Typage statique

Le but du *typage statique* est de détecter et de rejeter dès la compilation (jusqu'ici les erreurs étaient détectées à l'exécution) un certain nombre de programmes absurdes, comme 1 2 ou
$\mathrm{BinOp}\,(\mathrm{Add}, (\mathbf{fun}\,\mathrm{x} \to \mathrm{x}), 1)$.Pour cela un *type* est attribué à chaque sous-expression du programme (p.ex. *int* pour une expression arithmétique, ou $\mathit{int} \to \mathit{int}$ pour une fonction des entiers vers les entiers) et la *cohérence* de ces types est vérifiée.

Déterminer toutes les erreurs d'exécution pour tous les programmes est un problème indécidable. Or les systèmes de types sont souvent décidables car on souhaite que le compilateur termine pour tous les programmes. *Il est donc impossible de ne rejeter que les programmes erronés*. Tout système de types rejette des programmes innocents, c'est pourquoi la quête de meilleurs systèmes est sans fin.

Définition formelle des types de la calculette

- Syntaxe concrète

```
Type ::= "int" | Type -> Type | "(" Type ")"
       | "'a" | "'b" | ...
```

  On dénote les variables de type avec les méta-variables $\overline{\alpha}$, $\overline{\beta}$ etc.

- Syntaxe abstraite

  type *type_expr* = TEint | TEfun **of** *type_expr* * *type_expr* | TEvar **of** *string*;;

  On dénote les types avec la méta-variable τ.

Définition formelle des types de la calculette (suite)

- Analyse syntaxique
  La flèche est associative à droite.
  $\langle\langle int \rangle\rangle$ = TEint et $\langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle$ = TEfun $(\langle\langle \overline{\tau}_1 \rangle\rangle, \langle\langle \overline{\tau}_2 \rangle\rangle)$ et
  $\langle\langle$ 'a $\rangle\rangle$ = TEvar "a" etc.
  On note τ au lieu de $\langle\langle \overline{\tau} \rangle\rangle$. On note les variables de type α, β, γ etc. au lieu de TEvar "a", TEvar "b", TEvar "c" etc.
- Variables libres $\mathcal{L}\langle\langle int \rangle\rangle = \varnothing$ et $\mathcal{L}\langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle = \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2)$ et $\mathcal{L}\langle\langle \overline{\alpha} \rangle\rangle = \{\alpha\}$.

Système de types — typage monomorphe

Un jugement de typage est de la forme $\Gamma \vdash e : \tau$ et se lit « Dans l'environnement de typage Γ, l'expression e a pour type τ. » Un environnement de typage Γ lie des variables x à leur type $\Gamma(x)$. Un liaison de typage se note $x : \tau$. Soit

$$\Gamma \vdash \langle\!\langle \overline{n} \rangle\!\rangle : \langle\!\langle int \rangle\!\rangle \quad \text{Tconst} \qquad \frac{\Gamma \vdash e_1 : \langle\!\langle int \rangle\!\rangle \qquad \Gamma \vdash e_2 : \langle\!\langle int \rangle\!\rangle}{\Gamma \vdash \langle\!\langle \overline{e}_1 \ \overline{o} \ \overline{e}_2 \rangle\!\rangle : \langle\!\langle int \rangle\!\rangle} \ \text{Tbin}$$

$$\Gamma \vdash \langle\!\langle x \rangle\!\rangle : \Gamma(x) \quad \text{Tvar} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \oplus x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle : \tau_2} \ \text{Tlet}$$

Système de types — typage monomorphe (suite)

$$\frac{\Gamma \oplus x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \langle\langle \textbf{fun}\ x \to \overline{e} \rangle\rangle : \langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle}\ \mathsf{Tfun}$$

$$\frac{\Gamma \vdash e_1 : \langle\langle \overline{\tau}' \to \overline{\tau} \rangle\rangle \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \langle\langle \overline{e}_1\ \ \overline{e}_2 \rangle\rangle : \tau}\ \mathsf{Tapp}$$

Exemple de preuve de typage (ou dérivation de typage)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{x} : \langle\langle \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{x} \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle} \qquad \dfrac{}{\mathsf{x} : \langle\langle \mathit{int} \rangle\rangle \vdash \langle\langle 1 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\mathsf{x} : \langle\langle \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{x} + 1 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\varnothing \vdash \langle\langle \textbf{fun } \mathsf{x} \to \mathsf{x} + 1 \rangle\rangle : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle} \qquad \mathsf{f} : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{f} \rangle\rangle : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \qquad \mathsf{f} : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \vdash \langle\langle 2 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}{\mathsf{f} : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{f}\ 2 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\varnothing \vdash \langle\langle \textbf{let } \mathsf{f} = \textbf{fun } \mathsf{x} \to \mathsf{x} + 1 \textbf{ in } \mathsf{f}\ 2 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}$$

Voici d'autres jugements de typage dérivables :
$\varnothing \vdash \langle\langle \textbf{fun } \mathsf{x} \to \mathsf{x} \rangle\rangle : \langle\langle \overline{\alpha} \to \overline{\alpha} \rangle\rangle$ et $\varnothing \vdash \langle\langle \textbf{fun } \mathsf{x} \to \mathsf{x} \rangle\rangle : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle$
Voici des jugements non dérivables :
$\varnothing \vdash \langle\langle \textbf{fun } \mathsf{x} \to \mathsf{x} + 1 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle$ et $\varnothing \vdash \langle\langle \textbf{fun } \mathsf{x} \to \mathsf{x} + 1 \rangle\rangle : \langle\langle \overline{\alpha} \to \mathit{int} \rangle\rangle$

Quid de l'auto-application ?

Pour typer **fun** f $\to$ f f il faudrait construire une dérivation de la forme suivante :

$$\dfrac{\dfrac{\Gamma \oplus \mathsf{f} : \tau_1 \vdash \mathsf{f} : \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle \qquad \Gamma \oplus \mathsf{f} : \tau_1 \vdash \mathsf{f} : \tau_2}{\Gamma \oplus \mathsf{f} : \tau_1 \vdash \langle\!\langle \mathsf{f}\ \mathsf{f} \rangle\!\rangle : \tau_2}}{\Gamma \vdash \langle\!\langle \textbf{fun}\ \mathsf{f} \to \mathsf{f}\ \ \mathsf{f} \rangle\!\rangle : \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle}$$

Pour que les feuilles de la dérivation soient justifiée par l'axiome Tvar, il faudrait que $\tau_1 = \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle$ et $\tau_1 = \tau_2$. La première de ces égalités est impossible, car τ_1 serait un sous-terme strict de lui-même, ce qui est impossible pour tout terme τ_1 fini.

Quelques propriétés du typage

- Expressions bien typées ou typables
  Une expression e est typable s'il existe un environnement de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$
- Typage Un typage est une paire (Γ, τ) ou une dérivation de typage.
- Stabilité par substitution de variables de types
  Si on peut dériver un jugement non clos, c.-à-d. contenant des variables de types libres, p.ex. $\mathsf{f} : \langle\!\langle \overline{\alpha} \to \overline{\alpha} \rangle\!\rangle \oplus \mathsf{x} : \alpha \vdash \mathsf{f}(\mathsf{x}) : \alpha$, alors on peut aussi dériver tous les jugements obtenus en remplaçant ces variables par des types arbitraires, p.ex.
  $\mathsf{f} : \langle\!\langle \mathit{int} \to \mathit{int} \rangle\!\rangle \oplus \mathsf{x} : \mathit{int} \vdash \mathsf{f}(\mathsf{x}) : \mathit{int}$
- Sûreté du typage
  Si $\Gamma \vdash e : \tau$ et $\rho \vdash e \twoheadrightarrow r$ alors $r \neq \mathsf{Err}(\ldots)$
  Cette propriété est la motivation même du typage statique.

Normalisation forte

Le système de typage monomorphe présenté ici est *fortement normalisant*, c.-à-d. que les programmes bien typés terminent toujours.

Ces systèmes de types ne sont donc pas intéressants en programmation car on souhaite que le langage soit Turing-complet (p. 67), c.-à-d. qu'il accepte *tous* les programmes qui terminent. Si le système de types rejetait de plus tous les programmes qui ne terminent pas, on aurait résolu le problème de l'arrêt de la machine de Turing, qui est connu pour être indécidable (p. 69).

Tout langage Turing-complet muni d'un système de types décidable contient donc des programmes qui ne terminent pas.

Normalisation faible et opérateur de point fixe

C'est pourquoi on considère habituellement des systèmes de types qui ne garantissent pas que les programmes typables terminent (on parle de normalisation faible). Un bon exemple est le système de types monomorphe de mini-ML muni d'un opérateur de point fixe fix ou d'un **let rec** natif :

$$\frac{\Gamma \oplus x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma \oplus x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle : \tau_2} \text{ Tlet-rec}$$

Il n'y a ici aucune difficulté par rapport à Tlet, il faut juste ajouter la liaison $x : \tau_1$ dans l'environnement de typage de la première prémisse.

De la vérification pure à l'inférence complète

Pour un langage statiquement typé, un *typeur* est un algorithme qui, étant donné un programme, détermine si celui-ci est typable et, si oui, en produit un type. Si le programme admet plusieurs types, un typeur devra produire un *type principal*, c.-à-d. un type « plus général » que tous les autres types possibles (cette notion dépend du système de type considéré).

Exemple En mini-ML, le programme **fun** x $\rightarrow$ x a les types $\tau \rightarrow \tau$ pour n'importe quel type τ, mais le type $\alpha \rightarrow \alpha$ est principal, puisque tous les autres types s'en déduisent par substitution (de la variable α).

Selon la nature et la quantité d'informations, sous forme d'*annotations* de types, que le langage exige, la tâche du typeur est plus ou moins complexe. Il existe de nombreuses configurations.

Vérification pure

Dans le cas de la pure vérification de types, toutes les sous-expressions du programme, ainsi que tous les identificateurs, doivent être annotés par leur type.
Exemple

fun (x : *int*)→ (
    let y : *int* = (+ : *int* ∗ *int* → *int*) (x : *int*) (1 : *int*)
    in y : *int*
) : *int*

Le typeur est alors très simple, puisque le programmeur n'écrit en fait plus uniquement une expression, mais une dérivation de typage complète.

Bien sûr, un tel langage est inutilisable en pratique ; aucun langage réaliste n'adopte cette approche extrême.

Déclaration des types des variables et propagation des types

Le programmeur doit déclarer les types des paramètres de fonction et des variables locales. Le typeur infère alors le type de chaque expression à partir des types de ses sous-expressions. Autrement dit, l'information de typage est propagée à travers l'expression des feuilles vers la racine.

Exemple Sachant que x est de type *int*, le typeur peut non seulement vérifier que l'expression x + 1 est bien typée, mais aussi inférer qu'elle a le type *int*. Ainsi l'exemple précédent devient :

$$\textbf{fun}\ (\mathrm{x} : \mathit{int}) \rightarrow \textbf{let}\ \mathrm{y} : \mathit{int} = \mathrm{x} + 1\ \textbf{in}\ \mathrm{y}$$

Le typeur infère le type $\mathit{int} \rightarrow \mathit{int}$ pour cette expression.

Une approche similaire à celle-ci est adoptée par la plupart des langages impératifs, tels Pascal, C, Java etc.

Déclaration des types des paramètres et propagation des types

Le programmeur doit déclarer uniquement les types des paramètres. La différence par rapport au cas précédent est donc que les variables locales (liées par **let**) ne sont plus obligatoirement annotées lorsqu'elles sont introduites.

Le typeur en détermine alors le type en se fondant sur le type de l'expression à laquelle elles sont associées.

Exemple Notre exemple devient

$$\textbf{fun}\ (\mathrm{x} : \mathit{int}) \rightarrow \textbf{let}\ \mathrm{y} = \mathrm{x} + 1\ \textbf{in}\ \mathrm{y}$$

Ayant déterminé que x + 1 est de type *int*, le typeur associe le type *int* à y.

Inférence complète des types

Plus aucune annotation de types n'est exigée. Le typeur détermine le type des paramètres d'après l'utilisation qui en est faite par la fonction.

Exemple Notre exemple devient

$$\textbf{fun}\ \mathrm{x} \to \textbf{let}\ \mathrm{y} = \mathrm{x} + 1\ \textbf{in}\ \mathrm{y}$$

Puisque l'addition + n'opère que sur des entiers, x est nécessairement de type *int*. Cet intéressant processus de déduction, ou *inférence*, est celui utilisé par les langages de la famille ML (dont OCaml).

Inférence de types pour mini-ML avec typage monomorphe

Pour réaliser l'inférence de types pour mini-ML avec typage monomorphe, on procède en trois temps :

1. on annote l'arbre de syntaxe abstraite par des variables de types ;
2. à partir de cet arbre décoré on construit un système d'équations entre types, qui caractérise tous les typages possibles pour le programme ;
3. on résout ce système d'équations sachant que s'il n'a pas de solution le programme est alors mal typé, sinon on détermine une solution *principale* qui nous permet de déduire un typage principal du programme.

En combinant ces phases nous obtenons un algorithme qui détermine si un programme est typable, et, si oui, en fournit un typage principal.

Substitution

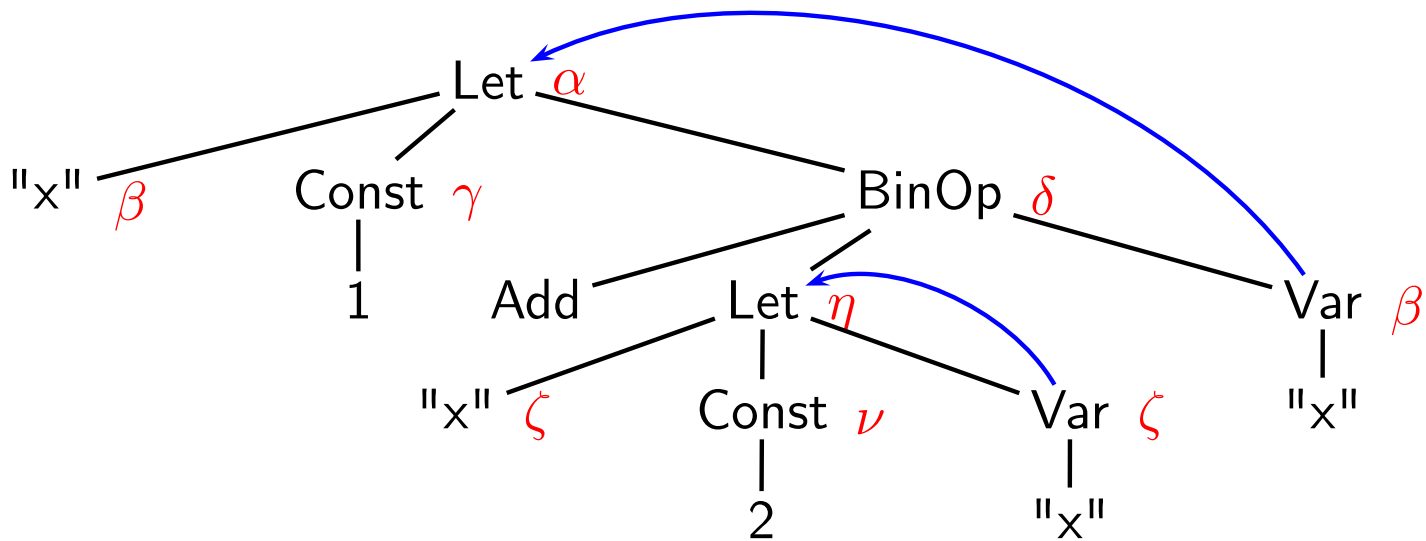
Nous avons besoin d'un nouveau concept pour décrire l'inférence de types : la *substitution*. C'est une fonction dont l'application a la forme générale $\tau[\alpha \leftarrow \tau']$, qu'on lit « La substitution de la variable de type α par le type τ' dans le type τ. » Elle se définit par induction sur la structure des types :

$$
\begin{aligned}
\langle\langle \mathit{int} \rangle\rangle[\alpha \leftarrow \tau'] &= \langle\langle \mathit{int} \rangle\rangle \\
\alpha[\alpha \leftarrow \tau'] &= \tau' \\
\beta[\alpha \leftarrow \tau'] &= \beta && \text{si } \beta \neq \alpha \\
\langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle[\alpha \leftarrow \tau'] &= \langle\langle \overline{\tau}'_1 \to \overline{\tau}'_2 \rangle\rangle && \text{où } \tau'_1 = \tau_1[\alpha \leftarrow \tau'] \text{ et } \tau'_2 = \tau_2[\alpha \leftarrow \tau']
\end{aligned}
$$

La notion de substitution peut s'étendre à d'autres objets, comme les expressions. On notera φ, ψ ou θ une substitution.

Décoration de l'arbre de syntaxe abstraite par des variables de types

Reprenons l'arbre de syntaxe abstraite page 51 et décorons-le avec des variables de type uniques respectant les liaisons, c'est-à-dire qu'une variable liée est annotée par la même variable de type que celle de son lieur et les autres expressions sont annotées par des variables uniques :



Notons en exposant d'une expression la variable de type qui l'annote, par exemple $\text{Let}^{\alpha}(x^{\beta}, {e_1}^{\gamma}, {e_2}^{\delta})$.

Construction du système d'équations (collection de contraintes)

À partir d'une expression annotée e^{α} nous construisons un système d'équations $C(e^{\alpha})$ capturant les contraintes de typage entre les sous-expressions de e. Ce système est défini inductivement sur la structure de e :

$$
\begin{aligned}
C(\mathsf{Var}^{\alpha}\, x) &= \varnothing \\
C(\mathsf{Const}^{\alpha}\, n) &= \{\alpha = \langle\!\langle \mathit{int} \rangle\!\rangle\} \\
C(\mathsf{BinOp}^{\alpha}(\_, {e_1}^{\beta}, {e_2}^{\gamma})) &= \{\alpha = \langle\!\langle \mathit{int} \rangle\!\rangle; \beta = \alpha; \gamma = \alpha\} \\
&\quad \cup\, C({e_1}^{\beta}) \cup C({e_2}^{\gamma}) \\
C(\mathsf{Let}^{\alpha}(x^{\beta}, {e_1}^{\gamma}, {e_2}^{\delta}) &= \{\beta = \gamma; \alpha = \delta\} \cup C({e_1}^{\gamma}) \cup C({e_2}^{\delta}) \\
C(\mathsf{Fun}^{\alpha}(x^{\beta}, e^{\gamma})) &= \{\alpha = \langle\!\langle \overline{\beta} \to \overline{\gamma} \rangle\!\rangle\} \cup C(e^{\gamma}) \\
C(\mathsf{App}^{\alpha}({e_1}^{\beta}, {e_2}^{\gamma})) &= \{\beta = \langle\!\langle \overline{\gamma} \to \overline{\alpha} \rangle\!\rangle\} \cup C({e_1}^{\beta}) \cup C({e_2}^{\gamma})
\end{aligned}
$$

Construction du système d'équations (exemple)

Poursuivons avec l'exemple précédent. Nous avons alors :

$$
\begin{aligned}
C(e^{\alpha}) = \quad \{ \quad & \beta = \gamma; \alpha = \delta; \\
& \gamma = \langle\langle int \rangle\rangle; \\
& \delta = \langle\langle int \rangle\rangle; \eta = \delta; \beta = \delta; \\
& \zeta = \nu; \eta = \zeta; \\
& \nu = \langle\langle int \rangle\rangle \}
\end{aligned}
$$

Lien entre jugements de typage et solutions des équations

Une *solution* de l'ensemble d'équations $C(e^\alpha)$ est une substitution φ telle que pour toute équation $\tau_1 = \tau_2 \in C(e^\alpha)$ on a $\varphi(\tau_1) = \varphi(\tau_2)$. Autrement dit, une solution est un *unificateur* du système d'équations. Les propositions suivantes établissent que les solutions de $C(e^\alpha)$ caractérisent exactement les typages de e.

Proposition (Correction des équations)

Si φ est une solution de $C(e^\alpha)$ alors $\Gamma \vdash e^\alpha : \varphi(\alpha)$, où Γ est l'environnement de typage $\{x^\beta : \varphi(\beta) \mid x^\beta \in \mathcal{L}(e)\}$.

Proposition (Complétude des équations)

Soit e une expression. S'il existe un environnement de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$, alors le système d'équations $C(e^\alpha)$ admet une solution φ telle que $\varphi(\alpha) = \tau$ et $\Gamma = \{x^\beta : \varphi(\beta) \mid x^\beta \in \mathcal{L}(e)\}$.

Résolution des équations (l'unificateur de Robinson)

- Définissons $\varphi \leqslant \psi$ s'il existe une substitution θ telle que $\psi = \theta \circ \varphi$.
- Par définition, une solution φ de $C(e^\alpha)$ est dite *principale* si toute solution ψ de $C(e^\alpha)$ vérifie $\varphi \leqslant \psi$.
- Il existe un algorithme mgu qui, étant donné un système d'équations C, soit échoue soit produit une solution principale de C :

$$\begin{aligned}
\mathsf{mgu}(\varnothing) &=_1 \forall x.x \mapsto x \\
\mathsf{mgu}(\{\tau = \tau\} \cup C') &=_2 \mathsf{mgu}(C') \\
\mathsf{mgu}(\{\alpha = \tau\} \cup C') &=_3 \mathsf{mgu}(C'[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \notin \mathcal{L}(\tau) \\
\mathsf{mgu}(\{\tau = \alpha\} \cup C') &=_4 \mathsf{mgu}(\{\alpha = \tau\} \cup C') \\
\mathsf{mgu}(C \cup C') &=_5 \mathsf{mgu}(\{\tau_1 = \tau_1'; \tau_2 = \tau_2'\} \cup C')
\end{aligned}$$

where $C = \{\langle\langle \overline{\tau_1} \to \overline{\tau_2} \rangle\rangle = \langle\langle \overline{\tau_1}' \to \overline{\tau_2}' \rangle\rangle\}$

Dans tous les autres cas, mgu échoue (pas de solutions).

Résolution des équations (exemple)

$$\mathsf{mgu}(C(e^{\alpha})) =_3 \varphi_0 \circ [\beta \leftarrow \gamma]$$

$$\begin{aligned}
\varphi_0 &= \mathsf{mgu}(\{\alpha = \delta; \gamma = \langle\!\langle int \rangle\!\rangle; \delta = \langle\!\langle int \rangle\!\rangle; \eta = \delta; \\
&\qquad\qquad \gamma = \delta; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle int \rangle\!\rangle\}) \\
&=_3 \varphi_1 \circ [\alpha \leftarrow \delta] \\
\varphi_1 &= \mathsf{mgu}(\{\gamma = \langle\!\langle int \rangle\!\rangle; \delta = \langle\!\langle int \rangle\!\rangle; \eta = \delta; \gamma = \delta; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle int \rangle\!\rangle\}) \\
&=_3 \varphi_2 \circ [\gamma \leftarrow \langle\!\langle int \rangle\!\rangle] \\
\varphi_3 &= \mathsf{mgu}(\{\delta = \langle\!\langle int \rangle\!\rangle; \eta = \delta; \langle\!\langle int \rangle\!\rangle = \delta; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle int \rangle\!\rangle\}) \\
&=_3 \varphi_4 \circ [\delta \leftarrow \langle\!\langle int \rangle\!\rangle] \\
\varphi_4 &= \mathsf{mgu}(\{\eta = \langle\!\langle int \rangle\!\rangle; \langle\!\langle int \rangle\!\rangle = \langle\!\langle int \rangle\!\rangle; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle int \rangle\!\rangle\}) \\
&=_3 \varphi_5 \circ [\eta \leftarrow \langle\!\langle int \rangle\!\rangle] \\
\varphi_5 &= \mathsf{mgu}(\{\langle\!\langle int \rangle\!\rangle = \langle\!\langle int \rangle\!\rangle; \zeta = \nu; \langle\!\langle int \rangle\!\rangle = \zeta; \nu = \langle\!\langle int \rangle\!\rangle\})
\end{aligned}$$

Résolution des équations (fin de l'exemple)

$$
\begin{array}{rcl}
\varphi_5 & =_2 & \mathsf{mgu}(\{\zeta = \nu; \langle\!\langle \mathit{int} \rangle\!\rangle = \zeta; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
 & =_3 & \varphi_6 \circ [\zeta \leftarrow \nu] \\
\varphi_6 & = & \mathsf{mgu}(\{\langle\!\langle \mathit{int} \rangle\!\rangle = \nu; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
 & =_4 & \varphi_5 \circ [\nu \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \\
\varphi_5 & = & \mathsf{mgu}(\{\langle\!\langle \mathit{int} \rangle\!\rangle = \langle\!\langle \mathit{int} \rangle\!\rangle; \langle\!\langle \mathit{int} \rangle\!\rangle = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
 & = & \mathsf{mgu}(\{\langle\!\langle \mathit{int} \rangle\!\rangle = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
 & =_2 & \mathsf{mgu}(\varnothing) \ =_1 \ \forall x.x \mapsto x \\
\mathsf{mgu}(C(e^\alpha)) & = & [\nu \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \circ [\zeta \leftarrow \nu] \circ [\eta \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \circ [\delta \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \\
 & & \circ [\gamma \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \circ [\alpha \leftarrow \delta] \circ [\beta \leftarrow \gamma] \\
\mathsf{mgu}(C(e^\alpha))(\alpha) & = & \alpha([\delta \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \circ [\alpha \leftarrow \delta]) = \delta[\delta \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] = \langle\!\langle \mathit{int} \rangle\!\rangle
\end{array}
$$

L'algorithme d'inférence et ses propriétés

Proposition (Correction de mgu)

Si mgu$(C) = \varphi$ *alors* φ *est une solution de* C*.*

Proposition (Complétude de mgu)

Si C *admet une solution* ψ *alors mgu*(C) *réussit et produit une solution* φ *telle que* $\varphi \leqslant \psi$*.*