Chapitre 2

Système de types de base

Nous présentons ici la technique utilisée pour formaliser les différents systèmes de types à base de saturation de contraintes auxquels nous aurons affaire par la suite. Un système de types de base est ensuite présenté dans ce formalisme, il formera le point de départ des systèmes des prochains chapitres. Nous donnerons ensuite une trame des preuves de validité de tels systèmes de types vis-à-vis de la sémantique du langage, et enfin prouverons la terminaison et la validité du système de base.

2.1 Contexte

Le but du « typage » est principalement d'analyser les programmes dans le but d'y détecter des erreurs. Pour atteindre un tel but, une idée pourrait être d'exécuter effectivement le programme que l'on souhaite analyser et d'observer son comportement. Néanmoins, les programmes ne sont pas toujours déterministes, ils peuvent dépendre d'un certain nombre de facteurs extérieurs sur lesquels nous n'avons pas le contrôle au moment de la vérification du programme comme la lecture d'informations dans un fichier, la réception de données venant d'un capteur, ou l'appel volontaire à une fonction renvoyant une valeur aléatoire. De plus, l'exécution d'un programme peut durer un temps très long, voire ne pas se terminer. Toutefois, les analyses que nous souhaitons faire sur les programmes avant leurs exécutions doivent se terminer en un temps fini, et si possible, en un temps « raisonnable ». Il est donc nécessaire, pour faire de telles analyses, de s'abstraire du comportement exact du programme et de se restreindre à en extraire des « propriétés ».

En réalité, les systèmes de types ont également un autre but que la simple « vérification de validité », à savoir extraire depuis le code des informations utiles pour le programmeur : les « types ». Les types représentent des ensembles de valeurs que peuvent prendre les sous-expressions du programme lors de l'exécution. Ils sont utiles à la fois en terme de documentation, car il est souvent plus facile de lire un type donnant une information synthétique sur un code plutôt que le code lui-même ; mais aussi en terme d'auto-protection de la part du programmeur. En effet, celui-ci peut être amené à annoter son code avec des types (potentiellement incomplets) et demander à l'analyseur (le « typeur », en l'occurrence) de vérifier que ses annotations sont correctes.

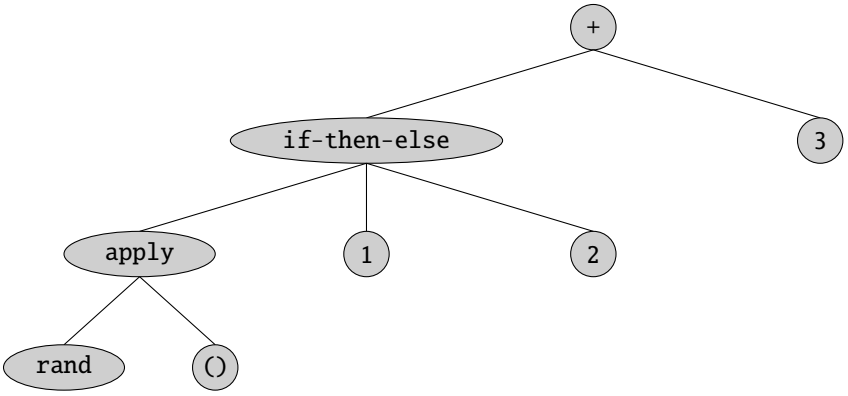
Un système de types a donc pour but de définir formellement quels types sont compatibles avec quelles expressions. Un système de types peut donc être utilisé directement en tant que « validateur » : étant donné une expression et un type, ce type est-il compatible avec cette expression ?

Dans certains cas, il est également possible de transformer un système de types en un algorithme dit d'« inférence » prenant en entrée une expression et générant, s'il y arrive, un type compatible avec cette expression.

Habituellement, la définition d'un système de types se fait indépendamment de la sémantique du langage. Pour que le système de types nous donne une information intéressante vis-à-vis de la sémantique, nous sommes amenés à les relier par un « théorème de validité » : étant donnée une expression `e`, s'il existe un type τ compatible avec `e`, alors l'évaluation de `e` devra se dérouler correctement. Ce théorème est détaillé en section 2.4.3.

2.1.1 Principes de base

Un algorithme de typage possède en général une phase consistant à effectuer un parcours récursif du programme, qui est alors vu comme un arbre, et à en extraire des contraintes. Par exemple, l'expression `((if rand () then 1 else 2) + 3)` sera représenté par l'arbre :



Pour chaque noeud de cet arbre, qui correspond donc à une sous-expression de l'expression originale, nous allons chercher à associer une « variable de type » représentant l'ensemble des valeurs obtenues lors des évaluations de cette sous-expression. Chacune de ces variables de type est contrainte par deux biais : d'une part par la sous-expression en question qui est susceptible de générer un certain ensemble de valeurs à l'exécution, mais aussi par le parent du noeud susceptible d'imposer des contraintes sur les valeurs acceptables comme résultat de l'évaluation de la sous-expression. Nous allons alors construire une conjonction de contraintes (notée « Φ ») contenant l'ensemble des contraintes satisfaites par chacune de ces variables de type.

Par exemple, dans l'arbre précédent, le noeud `if-then-else` s'évaluera parfois en `1` et parfois en `2`, qui sont des entiers, et son parent, le noeud `(+)`, lui impose d'être un entier. Ces deux ensembles étant les mêmes, cette partie du programme devrait s'exécuter sans erreur.

La différence principale entre « unification » et « sous-typage » tient à la forme des contraintes. Un algorithme à base d'unification imposera des contraintes d'égalité entre les types tandis qu'un algorithme à base de sous-typage imposera des contraintes d'inclusion.

Représenter les contraintes par des relations d'inclusion est en général plus « fin » que les représenter par des égalités car il est toujours possible d'encoder une égalité par une double inclusion, tandis que l'inverse est impossible. Les systèmes à base de sous-typage offrent donc plus d'expressivité dans les contraintes reliant les types et sont donc en général plus « puissants » au sens où plus de programmes valides sont acceptés par le typeur.

Néanmoins, les algorithmes à base d'unification peuvent avoir recours à un algorithme de résolution des contraintes d'égalité nommé « union-find » (cf. [CP15, H91]) qui est très performant en pratique, tandis que la saturation de contraintes d'inégalités est souvent plus complexe à implémenter et moins performante.

Malgré cette difficulté implémentatoire, cette thèse se concentre sur le problème du sous-typage. Le chapitre 6 sur l'implémentation donne un ensemble de techniques qui ont été éprouvées pour améliorer les performances d'un typeur à base de sous-typage et rendre la saturation de contraintes utilisable en pratique.

2.1.2 Approches classiques du sous-typage

Il est assez classique de considérer les types comme des structures arborescentes définies par une grammaire ressemblant à :

$$\tau \ ::= \alpha \mid \tau_b \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

$$\tau_b \ ::= \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \ldots$$

Dans un système de types à base d'unification, il est ainsi possible d'associer un type seul à une expression, comme par exemple le type « $\alpha \to \alpha \to \texttt{int}$ » dans lequel apparaît la variable de type α libre. Une telle syntaxe est simple et agréable à manipuler pour l'utilisateur.

Dans un monde avec sous-typage, une approche classique consiste à se débarrasser des contraintes de sous-typage en étendant la grammaire des types avec les constructions suivantes :

$$\tau \ +:= \top \mid \bot \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$$

Grâce à une telle extension, lorsque le langage des contraintes est restreint à une conjonction de contraintes de sous-typage, il est possible de représenter un schéma de type quelconque (possédant des contraintes de sous-typage arbitraire) par un type. Le principe de cette transformation d'un schéma de type en type consiste à réduire la conjonction de contraintes de sous-typage par des règles de la forme :

- $\alpha \leqslant \tau_1 \ \wedge \ \alpha \leqslant \tau_2 \ \rightsquigarrow \ \alpha \leqslant \tau_1 \sqcap \tau_2$
- $\tau_1 \leqslant \alpha \ \wedge \ \tau_2 \leqslant \alpha \ \rightsquigarrow \ \tau_1 \sqcup \tau_2 \leqslant \alpha$
- etc.

Sur cette nouvelle algèbre de types (munie de « $\top$ », « $\bot$ », « $\sqcup$ » et « $\sqcap$ »), il est alors d'usage de définir une « relation de sous-typage », notée également $(\leqslant)$, spécifiant si deux types τ_1 et τ_2

vérifient $\tau_1 \leqslant \tau_2$. Il existe deux approches pour définir cette relation de sous-typage, à savoir le « sous-typage syntaxique » et le « sous-typage sémantique ».

Le sous-typage syntaxique consiste à définir cette relation via un ensemble de règles décomposant les types en fonction de leur forme syntaxique et propageant la relation de sous-typage jusqu'à obtenir des relations élémentaires (comme « `int` $\leqslant$ `int` », « `int` $\leqslant$ `string` » ou « $\top \leqslant \bot$ ») sur lesquelles un ensemble d'axiomes spécifient si la relation de sous-typage est vérifiée ou non.

À l'inverse, le sous-typage sémantique va, en premier lieu, chercher à donner une « sémantique » aux types. Pour ce faire, le principe consiste à définir une « fonction d'interprétation », en général notée $(\llbracket \_ \rrbracket)$, associant un ensemble (une partie d'un certain domaine $\mathcal{D}$) à un type. La relation de sous-typage $(\leqslant)$ entre les types peut alors être définie via la relation d'inclusion sur les ensembles associés :

$$\tau_1 \leqslant \tau_2 \iff \llbracket \tau_1 \rrbracket \subset \llbracket \tau_2 \rrbracket$$

Les travaux de Haruo Hosoya et Benjamin Pierce [HP03, HP01], et en particulier leur travail sur XDuce, s'inscrivent dans la voie du sous-typage sémantique. Une limitation importante de leurs systèmes est l'absence de types fonctionnels. Cette restriction a été levée plus tard par Véronique Benzaken, Giuseppe Castagna et Alain Frisch ([CF05, FC08]) lors de leur travaux liés à $\mathbb{C}$Duce. Ils introduisent entre autres une technique générale permettant de définir la « fonction d'interprétation » sur des constructions de types plus complexes permettant en particulier de gérer le typage des fonctions $(\to)$, des références (`ref`) et de l'évaluation paresseuse (`lazy`).

Les travaux de Pottier en 2001 (cf. [P01]) s'inscrivent quant à eux dans la voie du sous-typage syntaxique en montrant différentes méthodes de simplification des ensembles de contraintes utilisant des mécanismes de résolution.

Les travaux de Gottlieb de 2011 (cf. [G11]) continuent dans cette direction. Ils montrent en particulier comment formaliser et implémenter simplement un système à base de sous-typage sur un langage muni de variants et d'enregistrements.

Stephen Dolan et Alan Mycroft (cf. [DM15]) ont suivi une approche très similaire récemment. En plus de conserver une notion de « type principal » et d'être implémentable, le système qu'ils présentent dans cet article met en évidence certains invariants concernant la forme de leurs contraintes et la position des opérateurs $(\sqcup)$ et $(\sqcap)$ dans leurs types. Ces invariants leur permettent de gagner en élégance en simplifiant leurs règles de typage et l'implémentation associée. Des propriétés très semblables apparaîtront dans nos systèmes. En effet, ces considérations se rapprochent beaucoup de la classification que nous introduirons entre nos types (τ^l et τ^r) et des propriétés de conservation les concernant lors de la saturation.

2.1.3 Manipulation de contraintes sans résolution

Une approche assez différente consiste à uniquement tenter de vérifier la cohérence entre les contraintes extraites lors du typage, sans chercher à les « résoudre ». Une telle approche ne cherche donc pas à faire disparaître les opérateurs logiques entre les contraintes au profit de

constructions de type, et a tendance à enrichir le langage des contraintes plutôt que le langage des types. Cette voie a en particulier été explorée dans les années 1995 par Trifonov et Smith (cf. [ES95, TS96]).

C'est cette direction que nous avons choisie de suivre, en premier lieu pour son élégance (ce qui est très subjectif), mais aussi parce que c'est la seule voie que nous ayons trouvée pour gérer proprement les disjonctions et les négations dans les ensembles de contraintes, et ainsi gagner une expressivité que les mécanismes de résolution nous interdisaient.

2.2 Formalisme utilisé

Le but de cette section est de présenter les mécanismes de base que nous utilisons pour définir les types et les systèmes associés. Les types manipulés dépendent bien entendu du système de types, ils seront donc enrichis au cours des chapitres de cette thèse.

2.2.1 Notre approche

Notre but dans cette thèse est d'étendre les mécanismes de sous-typage. Nous serons alors amenés à étendre le langage des contraintes avec des disjonctions, des négations, et d'autres formes de relations entre les types. Malheureusement, une extension du langage des types avec des unions, des intersections, top et bottom n'est plus suffisante pour représenter de telles contraintes de type. En conséquence, nous n'allons plus chercher à enrichir la grammaire des types pour supprimer les contraintes de sous-typage, mais au contraire, à l'épurer au maximum. À la place, la structure de base qui nous permettra de représenter un « ensemble de valeurs » sera un « schéma de type » et contiendra en particulier un ensemble de contraintes, mais dans un langage beaucoup plus riche.

Il est important de noter que la présence de sous-typage va faire intervenir une relation asymétrique entre les types que nous noterons « $\leqslant$ ». Cette relation représente l'inclusion des ensembles dénotés par les types qu'elle compare. De telles relations sont générées par les systèmes de types que nous allons définir. Cependant, les types qui seront générés à gauche d'un ($\leqslant$) n'ont pas exactement la même structure que ceux qui seront générés à droite. Cette différence sera très légère dans le système de base que nous décrivons dans ce chapitre, mais s'accentuera par la suite. Il serait bien évidemment possible de définir uniquement « l'ensemble de tous les types » comme l'union de l'ensemble des types pouvant apparaître à gauche d'un ($\leqslant$) et de ceux pouvant apparaître à droite. Une telle définition de la structure des types poserait néanmoins quelques problèmes d'élégance, à la fois dans le cadre d'une formalisation car certaines relations seraient syntaxiquement valides mais n'auraient aucun sens et ne pourraient jamais être générées ; mais aussi dans le cadre d'une implémentation car le code du typeur serait alors « pollué » par la gestion de cas impossibles pour des raisons algorithmiques.

Nous sommes donc amenés à définir deux ensembles de types :

$$
\begin{aligned}
\tau^l &\ \texttt{::=}\ \alpha \mid \texttt{(}\alpha_1\texttt{, ..., }\alpha_n\texttt{) t} \mid \texttt{K}\ \alpha \\
\tau^r &\ \texttt{::=}\ \alpha \mid \texttt{(}\alpha_1\texttt{, ..., }\alpha_n\texttt{) t} \\
&\quad \mid \texttt{\{ K}_1\ \alpha_1\ \texttt{‖ ... ‖ K}_n\ \alpha_n\ \texttt{\}} \\
&\quad \mid \texttt{\{ K}_1\ \alpha_1\ \texttt{‖ ... ‖ K}_n\ \alpha_n\ \texttt{‖}\ \alpha_d\ \texttt{\}}
\end{aligned}
$$

L'ensemble τ^l représente les types pouvant apparaître à gauche d'un ($\leqslant$) et τ^r ceux pouvant apparaître à droite.

D'après cette définition, un type est soit :

- une variable de type

- un constructeur de types paramétré par n variables de type (avec $n \geqslant 0$). Nous avons décidé de représenter tous les constructeurs de types de cette manière car ils sont manipulés uniformément dans les règles de saturation. Les constructeurs de types sont utilisés en particulier pour représenter :
  - les types prédéfinis associés aux constantes, typiquement d'arité 0, comme `int`, `bool`, `string`, etc.
  - le type $(\times)$ d'arité 2 utilisé pour les couples. Pour des raisons de lisibilité, on s'autorisera le raccourci de notation « $\alpha_1 \times \alpha_2$ » pour « $(\alpha_1, \alpha_2)(\times)$ »
  - le type $(\rightarrow)$ d'arité 2 utilisé pour les fonctions. De la même manière, on s'autorisera le raccourci de notation « $\alpha_1 \rightarrow \alpha_2$ » signifiant « $(\alpha_1, \alpha_2)(\rightarrow)$ ».
  - les types algébriques gardés (où « **GADT** ») définis par l'utilisateur (voir le chapitre 5).
- dans le cas τ^l, le type d'un variant polymorphe seul provenant de sa construction ; et dans le cas τ^r, un ensemble de variants polymorphes provenant d'une déconstruction via un filtrage de motifs.

Dans cette définition, la seule différence entre τ^l et τ^r concerne donc les types associés aux variants polymorphes, et cela pourrait passer pour un détail. Cependant, nous préférons introduire cette distinction entre τ^l et τ^r dès le départ car elle sera encore plus marquée dans le système de types du chapitre 4 sur la généralisation étendue (τ^l sera alors augmenté de « schémas de type » mais pas τ^r), et primordiale pour démontrer certaines propriétés dans le chapitre 5 sur les **GADT**.

En réalité, la distinction entre τ^l et τ^r est beaucoup plus profonde qu'une simple différence syntaxique. Ils sont chacun associé à un ensemble bien distinct de constructions du langage :

- Un type τ^l est toujours associé à la « construction » d'une valeur. Il en sera généré lors du typage des constantes, des couples, des fonctions, etc.
- Un type τ^r est quant à lui associé à la « déconstruction » d'une valeur. Il en sera généré lors du typage de l'application (impliquant la déconstruction de la fonction), de la conditionnelle (qui déconstruit le booléen associé au test), du filtrage de motifs, etc.

À différentes occasions, nous serons amenés à « renommer » les variables présentes dans les types. Pour ce faire, nous utiliserons la notation « $\tau[\alpha_1 \mapsto \alpha_2]$ » représentant le type τ dans lequel toutes les occurrences libres de α_1 ont été remplacées par α_2. Un renommage de variables peut être injectif (au sens où les variables ont des images deux à deux distinctes) ou non. Par défaut, dans ce manuscrit, le terme « renommage » désignera un renommage potentiellement non-injectif.

Une propriété notable des mécanismes de saturation de contraintes que nous présentons ici est que les types ne changent jamais de côté dans les comparaisons. Ainsi, si un type construit (c'est-à-dire autre qu'une variable de type) a été généré en tant que τ^l, il ne deviendra jamais un τ^r. De plus, la seule raison de « clash » de typage est lorsqu'un type construit τ^l est comparé avec un type construit τ^r incompatible. Autrement dit, un clash de typage ne peut apparaître que lorsque la construction d'une valeur se collisionne avec une déconstruction incompatible, par exemple lorsqu'une valeur est construite comme un couple puis utilisée en tant que fonction dans une application.

Cette propriété est très utile pour générer de « bons » messages d'erreur de sous-typage. En effet, il est alors possible d'associer à tout clash une position dans le code indiquant la construction d'une valeur, et une autre position indiquant l'endroit où elle est déconstruite de manière incompatible. Une version avancée peut même utiliser la chaîne de variables de type reliant ces deux types incompatibles, et la présenter au programmeur comme un « chemin d'exécution » invalide du programme.

Une telle propriété n'est néanmoins valide que dans un système à base de sous-typage. Un système à base d'unification ne vérifie pas cette propriété et des clash peuvent apparaître entre deux τ^l ou entre deux τ^r. Par exemple, l'expression (`if ... then 3 else "hello"`) sera typiquement rejetée par un système à base d'unification avec un clash `int`/`string` alors qu'il n'existe aucun chemin d'exécution reliant un `int` à une `string`.

Une autre différence importante entre notre définition de τ^l/τ^r et l'approche standard tient au fait que notre définition des types n'est pas récursive. Une telle définition a l'avantage de forcer les types et les schémas de type à rester sous une « forme normale ». De plus, cette représentation des types trivialise les preuves de terminaison des algorithmes de saturation car il suffit de borner le nombre de variables de type pour borner le nombre de types, et ainsi le nombre de contraintes de types. Cependant, cette définition nous empêche d'imbriquer des types les uns dans les autres, et donc en particulier d'écrire un type comme « $\alpha \to \alpha \to$ `int` ». Nous aurons donc besoin d'un « schéma de type » pour dénoter un tel ensemble.

2.2.2 Schémas de type

Dans ce chapitre, les contraintes de types que nous allons manipuler sont uniquement des relations de sous-typage. Elles seront enrichies par la suite avec d'autres sortes de relations entre les types. L'ensemble de ces contraintes, noté `C`, est donc défini pour l'instant par la grammaire suivante :

$$\texttt{C ::= } \tau^l \leqslant \tau^r$$

Pour le système de types de base, un ensemble de contraintes, noté Φ, se résume à une conjonction de `C` :

$$\Phi \texttt{ ::= \{ C}_1 \texttt{ } \wedge \texttt{ ... } \wedge \texttt{ C}_n \texttt{ \}}$$

La définition de Φ sera enrichie par la suite avec des disjonctions et des négations. Les Φ que nous manipulerons à partir de maintenant seront considérés comme des « ensembles » de contraintes dans lesquels l'ordre des relations dans la conjonction n'a pas d'importance. On s'autorisera en particulier les notations « $C \in \Phi$ » signifiant que l'un des membres de la conjonction Φ est égal à C, et « $\Phi_1 \subset \Phi_2$ » signifiant que tous les C de Φ_1 appartiennent à Φ_2.

Un schéma de type, quant à lui, est défini de la manière suivante :

$$\sigma \texttt{ ::= [} \forall\alpha_1 \texttt{ ... } \alpha_n \texttt{ . } \alpha \texttt{ | } \Phi \texttt{]}$$

Il contient :

- Un ensemble de n variables de type $\alpha_1 \ldots \alpha_n$: les variables « généralisées ».
- Une variable de type (appartenant à $\alpha_1 \ldots \alpha_n$ ou pas) : la racine du schéma.
- Un ensemble de contraintes Φ liant α avec les α_i, potentiellement d'autres variables de type et éventuellement des constructeurs de types et des variants.

Par exemple, l'ensemble des fonctions ayant deux paramètres d'un même type quelconque et renvoyant un entier est habituellement noté $(\alpha \to \alpha \to \texttt{int})$. Il sera ici représenté par le schéma de type :

$$\sigma = [\ \forall \alpha_1 \alpha_2 \alpha_3\ .\ \alpha \mid \{\ \alpha_1 \to \alpha_2 \leqslant \alpha \wedge \alpha_1 \to \alpha_3 \leqslant \alpha_2 \wedge \texttt{int} \leqslant \alpha_3\ \}\ ]$$

Un tel schéma de type est certes assez peu lisible. Son avantage est qu'il est basé sur des briques élémentaires très simples, ce qui facilite sa manipulation dans le système de types et dans les preuves. Pour être utilisable en pratique, une implémentation réelle d'un système de types basé sur une telle représentation devra définir une fonction d'affichage mettant les schémas de type sous une forme plus classique lorsqu'ils doivent être lus par un programmeur. Une telle technique est détaillée dans le chapitre 6 sur l'implémentation.

2.2.3 Environnement de typage

Le typage d'une expression se fait par un parcours récursif de l'arbre la représentant. Lors de la descente du typeur à travers un `let`, un λ ou un cas de filtrage, une nouvelle variable du programme est créée et peut alors apparaître dans le corps de la construction en question. Le typeur va donc être amené à stocker des contraintes de types concernant ces variables du programme dans une structure appelée « environnement de typage » et notée Γ. De telles contraintes sont simplement représentées par un schéma de type. L'ensemble Γ est donc défini par une liste de couples :

$$\Gamma\ ::= (\mathtt{x}_1,\ \sigma_1),\ \ldots,\ (\mathtt{x}_n,\ \sigma_n)$$

Nous définissons alors deux opérateurs sur Γ :

- L'opérateur d'ajout d'une nouvelle association $(\mathtt{x},\ \sigma)$ dans Γ, noté : « $\Gamma \oplus (\mathtt{x},\ \sigma)$ »
- L'accès à une variable `x` dans un environnement Γ, noté : « $\Gamma[\mathtt{x}]$ »

Lorsqu'un même `x` est lié à plusieurs σ dans Γ, la syntaxe $\Gamma[\mathtt{x}]$ donne accès au dernier σ ajouté pour `x`.

2.2.4 Règles d'inférence

Nous allons maintenant définir comment nous formalisons nos systèmes de types dans cette thèse. Les choix que nous faisons ici sont guidés par un but assez pragmatique : nous souhaitons que nos règles d'inférence soient transformables, de manière systématique, en un algorithme d'inférence.

Le but de l'inférence est, lorsque c'est possible, d'associer à une expression e un schéma de type σ représentant une approximation de l'ensemble des valeurs obtenues par évaluation de e. Chaque système de types que nous allons définir consistera en un ensemble de règles d'inférence. Il en existe trois catégories :

- Les « règles de typage », nommées «T...», de la forme :

$$\frac{\begin{array}{c}\text{T}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi, \Gamma \vdash \mathsf{e} : \alpha \rhd \Phi'}$$

  De telles règles se lisent de la manière suivante : étant donné un ensemble de contraintes Φ, dans l'environnement de typage Γ, pour que l'expression e soit de type α, Φ doit être enrichi avec de nouvelles contraintes pour obtenir Φ'.

- Les « règles de saturation », nommées «S...», de la forme :

$$\frac{\begin{array}{c}\text{S}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi \vdash \tau^l \leqslant \tau^r \rhd \Phi'} \qquad \text{et} \qquad \frac{\begin{array}{c}\text{S}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi \vdash \tau^l \leq \tau^r \rhd \Phi'}$$

  De telles règles se lisent : étant donné un ensemble de contraintes Φ, l'ajout de la contrainte de sous-typage $\tau^l \leqslant \tau^r$ nécessite d'enrichir Φ avec de nouvelles contraintes pour générer Φ'.

- Les « règles d'instanciation » de la forme :

$$\frac{\begin{array}{c}\text{I}\ldots\\ \ldots \quad \ldots \quad \ldots\end{array}}{\Phi \vdash \sigma \leqslant \tau^r \rhd \Phi'}$$

  Ces règles sont similaires aux règles de saturation, sauf qu'elles comparent un schéma de type avec un type plutôt que deux types entre eux.

L'algorithme d'inférence fonctionne de la manière suivante : à partir d'une expression e, nous générons une variable de type α et cherchons à construire un arbre d'inférence dont les noeuds sont des instances de nos règles de typage, de saturation et d'instanciation :

$$\dfrac{\dfrac{\ldots \quad \ldots}{\ldots} \quad \dfrac{\ldots \quad \ldots \quad \ldots}{\ldots} \qquad \dfrac{\dfrac{\ldots \quad \ldots}{\ldots} \quad \dfrac{\ldots \quad \ldots}{\ldots}}{\ldots} \qquad \dfrac{\dfrac{\ldots \quad \ldots}{\ldots} \quad \dfrac{\ldots \quad \ldots}{\ldots}}{\ldots}}{\varnothing, \varnothing \vdash \mathsf{e} : \alpha \rhd \Phi}$$

Lorsque la construction de cet arbre est impossible, l'expression e est dite « non-typable ». On dit alors qu'il se produit un « clash » de typage. Lorsque cet arbre est constructible, il représente une « preuve de typage » de notre expression e qui est alors dite typable. Nous extrayons alors de

cet arbre l'ensemble de contraintes de sous-typage Φ et l'ensemble $\{ \alpha_1, \ldots, \alpha_n \}$ des variables de type générées au cours de l'inférence. Le schéma de type inféré pour e représentant l'ensemble des valeurs possibles par évaluation e est alors :

$$\sigma = [\ \forall \alpha_1 \ \ldots \ \alpha_n \ .\ \alpha \mid \Phi\ ]$$

Le schéma de type σ généré par cet algorithme contient souvent des contraintes qui ne sont pas liées à α. Pour des raisons de performance, il peut être intéressant de le nettoyer et de le normaliser. Ce genre d'algorithme est détaillé dans le chapitre 6 sur l'implémentation.

La syntaxe de nos règles de typage se distingue des systèmes de types « standard » par le fait qu'elle impose de lier l'expression e à une variable de type et pas à un type comme habituellement. Toutes les contraintes de type associées à e sont en réalité regroupées dans Φ'. Il serait bien entendu possible d'autoriser la présence d'un τ^r à la place de α dans les règles de typage, mais ceci ne ferait que compliquer la forme des règles et n'apporterait aucune expressivité supplémentaire pour l'écriture de systèmes de types.

Nous remarquerons la présence de deux formes de règles de saturation différentes, chacune ayant en conclusion à une relation de sous-typage distincte : « $\leqslant$ » et « $\leq$ ». Il s'agit en réalité d'une simple astuce technique permettant de formaliser le « calcul de point fixe » effectué lors de la saturation en utilisant uniquement des « règles ». Il n'y a pas de « différence sémantique » entre ces deux relations. L'usage de deux relations permet simplement d'éviter de boucler lors de la saturation.

Le principe de la saturation est simple : lorsqu'une règle de typage, d'instanciation ou de saturation engendre une contrainte de sous-typage $(\tau_1 \leqslant \tau_2)$, si cette contrainte est déjà présente dans l'ensemble de contraintes Φ, on ne fait rien. Sinon, on l'ajoute à Φ et on génère une contrainte de la forme $(\tau_1 \leq \tau_2)$ qui est prise en charge par les autres règles de saturation.