dans $\langle\ \texttt{f : int} \to \texttt{int}\ \rangle \langle\ \texttt{0}\texttt{: int}\ \rangle$. Avec une substitution classique, le résultat serait :

$$\langle\ \lambda \langle\ \texttt{x} : \alpha\ \rangle.\ \langle\ \texttt{x} : \alpha\ \rangle : \alpha \to \alpha\ \rangle \langle\ \texttt{0}\texttt{: int}\ \rangle$$

Or, un tel résultat n'est pas un TAST correct : en effet, sachant la règle Abs (figure 3.14), 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

Substitution classique

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

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 \leqslant \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 & \langle\, \mathbf{let} \langle\, y : \sigma \,\rangle = \mathrm{e}_1 \,[\, x \mapsto \mathrm{e} ; \theta_\alpha \,]_\sigma \; \mathbf{in} \\
 & & \quad \mathrm{e}_2 \,[\, x \mapsto \mathrm{e} ; \theta_\alpha \,]_\sigma : \tau \,\rangle
\end{array}
$$

FIGURE 3.21 – Substitution naïve de variable dans une expression annotée

types vis-à-vis de cette sémantique. La solution est alors d'effectuer une substitution qui prend en compte les instanciations de variables. Celle-ci, ainsi que la substitution simple, est donnée figure 3.21.

La substitution dite classique est simple : il s'agit de remplacer toute occurence 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 occurences 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 à 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 à substituer : cela permet alors de connaître les instances pour chaque variable quantifiée. Ensuite, cette nouvelle substitution obtenue doit être appliquée sur l'expression à substituer.

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

$$[\ \mathtt{f} \mapsto \langle\ \lambda\ \langle\ \mathtt{x} : \alpha\ \rangle.\ \langle\ \mathtt{x} : \alpha\ \rangle : \alpha \to \alpha\ \rangle;\ \theta_{init}(\forall\alpha.\ \alpha \to \alpha)\ ]$$

La variable `f` étant annotée avec $\mathtt{int} \to \mathtt{int}$, il faut calculer la substitution permettant une telle instance (la substitution initiale étant vide) :

$$\dfrac{\dfrac{\alpha \notin \varnothing}{\varnothing \vdash \mathtt{int} \leqslant \alpha \Rightarrow [\alpha \mapsto \mathtt{int}]} \qquad \dfrac{\alpha \in dom([\alpha \mapsto \mathtt{int}]) \qquad \mathtt{int} \equiv \mathtt{int}}{[\alpha \mapsto \mathtt{int}] \vdash \mathtt{int} \leqslant \alpha \Rightarrow [\alpha \mapsto \mathtt{int}]}}{\varnothing \vdash \mathtt{int} \to \mathtt{int} \leqslant \alpha \to \alpha \Rightarrow [\alpha \mapsto \mathtt{int}]}$$

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'abbréviations 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 noeuds 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 noeuds, 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, abbréviations 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

$$\textsc{SemInst-Var-Unbound}\ \frac{\alpha \notin dom(\theta)}{\theta \vdash \tau \lhd\ \alpha \Rightarrow \theta \oplus [\alpha \to \tau]} \qquad \textsc{SemInst-Var-Bound}\ \frac{\alpha \in dom(\theta)}{\theta \vdash\ \tau \lhd\ \alpha \Rightarrow \theta}$$

$$\textsc{SemInst-Fun}\ \frac{\theta \vdash \tau_1 \lhd\ \tau_1' \Rightarrow \theta_1 \qquad \theta_1 \vdash \tau_2 \lhd\ \tau_2' \Rightarrow \theta_2}{\theta \vdash \tau_1 \to \tau_2 \lhd\ \tau_1' \to \tau_2' \Rightarrow \theta_2}$$

$$\textsc{SemInst-Construct}\ \theta \vdash \texttt{int} \lhd\ \texttt{int} \Rightarrow \theta \qquad \textsc{SemInst-Wildcard}\ \theta \vdash \_ \lhd\ \_ \Rightarrow \theta$$

FIGURE 3.22 – Instanciation sémantique

version simplifiée de l'instanciation, dite aussi instanciation sémantique donnée en figure 3.22.

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 $\theta_i \vdash \tau \leqslant \sigma \Rightarrow \theta$, alors $\theta_i \vdash \tau \lhd \sigma \Rightarrow \theta$.*

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 3.23 et 3.24. La substitution reste très classique, dans le cas des types il s'agit dont de remplacer chaque variable de type qui apparait dan la substitution par le type auquel elle est associée.

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 \lhd \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\, \mathtt{let} \langle\, x : \sigma \,\rangle = \mathrm{e}_1\ \mathtt{in}\ \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e} ; \theta_\alpha \,]_\sigma & \rightsquigarrow & \langle\, \mathtt{let} \langle\, x : \sigma \,\rangle = \mathrm{e}_1\ \mathtt{in}\ \mathrm{e}_2 : \tau \,\rangle \\
\langle\, \mathtt{let} \langle\, y : \sigma \,\rangle = \mathrm{e}_1\ \mathtt{in}\ \mathrm{e}_2 : \tau \,\rangle\, [\, x \mapsto \mathrm{e} ; \theta_\alpha \,]_\sigma & \rightsquigarrow & \langle\, \mathtt{let} \langle\, y : \sigma \,\rangle = \mathrm{e}_1\, [\, x \mapsto \mathrm{e} ; \theta_\alpha \,]_\sigma\ \mathtt{in} \\
 & & \quad \mathrm{e}_2\, [\, x \mapsto \mathrm{e} ; \theta_\alpha \,]_\sigma : \tau \,\rangle
\end{array}
$$

FIGURE 3.23 – Substitution de variables liée dans une expression annotée

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

FIGURE 3.24 – Substitution de variable de type dans un type

3.5.1 Représentation Coq de la sémantique de MiniML

La sémantique du langage est définie de manière inductive en Coq (figure 3.25). 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 substituées par des variables libres fraiches. Enfin, il faut générer une substitution initiale pour permettre l'instantiation sémantique lors de la substitution.

3.5.2 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 donné en figure 3.27. La fonction `is_value` permet de déterminer quelles expressions sont des

```
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 3.25 – 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 3.26 – Instanciation sémantique implantée en Coq

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 3.20. 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 substituer 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`** est généralisé. 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 la variable locale 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 3.28, pour référence.

```
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 3.27 – 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 3.28 – Implantation de la substitution et de l'instanciation sémantique en OCaml

3.6 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 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.

L'un des buts derrière cette vérification est de dériver un système de types exécutable, en d'autre terme 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 des 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 primaire de filtrage de motifs.

Le chapitre suivant 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 4

Implémentation d'un vérificateur de types pour OCaml

4.1 Préambule

Toute cette machinerie de vérification d'un arbre de syntaxe abstrait annoté peut-être mise en œuvre pour le compilateur OCaml, qui génère après sa phase d'inférence de types une telle structure de données. Le langage n'ayant qu'une implémentation, OCamldésignera les deux dans ce chapitre.

4.2 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 noeud 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 Typedtreeest 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 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 Typedtreecorrespond 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'abbréviations et de déclarations de types d'enregistrement ou de types algébriques. A 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.

4.2.1 Algèbre de types

L'algèbre de types tel qu'il est utilisé par le compilateur est donné en figure 4.1.

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

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

FIGURE 4.1 – Algèbre de types

expliqué section 4.2.5.

4.2.2 Typedtree : expressions

L'ensemble des expressions du langage est donné en figure 4.2.

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

4.2.3 Typedtree : modules, structures et signatures

Le langage est stratifié selon deux niveaux : ML, soit les expressions, et le langage de modules, à priori distinct de celui-ci et permettant d'exprimer l'architecture du programme et ce 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 4.3.

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.

Les modules, dans leur syntaxe et leur sémantique, ressemblent à un lambda calcul classique.

— P, représentant le chemin d'un module dans l'environnement (ou la portée) courant(e).

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

$p ::= \langle\, p' : \tau \,\rangle$	*pattern node*
p' : :=	*pattern description*
x | c | p **as** x	*variable, constant and aliases pattern*
| (p_1, .., p_n)	
| K $(\overline{p})$	
| 'K $(\overline{p})$	*tuple, variant and polymorphic variant constructor pattern*
| { $\overline{l = e;}$ } | [$\overline{p}$]	*record and array pattern*
| (p_1 **|** p_2)	*or pattern*
| **lazy** p	*lazy pattern*
| (p : t)	*coercions and constraints*

FIGURE 4.2 – Syntaxe des expressions

$M ::= \langle\, M' : \mathcal{M} \,\rangle$	*module node*
$M' ::=$	*module description*
$\quad P$	*module path*
$\quad \mid \langle\, \texttt{struct}\ \mathit{Items}\ \texttt{end} : \mathcal{S} \,\rangle$	*structure*
$\quad \mid \texttt{functor}\ (\mathtt{I} : M_{ty}) \to M$	*functor*
$\quad \mid M_1\ (M_1)$	*functor application*
$\quad \mid M : M_{ty}$	*module constraint*
$\quad \mid (\texttt{val}\ e : M_{ty})$	*first-class module unpacking*
$\mathit{Items} ::= I\,;\,;\,\mathit{Items} \mid \Sigma$	
$I ::=$	*structure item*
$\quad \texttt{let}\ \mathtt{i} = e$	*value binding*
$\quad \mid \texttt{type}\ (\alpha_1, .., \alpha_n)\ \mathtt{i} = T$	*type declaration*
$\quad \mid \texttt{type}\ \mathtt{i} \mathrel{+}= K\ \texttt{of}\ t$	*type extension*
$\quad \mid \texttt{external}\ \mathtt{i} : t = \text{“}prim\_name\text{”}$	*stubs import*
$\quad \mid \texttt{module}\ \mathtt{I} = M$	*module binding*
$\quad \mid \texttt{module type}\ \mathtt{I} = M_{ty}$	*module type abbreviation*
$\quad \mid \texttt{open}\ P$	*environment extension*
$\quad \mid \texttt{include}\ M$	*signature reexport*

FIGURE 4.3 – *Syntaxe du système de modules*

— 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*,

— 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 abbréviations : `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.

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

Signatures et types de modules

Tout comme ML, le système de modules et de structures possèdent un système de types. La figure 4.4 est une synthèse de son algèbre.

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

$\mathcal{S} ::= \mathcal{S}_i ;; \mathcal{S} \mid \Sigma$

$\mathcal{S}_i ::=$ *signature item*
- **val** $i : \tau$ *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 4.4 – *Algèbre des modules de types*

Ainsi, l'algèbre des types de modules comprend :

— L'alias de modules est 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'abbréviation 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. 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.

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

$S ::= S_i ;; S \mid \Sigma$

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

FIGURE *4.5 – Syntaxe des types de modules*

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.

En comparant la syntaxe des modules de types et signatures (figure 4.5) 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 abbrévié par `t` par le type `u`, et retourne donc la signature résultante. Le type `t` doit être une abbréviation déclarée dans `M`. La version `M with type t := u` est dite *destructive*, toutes les occurences de `t` sont remplacées et la déclaration de l'abbréviation est retirée de la signature.

4.2.4 Environnements

L'environnement d'OCaml, que l'on nommera Γ, reste relativement simple : il associe à 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'abbréviations ;

— $Class$, soit les déclarations de classes ;

— $Mods$, soit les identifiants associés à des modules ;

— $Modtypes$, 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 associés à un type, et respectivement pour les autres sous-environnement. Par la suite, on introduira un ensemble d'opérations permettant de manipuler cet environnement.

4.2.5 Moteur d'inference 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 4.6

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 $'\_a \to '\_b$.

$$\textsc{Var}\ \frac{\Gamma.Values(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \textbf{fun}\ x \rightarrow e\ : \tau_1 \rightarrow \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

FIGURE 4.6 – Règles d'inférence de ML

$$\Gamma \vdash \textbf{fun}\ x \rightarrow incr\ x\ :\ '\_a \rightarrow\ '\_b\ \textsc{Abs}$$

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

$$\frac{\Gamma, (x,\ '\_a) \vdash incr\ x :\ '\_b}{\Gamma \vdash \textbf{fun}\ x \rightarrow incr\ x\ :\ '\_a \rightarrow\ '\_b}\ \textsc{Abs}$$

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 occurences de $'\_a$ par `int`.

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

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

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

L'unification se fait *en place*, autrement dit la variable de type est physiquement remplacée par 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.

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 puisse être généralisée selon la restriction de valeurs, elle ne doit pas apparaitre dans l'environnement. La solution la plus naive 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 déjà liées, ce qui induit néanmoins d'allouer de la mémoire pour stocker cet ensemble. Le généralisation ainsi que la restriction de valeur relachée sont détaillées en sous-section 4.4.2.

La solution proposée par D.Rémy [35] et implémentée dans OCaml [21] 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. Une fois le corps de `g` typé, toutes les variables généralisables créées au niveau 2 ou plus peuvent être généralisées. Si une variable a été créée à un niveau inférieur, alors elle apparait dans l'environnement et n'aurait pas pu 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 l = ref []  (* niveau 2 *)

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

Selon la restriction de valeurs, `l` 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.

```
l := [ T1 ]
```

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

```
l := [ 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.

4.2.6 Environnements

Bien que l'environnement utilisé pendant l'inférence soit annoté dans chaque noeud du Typedtree, celui-ci 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 incohérent. Il est possible que deux environnements dont l'un est censé être un extension de l'autre lient le même identifiant [1] à un type différent : cela serait considéré comme une incohérence.

Lemme : Bonne formation d'un environnement On considère qu'un environnement est bien formé, si :

— Pour tout identifiant, celui-ci n'est lié qu'une fois ;

— Pour tout identifiant de type, de module ou de classe qui apparaitrait dans une valeur de l'environnement (au sens clé-valeur), alors celui-ci est déjà lié au moment de l'ajout de la dite valeur.

Le premier cas est assez simple à vérifier : durant sa phase de typage, le compilateur effectue une étape d'alpha-conversion. Tout identifiant possède un nombre unique permettant de le différencier des autres identifiants du même nom. Il n'est donc pas possible de lier deux fois le même identifiant *a priori*. La seconde condition découle de la vérification : supposons que l'environnement soit représenté par une liste d'identifiants, dont l'ordre est celui chronologique selon le typage du programme. Supposons que l'expression à vérifier soit $\langle$ `K l : t` $\rangle$, où `K` a le type `u` $\rightarrow$ `t`. Dans ce cas, la définition du type `u` doit avoir lieu avant le type `t`. Si `u` n'est pas présent dans l'environnement alors que `t` si, 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.

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

1. $\wedge$Le Typedtreeest 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

nom et permet donc une recherche rapide dans celui-ci. Il n'existe aucune véritable notion d'ordre chronologique [2] pour l'ajout des noeuds : 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'entièreté de l'environnement, tout en naviguant entre les différents champs de l'environnement 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 comparable structurellement : les identifiants sont accompagnés d'une fonction appelées 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.

4.3 Comparaison et vérification de types

Comme décrit précédemment, 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

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

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 pour des soucis 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'équivalences liées aux GADTS (on parlera aussi d'équations par la suite), que l'on détaillera en sous-section 4.3.5. Leur utilisation pour le moment n'a pas d'incidence sur la compréhension des différents algorithmes de vérification liés aux types.

4.3.1 Equivalence de types

L'équivalence de types (figure 4.7) permet de tester si deux types sont identiques, tout en expansant les abbréviations 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 :

$$\Gamma, \Phi \vdash \ \tau \equiv \ \tau'$$

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 fonctions sont équivalentes 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 des règles Equiv-Construct-Exp-Left et Equiv-Construct-Exp-Right). Le second cas est celui lié aux GADTs (règles Equiv-Rigid1 et Equiv-Rigid2) : 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

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

$$\textsc{Equiv-Tuple}\ \frac{\forall i_{>1}.\ \Gamma, \Phi \vdash \tau_i \equiv \tau'_i}{\Gamma, \Phi \vdash \tau_1 * .. * \tau_n \equiv \tau'_1 * .. * \tau'_n} \qquad \textsc{Equiv-Construct}\ \frac{\forall i_{>1}.\ \Gamma, \Phi \vdash \tau_i \equiv \tau'_i}{\Gamma, \Phi \vdash (\overline{\tau})\ \mathtt{p} \equiv (\overline{\tau'})\ \mathtt{p}}$$

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

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

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

$$\textsc{Equiv-Rigid1}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi \vdash \tau \equiv \mathtt{t}} \qquad \textsc{Equiv-Rigid2}\ \frac{\Phi(\mathtt{t}) = \Phi(\tau)}{\Gamma.Types \oplus (\mathtt{t} : \mathcal{R}), \Phi \vdash \mathtt{t} \equiv \tau}$$

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

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

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

FIGURE 4.7 – Règles d'équivalence de types

calcule une nouvelle signature), puis les deux signatures sont comparées de la même manière que les types.

4.3.2 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 :

$$\Gamma, \Phi, \theta \vdash \tau \leqslant \tau' \Rightarrow \theta'$$

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 function 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 a été remplacée par le type qui lui était associé dans l'environnement.

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 $\theta(\tau') = \tau$.

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 à 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 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 instancié avec le type `int`. Si on vérifie maintenant la variable `x` annotée `int` :

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

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 en 4.5, 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 un 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 était disponible sa quantification, 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.

Finalement, on peut définir l'ensemble des règles du calcul d'instanciation (figure 4.8).

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 la coercion de signature : un module de première classe est une instance d'un autre si on peut coercer cette dernière par son instance.

4.3.3 Vérification de bonne formation/construction

Un jugement de bonne formation est le suivant :

$$\Gamma, \Phi \vdash \tau wf$$

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

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{Inst-Nil}\ \Gamma, \Phi, \theta \vdash \epsilon \leqslant \epsilon \Rightarrow \theta$$

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

FIGURE 4.8 – Règles de vérification d'instanciation

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 que 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, alors le type de modules, qui est un chemin, doit être lié dans l'environnement. De plus, pour chaque type substitué dont la définition n'est pas abstraite, alors le nouveau type doit être une instance de ce dernier.

Les règles de vérification de bonne formation sont données en figure 4.9.

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

4.3.4 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-noeuds de celui-ci. Le jugement de filtrage est le suivant :

$$\Gamma, \Phi \vdash \tau \prec \tau_\rho$$

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

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

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

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

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

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

FIGURE 4.9 – Vérification de bonn formation des types

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-noeuds représentés par des métavariables peuvent être utilisés par la suite.

Un exemple concret, supposons que le jugement suivant apparait dans le contexte d'une règle :

$$\Gamma, \Phi \vdash \tau \prec \tau_1 \to \tau_2$$

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 abbréviations de types. Dans le cas où le type à filtrer serait une variable de type rigides (voir 4.3.5), 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 des classes d'équivalence courant (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

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

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 abbréviation 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'implémentation 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 abbréviation de type, et la fonction est rappelée sur le résultat.

4.3.5 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 noeud, la fonction prend en argument l'accumulateur et le noeud 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 noeuds, 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,

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

$$\equiv$$

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

Contrairement au *fold2* des structures de données d'OCaml, on considère que deux types n'ayant pas la même structure est acceptable et ne doit donc pas être une erreur, puisque c'est à la fonction passée en paramètre de gérer ce cas. Il est par exemple possible de réecrire 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 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 donnée 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 4.3.5. Il s'agit basiquement d'un mécanisme d'*union-find* :

— Au moins l'un des deux types n'apparait pas lié dans Φ, il s'agit donc de le rajouter et le

```
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
```

FIGURE 4.10 – Algorithme de vérification d'équations

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 [3]. 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.

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

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

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

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

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

FIGURE 4.11 – Règles de vérifications du noyau d'OCaml

4.4 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.

4.4.1 ML

On définit les règles nécessaires à la vérification des constructions minimales de ML en figure 4.11.

La vérification d'une constante (CONST) vérifie simplement que la constante appartient bien au type avec laquelle 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 $\overline{|p \to e}$ 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 [4]. 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 en traduite en

```
⟨ 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 4.3.4 :

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

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 pour chacun des types annotés au motif de la 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

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

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 accompagnée des les 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 de valeurs relachée, 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 4.4.2. 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 noeud doit être équivalent à ce type.

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

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

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

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

FIGURE 4.12 – Vérification des motifs

de types abstraits représentants des types existentiels générés par l'inférence (voir section 4.4.3). 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 annoté. 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'OCamlétant 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 occurences 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'équi-

valence 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 de la forme de `p` ou `q`. Cela implique de s'assurer que les valeurs liées par chacun sont les mêmes, en s'assurant bien sûr que le type annoté pour les deux motifs sont équivalents, soit

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

$$\Leftrightarrow$$

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

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 4.4.3 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 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 pour 1 d'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 OCamlrelativement facilement. Par exemple, la transcription de la règle Abs (les fonctions OCamlsont 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 guarde 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 permettre une meilleure propagation des erreurs et la traçabilité de celles-ci. Le résultat retourne ici 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

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

```
  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

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

$$\forall i.\ \Gamma, \Phi \vdash \tau_d \equiv \tau_i$$

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

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

$$\forall i.\ \Gamma_I, \Phi_i \vdash \tau_{c_i} \equiv \texttt{bool}$$

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

Finalement, le corps de chaque branche est vérifié avec le nouvel environnement généré. La fonc-

tion `type_expr` va effectuer les trois opérations restantes suivantes :

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

$$\forall i.\ \Gamma, \Phi \vdash \tau'_i\ wf$$

$$\forall i.\ \Gamma_I, \Phi_i \vdash \tau_{cd} \equiv \tau'_i$$

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

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

4.4.2 Généralisation et restriction de valeur relachée

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 [5].

Ainsi, il n'est pas possible d'écrire une fonction avec le type

$$\forall \alpha.\ \alpha \to \forall \beta.\ \beta \to \alpha * \beta$$

soit une fonction permettant de construire un couple et permettant une application partielle dont le type serait le plus général possible. En effet, une telle fonction dans ML s'écrirait

```
let f = fun x -> fun y -> x, y
```

et son type inféré

$$\forall \alpha\beta.\ \alpha \to \beta \to \alpha * \beta$$

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

Ce choix de quantification prénexe est nécessaire pour la décidabilité de l'inférence de types [6].

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 instantiées 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. Selon la restriction de valeur "classique", on considère que seules les expressions étant syntaxiquement des valeurs comme pouvant 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 fraiche 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 application partielle, 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)
```

6. ∧L'inférence de quantificateurs de rangs 2 est décidable, mais OCaml garde tout de même cette quantification prénexe.

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,

$$\Gamma, \Phi, \varnothing \vdash (\texttt{int} \to \texttt{int})\ \texttt{ref} \leqslant (\,'a \to\ 'a)\ \texttt{ref} \Rightarrow ['a \mapsto \texttt{int}]$$

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 n'est pas une valeur, et donc 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 4.13) [7], alors le type de 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 OCamlcomme étant un type mal formé.

OCaml utilise une version moins stricte : la restriction de valeur relachée. 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 $(\alpha \to \texttt{int}) \to \texttt{int}$, 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'autre terme, une variable qui n'apparaît pas à gauche d'une flêche peut être généralisée, que la valeur soit expansive ou non.

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

$$
\begin{aligned}
&\mathit{nonexp}\ (\texttt{x}) \ ::= \ \top \\
&\mathit{nonexp}\ (c) \ ::= \ \top \\
&\mathit{nonexp}\ (\texttt{let x = } e_1 \texttt{ in } e_2) \ ::= \\
&\qquad \mathit{nonexp}\ (e_1) \wedge \mathit{nonexp}\ (e_2) \\
&\mathit{nonexp}\ (\texttt{fun x -> e}) \ ::= \ \top \\
&\mathit{nonexp}\ (\texttt{match e with } \overline{\texttt{| p when g -> b}}) \ ::= \\
&\qquad \mathit{nonexp}\ (e) \wedge \bigwedge_{i=1}^{n} (\mathit{nonexp}\ (g_i) \wedge \mathit{nonexp}\ (b_i)) \\
&\mathit{nonexp}\ (e_1\texttt{, } e_2) \ ::= \ \mathit{nonexp}\ (e_1) \wedge \mathit{nonexp}\ (e_2) \\
&\mathit{nonexp}\ (K) \ ::= \ \top \\
&\mathit{nonexp}\ (K\ e) \ ::= \ \mathit{nonexp}\ (e) \\
&\mathit{nonexp}\ (\texttt{`}K) \ ::= \ \top \\
&\mathit{nonexp}\ (\texttt{`}K\ e) \ ::= \ \mathit{nonexp}\ (e) \\
&\mathit{nonexp}\ (\texttt{\{ } \overline{\texttt{l = e;}} \texttt{ \}}) \ ::= \ \bigwedge_{i=1}^{n} (\mathit{immutable}\ (l_i) \wedge \mathit{nonexp}\ (e_i)) \\
&\mathit{nonexp}\ (\texttt{e.l}) \ ::= \ \mathit{nonexp}\ (e) \\
&\mathit{nonexp}\ (\texttt{if c then } e_1 \texttt{ else } e_2) \ ::= \\
&\qquad \mathit{nonexp}\ (c) \wedge \mathit{nonexp}\ (e_1) \wedge \mathit{nonexp}\ (e_2) \\
&\mathit{nonexp}\ (e_1\texttt{; } e_2) \ ::= \ \mathit{nonexp}\ (e_2) \\
&\mathit{nonexp}\ (\texttt{lazy e}) \ ::= \ \mathit{nonexp}\ (e) \\
&\mathit{nonexp}\ (\texttt{let module I = M in e}) \ ::= \\
&\qquad \mathit{nonexp\_mod}\ (\texttt{M}) \wedge \mathit{nonexp}\ (e) \\
&\mathit{nonexp}\ (\texttt{(module M)}) \ ::= \ \mathit{nonexp\_mod}\ (\texttt{M}) \\
&\mathit{nonexp}\ (\_) \ ::= \ \bot
\end{aligned}
$$

FIGURE *4.13 – Valeurs syntaxiques, ou non expansives*

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 4.4.4.

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 en 4.2.5. 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 4.14 est relativement simple. Il s'agit de considérer donc 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 à priori des variables quantifiées par ce type. Le parcours du type se fait à l'aide du combinateur *fold* décrit en 4.3.5. 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.* qu'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

```
let isvar_nonexpans (Γ, Φ) 𝒯 τ =
  if Γ,Φ ⊢ '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 ⊢ 'a ≺ τ then
    if τ ∈ Γ.𝒱ars ∨ (τ ∈ 𝒯 ∧ pos ≠ Contravariant) then
      𝒯
    else
      𝒯 ⊕ pos
  else if Γ,Φ ⊢ 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 ⊢ τarg t ≺ τ then
    list_fold_indexed_left
      (fun 𝒯 index τarg →
        let pos =
          if get_variance (Γ, Φ) index t = Contravariant
          then Contravariant
          else pos in
        aggregate_expans (Γ, Φ) 𝒯 pos τarg)
      𝒯
      τarg
  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 4.14 – Algorithme de vérification de généralisation

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-noeud 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 les variables 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.

4.4.3 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 invariants sur certaines valeurs qui seront vérifiés par le système de types. On peut alors reprendre le type donné en 4.3.5, 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 malgré qu'il n'y ait qu'une seule branche, le typeur d'OCaml considère que le filtrage est complet : en effet, pour toutes les valeurs possible de `int t`, seul le constructeur `Int` est possible. 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 et 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` a 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 le 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 équivalent : `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 déclaré. Il est alors possible de représenter des types existentiels :

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

Ainsi, une fois *empaquetée* dans un constructeur `Ex`, le type d'une valeur est perdu. 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 donner un type à 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 utilisé 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 précédement, 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 : 'a 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, `'a list t` ne peut être une application correcte du constructeur, et doit être rejetée. Ensuite, comme expliqué en 4.4.3, certaines variables peuvent être quantifiées existentiellement. Dans la structure de donnée du compilateur qui représente les déclarations de constructeurs de données, les variables existentielles sont explicitement indiquées. Il faut donc pouvoir vérifier celles-ci, 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 4.4.3

La génération et vérification des équations est décrite en 4.3.5.

L'algorithme d'extraction des constructeurs dits *existentiels* durant le filtrage est donné en figure 4.4.3. 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 de 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 auquel il est instancié : si la variable a été considérée comme étant existentielle, alors son instance est un constructeur de type généré pour la représenter.

```
let find_universals τ_ret =
  let { τ_1, .., τ_n } = τ_ret ≺ (τ_1, .., τ_n) t in
  { τ | τ ∈ { τ_1, .., τ_n } ⋀ τ ≺ 'a }


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 4.15 – Algorithme d'extraction des types existentiels

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

FIGURE 4.16 – Vérification des motifs de constructeurs

Enfin, la règle de vérification des motifs de constructeur est donnée dans la figure 4.16. 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 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.

4.4.4 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 4.17) se fait de manière directe. Il s'agit de vérifier chacune

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

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

FIGURE 4.17 – Vérification des tuples

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

$$\textsc{Match}\ \frac{\begin{array}{c}\Gamma,\Phi \vdash \langle e : \tau_{scrut} \rangle \qquad \forall i.\ \Gamma,\Phi,\varnothing \vdash \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}),(\overline{\tau_{\exists_i}}),\Phi_i \\ \forall i.\ \Gamma,\Phi \vdash \tau_{scrut} \equiv \tau_i \qquad \forall i.\ let\ \Gamma_i = \Gamma \oplus_\mathcal{V} (\overline{v_i : \tau_{v_i}}) \oplus_\mathcal{T} (\overline{\tau_{\exists_i}}) \\ \forall i.\ \Gamma_i,\Phi_i \vdash \langle e_i : \tau_{e_i} \rangle \qquad \forall i.\ \Gamma,\Phi \vdash \tau_{e_i}\ wf \qquad \forall i.\ \Gamma_i,\Phi_i \vdash \tau \equiv \tau_{e_i}\end{array}}{\Gamma,\Phi \vdash \langle \texttt{match}\ e\ \texttt{with}\ \overline{|\ p \rightarrow e} : \tau \rangle}$$

FIGURE 4.18 – Vérification des constructeurs de données et du filtrage

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.

La vérification du filtrage (figure 4.18) 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, il faut d'abord s'assurer que chacune de ses sous-expression 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

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

$$\textsc{Field}\ \frac{\begin{array}{ccccc} & & \Gamma, \Phi \vdash \langle e : \tau_e \rangle & let\ \tau_{rec}, \mathcal{L} = find\_record(\Gamma, \Phi, \tau_e) & \\ l \in dom(\mathcal{L}) & let\ \tau_l = \mathcal{L}(l) & \Gamma, \Phi, \Sigma \vdash \tau \leqslant \tau_l \Rightarrow \theta & \Gamma, \Phi, \theta_\forall(\Gamma) \vdash \tau_e \leqslant \tau_{rec} & \end{array}}{\Gamma, \Phi \vdash \langle e.l : \tau \rangle}$$

FIGURE 4.19 – Vérification des enregistrements

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 vu comme une version polymorphe de ceux-ci. Ainsi, la vérification de la création d'un enregistrement (figure 4.19) 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 4.20). 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.* que 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 $\texttt{l} + \mathcal{P}' = \mathcal{P}$. Si le champs n'apparaît

```
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 4.20 – Vérification de la présence et de l'unicité de l'ensemble des champs d'un enregistrement

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 correctement instancié.

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 4.21. Si celle-ci est similaire à RECORD (figure 4.19), la principale différence réside dans la vérification des

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

FIGURE 4.21 – Vérification des copies d'enregistrements

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

FIGURE 4.22 – Vérification de l'assignation d'un champ d'enregistrement

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 instantiés. Ensuite, pour chacun des champs redéfinis, le type de sa valeur est vérifiée, et une instance entre la définition et le type annoté est calculé. Finalement, on vérifie que chacun des champs hérité 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.

La vérification de l'assignation d'un champ (figure 4.22) est similaire à celle décrite pour l'accès. La principale différence est de s'assurer que le noeud 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`.

Finalement, il est possible de filtrer sur des enregistrements (PAT-RECORD, figure 4.23). La

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

FIGURE 4.23 – Vérification de motifs d'enregistrement

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

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

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

FIGURE 4.24 – Vérification des instructions impératives

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 champs filtré celui-ci appartient bien à l'enregistrement et n'est pas déclaré plus d'une fois. L'algorithme de `check_atmost` vérifie cette propriété et est donné figure 4.20.

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 des boucles conditionnelles.

Ces trois règles en figure 4.24 sont relativement simples et supposent simplement des équi-

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

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

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

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

FIGURE 4.25 – Vérification des constructions liées aux exceptions

valences de types. Dans le cas de la séquence, le premier noeud doit être annoté avec un type équivalent à `unit` [8] et le second à celui de la séquence. Le type de l'expression **`while`** doit toujours être équivalent à `unit`, tout comme le type annoté de son corps. La condition quand à elle doit être booléenne, et donc annoté 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 vérifica-

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

tion 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

```
(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 d'abord substitués dans le corps de celle-ci. En revanche, il existe des cas où il peut être intéressant de n'évaluer l'argument que lorsque celui-ci doit être utilisé : il s'agit de la stratgie dite d'*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, 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 [9], 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. La construction `lazy e` permet donc d'exprimer une valeur `e` dont le résultat sera calculé explicitement la

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

première fois qu'elle sera accédée via `Lazy.force` ou à l'aide du filtrage, grâce au motif

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

On pourrait alors définir un type OCaml similaire au fonctionnement du type `'a lazy_t` [10] :

```
type 'a thunk = Res of 'a | Thunk of (unit -> 'a)
type 'a t = {
  mutable thunk: 'a thunk;
}

let force (l: 'a t) : 'a =
  match l.thunk with
    Thunk f -> let res = f () in l.thunk <- Res res; res
  | Res res -> res
```

Ainsi, `lazy e` peut être un sucre syntaxique de

```
{ thunk = Thunk (fun () -> e) }
```

En revanche, cette représentation ne sera pas strictement identique au type `'a lazy_t`, car on considère que la création d'un enregistrement dont au moins l'un des champs est mutable comme étant un effet de bord observable, et cela rend certaines valeurs l'utilisant comme potentiellement non généralisables. Prenons l'exemple suivant :

```
let ⟨ f : '_a → '_a ⟩ =
  let _ = { thunk = Thunk (fun () -> 0) } in
  (fun x -> x)

let ⟨ g : 'a → 'a ⟩ =
  let _ = lazy 0 in
  (fun x -> x)
```

10. ∧La représentation interne d'une valeur paresseuse est similaire : le bloc est un étiqueté avec une valeur indiquant s'il s'agit d'un bloc lazy dont la valeur a été calculée ou non, le bloc lui-même contenant un *thunk* ou directement la valeur si elle a été calculée. Cette version est plus compacte que la représentation proposée, mais n'est pas typable dans OCamlpuisqu'elle induit de modifier la représentation d'un bloc déjà typé.

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

FIGURE 4.26 – Vérification des expressions paresseuses

Bien que la sémantique de notre type `t` et `lazy_t` soit la même, le type inféré ici n'est pas le même, puisque le champ `thunk` est mutable, donc l'expression est elle-même considérée expansive, et donc la variable `'a` qui apparaît à une position contravariante ne peut être généralisée. En revanche, dans le second cas, pour que `lazy ()` soit considéré comme étant expansif il faudrait que `()` le soit, ce qui n'est bien évidemment pas le cas, alors l'expression n'est pas expansive et on obtient bien une fonction d'identité.

La vérification d'une expression `lazy_t` est donc relativement simple : il s'agit de s'assurer que l'expression à calculée est correcte, que le type du noeud 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 4.26.

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
```

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 décrit par la metavariable 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 metavariable de rangée est $>$ s'écrira $[> \mathcal{T}]$, puisque par définition tous les constructeurs $\mathcal{K}$ sont forcément présent 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 metavariable 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-même, 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 à postériori de rajouter de nouveaux constructeurs à cette liste.

La vérification d'un constructeur de variant polymorphe est alors simple (voir figure 4.27). D'abord, le type τ du noeud doit être un variant. le constructeur doit apparaître dans l'ensemble des constructeurs possible, 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

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

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

FIGURE 4.27 – Vérification des constructeurs de variants polymorphes

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 ] ⟩ = ..
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.

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

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

FIGURE 4.28 – Vérification des motifs de variants polymorphes

4.5 Sémantique opérationnelle

Le Typedtreen'est pas prévu pour être exécuté, il peut néanmoins être intéressant d'en définir une sémantique opérationnelle. Celle-ci suit le principe énoncé en 3.5. 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 significatives. Chacune des constructions ayant une sémantique distincte et orthogonale des autres, la description de celle-ci est décomposée en différentes catégories : noyau, mutabilité, extensions fonctionnelles, extensions impératives, etc. Une description complète comprenant chacun des ajouts nécessaire de cette sémantique sera donnée en annexe.

4.5.1 Sémantique du noyau ML d'OCaml

La sémantique du noyau est décrite en figure 4.29. La principale différence avec celle donnée pour MiniML annoté est l'utilisation du filtrage de motifs. Dans le cas de l'application, si les deux membres dont 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 continue à tester les cas suivants.

$v ::=$ *valeurs*

$\langle\, x : \tau \,\rangle$ *variable*

$\mid \langle\, \mathrm{c} : \tau \,\rangle$ *constante*

$\mid \langle\, \texttt{function} \mid \overline{p \to e} : \tau \,\rangle$ *abstraction*

Réduction des expressions annotées

$$
\begin{array}{lcl}
\mathrm{e}_1\ \mathrm{e}_2 & \rightsquigarrow & v_1\ \mathrm{e}_2 \\
v_1\ \mathrm{e}_2 & \rightsquigarrow & v_1\ \mathrm{v}_2 \\
\langle\, \texttt{function} \mid p \to e \mid \overline{p' \to e' : \tau} \,\rangle\ v_2 & \rightsquigarrow & si\ \overline{(x, v)} = \mathrm{match}\ (p, v_2) : e[x_1 \mapsto \mathrm{e}_1, .., x_n \mapsto \mathrm{e}_n] \\
\langle\, \texttt{function} \mid p \to e \mid \overline{p' \to e' : \tau} \,\rangle\ v_2 & \rightsquigarrow & si\ \neg\ \overline{(x, v)} = \mathrm{match}\ (p, v_2) : \langle\, \texttt{function} \mid \overline{p' \to e' : \tau} \,\rangle\ v_2 \\
\texttt{let}\ \overline{p = e}\ \texttt{in}\ \mathrm{e} & \rightsquigarrow & \texttt{let}\ \overline{p = v}\ \texttt{in}\ \mathrm{e} \\
\texttt{let}\ \overline{p = v}\ \texttt{in}\ \mathrm{e} & \rightsquigarrow & \forall i.\overline{(x_i, v_i)} = \mathrm{match}(p_i, v_i)\,; \\
 & & \mathrm{e}\ [x_{11} \mapsto v_{11}, .., x_{nk} \mapsto v_{nk}]_{\triangleleft}
\end{array}
$$

Filtrage de motifs

$$
\begin{array}{lcll}
\mathrm{match}\ (x, \mathrm{e}) & = & (\mathrm{x}, \mathrm{e}) & \\
\mathrm{match}\ (\_, \mathrm{e}) & = & \varnothing & \\
\mathrm{match}\ (\mathrm{c}, \mathrm{c}) & = & \varnothing & \\
\mathrm{match}\ (\mathrm{p}_1 \mid \mathrm{p}_2, \mathrm{e}) & = & -\ \mathrm{match}\ (\mathrm{p}_1, \mathrm{e}) & \\
 & & -\ \mathrm{match}\ (\mathrm{p}_2, \mathrm{e}) & si\ \neg\ match\ (p_1, e) \\
\mathrm{match}\ (\_, \_) & = & \bot &
\end{array}
$$

FIGURE 4.29 – Sémantique opérationnelle du noyau d'OCaml

$$
\begin{array}{lll}
v ::= & & \textit{valeurs} \\
\quad .. & & \\
\quad \mid \langle\, (\mathrm{v}_1, .., \mathrm{v}_n) : \tau \,\rangle & & \textit{n-uplets} \\
\quad \mid \langle\, \mathrm{K}\ (\mathrm{v}_1, .., \mathrm{v}_n) : \tau \,\rangle & & \textit{n-uplets}
\end{array}
$$

Réduction des expressions annotées

$$
\begin{array}{lcl}
(\mathrm{e}_1, .. \mathrm{e}_n) & \leadsto & (v_1, .., \mathrm{e}_n) \\
(v_1, .., \mathrm{e}_i, .., \mathrm{e}_n) & \leadsto & (v_1, .. \mathrm{v}_i, .. \mathrm{e}_n) \\
\mathrm{K}\ (\mathrm{e}_1, .. \mathrm{e}_n) & \leadsto & \mathrm{K}\ (v_1, .., \mathrm{e}_n) \\
\mathrm{K}\ (v_1, .., \mathrm{e}_i, .., \mathrm{e}_n) & \leadsto & \mathrm{K}\ (v_1, .. \mathrm{v}_i, .. \mathrm{e}_n) \\
\langle\, \texttt{match}\ \mathrm{e}\ \texttt{with}\ \overline{\mid p \to e : \tau} \,\rangle & \leadsto & \langle\, \texttt{match}\ \mathrm{v}\ \texttt{with}\ \overline{\mid p \to e : \tau} \,\rangle \\
\langle\, \texttt{match}\ v\ \texttt{with}\ p \to e \ \overline{\mid p' \to e' : \tau} \,\rangle & \leadsto & \text{si } \overline{(x, v)} = \text{match } (p, v) : \\
 & & e[x_1 \mapsto \mathrm{e}_1, .., x_n \mapsto \mathrm{e}_n] \\
\langle\, \texttt{match}\ v\ \texttt{with}\ p \to e \ \overline{\mid p' \to e' : \tau} \,\rangle & \leadsto & \text{si } \neg\, \overline{(x, v)} = \text{match } (p, v) : \\
 & & \langle\, \texttt{match}\ v\ \texttt{with}\ \overline{\mid p' \to e' : \tau} \,\rangle
\end{array}
$$

Filtrage de motifs

$$
\begin{array}{lcl}
\text{match } ((\mathrm{p}_1, .., \mathrm{p}_n), (\mathrm{e}_1, .., \mathrm{e}_n)) & = & \text{match } (\mathrm{p}_1, \mathrm{e}_1) \cup .. \cup \text{match } (\mathrm{p}_n, \mathrm{e}_n) \\
\text{match } (\mathrm{K}\ (\mathrm{p}_1, .., \mathrm{p}_n), \mathrm{K}\ (\mathrm{e}_1, .., \mathrm{e}_n)) & = & \text{match } (\mathrm{p}_1, \mathrm{e}_1) \cup .. \cup \text{match } (\mathrm{p}_n, \mathrm{e}_n)
\end{array}
$$

FIGURE 4.30 – Sémantique opérationnelle du filtrage et des constructeurs

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 un 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.

4.5.2 Filtrage, n-uplets et constructeurs de données

La sémantique opérationnelle des n-uplets, constructeurs et du filtrage de motifs est décrite en figure 4.30. Un n-uplet (ou un constructeur) est un valeur si ses composantes sont elles-même des

valeurs. L'expression `match` n'est pas une valeur, puisqu'elle peut se réduire. De la même manière que `function` (figure 4.29), sa réduction considère d'abord l'expression filtrée qui doit être une valeur, puis teste une à une les branches et se réduit en la première dans l'ordre de déclaration qui correspond à la valeur filtrée.

4.5.3 Enregistrements

La réduction des enregistrements est donnée en figure 4.31. Une première remarque importante est l'ajout d'une nouvelle construction pour les expressions : m, qui correspond à un emplacement réservé en mémoire. Celle-ci est donc le résultat de la réduction d'un enregistrement et permet donc l'introduction de la mutabilité. A cela s'ajoute un nouvel environnement $\mathcal{M}$ sous la forme d'une liste de triplets. Chacun de ceux-ci contient :

— 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 champs dans le Typedtreeest 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 champs 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 associé, et se réduit en la valeur `()`.

e' : := *expressions*

..

| m *emplacement mémoire*

v : := *valeurs*

..

| $\langle\, m : \tau \,\rangle$ *emplacement mémoire*

Mémoire

$\mathcal{M}$: := $\varnothing$

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

— dom ($\mathcal{M}$) : ensemble des emplacements mémoire alloués
— $\mathcal{M}_\tau(m)$: type associé à l'emplacement mémoire
— $\mathcal{M}\,(m, \mathrm{l})$: 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.

Réduction des expressions annotées

$$
\begin{array}{lcl}
\{\mathrm{l}_1 = \mathrm{e}_1 ; .. ; \mathrm{l}_n = \mathrm{e}_n\} \mid \mathcal{M} & \rightsquigarrow & \{\mathrm{l}_1 = v_1 ; .. ; \mathrm{l}_n = \mathrm{e}_n\} \mid \mathcal{M}_1 \\
\{\mathrm{l}_1 = v_1, .. ; \mathrm{l}_i = \mathrm{e}_i ; .. ; \mathrm{l}_n = \mathrm{e}_n\} \mid \mathcal{M} & \rightsquigarrow & \{\mathrm{l}_1 = v_1, .. ; \mathrm{l}_i = \mathrm{v}_i ; .. ; \mathrm{l}_n = \mathrm{e}_n\} \mid \mathcal{M}_i \\
\langle\, \{\mathrm{l}_1 = v_1 ; .. ; \mathrm{l}_n = v_n\} : \tau \,\rangle \mid \mathcal{M} & \rightsquigarrow & \langle\, m : \tau \,\rangle \mid \mathcal{M}, (m : \tau = [v_1 ; .. ; v_n]) ; \\
 & & \text{t.q. } m \notin dom(\mathcal{M}) \\
\mathrm{e.l} \mid \mathcal{M} & \rightsquigarrow & v.\mathrm{l} \mid \mathcal{M}' \\
\langle\, \mathrm{m} : \tau \,\rangle.\mathrm{l} \mid \mathcal{M} & \rightsquigarrow & \mathcal{M}\,(m, \mathrm{l}) \mid \mathcal{M} \\
\mathrm{e}_1.\mathrm{l} \leftarrow \mathrm{e}_2 \mid \mathcal{M} & \rightsquigarrow & \mathrm{e}_1.\mathrm{l} \leftarrow v_2 \mid \mathcal{M}' \\
\mathrm{e}_1.\mathrm{l} \leftarrow v_2 \mid \mathcal{M} & \rightsquigarrow & v_1.\mathrm{l} \leftarrow v_2 \mid \mathcal{M}' \\
\langle\, \mathrm{m} : \tau \,\rangle.\mathrm{l} \leftarrow v_2 \mid \mathcal{M} & \rightsquigarrow & () \mid \mathcal{M}\,(m, \mathrm{l}) \leftarrow v_2
\end{array}
$$

Filtrage de motifs

$$
\text{match } (\{\mathrm{l}_i = \mathrm{p}_i ; .. ; l_j = \mathrm{l}_j\}), \langle\, m : \tau \,\rangle) \quad = \quad \bigcup_{k \in [i..j]} \text{match } (\mathrm{p}_k, \mathcal{M}\,(m, \mathrm{l}_k))
$$

FIGURE 4.31 – Sémantique opérationnelle des enregistrements

$$\textsc{Location}\ \frac{\Gamma, \Phi \vdash \tau \equiv \mathcal{M}_\tau(m)}{\Gamma, \mathcal{M}, \Phi \vdash \langle m : \tau \rangle}$$

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

FIGURE 4.32 – Règles de vérification des emplacements mémoires

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 4.32). 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. De base, 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 où là où intervient la vérification d'une expression.

4.5.4 Récursion

La récursion nécessite l'ajout d'un combinateur de point fixe **`fix`**[32]. La sémantique de celui-ci est décrite dans la figure 4.33. Ce combinateur prend donc une fonction en argument et retourne une nouvelle fonction 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 équvalent, et le type du noeud **`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`
Néanmoins, OCamlpermettant de définir des fonctions mutuellement récursives, la transformation doit être adaptée. Dans un cas avec deux fonctions récursives :

```
let rec f x = e1 and g y = e2 in f ()
```

e' : := *expressions*
..
| **fix** e *combinateur de point fixe*

Vérification

$$\textsc{Fix}\ \frac{\Gamma,\Phi \vdash \langle e : \tau_e \rangle \qquad \Gamma,\Phi \vdash \tau_e \prec \tau_d \to \tau_{cd} \qquad \Gamma,\Phi \vdash \tau_e \equiv \tau_d \qquad \Gamma,\Phi \vdash \tau_e \equiv \tau_{cd} \qquad \Gamma,\Phi \vdash \tau_{cd} \equiv \tau}{\Gamma,\Phi \vdash \langle \textbf{fix}\ e : \tau \rangle}$$

Expansion de **let rec**

```
let rec ⟨ f1 : τ1 ⟩ = ⟨ e1 : τ1 ⟩
..
and ⟨ fn : τn ⟩ = ⟨ en : τn ⟩
```

⇝
(Sachant $\tau_{fs} = \tau_1 * .. * \tau_n$)

```
let ⟨ (f1, .., fn) : τfs ⟩ =
  fix (function ⟨ fs : τfs ⟩ →
    ( e1 [ ⟨ f1 : τf1 ⟩ ↦
                let < (f1, _, ..) : τfs ⟩ = < fs : τfs ⟩
                in ⟨ f1 : τf1 ⟩; ..

                ⟨ fn : τfn ⟩ ↦
                let < (_, .., fn) : τfs ⟩ = < fs : τfs ⟩
                in ⟨ fn : τfn ⟩ ],
      ..,

      en [ ⟨ f1 : τf1 ⟩ ↦
                let < (f1, _, ..) : τfs ⟩ = < fs : τfs ⟩
                in ⟨ f1 : τf1 ⟩; ..

                ⟨ fn : τfn ⟩ ↦
                let < (_, .., fn) : τfs ⟩ = < fs : τfs ⟩
                in ⟨ fn : τfn ⟩ ]))
```

Réduction des expressions annotées

fix e	⇝	**fix** *v*
fix (**function** $\langle x : \tau_x \rangle \to$ e)	⇝	e [x ↦ **fix** (**function** $\langle x : \tau_x \rangle \to$ e)]

FIGURE 4.33 – Sémantique opérationnelle de la récursion mutuelle

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 récusifs seront remplacés respectivement par `fst f` et `snd g`, soit : e'_1 = `e1 [ f ↦ (fst fg); g ↦ (snd fg) ]` et e'_2 = `e2 [ f ↦ (fst fg); g ↦ (snd f`
On note cependant qu'il n'est pas possible d'écrire `let rec (f, g) = ..`, 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 des valeurs (et n'étant plus une valeur récursive), on peut récupérer les noms de fonctions pour la suite du programme :

```
let (f, g) = fix (fun fg -> (fun x -> e1', fun y -> e2')) in f ()
```

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 : 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 significative) :

```
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 *)
↝
```

$$
\begin{array}{lcl}
\mathrm{e}_1 \,;\, \mathrm{e}_2 & \rightsquigarrow & v_1 \,;\, \mathrm{e}_2 \\
v_1 \,;\, \mathrm{e}_2 & \rightsquigarrow & \mathrm{e}_2 \\
\texttt{while}\ \mathrm{e}_1\ \texttt{do}\ \mathrm{e}_2\ \texttt{done} & \rightsquigarrow & \begin{array}{l} \mathrm{e}_1 \rightsquigarrow \text{true} : \\ \qquad \mathrm{e}_2 \,;\, \texttt{while}\ \mathrm{e}_1\ \texttt{do}\ \mathrm{e}_2\ \texttt{done} \\ \mathrm{e}_1 \rightsquigarrow \text{false} : \\ \qquad \texttt{()} \end{array} \\
\texttt{for}\ i = \mathrm{e}_i\ \texttt{to}\ \mathrm{e}_j\ \texttt{do}\ \mathrm{e}\ \texttt{done} & \rightsquigarrow & \texttt{for}\ i = v_i\ \texttt{to}\ \mathrm{e}_j\ \texttt{do}\ \mathrm{e}\ \texttt{done} \\
\texttt{for}\ i = v_i\ \texttt{to}\ \mathrm{e}_j\ \texttt{do}\ \mathrm{e}\ \texttt{done} & \rightsquigarrow & \texttt{for}\ i = v_i\ \texttt{to}\ v_j\ \texttt{do}\ \mathrm{e}\ \texttt{done} \\
\texttt{for}\ i = v_i\ \texttt{to}\ v_j\ \texttt{do}\ \mathrm{e}\ \texttt{done} & \rightsquigarrow & \begin{array}{l} v_i \leqslant v_j : \\ \qquad \mathrm{e}\,[\, i \mapsto v_i \,]\,;\, \texttt{for}\ i = v_i + 1\ \texttt{to}\ v_j\ \texttt{do}\ \mathrm{e}\ \texttt{done} \\ v_i > v_j : \\ \qquad \texttt{()} \end{array}
\end{array}
$$

FIGURE 4.34 – Sémantique opérationnelle des constructions impératives

. .

L'évaluation d'OCamlétant stricte, 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 opérationelle du Typedtree.

4.5.5 Extensions impératives

La figure 4.34 décrit la sémantique des construtions 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 pour la séquence dans le cas où la première expression est une valeur :

$$\texttt{()}\,;e_2 \qquad \rightsquigarrow \qquad e_2$$

Celle-ci oblige alors la première valeur à se réduire forcément en `()` (unit), néanmoins OCaml ne force pas cette condition dans son comportement normal [11]. Les boucles se réduisent de la manière suivante :

1. Dans le cas de **`for`**, la condition de départ et d'arrêt sont réduites.
2. Si la condition pour accéder au corps de la boucle est correct, on réduit celle-ci à une séquence qui correspond à son corps, puis la boucle à nouveau. Dans le cas de **`for`**, la condition initiale est incrémentée.
3. Si la condition n'est pas vérifiée, celle-ci se réduit en `()`.

4.5.6 Exceptions

La figure 4.35 présente les règles de réductions propres aux exceptions, ainsi que les règles additionnelles nécessaires dans le noyau pour la propagation de celles-ci jusqu'au gestionnaire (**`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 **`try .. with`**, il est d'abord nécessaire de réduire le corps de celui-ci. S'il s'agit d'une valeur régulière, l'instruction se réduit en celle-ci. En revanche, si son corps est une exception, l'instruction se réduit en un filtrage de l'exception avec un ensemble de motifs correspondant aux exceptions gérées par le gestionnaire courant. A ceux-ci est ajouté un motif joker, dont le but est simplement de propager l'exception si le filtrage a échoué.

La propagation des exceptions nécessite d'ajouter de nouvelles règles pour toutes les constructions où il y a une réduction. Un exemple du noyau est donné en figure 4.35 : les nouvelles règles sont celles précédées par le symbole •. Dans le cas de l'application, si la fonction ou l'argument se réduit en une levée d'exception, alors le résultat de cette application est la propagation de cette exception. Pour celui des expressions locales, si l'une de celles-ci se réduit en une exception, la réduction est également sa propagation. L'ajout des exceptions dans la sémantique est mécanique, et se fait de la même manière pour le reste du langage.

11. ∧Le système de types présenté ici force néanmoins cette condition.

Réduction des expressions annotées

$$
\begin{array}{lcl}
\texttt{raise}\ \mathrm{e} & \rightsquigarrow & \texttt{raise}\ v \\
\texttt{raise}\ (\texttt{raise}\ v) & \rightsquigarrow & \texttt{raise}\ v \\
\texttt{try}\ \mathrm{e}\ \texttt{with}\ \overline{p \rightarrow e} & \rightsquigarrow & \texttt{try}\ \mathrm{e}\ \texttt{with}\ \overline{p \rightarrow e} \\
\texttt{try}\ v\ \texttt{with}\ \overline{p \rightarrow e} & \rightsquigarrow & v \\
\texttt{try}\ \langle\ \texttt{raise}\ v : \tau\ \rangle\ \texttt{with}\ \overline{p \rightarrow e} & \rightsquigarrow & \begin{array}{l}\texttt{match}\ v\ \texttt{with} \\ \mid\ \overline{p \rightarrow e} \\ \mid\ \langle\ \_ : \texttt{exn}\ \rangle \rightarrow \langle\ \texttt{raise}\ v : \tau\ \rangle\end{array}
\end{array}
$$

Sémantique du noyau en présence d'exceptions

$$
\begin{array}{lcl}
\mathrm{e}_1\ \mathrm{e}_2 & \rightsquigarrow & v_1\ \mathrm{e}_2 \\
v_1\ \mathrm{e}_2 & \rightsquigarrow & v_1\ \mathrm{v}_2 \\
\bullet\ (\texttt{raise}\ v_1)\ \mathrm{e}_2 & \rightsquigarrow & \texttt{raise}\ v_1 \\
\bullet\ v_1\ (\texttt{raise}\ v_2) & \rightsquigarrow & \texttt{raise}\ v_2 \\
\langle\ \texttt{function} \mid \overline{p \rightarrow e \mid p' \rightarrow e' : \tau}\ \rangle\ v_2 & \rightsquigarrow & si\ \overline{(x, v)} = \mathrm{match}\ (p, v_2) : e[x_1 \mapsto \mathrm{e}_1, .., x_n \mapsto \mathrm{e}_n] \\
\langle\ \texttt{function} \mid \overline{p \rightarrow e \mid p' \rightarrow e' : \tau}\ \rangle\ v_2 & \rightsquigarrow & si\ \neg\ \overline{(x, v)} = \mathrm{match}\ (p, v_2) : \langle\ \texttt{function} \mid \overline{p' \rightarrow e' : \tau}\ \rangle\ v_2 \\
\texttt{let}\ \overline{p = e}\ \texttt{in}\ \mathrm{e} & \rightsquigarrow & \texttt{let}\ \overline{p = v}\ \texttt{in}\ \mathrm{e} \\
\texttt{let}\ \overline{p = v}\ \texttt{in}\ \mathrm{e} & \rightsquigarrow & \forall i.\overline{(x_i, v_i)} = \mathrm{match}(p_i, v_i)\,; \mathrm{e}\ [x_{11} \mapsto v_{11}, .., x_{nk} \mapsto v_{nk}]_{\triangleleft} \\
\texttt{let}\ \mathrm{p}_1 = v_1 \quad .. & & \\
\bullet\ \texttt{and}\ \mathrm{p}_i = \texttt{raise}\ v_i & \rightsquigarrow & \texttt{raise}\ v_i \\
\quad .. \quad \texttt{in}\ \mathrm{e} & &
\end{array}
$$

FIGURE 4.35 – Sémantique opérationnelle en présence d'exceptions

Implémentation de `Lazy.force` :

```
⟨ Lazy.force ⟨ m : τ ⟩ : τ_arg ⟩ ::=

  match ⟨ M (m, thunk) : M_τ (m) ⟩ with
  | ⟨ Thunk ⟨ f : unit → τ_arg ⟩ : M_τ (m) ⟩ →
    ⟨ let ⟨ x : τ_arg ⟩ = ⟨ f : unit → τ_arg ⟩ ⟨ () : unit ⟩ in
    ⟨ M (m, thunk) : τ ⟩ ← ⟨ Res x : M_τ (m) ⟩;
    ⟨ x : τ_arg ⟩

  | ⟨ Res ⟨ x : τ_arg ⟩ : M_τ (m) ⟩ →
    ⟨ x : τ_arg ⟩
```

Réduction des expressions annotées

$$\texttt{lazy}\ \text{e} \mid \mathcal{M} \quad \rightsquigarrow \quad \begin{array}{l} \langle\, m : \tau \,\rangle \mid \mathcal{M}, (m : \tau = [\text{e}]) ; \\ \text{t.q. } m \notin dom(\mathcal{M}) \end{array}$$

Filtrage de motifs

$$\text{match } (\texttt{lazy}\ \text{p}, \langle\, m : \tau \,\rangle) \quad = \quad x \leftarrow \text{Lazy.force } m \,;\, \text{match } (\text{p}, x)$$

FIGURE 4.36 – Sémantique opérationnelle des expressions paresseuses

4.5.7 Evaluation paresseuse

Si la construction **`lazy`** n'est pas exprimable directement en OCaml à cause de la généralisation (voir 12), on peut néanmoins exprimer sa sémantique de manière similaire à l'implémentation donnée dans la-dite sous-section. Pour cela, on se basera sur la sémantique des enregistrements et notamment des emplacements mémoire. La sémantique est donnée en figure 4.36.

Une valeur de type **`lazy`** est donc représentée par une cellule mémoire, laquelle contient une valeur de type `thunk` présentée 121. 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 noeuds. Heureusement, tous les types nécessaires aux annotations sont présents dans la mémoire courante ainsi que sur le noeud d'application `Lazy.force e`. L'implémentation est donc donnée figure 4.36 [12].

12. ∧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 noeuds composés

(applications, **let** .. **in** .. ou séquences) étant généralement le résultat des expressions qui les composent.

Chapitre 5

Conclusion

L'implémentation de l'inférence n'est pas infaillible, en particulier dans le cas d'un compilateur aussi riche et complexe qu'OCaml. Le système de types garantit l'absence d'erreurs à l'exécution, mais écrire un moteur d'inférence lui-même typé n'apporte aucune garantie sur son son résultat si ce n'est qu'il ne s'arrêtera pas en cours d'exécution sans que cela puisse être la volonté de son développeur (par une exception par exemple). Pour cela, il faudrait être capable d'écrire un typeur dans un langage avec types dépendants et prouver que cette implémentation respecte une spécification donnée.

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 annoté par des informations de types à tous les niveaux du programme. Vérifier la cohérence de cet arbre revient alors à s'assurer que l'implémentation de l'inférence est correcte pour un programme donné, 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 a vocation à être scolaire, elle n'est pas écrite dans le but d'être efficace 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 regard sur les performances.

L'objectif de cette thèse est double : formaliser un sous-ensemble du système de types d'OCaml et écrire un vérificateur d'arbre annoté suivant cette spécification. Dans le premier cas, l'intérêt est 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'agit 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. Le second est 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.

Le seconde partie décrit un sous-ensemble relativement important du langage OCaml. Ce chapitre formalise plusieurs aspects du langage d'une approche plus pratique : chaque construction

est étudiée sachant sa représentation concrète et non selon 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 abbréviations. Ces deux approches se justifient : l'inférence est implémentée dans un compilateur, écrit 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 autant 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.

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. Etre 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 sortes 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.

Finalement, est-il possible de répondre à la question qui a engendré tout ce travail : le Typedtree contient-il vraiment toute les informations issues de l'inférence ? Si oui, peut-on le voir comme un arbre de preuve du typage de notre programme ? Et donc, peut-on en vérifier la cohérence ? Considérant les quelques pages écrites entre le début de ce manuscrit et ces derniers mots, j'aurais tendance à penser que oui.

Annexe A

Implémentation d'un MiniML en Coq avec deconstruction

Finalement, cette rapide annexe présente une implémentation légèrement différente d'un MiniML dans lequel ont été ajouté des couples et dont les **let** sont dits destructurants : leur partie gauche de la valeur locale n'est pas forcément une variable mais un motif pouvant contenir plusieurs variables. Les arguments de fonctions peuvent également être des motifs. Cette implantation 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 suivant. Cette implantation 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 noeud de TAST contient un type et une expression. Ces expressions contiennent elles-même des sous-noeuds annotés. Cette représentation nécessite de redéfinir les principes d'induction normalement générés par Coq, ceux-ci n'étant pas suffisant. La technique utilisée est empruntée à Adam Chlipala [9] 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.

```
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

Les différentes opérations sont écrites de manière inductives, de la même manière que montrée précédemment. Pour chacune de celle-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 t1t2` se réduit en **`true`**, et inversement. La preuve se fait alors 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 implantant celle-ci en figure A.5. Dans un soucis 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 :

$$\Gamma, \varnothing \vdash \langle p : \sigma \rangle \Rightarrow (\overline{v : \sigma_v})$$

$$\Gamma \vdash \langle e : \tau \rangle \qquad \Gamma, \theta_\forall(\Gamma) \vdash \tau \leqslant \sigma \Rightarrow \theta \qquad \mathit{check\_gen}(\Gamma, \sigma, e)$$

$$\textsc{Let}\ \frac{\mathit{let}\ \Gamma' = \Gamma \oplus_{\mathcal{V}} (\overline{v : \sigma_v}) \qquad \Gamma' \vdash \langle e' : \tau' \rangle \qquad \Gamma \vdash \tau\ wf \qquad \Gamma' \vdash \tau \equiv \tau'}{\Gamma \vdash \langle \textbf{let}\ p = e\ \textbf{in}\ e' : \tau \rangle}$$

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

```
(** 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 implantation 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

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 qu'elle présente des effets de bord observables, il faut s'assurer que les variables généralisées peuvent effectivement l'être.
- L'expression courante est cohérente dans l'environnement initial auquel on a ajouté les variables liées dans le motifs.
- 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'implantation du vérificateur suit exactement ces étapes de vérification en appelant chaque fois

```
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 – Implantation 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'implantation 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 à la variable par laquelle la substituer.

L'implantation 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 montré par la décroissance du nombre de noeuds au fur et à mesure des appels récursifs successifs, via la condition en entête `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), autrement dit que pour toute expression dont il est possible de construire une preuve de cohérence, alors l'implantation retourne `true`, et inversement.

Bibliographie

[1] Hassan Aït-Kaci et Jacques Garrigue. Label-selective lambda-calculus syntax and confluence. In R. K. Shyamasundar, éditeur, *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15-17, 1993, Proceedings*, tome 761 de *Lecture Notes in Computer Science*, pages 24–40. Springer, 1993. isbn : 3-540-57529-4. doi : 10.1007/3-540-57529-4_41. url : https://doi.org/10.1007/3-540-57529-4_41 (cf. page 8).

[2] François Pottier Arthur Chargueraud. Locally nameless representation with cofinite quantification. http://chargueraud.org/softs/ln/ (cf. page 27).

[3] François Pottier Arthur Chargueraud. Tlc : a non-constructive library for coq. http://chargueraud.org/softs/tlc/ (cf. page 44).

[4] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack et Stephanie Weirich. Engineering formal metatheory. In George C. Necula et Philip Wadler, éditeurs, *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. ACM, 2008. isbn : 978-1-59593-689-9. doi : 10.1145/1328438.1328443. url : http://doi.acm.org/10.1145/1328438.1328443 (cf. pages 27, 143).

[5] 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. URL : http://gallium.inria.fr/~xleroy/publi/cfront.pdf (cf. page 21).

[6] Sandrine BLAZY et Xavier LEROY. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3) :263–288, 2009. URL : http://gallium.inria.fr/~xleroy/publi/Clight.pdf (cf. page 21).

[7] Arthur CHARGUÉRAUD. Program verification through characteristic formulae. In Paul HUDAK et Stephanie WEIRICH, éditeurs, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 321–332. ACM, 2010. ISBN : 978-1-60558-794-3. DOI : 10.1145/1863543.1863590. URL : http://doi.acm.org/10.1145/1863543.1863590 (cf. page 20).

[8] Arthur CHARGUÉRAUD. The locally nameless representation. *J. Autom. Reasoning*, 49(3) :363–408, 2012. DOI : 10.1007/s10817-011-9225-2. URL : https://doi.org/10.1007/s10817-011-9225-2 (cf. page 27).

[9] Adam CHLIPALA. *Certified programming with dependent types - A pragmatic introduction to the coq proof assistant*. In MIT Press, 2013. Partie 3, pages 64–69. ISBN : 978-0-262-02665-9. URL : http://mitpress.mit.edu/books/certified-programming-dependent-types (cf. page 143).

[10] Guy COUSINEAU, Pierre-Louis CURIEN et Michel MAUNY. The Categorical Abstract Machine. *Science of Computer Programming*, 8 :173–202, 1987 (cf. page 8).

[11] Zaynah DARGAYE. *Vérification formelle d'un compilateur pour langages fonctionnels*. Thèse de doctorat, Université Paris 7 Diderot, juillet 2009 (cf. page 22).

[12] Damien DOLIGEZ et Xavier LEROY. A concurrent, generational garbage collector for a multi-threaded implementation of ML. In Mary S. Van DEUSEN et Bernard LANG, éditeurs, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 113–

123. ACM Press, 1993. ISBN : 0-89791-560-7. DOI : 10.1145/158511.158611. URL : http://doi.acm.org/10.1145/158511.158611 (cf. page 8).

[13] Catherine DUBOIS. Proving ML type soundness within coq. In Mark AAGAARD et John HARRISON, éditeurs, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, tome 1869 de *Lecture Notes in Computer Science*, pages 126–144. Springer, 2000. ISBN : 3-540-67863-8. DOI : 10.1007/3-540-44659-1_9. URL : https://doi.org/10.1007/3-540-44659-1_9 (cf. page 22).

[14] 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. DOI : 10.1023/A:1006285817788. URL : https://doi.org/10.1023/A:1006285817788 (cf. page 22).

[15] Jacques GARRIGUE. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25(4) :867–891, 2015. DOI : 10.1017/S0960129513000066. URL : https://doi.org/10.1017/S0960129513000066 (cf. page 22).

[16] Jacques GARRIGUE. Programming with polymorphic variants. In *ML Workshop*, tome 13. Baltimore, 1998 (cf. page 8).

[17] Jacques GARRIGUE et Didier RÉMY. Ambivalent types for principal type inference with gadts. In Chung-chieh SHAN, éditeur, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, tome 8301 de *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013. ISBN : 978-3-319-03541-3. DOI : 10.1007/978-3-319-03542-0_19. URL : https://doi.org/10.1007/978-3-319-03542-0_19 (cf. page 8).

[18] Michael J. C. GORDON, Robin MILNER, L. MORRIS, Malcolm C. NEWEY et Christopher P. WADSWORTH. A metalanguage for interactive proof in LCF. In Alfred V. AHO, Stephen N.

Zilles et Thomas G. Szymanski, éditeurs, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 119–130. ACM Press, 1978. doi : 10.1145/512760.512773. url : http://doi.acm.org/10.1145/512760.512773 (cf. page 7).

[19] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar et Michael Norrish. Verified characteristic formulae for cakeml. In Hongseok Yang, éditeur, *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*, tome 10201 de *Lecture Notes in Computer Science*, pages 584–610. Springer, 2017. isbn : 978-3-662-54433-4. doi : 10.1007/978-3-662-54434-1_22. url : https://doi.org/10.1007/978-3-662-54434-1_22 (cf. page 20).

[20] Stefan Kaes. Parametric overloading in polymorphic programming languages. In Harald Ganzinger, éditeur, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, tome 300 de *Lecture Notes in Computer Science*, pages 131–144. Springer, 1988. isbn : 3-540-19027-9. doi : 10.1007/3-540-19027-9_9. url : https://doi.org/10.1007/3-540-19027-9_9 (cf. page 14).

[21] 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 75).

[22] Daniel K. Lee, Karl Crary et Robert Harper. Towards a mechanized metatheory of standard ML. In Martin Hofmann et Matthias Felleisen, éditeurs, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173–184. ACM, 2007. isbn : 1-59593-575-4. doi : 10.1145/1190216.1190245. url : http://doi.acm.org/10.1145/1190216.1190245 (cf. page 22).

[23] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009. URL : http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf (cf. page 21).

[24] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3) :269–303, 2000. URL : http://journals.cambridge.org/action/displayAbstract?aid=54525 (cf. page 8).

[25] Michel Mauny. *Compilation des Langages Fonctionnels dans les Combinateurs Cat'egoriques – Application au langage ML.* Thèse de doctorat, Universit'e Paris 7, 1985 (cf. page 8).

[26] 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).

[27] 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. DOI : 10.1007/s10990-011-9065-0. URL : http://dx.doi.org/10.1007/s10990-011-9065-0 (cf. page 14).

[28] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3) :348–375, 1978. DOI : 10.1016/0022-0000(78)90014-4. URL : https://doi.org/10.1016/0022-0000(78)90014-4 (cf. page 7).

[29] 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. DOI : 10.1017/S0956796813000282. URL : https://doi.org/10.1017/S0956796813000282 (cf. page 20).

[30] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein et Neil D. Jones, éditeurs, *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. ACM Press, 1997. ISBN : 0-89791-853-3. DOI : 10.1145/263699.263712. URL : http://doi.acm.org/10.1145/263699.263712 (cf. page 19).

[31] Simon PEYTON JONES. Haskell 98 Language and Libraries : the Revised Report, janvier 2003 (cf. page 14).

[32] Benjamin C. PIERCE. *Types and programming languages*. MIT Press, 2002. ISBN : 978-0-262-16209-8 (cf. page 130).

[33] Richard PORTER. *Top Gear : Ambitious but Rubbish : The Secrets Behind Top Gear's Craziest Creations*. BBC Books, 2012. ISBN : 978-1849905039 (cf. page 7).

[34] François POTTIER et Didier RÉMY. The essence of ML type inference. In Benjamin C. PIERCE, éditeur, *Advanced Topics in Types and Programming Languages*, partie 10, pages 389–489. MIT Press, 2005. URL : http://cristal.inria.fr/attapl/ (cf. page 26).

[35] Didier REMY. Extension of ML Type System with a Sorted Equational Theory on Types. Rapport technique, 1992 (cf. page 75).

[36] Didier RÉMY et Jerome VOUILLON. Objective ML : an effective object-oriented extension to ML. *TAPOS*, 4(1) :27–50, 1998 (cf. page 8).

[37] 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. DOI : 10.1145/1190315.1190324. URL : http://doi.acm.org/10.1145/1190315.1190324 (cf. page 14).

[38] 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. DOI : 10.1145/989393.989449. URL : http://doi.acm.org/10.1145/989393.989449 (cf. page 14).

[39] 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. DOI : 10.1145/2544174.2500599. URL : http://doi.acm.org/10.1145/2544174.2500599 (cf. page 14).

[40] 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. DOI : 10.1145/1925844.1926411. URL : http://doi.acm.org/10.1145/1925844.1926411 (cf. page 14).

[41] Pierre WEIS et Xavier LEROY. *Le langage Caml*. InterEditions, 1993. ISBN : 978-2-7296-0493-6 (cf. page 8).