

Chapitre 10

Vérification et inférence de types

La programmation non typée est d'une grande souplesse : elle permet le meilleur . . . comme le pire. C'est elle qui permet par exemple de coder dans le λ -calcul (et dans tout langage non typé, ou typé dynamiquement) des choses aussi puissantes que les combinateurs de point fixe, les entiers de Church, l'interprétation d'entiers machine comme des peinteurs, les écrans bleus de Windows, les erreurs de segmentation et les *kernel panic* d'Unix. Une grande expressivité et liberté de codage, en échange d'une absence de garde-fous.

Le langage C est un exemple de langage faiblement typé : il permet la conversion d'entier en peinteur sans vérification aucune, tout au plus le compilateur vous avertira qu'il décline toute responsabilité en cas d'accident. L'arithmétique qu'il autorise sur les peinteurs lui confère une grande efficacité dans les parcours de données stockées dans des espaces mémoire contigus comme les tableaux, mais fait aussi sortir très rapidement de l'espace mémoire autorisé. Un langage peu ou faiblement typé comme C permet ainsi à la fois une programmation structurée d'assez haut niveau et une programmation de très bas niveau, formant ainsi un outil très apprécié pour la réalisation de couches logicielles proches du matériel, comme le système d'exploitation ou les pilotes de périphériques. Le prix de cette liberté de codage est bien sûr la gravité des erreurs d'exécution, dues par exemple à des accès hors de la mémoire ou à des tentatives d'exécuter des codes arbitraires : en effet, tout est *a priori* autorisé, et aucune vérification n'a lieu à l'exécution. Il est d'ailleurs probable que l'efficacité recherchée dans les logiciels d'aussi bas niveau est telle que les tests dynamiques qui leur confèreraient un peu plus de sécurité seraient sans doute jugés trop coûteux.

Au lieu de ne presque rien vérifier, comme le langage C (dans sa version classique), il existe des langages qui vous laissent écrire des programmes, et qui vont vérifier à l'exécution la validité des opérations effectuées par les programmes. C'est le cas du langage Scheme (et tous les langages Lisp en général), et c'est aussi le cas des langages de script comme Perl, Javascript, Python, Ruby et autres. L'avantage de cette technique, est qu'elle laisse la même liberté de codage que celle dont vous bénéficiez lorsque vous programmez dans un langage n'effectuant que peu ou pas du tout de vérification. Bien sûr, l'inconvénient est la pénalité en termes d'efficacité des tests nécessaires à *chaque pas d'exécution* pour provoquer un échec relativement gracieux là où le langage C non typé vous aurait laissé aller dans le mur, voire même un peu au-delà du mur ! Un langage typé dynami-

quement teste que chaque addition que vous effectuez implique bien des entiers, que la procédure que vous appelez existe bien, et que l'accès à tel ou tel champ d'enregistrement est bien possible, c'est-à-dire que le champ est effectivement présent. Le coût des tests supplémentaires n'est pas le seul surcoût de cette technique : il est aussi probable que des informations relevant du typage soient présents dans les données manipulées par les programmes, de sorte à faciliter les vérifications (noms de champs d'enregistrements, par exemple).

À l'autre extrémité du spectre, se trouvent les langages à typage statique : les types de leurs programmes sont vérifiés à la compilation, une fois pour toutes, factorisant ainsi toutes les vérifications dynamiques correspondantes. L'avantage est évident en termes d'efficacité – les programmes peuvent être exempts de tests dynamiques de type –, et en termes de fiabilité. Celle-ci est en effet accrue pour les programmes qui passent l'épreuve du typage statique : on a généralement pour eux la garantie supplémentaire que le programme échappe à toute une classe d'erreurs, suite à la vérification dont il a été l'objet. Par contre, un compilateur effectuant une vérification statique de types peut compliquer – à première vue – la tâche du programmeur, en l'empêchant d'écrire des programmes qui ne satisfont pas la discipline de typage du langage tout en étant parfaitement correct, au sens où ils ne provoqueraient pas d'erreur d'exécution. Les langages de programmation qui imposent une vérification statique des types sont donc, en première approximation, moins expressifs que les autres. En fait, ce n'est pas vrai en théorie : les langages généralistes, fussent-ils statiquement typés, sont généralement complets au sens de Turing : ils permettent d'encoder toute fonction calculable. C'est par contre quelquefois vrai dans la pratique : la vérification statique de typage peut ne pas apprécier votre style de programmation ; mais, qu'on se rassure, il existe toujours un style alternatif, ne serait-ce que par le « théorème » précédent sur l'expressivité théorique des langages.

C'est à ces techniques de vérification statique et d'inférence, que ce chapitre est consacré. Nous nous concentrerons essentiellement sur l'inférence, qui relègue la vérification au rang de sous-problème : en présence d'annotations de type en assez grand nombre, le problème de l'inférence devient un problème de propagation et de vérification des informations de typage.

10.1 Le langage PCF, et le but du typage statique

Le langage que nous allons considérer ici est le noyau fonctionnel de PCF dans lequel nous précisons un peu les constantes (entières et booléennes). La syntaxe du langage s'écrit donc :

$$\begin{aligned}
 e ::= & n \in \mathbb{N} \mid b \in \mathbb{B} \mid x \mid e_1 e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
 & \mid e_1 + e_2 \mid e_1 = e_2 \\
 & \mid \text{let } x = e_1 \text{ in } e_2 \\
 & \mid \text{letrec } f(x) = e_1 \text{ in } e_2 \\
 & \mid \text{fun } x \rightarrow e
 \end{aligned}$$

La discipline de typage que nous allons imposer aux programmes PCF est assez évidente : nous allons par exemple limiter l'emploi des opérateurs d'addition et de comparaison aux

entiers, et exiger que la conditionnelle ne teste que des conditions booléennes. Bien sûr, nous devons vérifier qu'une expression en position « fonction » dans une application a effectivement un type fonctionnel. On voit clairement que le but de ces vérifications est précisément d'éviter les tests correspondants, qui devraient être inclus dans le code produit si la vérification de typage n'était pas effectuée à la compilation. Si les tests en question sont assez faciles à énumérer et à mettre en œuvre à l'exécution, garantir leur inutilité n'est pas aussi simple. Pour que les tests qui vérifient que les valeurs que l'on applique sont toujours des fonctions, il faut que dans le programme entier, *toutes* les applications disposent de la garantie que la valeur calculée par l'évaluation de la partie « fonction ». de l'application sera bien une fonction. Or, on n'a pas le droit d'effectuer des calculs à ce stade (ils pourraient ne pas terminer, ou nécessiter des lectures ou des écritures qu'il est hors de question de réaliser à la compilation. Puisqu'il est impossible de calculer les valeurs exactes à la compilation, on doit se contenter d'approximations, et c'est exactement ce que sont les types : des approximations de valeurs.

10.2 Prédire statiquement impose de faire des choix

Puisque la prédiction statique des valeurs exactes de telle ou telle expression est impossible, il faut prédire les résultats de calculs non pas comme des valeurs précises, mais comme des ensembles de valeurs possibles. Ainsi, on prédira que le résultat de $e_1 + e_2$ sera un entier (en supposant que le calcul termine), si tant est que e_1 et e_2 sont eux aussi (prédits comme étant) des entiers.

Se parer à toutes les éventualités implique aussi (et surtout) de choisir de refuser des programmes contenant des sous-expressions dont le typage ne peut prédire aisément le résultat. Par exemple, si e_2 est un entier et si e_3 est une fonction, on interdira la conditionnelle « if e_1 then e_2 else e_3 », car si cette conditionnelle était appliquée, ou si on en additionnait le résultat avec un entier, on ne pourrait pas dire si on devait l'autoriser ou l'interdire. En résumé, une discipline de typage impose généralement que chaque sous-expression ait un type, et on va utiliser ce type pour représenter l'expression là où sa valeur peut apparaître à l'exécution.

Définir une discipline de typage, c'est donc définir d'une part quels sont les types qui sont possibles, et d'autre part comment on calcule le type de chaque expression, comment on propage l'information de type dans les programmes, et quelles sont les vérifications auxquelles doivent être soumis les programmes afin de passer avec succès l'épreuve du typage.

10.3 Les types de PCF

Définissons maintenant les types que nous souhaitons attribuer à notre langage. Clairement, nous avons besoin d'un type `int` pour les entiers, d'un autre que nous appellerons `bool` pour les booléens, et d'un *constructeur de types* à deux places que nous noterons $(_ \rightarrow _)$ pour les fonctions, de sorte à pouvoir construire des types fonctionnels comme `int \rightarrow bool`, ou `bool \rightarrow (int \rightarrow int)`. Formellement, notre algèbre de types est constituée des

termes suivants :

$$\tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2$$

Nous pouvons maintenant considérer chacune des formes d'expressions constituant notre langage, et tenter d'exprimer – informellement dans un premier temps – comment attribuer un type et effectuer au passage les vérifications que nous avons mentionnées ci-dessus. Nous décrivons ci-dessous comment *calculer* les types ; leur *vérification* consistera à tester l'égalité (ou, plus généralement, la compatibilité) du type calculé avec le type attendu.

cas $n \in \mathbb{N}$: clairement, cette expression est du type `int` ;

cas $b \in \mathbb{B}$: de façon identique au cas précédent, cette expression est du type `bool` ;

cas $(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)$: on va ici calculer le type de e_1 , et exiger qu'il soit compatible avec (c'est-à-dire égal à) `bool` ; à la suite de quoi on va calculer les types de e_2 et e_3 , et exiger qu'ils soient compatibles entre eux, c'est-à-dire vérifier qu'ils sont égaux, voire les rendre égaux, comme on le verra plus tard ;

cas $(e_1 + e_2)$ **et** $(e_1 = e_2)$: ces deux cas sont similaires : on va calculer les types de e_1 et e_2 , et les identifier à `int`, ensuite, on produira le type `int` dans le premier cas, `bool` dans le second ;

cas $(\text{let } x = e_1 \text{ in } e_2)$: ici, on va calculer un type pour e_1 et l'associer à x durant le typage de e_2 . Tout comme pour l'évaluation, le typage de e_2 nécessitera un *environnement* qui va associer un type à chacune des variables qui sont accessibles à cet endroit du programme. On va nommer Γ un tel environnement, et on le dotera d'opérations similaires à celles dont étaient équipés nos environnements d'évaluation ou d'exécution ρ .

cas x : puisque nous sommes maintenant munis d'un environnement de typage Γ , le type de x sera le type $\Gamma(x)$ qui lui est associé dans Γ ;

cas $(\text{fun } x \rightarrow e)$: le type de cette expression sera nécessairement un type fonctionnel $\tau_1 \rightarrow \tau_2$; même si cela a l'air un peu magique à ce stade, on peut dire que l'on va obtenir le type τ_2 comme le type de e en faisant l'hypothèse que x est de type τ_1 . La question qui se pose alors est naturellement la suivante : « comment fait-on le choix de τ_1 dans ce cas ? » Nous avons laissé entendre ci-dessus que nous étions capables de nous « arranger » pour rendre deux types égaux : c'est exactement ce phénomène qui va agir ici. Un type initial sera choisi pour x , et il sera raffiné au fur et à mesure que l'on découvre les usages faits de x dans e . Ce type initial, si on ne dispose d'absolument aucune information, ce sera une inconnue de type, une variable de type α . Le processus de raffinement qui va être utilisé, basé sur la possibilité de rendre deux types égaux, sera l'*unification* que nous avons déjà vue au chapitre 4, où les termes à unifier vont représenter des types à rendre égaux.

cas $(e_1 e_2)$: le calcul du type de cette expression passe par le calcul du type de e_1 , en lui imposant d'être un type fonctionnel $\tau_2 \rightarrow \tau$. On calcule aussi un type τ'_2 pour e_2 , et on va forcer l'égalité de τ_2 et τ'_2 . Plus précisément, on va calculer μ , le meilleur unificateur de τ_2 et τ'_2 , et on va produire $\mu(\tau)$ comme type résultant.

cas $(\text{letrec } f(x) = e_1 \text{ in } e_2)$: ce cas n'est en rien spécial, si ce n'est qu'il faut deviner le type de f et de x en calculant le type de $(\text{fun } f \rightarrow (\text{fun } x \rightarrow e_1))$ et l'identifiant au type de

($\text{fun } x \rightarrow e_1$), et ensuite calculer le type de e_2 dans un environnement associant le type produit à f . On note que si le type de f n'est pas complètement déterminé lors de sa mise dans l'environnement, il pourra être automatiquement précisé durant le typage de e_2 , ou, plus généralement, du contexte de cette déclaration.

Nous avons maintenant une idée assez précise du mécanisme opérationnel de synthèse de types pour PCF : on a une algèbre de types avec des variables de types jouant le rôle d'inconnues, on matérialise les contraintes de type par des équations entre types, et on fait appel à l'unification pour résoudre ces équations. Nous allons maintenant donner une présentation plus formelle de la discipline de typage de PCF, dans un formalisme proche de celui que nous avons vu au chapitre précédent : des règles d'inférence.

10.4 Inférence de types pour PCF

Notre algèbre de types s'est maintenant enrichie de variables de types et devient :

$$\tau ::= \alpha \mid \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2$$

La discipline de typage de PCF a été donnée indépendamment par Roger Hindley et Robin Milner à la fin des années 70. Le système de Milner, qui équipait la toute première version du langage ML, était doté de polymorphisme (voir section suivante), ce qui a largement contribué à son succès. Cette discipline de typage été donnée initialement dans un formalisme bien différent de celui qui est présenté ici. Celui que nous utilisons, qui est maintenant devenu classique, a été proposé initialement par Kahn *et al.* dans un article où étaient présentées des applications de la sémantique naturelle.

Tout comme nous l'avons fait au chapitre précédent pour la sémantique opérationnelle, le système de types (présentation formelle de la discipline de typage) est donné par un système de règles définissant des jugements de la forme $\Gamma \vdash e : \tau$, que l'on lit comme suit : « dans l'environnement de typage Γ , l'expression e a le type τ ». Les règles sont à rapprocher des explications que nous avons données à la section précédente, qui en forment un commentaire de nature opérationnelle.

$$\begin{array}{c} \Gamma \vdash n : \text{int} \qquad \Gamma \vdash b : \text{bool} \qquad \frac{x \in \text{dom}(\Gamma) \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \\ \\ \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 + e_2) : \text{int}} \\ \\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 = e_2) : \text{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \oplus [x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2} \qquad \frac{\Gamma \oplus [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\text{fun } x \rightarrow e) : \tau_1 \rightarrow \tau_2} \\ \\ \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 e_2) : \tau} \qquad \frac{\Gamma \oplus [f : \tau \rightarrow \tau_1; x : \tau] \vdash e_1 : \tau_1 \quad \Gamma \oplus [f : \tau \rightarrow \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash (\text{letrec } f(x) = e_1 \text{ in } e_2) : \tau_2} \end{array}$$

Ce système appelle quelques remarques :

- tout comme au chapitre précédent, il n’y a pas de règle d’erreur : pour un jugement donné, ou bien on peut construire une dérivation qui le prouve, ou alors on ne le peut pas, et le jugement sera considéré comme invalide ;
- il y a exactement une règle par construction syntaxique : on est donc assez proche d’un algorithme qui transcrirait chacune de ces règles en un cas de filtrage d’une fonction récursive, comme nous l’avons suggéré pour l’interprète du chapitre précédent ;
- dans certaines règles, on peut trouver plusieurs occurrences d’un même type¹ τ : opérationnellement, c’est l’algorithme unification qui forcera ces différents types à être identiques, forçant au passage certaines inconnues (variables de types) à prendre des valeurs particulières. La synthèse de types est donc une affaire de résolution d’équations.

10.5 Polymorphisme à la ML, ou polymorphisme paramétrique

Nous allons maintenant rapprocher le système de types de celui du noyau fonctionnel des langages de la famille ML (donc OCaml fait partie). En effet, dans le système de types présenté ci-dessus, le programme suivant n’est pas typable :

$$\text{let } f = \text{fun } x \rightarrow x \text{ in } (f f)(1)$$

Pour s’en rendre compte, il suffit de construire une preuve de typage pour le jugement

$$\Gamma \vdash (\text{let } f = \text{fun } x \rightarrow x \text{ in } (f f)(1)) : \text{int}$$

où le choix de int pour le type du résultat semble judicieux, puisque l’entier 1 est le résultat de l’évaluation de ce programme. En effet, quand on tente de construire une preuve de typage pour ce jugement, on se heurte très vite à un conflit entre les types attendus pour les deux occurrences de f : la première prenant la seconde en argument, et cette dernière étant le résultat de l’application $(f f)$, la seconde occurrence de f devrait avoir pour type $(\text{int} \rightarrow \text{int})$, alors que la première devrait être de type $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$, ce qui est impossible puisque ces deux termes ne sont pas unifiables. La solution adoptée par Milner lorsqu’il a conçu le système de types de ML, est d’autoriser dans certaines circonstances la *généralisation* de variables de types dont la spécialisation n’apporterait rien de plus. C’est le cas du type $(\alpha \rightarrow \alpha)$ qu’il est possible d’affecter à la fonction identité $(\text{fun } x \rightarrow x)$. On pourrait bien sûr spécialiser α en un type quelconque, mais cela ne ferait que restreindre artificiellement et inutilement les utilisations possibles de cette fonction. Au lieu de cela, on peut généraliser la variable de type α et attribuer à la fonction identité le *schéma de type* $\forall \alpha. (\alpha \rightarrow \alpha)$, nous autorisant ainsi à utiliser la fonction identité avec autant de types qu’il nous plaira, pourvu que chacun d’entre eux soit une instance de ce schéma. Un tel schéma est appelé *type polymorphe*. Ce polymorphisme est appelé *polymorphisme paramétrique*, car il est basé sur ces schémas de types, où les variables quantifiées universellement sont

¹La règle concernant les applications, par exemple, force le domaine τ_2 de la fonction à être identique au type de l’argument.

comme des paramètres formels, qui sont remplacés par des types lors du calcul d'instances de ces schémas.

La présentation formelle du système de types pour PCF avec polymorphisme nécessite la définition des schémas, et la méthode utilisée pour instancier un schéma en un type. En rappelant la définitions des types de notre langage, nous définissons les schémas de types comme des types préfixés par la quantification d'un ensemble de variables que l'on note $\vec{\alpha}$. Lorsque l'ensemble de variables quantifiées est vide, un schéma de types n'est rien d'autre qu'un type : les schémas de types forment donc un sur-ensemble de l'ensemble des types.

$$\begin{aligned}\tau & ::= \alpha \mid \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \\ \sigma & ::= \forall \vec{\alpha}. \tau\end{aligned}$$

On remarque aussi que la quantification des variables n'est autorisée qu'à l'extérieur des schémas : nous n'autorisons pas ici la présence de schémas non triviaux à l'intérieur des types.

Instanciation générique Le passage d'un schéma de types à un type est appelé *instanciation générique*, et consiste à remplacer dans le corps d'un schéma, les variables quantifiées universellement par des types. On dit que τ_2 est une instance générique du schéma $\forall \vec{\alpha}. \tau_1$, et on note $\tau_2 \leq \forall \vec{\alpha}. \tau_1$, si il existe une substitution θ dont le domaine est inclus dans $\vec{\alpha}$ et telle que $\tau_2 = \theta(\tau_1)$.

Généralisation L'opération inverse, appelée généralisation, consiste, étant donné un type τ et un environnement de typage Γ , à identifier quelles sont les variables de types apparaissant dans τ qui sont à même d'être généralisées. La définition de la généralisation considère que seules les variables de types qui n'apparaissent pas (libres) dans l'environnement Γ sont généralisables :

$$\text{gen}(\tau, \Gamma) = \forall \vec{\alpha}. \tau \text{ où } \vec{\alpha} = \text{vars}(\tau) \setminus \text{fv}(\Gamma)$$

où on note $\text{vars}(\tau)$ l'ensemble des variables apparaissant dans τ et $\text{fv}(\Gamma)$ l'ensemble des variables *libres* de Γ . Comme nous le verrons dans quelques instants, la généralisation aura lieu dès que l'on a calculé le type d'une variable définie localement, et Γ est l'environnement à l'aide duquel ce type a été calculé.

Système de règles Nous donnons ici le système de règles permettant de calculer des types avec du polymorphisme à la ML à partir des programmes PCF.

$$\begin{array}{c}
\Gamma \vdash n : \text{int} \qquad \Gamma \vdash b : \text{bool} \qquad \boxed{\frac{x \in \text{dom}(\Gamma) \quad \tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}} \\
\\
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 + e_2) : \text{int}} \\
\\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 = e_2) : \text{bool}} \qquad \boxed{\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \oplus [x : \text{gen}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}} \\
\\
\frac{\Gamma \oplus [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\text{fun } x \rightarrow e) : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 e_2) : \tau} \\
\\
\boxed{\frac{\Gamma \oplus [f : \tau \rightarrow \tau_1; x : \tau] \vdash e_1 : \tau_1 \quad \Gamma \oplus [f : \text{gen}(\tau \rightarrow \tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash (\text{letrec } f(x) = e_1 \text{ in } e_2) : \tau_2}}
\end{array}$$

Ces règles ne présentent que deux différences très légères avec le système précédent : nous avons encadré ci-dessus les règles qui ont varié. D'une part, les règles sur les déclarations locales généralisent maintenant les types qu'elles ont calculés pour la valeur déclarée localement, ce qui fait que les environnements de typage contiennent maintenant des types ou des schémas de types. D'autre part, la règle de typage des identificateurs prend maintenant des instances du schéma de type associé à la variable correspondante dans l'environnement de typage courant.

Propriétés Nous énonçons seulement la propriété essentielle de la discipline de typage de PCF, et tout-à-fait informellement, de surcroît. Étant donné un programme e , un type τ et un environnement de typage Γ , si $\Gamma \vdash e : \tau$ est valide, alors étant donné un environnement d'exécution ρ compatible avec Γ (c'est-à-dire tel que $\rho : \Gamma$, en quelque sorte), si $\rho \vdash e \Rightarrow r$, alors r est une valeur v et son type est τ . En d'autres termes, cette propriété indique que le typage de PCF prédit correctement le type des valeurs effectivement calculées par les programmes. Un corollaire de cette propriété est que l'exécution d'un programme typable ne peut produire une erreur dynamique de type. Le typage statique élimine donc effectivement toute une classe d'erreurs.

10.6 Algorithme d'inférence

Nous terminons ce chapitre en donnant une idée de l'algorithme. Une présentation assez formelle de l'algorithme utilise un calcul d'unificateur le plus général (appels à la fonction `unif` ci-dessous), et produit, pour une expression e à typer dans un environnement de typage Γ , un type pour cette expression, ainsi qu'une substitution à appliquer à Γ pour que l'expression puisse avoir effectivement ce type.

```
let rec infer  $\Gamma$   $e$  = match  $e$  with
  «  $n$  »  $\rightarrow$  (int, [ ])
| «  $x$  »  $\rightarrow$  (instance( $\Gamma(x)$ , [ ]))
| «  $e_1 e_2$  »  $\rightarrow$ 
  let  $(\tau_1, \theta_1)$  = infer  $\Gamma$   $e_1$  in
  let  $(\tau_2, \theta_2)$  = infer ( $\theta_1(\Gamma)$ )  $e_2$  in
  let  $\tau_3$  = nouvelle variable in
  let  $\theta_3$  = unif ( $\theta_2(\tau_1)$ ) ( $\tau_2 \rightarrow \tau_3$ ) in
  ( $\theta_3(\tau_3)$ ,  $\theta_3 \circ \theta_2 \circ \theta_1$ )
| « (fun  $x \rightarrow e$ ) »  $\rightarrow$ 
  let  $\tau_0$  = nouvelle variable in
  let  $(\tau_1, \theta_1)$  = infer ( $\Gamma \oplus [x : \tau_0]$ )  $e$  in
  ( $(\theta_1(\tau_0) \rightarrow \tau_1)$ ,  $\theta_1$ )
| « (let  $x = e_1$  in  $e_2$ ) »  $\rightarrow$ 
  let  $(\tau_1, \theta_1)$  = infer  $\Gamma$   $e_1$  in
  let  $(\tau_2, \theta_2)$  = infer ( $\theta_1(\Gamma) \oplus [x : \text{gen}(\tau_1, \theta_1(\Gamma))]$ )  $e_2$  in
  ( $\tau_2$ ,  $\theta_2 \circ \theta_1$ )
| ...
```

L'algorithme est correct : si $\text{infer } \Gamma e = (\tau, \theta)$, alors $\theta(\Gamma) \vdash e : \tau$.

L'algorithme est complet : pour un programme typable, l'algorithme va calculer son type *le plus général*, c'est-à-dire celui dont tous ses autres types sont des instances.

Synthèse destructive Pour conclure, il nous faut noter que les algorithmes d'inférence effectifs sont très nettement optimisés par rapport à celui ci-dessus. Ces algorithmes utilisent notamment une unification *destructive* qui égalise physiquement deux types à unifier, au lieu de calculer une substitution.