Inférence de types nullables (version longue)

Michel Mauny & Benoît Vaugon

ENSTA-ParisTech, Université Paris-Saclay,
U2IS, Équipe Sûreté et Fiabilité des Logiciels,
828, boulevard des Maréchaux, F-91762 Palaiseau Cedex

Résumé

De nombreux langages de programmation utilisent une valeur spéciale pour représenter un résultat indéterminé ou la valeur d'une variable non initialisée. Afin d'éviter les erreurs classiques résultant de la confusion entre cette valeur (nommée `NULL`, `nil`, `None`) et une valeur définie, dans certains langages (comme Hack ou Swift, pour ne citer que deux langages de conception industrielle récente), la présence possible d'une valeur indéfinie est détectée au moment du typage. Les types contenant une valeur indéfinie sont alors dits *nullables*. *A contrario*, un type non nullable ne contient que des valeurs définies qui peuvent donc être utilisées librement pour des opérations classiques (arithmétiques pour des valeurs numériques, dé-référencements pour des pointeurs, appels pour des valeurs fonctionnelles, etc.). Nous présentons dans cet article un algorithme d'inférence de types pour un langage statiquement typé, d'ordre supérieur, doté de types nullables. Nous décrivons d'abord deux techniques, la première étant fondée sur un système de types très simple, et la seconde utilisant un encodage à base des variants polymorphes du langage OCaml. Nous soulignons les faiblesses de ces deux approches et introduisons un algorithme d'inférence, fondé sur la collection et la résolution de contraintes de sous-typage, qui ne présente pas les faiblesses des deux systèmes précédents. Cette inférence a été prouvée correcte et nous donnons ici le schéma de cette preuve.

1. Types nullables

Les langages de programmation impératifs, comme C ou les dérivés de Java, utilisent abondamment la constante `NULL`, ou bien comme représentation d'une référence inconnue, ou alors comme valeur d'un calcul qui échoue. À l'exécution : un simple test suffit à détecter la « `NULL`-ité » d'une valeur.

Cependant, sans autre mécanisme offert par le langage, la confusion entre `NULL` et une valeur légale est une source classique d'erreurs [8]. Les dispositifs permettant d'éviter ou de détecter ces erreurs potentielles à la compilation relèvent de l'analyse statique des programmes. Nous nous intéressons dans cet article au *typage statique*, et plus précisément à l'*inférence de types*, qui permet au programmeur de bénéficier de la sûreté inhérente au typage statique tout en ne s'embarrassant pas de l'insertion d'annotations de typage.

Un type *nullable* est constitué des valeurs d'un type classique, augmenté d'une valeur spéciale représentant l'indéfini. On distingue trois façons de typer les valeurs potentiellement indéfinies selon les langages de programmation qui le proposent :

1. à la Java : `null` est une référence compatible, du point de vue du typage, avec toute autre référence et son déréférencement provoque dynamiquement une levée d'exception ;
2. à la OCaml (ou Haskell) : le type (α `option`) d'OCaml est une union discriminée résultant de l'injection des valeurs du type argument à l'aide du constructeur `Some(_)` et de la constante `None`. Les valeurs injectées par `Some(_)` sont empaquetées [1] afin de permettre de distinguer dynamiquement, par filtrage, la représentation de `None` de celle de `Some(_)`.

1. *boxed*, en anglais : stockées dans un bloc mémoire fraîchement alloué.

3. à la Swift (ou Hack) : la constante `nil` étend un type t en un type $t?$ distinct de t. Dans ce cas, une valeur de type t est automatiquement vue comme pouvant être de type $t?$, et ce, sans empaquetage.

Remarquons que l'empaquetage réalisé par OCaml ou Haskell (cas 2 ci-dessus) permet de distinguer aisément les valeurs `None` et `Some(None)`, ou, plus généralement, les différentes valeurs $\texttt{Some}^n\texttt{(None)}$, pour $n \geqslant 0$. Remarquons aussi que l'absence d'empaquetage dans le cas 3 ne permet pas *a priori* de construire le type $t??$: on ne peut en effet distinguer dynamiquement le `nil` de cet hypothétique $t??$ de celui de $t?$, sauf à disposer de suffisamment de constantes `nil` distinctes. Sauf mention du contraire, la suite de cet article se place dans le cas 3 où les types t et $t?$ sont incompatibles, mais où une valeur de type t peut aussi être considérée du type $t?$, lorsque c'est nécessaire.

Nous nous proposons dans cet article d'étudier l'inférence de types nullables au travers d'un petit langage fonctionnel donné à la figure 1. Ce langage est un mini-ML classique, étendu avec une constante `NULL`, et un test dédié, noté $\texttt{case}\ e_1\ \texttt{of NULL} \to e_2 \parallel x \to e_3$, qui produit la valeur de e_2 lorsque e_1 vaut `NULL`, et, sinon, lie la valeur de e_1 à x et produit la valeur de e_3.

$$
\begin{array}{lcl}
c & ::= & () \mid \mathit{true} \mid \mathit{false} \mid 0 \mid 1 \mid 2 \mid \ldots \mid (+) \mid \ldots \\
v & ::= & c \mid \lambda x.e \mid \texttt{NULL} \\
e & ::= & v \mid x \mid e_1\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{case}\ e_1\ \texttt{of NULL} \to e_2 \parallel x \to e_3
\end{array}
$$

FIGURE 1 – Le langage

La section 2 considère d'abord une approche naïve, où les types τ associés aux expressions e, sont des paires (t, ν) constituées d'un type habituel t et d'une information de « nullabilité » ν. Un type $(t, ?)$ représente alors des valeurs qui *peuvent* être `NULL`, alors que (t, Δ) représente des valeurs qui *ne peuvent pas* être `NULL`. Les variables de nullabilité sont notées δ. Ce système nous sert à introduire le formalisme que nous utilisons dans la suite de cet article pour écrire nos algorithmes d'inférence.

La section 3 montre une traduction de valeurs nullables en variants polymorphes à la OCaml : le typage (par le compilateur OCaml) des programmes produits souffre des même faiblesses que notre premier système.

La section 4 présente un mécanisme de typage plus sophistiqué, où l'unification est remplacée par la résolution de contraintes de sous-typage, et la section 5 en donne une preuve de correction sémantique.

2. Typage par unification

Nous présentons d'abord un système de typage assez simple, où les types portent une information de « nullabilité » qui indique si la valeur de l'expression peut être `NULL` ou non.

$$
\begin{array}{lcllcl}
\tau & ::= & (t, \nu) & \quad t & ::= & t_b \mid \alpha \mid \tau_1 \to \tau_2 \mid \mu\alpha.\tau \\
t_b & ::= & \mathit{unit} \mid \mathit{bool} \mid \mathit{int} & \quad \nu & ::= & ? \mid \delta \mid \Delta
\end{array}
$$

FIGURE 2 – Les types du système à base d'unification

Les jugements de ce système sont de la forme :

$$\Phi, \Gamma \vdash e : (t, \nu) \rhd \Phi'$$

où Φ et Φ' sont des substitutions, Γ est un environnement de typage qui à des identificateurs associe des schémas de types, c'est-à-dire des types avec une quantification prénexe de variables de types et de nullabilité. Un tel jugement se lit : *étant donné* Φ*, sous l'hypothèse* Γ*, l'expression* e *a pour type* (t, ν) *avec la substitution* Φ'.

Les règles de typage données à la figure 3 sont les différents cas d'un algorithme qui, étant donnés Φ, Γ, e, et τ, calcule la substitution Φ' nécessaire à la production d'un type $\Phi'(\tau)$ pour e. Plus précisément, sous

TConst
$$\frac{\Phi \vdash \tau = T(c) \rhd \Phi'}{\Phi, \Gamma \vdash c : \tau \rhd \Phi'}$$

TInstVar
$$\frac{\Phi \vdash \tau' = \tau \rhd \Phi'}{\Phi, \Gamma \oplus \mathtt{x} : \tau' \vdash \mathtt{x} : \tau \rhd \Phi'}$$

TInst(α)
$$\frac{\textbf{soit } \alpha' \textbf{ fraîche} \qquad \Phi, \Gamma \oplus \mathtt{x} : \sigma[\alpha := \alpha'] \vdash \mathtt{x} : \tau \rhd \Phi'}{\Phi, \Gamma \oplus \mathtt{x} : \forall \alpha.\sigma \vdash \mathtt{x} : \tau \rhd \Phi'}$$

TInst(δ)
$$\frac{\textbf{soit } \delta' \textbf{ fraîche} \qquad \Phi, \Gamma \oplus \mathtt{x} : \sigma[\delta := \delta'] \vdash \mathtt{x} : \tau \rhd \Phi'}{\Phi, \Gamma \oplus \mathtt{x} : \forall \delta.\sigma \vdash \mathtt{x} : \tau \rhd \Phi'}$$

TLambda
$$\frac{\textbf{soit } \alpha_1, \delta_1, \alpha_2, \delta_2 \textbf{ fraîches} \qquad \Phi, \Gamma \oplus x : (\alpha_1, \delta_1) \vdash e : (\alpha_2, \delta_2) \rhd \Phi' \qquad \Phi' \vdash \tau = (\alpha_1, \delta_1) \to (\alpha_2, \delta_2) \rhd \Phi''}{\Phi, \Gamma \vdash \lambda x.e : \tau \rhd \Phi''}$$

TApp
$$\frac{\textbf{soit } \alpha, \delta \textbf{ fraîche} \qquad \Phi, \Gamma \vdash e_1 : ((\alpha, \delta) \to \tau), \Delta) \rhd \Phi' \qquad \Phi', \Gamma \vdash e_2 : (\alpha, \delta) \rhd \Phi''}{\Phi, \Gamma \vdash e_1\ e_2 : \tau \rhd \Phi''}$$

TLet
$$\frac{\textbf{soit } \alpha, \delta \textbf{ fraîche} \qquad \Phi, \Gamma \vdash e_1 : (\alpha, \delta) \rhd \Phi' \qquad \Phi', \Gamma \oplus x : gen(\Phi', \Gamma, (\alpha, \delta)) \vdash e_2 : \tau \rhd \Phi''}{\Phi, \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau \rhd \Phi''}$$

TIfThenElse
$$\frac{\Phi, \Gamma \vdash e_1 : (bool, \Delta) \rhd \Phi' \qquad \Phi', \Gamma \vdash e_2 : \tau \rhd \Phi'' \qquad \Phi'', \Gamma \vdash e_3 : \tau \rhd \Phi'''}{\Phi, \Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau \rhd \Phi'''}$$

TNull
$$\frac{\textbf{soit } \alpha \textbf{ fraîche} \qquad \Phi \vdash \tau = (\alpha, ?) \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{NULL} : \tau \rhd \Phi'}$$

TCase
$$\frac{\textbf{soit } \delta \textbf{ fraîche} \qquad \Phi, \Gamma \vdash e_1 : (t, ?) \rhd \Phi' \qquad \Phi', \Gamma \vdash e_2 : \tau \rhd \Phi'' \qquad \Phi'', \Gamma \oplus x : \forall \delta.(t, \delta) \vdash e_3 : \tau \rhd \Phi'''}{\Phi, \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ \mathtt{NULL} \to e_2 \parallel x \to e_3 : \tau \rhd \Phi'''}$$

Figure 3 – Règles de typage (à base d'unification). Définissent les jugements $\Phi, \Gamma \vdash e : \tau \rhd \Phi'$

les hypothèses $\Phi'(\Gamma)$, l'expression e a pour type $\Phi'(\tau)$. La règle de typage de la construction `let` , source de polymorphisme, généralise les variables de type et de nullabilité qui ne sont pas libres dans l'environnement ; les schémas de type sont donc de la forme :

$$\sigma ::= \forall \alpha.\sigma \mid \forall \delta.\sigma \mid \tau$$

La fonction de généralisation, classique, est donnée à la figure 4. On y note $FTV(\sigma)$ (resp. $FTV(\Gamma)$) l'ensemble des variables de type qui apparaissent libres dans σ (resp. Γ).

$$gen(\Phi, \Gamma, \tau) = gen'(\Phi(\Gamma), \Phi(\tau))$$

$$\begin{array}{ll} gen'(\Gamma, \sigma) = gen'(\Gamma, \forall \alpha.\sigma) & \textbf{lorsque } \alpha \in FTV(\sigma) \wedge \alpha \notin FTV(\Gamma) \\ gen'(\Gamma, \sigma) = gen'(\Gamma, \forall \delta.\sigma) & \textbf{lorsque } \delta \in FTV(\sigma) \wedge \delta \notin FTV(\Gamma) \\ gen'(\Gamma, \sigma) = \sigma & \textbf{sinon} \end{array}$$

Figure 4 – Généralisation (système à base d'unification)

Puisque nous avons deux sortes de variables, nous avons aussi deux mécanismes d'instanciation qui sont donnés dans des règles distinctes : TInst(α) s'occupe des variables de types universellement quantifiées, et TInst(δ) instancie les variables de nullabilité.

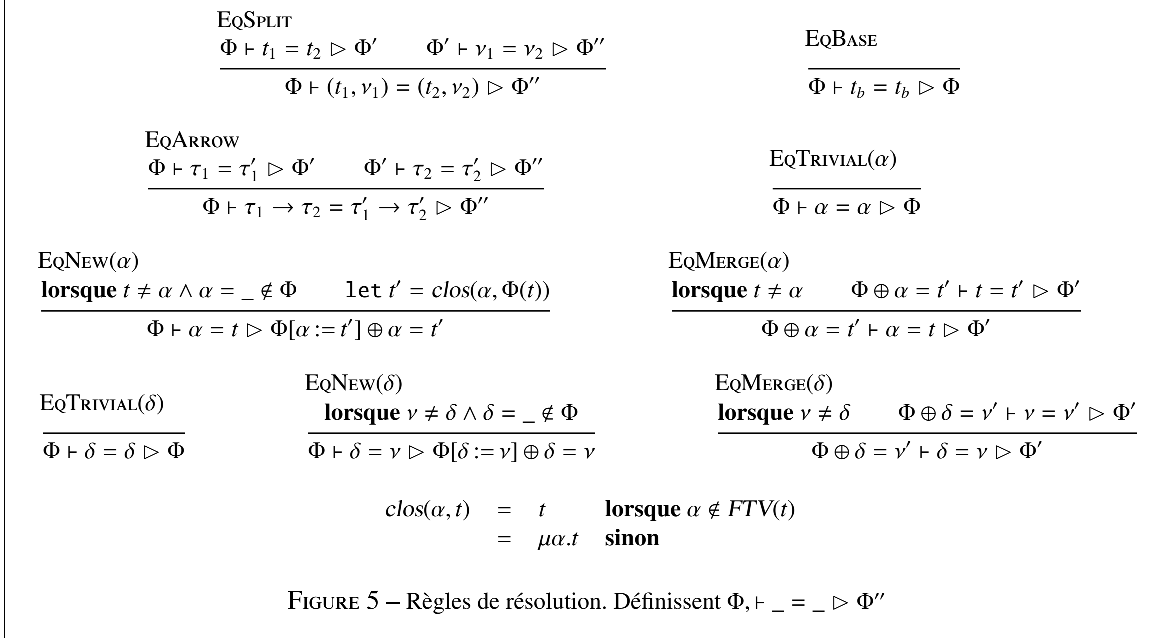

FIGURE 5 – Règles de résolution. Définissent $\Phi, \vdash _ = _ \rhd \Phi''$

Parmi les règles de la figure 3, il est intéressant de s'attarder sur la règle TCASE, qui, en généralisant la nullabilité de e_1, autorise x à être considéré à la fois comme défini (de type (t, Δ), donc non-NULL) ou nullable (de type $(t, ?)$) durant le typage de e_3. Ainsi, par exemple, les occurrences de x dans e_3 peuvent être passé indifféremment à des fonctions attendant des valeurs définies ou bien à d'autres, acceptant des arguments nullables.

Les contraintes d'égalité de types sont introduites par les règles de typage et sont résolues par un ensemble de règles donné à la figure 5. Les seules règles de résolution réellement intéressantes sont celles qui introduisent (EQNEW) ou fusionnent (EQMERGE) les images de variables dans la substitution Φ. EQNEW(α) installe une liaison $\alpha = t'$ dans la substitution $\Phi[\alpha := t']$ (c'est-à-dire Φ dans laquelle les occurrences libres de α sont changées en t') lorsqu'il n'existe pas de liaison précédente de α dans Φ. Ici, t' est de la forme $\mu\alpha.t$ ou simplement t, selon qu'α apparaît ou non dans t. EQNEW(δ) a le même effet sur les variables de nullabilité, dans un contexte plus simple.

EQMERGE(α) fusionne une nouvelle contrainte $\alpha = t'$ avec une autre $\alpha = t$ apparaissant dans Φ. Ici, la substitution résultante Φ' est obtenue par résolution de la contrainte $t = t'$. EQMERGE(δ) effectue la même opération pour les variables de nullabilité.

En dépit de sa simplicité et de sa praticité apparente, ce typage impose un style de programmation où `NULL` reste aussi isolé que possible du reste des programmes. Dans un langage essentiellement fonctionnel, comme ML, où `NULL` (ou une valeur similaire telle que `None`) n'est pas utilisé aussi couramment qu'en C ou Java, cela peut être acceptable. Néanmoins, on peut désirer accepter comme typable le programme suivant (que nous nommerons P par la suite) : Ce programme n'est pas typable car le paramètre k doit être à la fois de type (`int`,Δ) pour être additionné et de type (`int`,?) pour être compatible avec `NULL` au sortir de la conditionnelle.

3. Encodage de la nullabilité par des variants polymorphes

Le typage de la nullabilité pour notre langage peut être facilement réalisé par un typeur de variants polymorphes tel que ceux offerts par le langage OCaml. Il suffit de traduire `NULL` en le variant `‘None`, et d'empaqueter les autres valeurs par le variant `‘Some(_)`. La traduction d'une application vérifie que l'expression en position fonction représente bien une fonction définie (empaquetée par `‘Some(_)`), et un test de nullabilité

$$
\begin{array}{lcl}
[\![\, \texttt{NULL} \,]\!] & = & \texttt{`None} \\
[\![\, x \,]\!] & = & x \\
[\![\, c \,]\!] & = & \texttt{`Some}(c) \\
[\![\, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,]\!] & = & \texttt{if}\,(\texttt{match}\,[\![\, e_1 \,]\!]\,\texttt{with}\ \texttt{`Some}(b) \to b)\ \ \texttt{then}\,[\![\, e_2 \,]\!]\,\texttt{else}\,[\![\, e_3 \,]\!] \\
[\![\, \lambda x.e \,]\!] & = & \texttt{`Some}(\lambda x.[\![\, e \,]\!]) \\
[\![\, \texttt{case}\ e_1\ \texttt{of NULL} \to e_2 \parallel x \to e_3 \,]\!] & & \\
 & = & \texttt{match}\,[\![\, e_1 \,]\!]\,\texttt{with}\ \texttt{`None}\ \to [\![\, e_2 \,]\!] \mid \texttt{`Some}(x) \to (\lambda x.[\![\, e_3 \,]\!])(\texttt{`Some}(x)) \\
[\![\, e_1\ e_2 \,]\!] & = & (\texttt{match}\,[\![\, e_1 \,]\!]\,\texttt{with}\ \texttt{`Some}(f) \to f)\,[\![\, e_2 \,]\!] \\
[\![\, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \,]\!] & = & \texttt{let}\ x = [\![\, e_1 \,]\!]\ \texttt{in}\ [\![\, e_2 \,]\!] \qquad \text{(trivialement optimisé de } [\![\, (\lambda x.e_2)\ e_1 \,]\!])
\end{array}
$$

FIGURE 6 – Encodage de NULL par des variants

est traduit en un filtrage qui extrait la valeur empaquetée dans le cas non-NULL.

Par exemple, le programme suivant :

```
case read_int () of
| NULL -> failwith "not an integer"
| n -> n + 42
```

se traduira en OCaml en :

```
let (+) = `Some (fun n -> `Some (fun p ->
  match n, p with `Some n, `Some p -> `Some (n+p)))
let read_int = `Some (fun n ->
  match n with `Some () -> try `Some (read_int ()) with _ -> `None)
let failwith = `Some (fun msg ->
  match msg with `Some msg -> `Some (failwith msg))

match (match read_int with `Some f -> f) (`Some ()) with
| `None -> (match failwith with `Some f -> f) (`Some "not an integer")
| `Some n ->
    (fun n -> (match (match (+) with `Some f -> f) n with `Some f -> f)
               (`Some 42))
    (`Some n)
```

L'algorithme complet de traduction est donné à la figure 6. La typabilité des valeurs nullables peut être ainsi (trivialement) réduite en celle des variants polymorphes.

Cependant, comme on peut s'y attendre, la traduction du programme (P) de la section précédente ne passe pas l'épreuve du typage en OCaml :

```
let f = `Some (fun b -> `Some (fun k ->
  let p = (match (match (+) with `Some f0 -> f0) k with
            `Some f0 -> f0) (`Some 1) in
  if (match b with `Some b0 -> b0)
  then k else `None)) in ...
               ^^^^^
Error: this expression has type [> `None ] but an
expression was expected of type [< `Some of int ].
```

En effet, l'inférence de types de variants polymorphes utilise l'unification, même s'il simule une forme de sous-typage à l'aide d'une riche algèbre de types [7]. C'est le caractère symétrique de l'unification qui empêche le typage de tels programmes. Pour les accepter, il est nécessaire d'utiliser un mécanisme de typage plus puissant que l'unification : le sous-typage.

4. Une approche avec sous-typage

$$\textsc{TConst}\quad \frac{\Phi \vdash T(c) \leqslant \tau \rhd \Phi'}{\Phi, \Gamma \vdash c : \tau \rhd \Phi'}$$

$$\textsc{TInst}\quad \frac{\textbf{soit } \{\, \alpha'_k \,\}_1^n \textbf{ fraîches} \qquad \Phi, \Phi''[\alpha_k := \alpha'_k]_1^n \vdash \tau[\alpha_k := \alpha'_k]_1^n \leqslant \tau' \rhd \Phi'}{\Phi, \Gamma \oplus x : \forall \{\, \alpha_k \,\}_1^n[\Phi''].\tau \vdash x : \tau' \rhd \Phi'}$$

$$\textsc{TLambda}\quad \frac{\textbf{soit } \alpha_1, \alpha_2 \textbf{ fraîche} \qquad \Phi, \Gamma \oplus x : \alpha_1 \vdash e : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \alpha_1 \to \alpha_2 \leqslant \tau \rhd \Phi''}{\Phi, \Gamma \vdash \lambda x.e : \tau \rhd \Phi''}$$

$$\textsc{TApp}\quad \frac{\textbf{soit } \alpha \textbf{ fraîche} \qquad \Phi, \Gamma \vdash e_1 : \alpha \to \tau \rhd \Phi' \qquad \Phi', \Gamma \vdash e_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash e_1\ e_2 : \tau \rhd \Phi''}$$

$$\textsc{TLet}\quad \frac{\textbf{soit } \alpha \textbf{ fraîche} \qquad \Phi, \Gamma \vdash e_1 : \alpha \rhd \Phi' \qquad \Phi', \Gamma \oplus x : gen(\Phi', \Gamma, \alpha) \vdash e_2 : \tau \rhd \Phi''}{\Phi, \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \rhd \Phi''}$$

$$\textsc{TIfThenElse}\quad \frac{\Phi, \Gamma \vdash e_1 : bool \rhd \Phi_1 \qquad \Phi_1, \Gamma \vdash e_2 : \tau \rhd \Phi_2 \qquad \Phi_2, \Gamma \vdash e_3 : \tau \rhd \Phi_3}{\Phi, \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \rhd \Phi_3}$$

$$\textsc{TNull}\quad \frac{\textbf{soit } \alpha \textbf{ fraîche} \qquad \Phi \vdash \alpha? \leqslant \tau \rhd \Phi'}{\Phi, \Gamma \vdash \texttt{NULL} : \tau \rhd \Phi'}$$

$$\textsc{TCase}\quad \frac{\textbf{soit } \alpha \textbf{ fraîche} \qquad \Phi_0, \Gamma \vdash e_1 : \alpha? \rhd \Phi_1 \qquad \Phi_1, \Gamma \vdash e_2 : \tau \rhd \Phi_2 \qquad \Phi_2, \Gamma \oplus x : \alpha \vdash e_3 : \tau \rhd \Phi_3}{\Phi_0, \Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ \texttt{NULL} \to e_2 \parallel x \to e_3 : \tau \rhd \Phi_3}$$

FIGURE 7 – Règles de sous-typage. Définissent les jugements de la forme $\Phi, \Gamma \vdash e : \tau \rhd \Phi'$

(a) Comparaison, règles principales :

$$\textsc{LEQNew}\quad \frac{\textbf{lorsque } \tau_1 \leqslant \tau_2 \notin \Phi \qquad \Phi, \tau_1 \leqslant \tau_2 \vdash \tau_1 \leq \tau_2 \rhd \Phi'}{\Phi \vdash \tau_1 \leqslant \tau_2 \rhd \Phi'}$$

$$\textsc{LEQAlreadyProved}\quad \frac{}{\Phi, \tau_1 \leqslant \tau_2 \vdash \tau_1 \leqslant \tau_2 \rhd \Phi, \tau_1 \leqslant \tau_2}$$

FIGURE 8 – Sous-typage : règles de comparaison (1/2). Définissent $\Phi \vdash \tau \leq \tau' \rhd \Phi'$

Nous avons vu dans l'exemple précédent que la propagation d'informations « en arrière » dans l'environnement de typage interdit des programmes que l'on aimerait pouvoir accepter. Sur cet exemple, le type du paramètre `k` pourrait en effet ne pas être *identifié* avec le type de l'expression conditionnelle : il serait suffisant de garantir que le second peut inclure le premier, ou, en d'autres termes, que le type de `k` est un sous-type du type de retour de la fonction. L'algèbre de types pour ce nouveau système est la suivante :

$$t ::= t_b \mid \alpha \mid \tau_1 \to \tau_2 \qquad\qquad \tau ::= t \mid t?$$

Remplacer l'unification, qui résout des contraintes d'égalité, par la résolution de contraintes de sous-typage donne accès, en acceptant plus de programmes, à des styles de programmation plus naturels. Bien sûr, la résolution de contraintes de sous-typage peut échouer sur des problèmes où l'unification échouerait elle aussi, puisque certaines contraintes, se présentant comme des doubles inégalités sont *de facto* des contraintes d'égalité, comme celles qui sont produites lorsqu'on tente de typer `(1 + "hello")`. De plus, certaines contraintes de sous-typage sont clairement non satisfiables, comme par exemple celles produites durant le typage d'une conditionnelle `(if NULL then ... else ...)`, où on échoue à prouver $\alpha? \leqslant$ `bool`, ou bien d'une application `NULL(...)`, où on ne peut prouver $\alpha? \leqslant \tau_1 \to \tau_2$. Enfin, de nombreuses opérations primitives, comme l'addition entière, n'acceptent pas d'argument nullable.

(b) Comparaison, règles standard :

$$\textsc{LeqBaseTy}\quad \frac{}{\Phi \vdash t_b \leq t_b \rhd \Phi}$$

$$\textsc{LeqArrow}\quad \frac{\Phi \vdash \tau_1' \leqslant \tau_1 \rhd \Phi' \qquad \Phi' \vdash \tau_2 \leqslant \tau_2' \rhd \Phi''}{\Phi \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \rhd \Phi''}$$

$$\textsc{LeqBaseNull}\quad \frac{}{\Phi \vdash t_b \leq t_b? \rhd \Phi}$$

$$\textsc{LeqArrowNull}\quad \frac{\Phi \vdash \tau_1' \leqslant \tau_1 \rhd \Phi' \qquad \Phi' \vdash \tau_2 \leqslant \tau_2' \rhd \Phi''}{\Phi \vdash \tau_1 \to \tau_2 \leq (\tau_1' \to \tau_2')? \rhd \Phi''}$$

(c) Comparaison entre types et variables :

$$\textsc{LeqSameVar}\quad \frac{}{\Phi \vdash \alpha \leq \alpha \rhd \Phi}$$

$$\textsc{LeqVarLeqTy}\quad \frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \leqslant \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \leqslant \tau \vdash \alpha \leq \tau' \rhd \Phi''}$$

$$\textsc{GeqVarLeqTy}\quad \frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau' \leqslant \tau \notin \Phi \qquad \Phi, \alpha \leqslant \tau, \tau' \leqslant \tau \vdash \tau' \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau' \leq \tau \rhd \Phi''}{\Phi, \alpha \leqslant \tau \vdash \tau' \leq \alpha \rhd \Phi''}$$

$$\textsc{LeqTyLeqVar}\quad \frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \leqslant \tau' \notin \Phi \qquad \Phi, \tau \leqslant \alpha, \tau \leqslant \tau' \vdash \alpha \leq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \leq \tau' \rhd \Phi''}{\Phi, \tau \leqslant \alpha \vdash \alpha \leq \tau' \rhd \Phi''}$$

$$\textsc{GeqTyLeqVar}\quad \frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \tau \leqslant \alpha, \tau \approx \tau' \vdash \tau' \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \tau \leqslant \alpha \vdash \tau' \leq \alpha \rhd \Phi''}$$

$$\textsc{LeqVarCptTy}\quad \frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \approx \tau \vdash \alpha \leq \tau' \rhd \Phi''}$$

$$\textsc{GeqVarCptTy}\quad \frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \tau' \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \approx \tau \vdash \tau' \leq \alpha \rhd \Phi''}$$

$$\textsc{LeqVarEnd}\quad \frac{\begin{array}{c}\textbf{lorsque } \tau' \neq \alpha \\ \textbf{et } (\forall \tau \mid \alpha \leqslant \tau \in \Phi \implies \tau \approx \tau' \in \Phi) \quad \textbf{et } (\forall \tau \mid \tau \leqslant \alpha \in \Phi \implies \tau \leqslant \tau' \in \Phi) \quad \textbf{et } (\forall \tau \mid \alpha \approx \tau \in \Phi \implies \tau \approx \tau' \in \Phi)\end{array}}{\Phi \vdash \alpha \leq \tau' \rhd \Phi}$$

$$\textsc{GeqVarEnd}\quad \frac{\begin{array}{c}\textbf{lorsque } \tau' \neq \alpha \\ \textbf{et } (\forall \tau \mid \alpha \leqslant \tau \in \Phi \implies \tau' \leqslant \tau \in \Phi) \quad \textbf{et } (\forall \tau \mid \tau \leqslant \alpha \in \Phi \implies \tau \approx \tau' \in \Phi) \quad \textbf{et } (\forall \tau \mid \alpha \approx \tau \in \Phi \implies \tau \approx \tau' \in \Phi)\end{array}}{\Phi \vdash \tau' \leq \alpha \rhd \Phi}$$

FIGURE 9 – Sous-typage : règles de comparaison (2/2). Définissent $\Phi \vdash \tau \leq \tau' \rhd \Phi'$

Le nouvel ensemble de règles de typage est donné à la figure 7. La différence essentielle entre ce système et notre système initial se situe dans la règle TApp, où le domaine de la fonction est contraint à être « plus grand » que le type de l'argument pour que ce dernier soit accepté par la fonction. Intuitivement, une fonction

(a) Compatibilité, règles principales :

CPTNew
$$\frac{\textbf{lorsque } \tau_1 \approx \tau_2 \notin \Phi \qquad \Phi, \tau_1 \approx \tau_2 \vdash \tau_1 \simeq \tau_2 \rhd \Phi'}{\Phi \vdash \tau_1 \approx \tau_2 \rhd \Phi'}$$

CPTAlreadyProved
$$\frac{}{\Phi, \tau_1 \approx \tau_2 \vdash \tau_1 \approx \tau_2 \rhd \Phi, \tau_1 \approx \tau_2}$$

(b) Compatibilité : règles standard :

CptBaseTy
$$\frac{}{\Phi \vdash t_b \simeq t_b \rhd \Phi}$$

CptArrow
$$\frac{\Phi \vdash \tau_1 \approx \tau_1' \rhd \Phi' \qquad \Phi' \vdash \tau_2 \approx \tau_2' \rhd \Phi''}{\Phi \vdash \tau_1 \to \tau_2 \simeq \tau_1' \to \tau_2' \rhd \Phi''}$$

CptBaseNull
$$\frac{}{\Phi \vdash t_b \simeq t_b? \rhd \Phi}$$

CptArrowNull
$$\frac{\Phi \vdash \tau_1 \approx \tau_1' \rhd \Phi' \qquad \Phi' \vdash \tau_2 \approx \tau_2' \rhd \Phi''}{\Phi \vdash \tau_1 \to \tau_2 \simeq (\tau_1' \to \tau_2')? \rhd \Phi''}$$

(c) Compatibilité entre types et variables :

CptSameVar
$$\frac{}{\Phi \vdash \alpha \simeq \alpha \rhd \Phi}$$

CptVarLeqTy
$$\frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \leqslant \tau, \tau \approx \tau' \vdash \alpha \simeq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \leqslant \tau \vdash \alpha \simeq \tau' \rhd \Phi''}$$

CptTyLeqVar
$$\frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \tau \leqslant \alpha, \tau \approx \tau' \vdash \alpha \simeq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \tau \leqslant \alpha \vdash \alpha \simeq \tau' \rhd \Phi''}$$

CptVarCptTy
$$\frac{\textbf{lorsque } \tau' \neq \alpha \textbf{ et } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \alpha \simeq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \approx \tau \vdash \alpha \simeq \tau' \rhd \Phi''}$$

CptVarEnd
$$\frac{\begin{array}{c}\textbf{lorsque } \tau' \neq \alpha \\ \textbf{et}(\forall \tau \mid \alpha \leqslant \tau \in \Phi \implies \tau \approx \tau' \in \Phi) \quad \textbf{et } (\forall \tau \mid \tau \leqslant \alpha \in \Phi \implies \tau \approx \tau' \in \Phi) \quad \textbf{et } (\forall \tau \mid \alpha \approx \tau' \in \Phi \implies \tau \approx \tau' \in \Phi)\end{array}}{\Phi \vdash \alpha \simeq \tau' \rhd \Phi}$$

Figure 10 – Sous-typage : règles de compatibilité. Définissent $\Phi \vdash \tau_1 \approx \tau_2 \rhd \Phi'$, et $\Phi \vdash \tau_1 \simeq \tau_2 \rhd \Phi'$

$$\begin{array}{ll}
gen(\Phi, \Gamma, \tau) = \forall \{ \alpha_k \} [\Phi'].\tau & \textbf{où } \{ \alpha_k \} = gen_vars(\Phi, \Gamma, \tau) \\
 & \textbf{et } \Phi' = \Phi|_{\{ \alpha_k \}} \\
gen_vars(\Phi, \Gamma, \tau) = \bigcup_{\alpha \in gen_FTV(\Phi,\Gamma,\tau)} deps^*(\Phi, \{ \alpha \}) & \\
gen_FTV(\Phi, \Gamma, \tau) = \{ \alpha \mid \alpha \in FTV(\tau) \wedge deps^*(\Phi, \{ \alpha \}) \cap FTV(\Gamma) = \emptyset \} & \\
deps^*(\Phi, A) = A & \textbf{lorsque } deps(\Phi, A) = A \\
deps^*(\Phi, A) = deps^*(\Phi, deps(\Phi, A)) & \textbf{sinon} \\
deps(\Phi, A) = \bigcup_{\alpha \in A} dep_tvars(\Phi, \alpha) & \\
dep_tvars(\Phi, \alpha) = \bigcup_{\tau \in dep_tys(\Phi,\alpha)} FTV(\tau) & \\
dep_tys(\Phi, \alpha) = \{ \tau \mid \alpha \leqslant \tau \in \Phi \vee \tau \leqslant \alpha \in \Phi \vee \alpha \approx \tau \in \Phi \} &
\end{array}$$

Figure 11 – Sous-typage : généralisation

qui accepte un argument nullable accepte *a fortiori* un argument dont on a prouvé qu'il ne peut être `NULL`.

Les contraintes d'inégalité, notées $\tau_1 \leqslant \tau_2$, sont intégrées dans le composant Φ des règles de typage par

les règles de résolutiondonnées aux figures 8 et 9 . Les notations $\tau_1 \leqslant \tau_2$ et $\tau_1 \leq \tau_2$ ne diffèrent que pour des raisons techniques : pour prouver $\tau_1 \leqslant \tau_2$, on installe $\tau_1 \leqslant \tau_2$ dans Φ et on cherche à prouver $\tau_1 \leq \tau_2$ qui va décomposer les types τ_1 et τ_2 et saturer Φ. Si la preuve de $\tau_1 \leq \tau_2$ nécessite de prouver à nouveau $\tau_1 \leqslant \tau_2$, l'axiome LEQALREADYPROVED le fera immédiatement. Au moment de la résolution, la cohérence des contraintes nouvellement intégrées avec celles déjà présentes dans Φ est vérifiée. En particulier, la résolution procède aux vérifications basiques de sous-typage par le biais des règles LEQBASENULL et LEQARROWNULL, données à la figure 8(b).

Quand une variable de type α doit être « plus petite » (resp. « plus grande ») que deux types τ_1 et τ_2, les τ_i deviennent contraints à être « compatibles » (voir figure 10), c'est-à-dire à différer seulement par leurs annotations “?” (possiblement internes). Pour ce faire, nous engendrons des contraintes de compatibilité $\approx$ et $\simeq$, qui servent à rejeter des programmes qui produisent des contraintes qu'aucune valeur ne pourra satisfaire. Par exemple, aucune valeur de type τ ne pourra satisfaire simultanément les deux contraintes $\tau \leqslant \tau_1 \rightarrow \tau_2$ et $\tau \leqslant int$. La différence entre $\approx$ et $\simeq$ est purement technique, et est de même nature que celle entre $\leqslant$ et $\leq$.

La généralisation (figure 11) quantifie universellement les variables de types α ainsi que la fermeture transitive des contraintes qui leur sont associées, que l'on note $\Phi|_\alpha$, quand l'ensemble $\{ \alpha \} \cup FTV(\Phi|_\alpha)$ n'intersecte pas l'ensemble $FTV(\Gamma)$ des variables de types apparaissant libres dans Γ. Au moment de l'instanciation, une instance « fraîche » de contraintes est ré-injectée dans Φ.

Un autre changement important dans le nouveau système concerne la conditionnelle, la sélection de cas et plus généralement les constructions qui relèvent du filtrage. Dans ces cas, au lieu d'unifier les types des branches de retour, chacun d'eux se voit contraint d'être « plus petit » que le type de la construction elle-même. Cela force ainsi tous les types des branches à être compatibles entre elles, mais pas plus. Voir par exemple les règles TIFTHENELSE et TCASE sur la figure 7. C'est là la raison précise qui fait que l'exemple (P) donné à la fin de la section 2 est accepté par ce nouveau système.

5. Validité de l'inférence avec sous-typage

Nous énonçons et démontrons dans cette section la validité du système présenté à la section 4.

5.1. Sémantique du langage

Nous décrivons ici formellement la sémantique dynamique de notre langage. Les définitions des constantes c, des valeurs v et des expressions e sont données à la figure 1.

Definition 1 ($e_1 \longrightarrow e_2$) *On note $e_1 \longrightarrow e_2$ l'évaluation de e_1 en e_2 par un pas d'évaluation. Il s'agit simplement d'une fonction partielle allant de l'ensemble e des expressions dans lui même. La flèche ($\longrightarrow$) est définie au cas par cas ainsi :*

$$\begin{array}{rcl}
(\lambda x.e)\ v & \longrightarrow & e[x := v] \\
\texttt{let}\ x = v\ \texttt{in}\ e & \longrightarrow & e[x := v] \\
\texttt{if}\ \mathit{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_1 \\
\texttt{if}\ \mathit{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_2 \\
\texttt{case}\ \texttt{NULL}\ \texttt{of}\ \texttt{NULL} \rightarrow e_1 \parallel x \rightarrow e_2 & \longrightarrow & e_1 \\
\texttt{case}\ \ c\ \ \texttt{of}\ \texttt{NULL} \rightarrow e_1 \parallel x \rightarrow e_2 & \longrightarrow & e_2[x := c] \\
\texttt{case}\ \lambda x.e\ \texttt{of}\ \texttt{NULL} \rightarrow e_1 \parallel x \rightarrow e_2 & \longrightarrow & e_2[x := \lambda x.e]
\end{array}$$

Definition 2 ($E[]$) *Dans la suite, E représente un* ***contexte d'évaluation*** *et est défini par :*

$$E ::= [] \mid E\ e \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{case}\ E\ \texttt{of}\ \texttt{NULL} \rightarrow e_1 \parallel x \rightarrow e_2$$

Definition 3 ($e_1 \longmapsto e_2$) *À partir de E et de la flèche ($\longrightarrow$), nous construisons la flèche ($\longmapsto$) permettant d'évaluer une expression en évaluant une de ses sous-expression, identifiée par le contexte E. Ce dernier est construit de sorte que la définition de ($\longmapsto$) soit déterministe :*

$$E[e_1] \longmapsto E[e_2] \quad ssi \quad e_1 \longrightarrow e_2$$

Definition 4 ($e_1 \longmapsto\!\!\!\!\!\twoheadrightarrow e_2$) *On définit l'**évaluation de plusieurs pas** d'une expression e_1 en une expression e_2, que l'on note $e_1 \longmapsto\!\!\!\!\!\twoheadrightarrow e_2$, comme la fermeture réflexive et transitive de $\longmapsto$.*

$$e \longmapsto\!\!\!\!\!\twoheadrightarrow e \qquad \frac{e_1 \longmapsto e_2}{e_1 \longmapsto\!\!\!\!\!\twoheadrightarrow e_2} \qquad \frac{e_1 \longmapsto\!\!\!\!\!\twoheadrightarrow e_2 \quad e_2 \longmapsto\!\!\!\!\!\twoheadrightarrow e_3}{e_1 \longmapsto\!\!\!\!\!\twoheadrightarrow e_3}$$

Definition 5 (expression bloquée) *Une expression e est dite **bloquée** si e n'est pas une valeur et s'il n'existe aucune expression e' telle que $e \longmapsto e'$.*

Definition 6 ($e \Uparrow$) *L'évaluation d'une expression e boucle indéfiniment (ce que nous noterons $e \Uparrow$) s'il n'existe aucune expression bloquée e_b telle que $e \longmapsto\!\!\!\!\!\twoheadrightarrow e_b$ et aucune valeur v telle que $e \longmapsto\!\!\!\!\!\twoheadrightarrow v$.*

5.2. Théorème de validité

Notre langage est maintenant doté d'une sémantique formelle et d'un système de types, présenté sous la forme d'un mécanisme déterministe, implantable en algorithme, qui termine et qui permet d'accepter ou refuser un programme. La validité de ce mécanisme est sa cohérence avec la sémantique du langage, et signifie intuitivement qu'un programme bien typé ne peut jamais provoquer d'erreur de typage à l'exécution. Plus précisément, le théorème que nous cherchons à démontrer s'énonce comme suit :

Si $\emptyset, \emptyset \vdash e : \tau \rhd \Phi$ alors e vérifie une et une seule des propriétés suivantes :

- *L'évaluation de e boucle indéfiniment ($e \Uparrow$)*
- *Il existe une valeur v telle que $e \longmapsto\!\!\!\!\!\twoheadrightarrow v$*

5.3. Démonstration

À cause de la présence de `NULL`, de nouveaux types d'erreurs à l'exécution doivent être considérées comme l'utilisation `NULL` en tant qu'opérande d'une opération arithmétique, ou en tant que fonction lors d'une application. Nous commençons par donner quelques définitions et lemmes utiles à la preuve, puis démontrons les deux lemmes principaux :

- lemme 6 : une expression bien typée reste bien typée après un pas d'évaluation.
- lemme 9 : une expression bloquée n'est pas typable.

De ces deux lemmes se déduit le lemme 10 d'évaluation uniforme dont le théorème de validité est un simple corollaire.

Lemme 1 (Inclusion des Φ) *Si $\Phi, \Gamma \vdash e : \tau \rhd \Phi'$ ou $\Phi \vdash \tau_1 \mathcal{R}\, \tau_2 \rhd \Phi'$ (avec $\mathcal{R} \in \{\leqslant, \leq, \geqslant, \geq, \approx, \simeq\}$) alors $\Phi \subset \Phi'$.*

Preuve Aucune règle de typage ne supprime de relation (que ce soit une relation de comparaison ou de compatibilité) de Φ. Seules les règles TInst, LEQNew et CPTNew modifient Φ en y **ajoutant** de nouvelles relations.

Definition 7 (Saturation des Φ) *Un Φ est dit **saturé** si pour tout α, τ et τ', toutes les propriétés suivantes sont vérifiées :*

$$\left.\begin{array}{l}\bullet\ \ \alpha \leqslant \tau' \in \Phi \ \wedge\ \alpha \leqslant \tau \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi \\ \bullet\ \ \alpha \leqslant \tau' \in \Phi \ \wedge\ \tau \leqslant \alpha \in \Phi \ \Rightarrow\ \tau \leqslant \tau' \in \Phi \\ \bullet\ \ \alpha \leqslant \tau' \in \Phi \ \wedge\ \alpha \approx \tau \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi\end{array}\right\}\ \textit{(cf. }\textsc{LeqVarEnd}\textit{, figure 9)}$$

$$\left.\begin{array}{l}\bullet\ \ \tau' \leqslant \alpha \in \Phi \ \wedge\ \alpha \leqslant \tau \in \Phi \ \Rightarrow\ \tau' \leqslant \tau \in \Phi \\ \bullet\ \ \tau' \leqslant \alpha \in \Phi \ \wedge\ \tau \leqslant \alpha \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi \\ \bullet\ \ \tau' \leqslant \alpha \in \Phi \ \wedge\ \alpha \approx \tau \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi\end{array}\right\}\ \textit{(cf. }\textsc{GeqVarEnd}\textit{, figure 9)}$$

$$\left.\begin{array}{l}\bullet\ \ \alpha \approx \tau' \in \Phi \ \wedge\ \alpha \leqslant \tau \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi \\ \bullet\ \ \alpha \approx \tau' \in \Phi \ \wedge\ \tau \leqslant \alpha \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi \\ \bullet\ \ \alpha \approx \tau' \in \Phi \ \wedge\ \alpha \approx \tau \in \Phi \ \Rightarrow\ \tau \approx \tau' \in \Phi\end{array}\right\}\ \textit{(cf. }\textsc{CptVarEnd}\textit{, figure 10)}$$

Lemme 2 (Préservation de la saturation des Φ) *Si Φ est saturé et $\Phi, \Gamma \vdash e : \tau \rhd \Phi'$, alors Φ' est saturé.*

Preuve Considérons l'arbre de typage de $\Phi, \Gamma \vdash e : \tau \rhd \Phi'$. Des nœuds de type TINST de cet arbre peuvent ajouter des relations de comparaison ou de compatibilité dans Φ en provenance de schémas de type de Γ. Par construction, chaque schéma de type présent dans Γ est obtenu par généralisation d'un type. La fonction de généralisation extrait un ensemble saturé de relations de Φ. Leur réintroduction dans Φ ne casse donc pas la saturation de Φ.

L'unique autre règle ajoutant une relation de comparaison (resp. de compatibilité) C dans Φ est LEQNEW (resp. CPTNEW). Dans l'arbre de typage, les feuilles droites des sous-arbres correspondant à chaque utilisation de la règle LEQNEW (resp. CPTNEW) sont des LEQVAREND ou GEQVAREND (resp. CPTVAREND). Les règles nommées ...END imposent, *via* les contraintes qu'elles mentionnent dans leurs prémisses, la saturation du Φ généré par l'ajout de C.

Enfin, le lemme 1 (inclusion des Φ) nous assure que le Φ' généré contient tous les Φ intermédiaires présents dans l'arbre de typage, il est donc saturé vis à vis de toutes les relations de comparaison et de compatibilité, qu'elles soient introduites par l'utilisation des règles LEQNEW, CPTNEW ou TINST.

Definition 8 ($\Phi_1 \geqslant_R \Phi_2$) *On dit que Φ_1 est **plus grand que** Φ_2 **modulo** R (ce que l'on note $\Phi_1 \geqslant_R \Phi_2$) si R est une fonction de renommage des variables de type non nécessairement injective*[2] *telle que $\Phi_1 \supset R(\Phi_2)$.*

Lemme 3 (Transitivité de $\geqslant_R$) *Si $\Phi_1 \geqslant_{R_2} \Phi_2$ et $\Phi_2 \geqslant_{R_3} \Phi_3$ alors $\Phi_1 \geqslant_{R_2 \circ R_3} \Phi_3$.*

Preuve Puisque $\Phi_1 \geqslant_{R_2} \Phi_2$, par définition, on a $\Phi_1 \supset R_2(\Phi_2)$. Puisque $\Phi_2 \geqslant_{R_3} \Phi_3$, par définition, on a $\Phi_2 \supset R_3(\Phi_3)$. On en déduit que $R_2(\Phi_2) \supset R_2(R_3(\Phi_3))$. Par transitivité de $\supset$, on obtient $\Phi_1 \supset R_2(R_3(\Phi_3))$, ce qui signifie bien $\Phi_1 \geqslant_{R_2 \circ R_3} \Phi_3$.

Definition 9 ($\sigma_1 \geqslant \sigma_2$) *Un schéma de type $\forall \Phi_1.\tau_1$ est **plus grand** qu'un autre schéma de type $\forall \Phi_2.\tau_2$ s'il existe une fonction de renommage (noté R) des variables de type, non nécessairement injective, telle que $\Phi_1 \geqslant_R \Phi_2$ et $\tau_1 = R(\tau_2)$.*

Definition 10 ($\Gamma_1 \geqslant \Gamma_2$) *Un environnement Γ_1 est **plus grand** qu'un autre environnement Γ_2 si les variables du programme liées dans Γ_1 et Γ_2 sont exactement les mêmes, et si pour toute variable x de cette sorte, on a $\Gamma_1(x) \geqslant \Gamma_2(x)$.*

Lemme 4 (Réduction des contraintes par les règles de résolution) *Si Φ_1 et Φ_2 sont saturés et vérifient $\Phi_1 \geqslant_R \Phi_2$, et si $\Phi_1 \vdash R(\tau)\ \mathcal{R}\ R(\tau') \rhd \Phi'_1$ avec $\mathcal{R} \in \{\leqslant, \leq, \geqslant, \geq, \approx, \simeq\}$ alors $\Phi_2 \vdash \tau\ \mathcal{R}\ \tau' \rhd \Phi'_2$ et il existe R' tel que $\Phi'_1 \geqslant_{R'} \Phi'_2$.*

Preuve L'arbre de typage (que nous nommerons T_2) de $\Phi_2 \vdash \tau\ \mathcal{R}\ \tau' \rhd \Phi'_2$ est obtenu à partir de l'arbre de typage T_1 de $\Phi_1 \vdash R(\tau)\ \mathcal{R}\ R(\tau') \rhd \Phi'_1$ en supprimant certains sous-arbres de T_1 concernant des contraintes de

2. Il peut donc exister α_1 et α_2 distincts tels que $R(\alpha_1) = R(\alpha_2)$.

Φ_1 ayant disparu dans Φ_2, et en répliquant certains sous-arbres concernant des contraintes sur des variables de type ayant la même image par R.

Considérons l'ensemble $\mathcal{N}$ des nœuds de T_1 étiquetés LeqVarLeqTy, LeqTyLeqVar, LeqVarCptTy, GeqVarLeqTy, GeqTyLeqVar, GeqVarCptTy, CptVarLeqTy, CptTyLeqVar ou CptVarCptTy qui référencent une comparaison ou une compatibilité de $\Phi_1 \setminus R(\Phi_2)$. Pour chaque nœud $N \in \mathcal{N}$, nous remplaçons simplement le sous-arbre associé à N par le sous-arbre droit de N.

Enfin, pour chaque ensemble de variables de type $\alpha_1, \ldots, \alpha_n$ possédant la même image α par R, chaque nœud de T_1 de type LeqVarLeqTy, LeqTyLeqVar, LeqVarCptTy, GeqVarLeqTy, GeqTyLeqVar, GeqVarCptTy, CptVarLeqTy, CptTyLeqVar ou CptVarCptTy référençant α est répliqué n fois et introduit dans T_2 *via* son fils droit.

Lemme 5 (Réduction des contraintes par les règles de typage) *Si* Φ_1 *et* Φ_2 *sont saturés et vérifient* $\Phi_1 \geqslant_R \Phi_2$*, si* Γ_1 *et* Γ_2 *sont deux environnement de typage vérifiant* $\Gamma_1 \geqslant \Gamma_2$ *et si* $\Phi_1, R(\Gamma_1) \vdash e : R(\tau) \rhd \Phi'_1$ *alors* $\Phi_2, \Gamma_2 \vdash e : \tau \rhd \Phi'_2$ *et il existe une fonction de renommage* R' *telle que* $\Phi'_1 \geqslant_{R'} \Phi'_2$.

Preuve L'arbre de typage (nommé T_2) de $\Phi_2, \Gamma_2 \vdash e : \tau \rhd \Phi'_2$ est obtenu par transformation de l'arbre de typage T_1 de $\Phi_1, R(\Gamma_1) \vdash e : R(\tau) \rhd \Phi'_1$. La « partie typage » de T_1, c'est-à-dire le sous-arbre de T_1 constitué de nœuds étiquetés « T... », est structurellement inchangée (seules des variables de type sont renommées par R) puisqu'aucune hypothèse n'est faite sur le contenu de Φ par ce genre de règle.

Seule la "partie résolution" de l'arbre change, comme décrite dans la preuve du lemme de **résolution des contraintes par les règles de résolution**. Puisque $\Gamma_1 \geqslant \Gamma_2$, chaque nœud d'instanciation (ie. de type TInst) introduit dans Φ des comparaisons et compatibilités en provenance de Γ mentionnant des variables de type distinctes dans T_2 qui étaient égales dans T_1. Cela préserve l'existence d'une fonction de renommage (R') permettant de construire Φ'_2 à partir de Φ'_1.

Lemme 6 (Préservation du typage par "$\longrightarrow$") *Si* $\Phi, \Gamma \vdash e_1 : \tau \rhd \Phi_1$ *et* $e_1 \longrightarrow e_2$ *alors* $\Phi, \Gamma \vdash e_2 : \tau \rhd \Phi_2$ *et il existe* R *tel que* $\Phi_1 \geqslant_R \Phi_2$.

Preuve par analyse de cas sur la définition de $\longrightarrow$:

- Cas $(\lambda x.e)\ v \longrightarrow e[x := v]$. Considérons l'arbre de typage de $\Phi, \Gamma \vdash (\lambda x.e)\ v : \tau \rhd \Phi_1$ et construisons l'arbre de typage de $\Phi, \Gamma \vdash e[x := v] : \tau \rhd \Phi_2$. L'arbre de typage de $\Phi, \Gamma \vdash (\lambda x.e)\ v : \tau \rhd \Phi_1$ est obligatoirement de la forme :

$$
\textsc{TApp}\ \dfrac{\textsc{TLambda}\ \dfrac{\dfrac{T_1\ (x:\tau' \quad x:\tau'')}{\Phi_1, \Gamma \oplus x : \alpha'_1 \vdash e : \alpha'_2 \rhd \Phi_2} \quad \textsc{LeqArrow}\ \dfrac{\dfrac{T_2}{\Phi_2 \vdash \alpha \leqslant \alpha'_1 \rhd \Phi_3} \quad \dfrac{T_3}{\Phi_3 \vdash \alpha'_2 \leqslant \tau \rhd \Phi_4}}{\Phi_2 \vdash \alpha'_1 \to \alpha'_2 \leqslant \alpha \to \tau \rhd \Phi_4}}{\Phi_1, \Gamma \vdash \lambda x.e : \alpha \to \tau \vdash \Phi_4} \quad \dfrac{T_4}{\Phi_4, \Gamma \vdash v : \alpha \rhd \Phi_5}}{\Phi_1, \Gamma \vdash (\lambda x.e)\ v : \tau \rhd \Phi_5}
$$

  L'arbre de typage de $\Phi, \Gamma \vdash e[x := v] : \tau \rhd \Phi_2$ est construit à partir de T_1 en remplaçant les sous-arbres ayant pour racine les différentes instanciations de x par une adaptation de T_4 :

$$
\dfrac{T_1\ \left(\dfrac{T'_4}{v : \tau'} \quad \dfrac{T''_4}{v : \tau''}\right)}{\Phi_1, \Gamma \vdash e[x := v] : \tau \vdash \Phi_2}
$$

  Plus précisément, l'arbre T_1 original contient des preuves de $x : \tau'$ pour certains τ'. Pour chacun de ces τ', chaque preuve de $x : \tau'$ est remplacée dans T_1 par une preuve de $v : \tau'$ dérivée de T_4. On distingue différents cas selon la structure de v :

- si $v = c$, alors T_4 est de la forme :

$$\text{TConst}\ \dfrac{\dfrac{\bigtriangledown}{\Phi_4 \vdash T(c) \leqslant \alpha \rhd \Phi_5}}{\Phi_4, \Gamma \vdash c : \alpha \rhd \Phi_5}$$

L'arbre de typage T_4 contient alors une preuve de $T(c) \leqslant \alpha$ et nous devons construire une preuve de $T(c) \leqslant \tau'$.

- si $v = \texttt{NULL}$, alors T_4 est de la forme :

$$\text{TNull}\ \dfrac{\dfrac{\bigtriangledown}{\Phi_4 \vdash \alpha' \leqslant \alpha \rhd \Phi_5}}{\Phi_4, \Gamma \vdash \texttt{NULL} : \alpha \rhd \Phi_5}$$

où α' est une variable fraîche. Nous devons alors construire une preuve de $\alpha' \leqslant \tau'$ à partir de la preuve de $\alpha' \leqslant \alpha$ présent dans T_4.

- si $v = \lambda x'.e'$, alors T_4 est de la forme :

$$\text{TLambda}\ \dfrac{\dfrac{\bigtriangledown}{\Phi_4, \Gamma \oplus x' : \alpha'_1 \vdash e' : \alpha'_2 \rhd \Phi_{4.5}} \qquad \dfrac{\bigtriangledown}{\Phi_{4.5} \vdash \alpha'_1 \to \alpha'_2 \leqslant \alpha \rhd \Phi_5}}{\Phi_4, \Gamma \vdash \lambda x'.e' : \alpha \rhd \Phi_5}$$

Le sous-arbre gauche est inchangé. Nous devons construire une preuve de $\alpha'_1 \to \alpha'_2 \leqslant \tau'$ en partant de la preuve de $\alpha'_1 \to \alpha'_2 \leqslant \alpha$ présente dans T_4.

Dans tous les cas, nous avons à construire une preuve de $\tau_0 \leqslant \tau'$ en partant d'une preuve de $\tau_0 \leqslant \alpha$ où τ_0 est un type.

Cependant, le T_1 original contient une preuve de $x : \tau'$ de la forme :

$$\text{TInst}\ \dfrac{\dfrac{\bigtriangledown}{\Phi \vdash \alpha'_1 \leqslant \tau' \rhd \Phi'}}{\Phi, \Gamma \oplus x : \alpha'_1 \vdash x : \tau' \rhd \Phi'}$$

Cela implique que Φ_2 contient $\alpha'_1 \leqslant \tau'$. De plus, T_2 prouve que $\alpha \leqslant \alpha'_1 \in \Phi_5$. D'après le lemme 1 (inclusion des Φ), Φ_5 contient obligatoirement $\tau_0 \leqslant \alpha$, $\alpha \leqslant \alpha'_1$ et $\alpha'_1 \leqslant \tau'$.

Puisque Φ_1 est saturé, d'après le lemme 2 (préservation de la saturation des Φ), Φ_5 est lui aussi saturé. En particulier, on déduit des comparaisons précédentes que Φ_5 contient $\tau_0 \leqslant \tau'$.

La contrainte $\tau_0 \leqslant \tau'$ est alors compatible avec toutes les autres contraintes présentes dans Φ_5 et il existe un sous-arbre, quelque part dans l'arbre de typage original, contenant une preuve de $\tau_0 \leqslant \tau'$. Il suffit d'utiliser ce sous-arbre pour construire les preuves des différents $v : \tau'$ qui remplacent, dans T_1, les preuves des différents $x : \tau'$.

► Cas `let` $x = v$ `in` $e \longrightarrow e[x := v]$. L'arbre de typage prouvant $\Phi, \Gamma \vdash$ `let` $x = v$ `in` $e \rhd \Phi'$ est de la forme :

$$\textsc{TLet}\ \dfrac{\dfrac{T_1}{\Phi_1, \Gamma \vdash v : \alpha \rhd \Phi_2} \qquad \dfrac{T_2\ (x : \tau',\ x : \tau'')}{\Phi_2, \Gamma \oplus x : gen(\Phi_1, \Gamma, \alpha) \vdash e : \tau \rhd \Phi_3}}{\Phi_1, \Gamma \vdash \texttt{let}\ x = v\ \texttt{in}\ e : \tau \rhd \Phi_3}$$

Nous construisons l'arbre de typage de $\Phi, \Gamma \vdash e[x := v] \rhd \Phi'$ en remplaçant dans T_2 les sous-arbres prouvant $x : \tau'$ pour certains types τ' par une adaptation de T_1 :

$$\dfrac{T_2\ (T'_1 : v : \tau',\ T''_1 : v : \tau'')}{\Phi_1, \Gamma \vdash e[x := v] : \tau \rhd \Phi_3}$$

Comme dans le cas précédent, on distingue différents cas en fonction de la structure de v. Que v soit une constante, un constructeur, un λ, ou un tuple, T_1 revient à prouver $\tau_0 \leqslant \alpha$ pour un certains type τ_0.
Soit $\forall \Phi_\alpha.\alpha$ la décomposition du schéma de type $gen(\Phi_1, \Gamma, \alpha)$. Le T_2 original contient une preuve de $x : \tau'$ de la forme :

$$\textsc{TInst}\ \dfrac{\dfrac{}{\Phi \cup \Phi_\alpha[\alpha_k := \alpha'_k]_{k=1}^n \vdash gen(\Phi_1, \Gamma, \alpha)[\alpha_k := \alpha'_k]_{k=1}^n \leqslant \tau' \rhd \Phi'}}{\Phi, \Gamma \oplus x : gen(\Phi_1, \Gamma, \alpha) \vdash x : \tau' \rhd \Phi'}$$

Puisque cette instanciation ne fait que répliquer des variables de type et des contraintes de comparaison/compatibilité qui sont déjà présentes dans Φ_1 par des variables fraîches, les hypothèses du lemme 4 (réduction des contraintes par les règles de résolution) sont alors vérifiées et nous en déduisons que l'arbre original contient une preuve de $\alpha \leqslant \tau'$.
Tout comme dans le cas précédent, puisqu'il existe des preuves de $\tau_0 \leqslant \alpha$ et $\alpha \leqslant \tau'$, nous en déduisons qu'il existe une preuve de $\tau_0 \leqslant \tau'$ que nous utilisons pour remplacer le sous-arbre de T_2 prouvant $x : \tau'$ par une preuve de $v : \tau'$.

► Cas `if` *true* `then` e_1 `else` $e_2 \longrightarrow e_1$. L'arbre de typage de l'expression `if` *true* `then` e_1 `else` e_2 est obligatoirement de la forme :

$$\textsc{TIfThenElse}\ \dfrac{\textsc{TConst}\ \dfrac{T_0}{\Phi_0, \Gamma \vdash true : bool \rhd \Phi_1} \qquad \dfrac{T_1}{\Phi_1, \Gamma \vdash e_1 : \tau \rhd \Phi_2} \qquad \dfrac{T_2}{\Phi_2, \Gamma \vdash e_2 : \tau \rhd \Phi_3}}{\Phi_0, \Gamma \vdash \texttt{if}\ true\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \rhd \Phi_3}$$

D'après le lemme 1 (inclusion des Φ), on sait que en particulier que $\Phi_0 \subset \Phi_1$. L'arbre T_1 prouve $\Phi_1, \Gamma \vdash e_1 : \tau \rhd \Phi_2$. D'après le lemme 5 (réduction des contraintes par les règles de typage), nous concluons directement qu'il existe un $\Phi_{2'}$ tel que $\Phi_0, \Gamma \vdash e_1 : \tau \rhd \Phi_{2'}$.

► Cas `if` *false* `then` e_1 `else` $e_2 \longrightarrow e_2$. Similaire au cas précédent.

- Cas `case NULL of NULL` $\to e_1 \parallel x \to e_2 \longrightarrow e_1$. Démonstration très similaire au cas concernant le `if` *true* `then` e_1 `else` e_2.
- Cas `case` c `of NULL` $\to e_1 \parallel x \to e_2 \longrightarrow e_2[x := c]$. Démonstration très similaire au cas concernant le `let` .
- Cas `case` $\lambda x.e$ `of NULL` $\to e_1 \parallel x \to e_2 \longrightarrow e_2[x := \lambda x.e]$. Similaire au cas précédent.

Lemme 7 (Préservation du typage par "$\longmapsto$") *Si* $\Phi, \Gamma \vdash e_1 : \tau \rhd \Phi_1$ *et* $e_1 \longmapsto e_2$ *alors* $\Phi, \Gamma \vdash e_2 : \tau \rhd \Phi_2$ *et il existe* R *tel que* $\Phi_1 \geqslant_R \Phi_2$.

Preuve par induction et analyse de cas sur la définition du contexte d'évaluation E.

- Cas []. Utilisation directe du lemme 6 (préservation du typage par "$\longrightarrow$").
- Cas $E\ e$. Nous construisons l'arbre de typage de $\Phi, \Gamma \vdash \mathring{e}_2\ e : \tau \rhd \Phi_2$ à partir de l'arbre de typage de $\Phi, \Gamma \vdash \mathring{e}_1\ e : \tau \rhd \Phi_1$ avec $\mathring{e}_1 \longmapsto \mathring{e}_2$. La première règle de typage appliquée est forcément TApp. Nous appliquons l'hypothèse d'induction sur le sous-arbre gauche et le lemme 5 (réduction des contraintes par les règles de typage) sur les sous-arbres milieu et droit.
- Cas $v\ E$. Nous construisons l'arbre de typage de $\Phi, \Gamma \vdash v\ \mathring{e}_2 : \tau \rhd \Phi_2$ grâce à l'arbre de typage de $\Phi, \Gamma \vdash v\ \mathring{e}_1 : \tau \rhd \Phi_1$ avec $\mathring{e}_1 \longmapsto \mathring{e}_2$. La première règle de typage appliquée est TApp. Le sous-arbre gauche est inchangé. Nous utilisons l'hypothèse d'induction sur le sous-arbre du milieu et le lemme 5 (réduction des contraintes par les règles de typage) sur le sous-arbre droit.
- Cas `let` $x = E$ `in` e. Nous construisons l'arbre de typage de $\Phi, \Gamma \vdash$ `let` $x = \mathring{e}_2$ `in` $e \rhd \Phi_2$ depuis l'arbre de typage de $\Phi, \Gamma \vdash$ `let` $x = \mathring{e}_1$ `in` $e \rhd \Phi_1$ avec $\mathring{e}_1 \longmapsto \mathring{e}_2$. La première règle de typage appliquée est TLet. Nous appliquons l'hypothèse d'induction sur le sous-arbre gauche pour construire l'arbre de typage de $\Phi, \Gamma \vdash \mathring{e}_2 : \alpha \rhd \Phi'_2$ à partir de l'arbre de typage de $\Phi, \Gamma \vdash \mathring{e}_2 : \alpha \rhd \Phi'_1$ avec $\Phi'_1 \geqslant_{R'} \Phi'_2$. D'après la définition de la généralisation, puisque $\Phi'_1 \geqslant_{R'} \Phi'_2$, le schéma $gen(\Phi'_2, \Gamma, \alpha)$ contient au moins toutes les variables de $gen(\Phi'_1, \Gamma, \alpha)$ et les contraintes correspondantes. Pour construire l'arbre de typage de $\Phi'_2, \Gamma \oplus x : gen(\Phi'_2, \Gamma, \alpha) \vdash e : \tau \rhd \Phi_2$ à partir de l'arbre de typage de $\Phi'_1, \Gamma \oplus x : gen(\Phi'_1, \Gamma, \alpha) \vdash e : \tau \rhd \Phi_1$ nous appliquons le lemme 5 (réduction des contraintes par les règles de typage). Lorsque x est instancié dans le nouvel arbre de typage, potentiellement plus de variables de type sont générées qu'avant, la propriété $\Phi_2 \subset \Phi_1$ est alors préservée.
- Cas `if` E `then` e `else` e'. Nous construisons l'arbre de typage de $\Phi, \Gamma \vdash$ `if` $\mathring{e}_2$ `then` e `else` $e' : \tau \rhd \Phi_2$ à partir de l'arbre de typage de $\Phi, \Gamma \vdash$ `if` $\mathring{e}_1$ `then` e `else` $e' : \tau \rhd \Phi_1$ avec $\mathring{e}_1 \longmapsto \mathring{e}_2$. La première règle de typage appliquée est forcément TIfThenElse. Nous appliquons l'hypothèse d'induction sur le sous-arbre gauche et le lemme 5 (réduction des contraintes par les règles de typage) sur les sous-arbres milieu et droit.
- Cas `case` E `of NULL` $\to e_1 \parallel x \to e_2$. Nous construisons l'arbre de typage de $\Phi, \Gamma \vdash$ `case` $\mathring{e}_2$ `of NULL` $\to e_1 \parallel x \to e_2$ à partir de l'arbre de typage de $\Phi, \Gamma \vdash$ `case` $\mathring{e}_1$ `of NULL` $\to e_1 \parallel x \to e_2$ avec $\mathring{e}_1 \longmapsto \mathring{e}_2$. Nous appliquons l'hypothèse d'induction sur le sous-arbre gauche et le lemme 5 (réduction des contraintes par les règles de typage) sur les autres sous-arbres.

Lemme 8 (Préservation du typage par "$\longmapsto\!\!\!\!\!\rightarrow$") *Si* $\Phi, \Gamma \vdash e_1 : \tau \rhd \Phi_1$ *et* $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$ *alors* $\Phi, \Gamma \vdash e_2 : \tau \rhd \Phi_2$ *et il existe* R *tel que* $\Phi_1 \geqslant_R \Phi_2$.

Preuve Par définition, $\longmapsto\!\!\!\!\!\rightarrow$ est la fermeture réflexive et transitive de $\longmapsto$. Nous démontrons ce lemme par induction et analyse de cas sur la définition de $\longmapsto\!\!\!\!\!\rightarrow$:

- Si $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$ car $e_1 = e_2$: trivial.
- Si $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$ car $e_1 \longmapsto e_2$: utilisation directe du lemme 7 (préservation du typage par "$\longmapsto$").
- Si $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$ car il existe e_0 tel que $e_1 \longmapsto\!\!\!\!\!\rightarrow e_0$ et $e_0 \longmapsto\!\!\!\!\!\rightarrow e_2$: en appliquant une première fois l'hypothèse d'induction, nous obtenons $\Phi, \Gamma \vdash e_0 : \tau \rhd \Phi_0$ et $\Phi_1 \geqslant_{R_1} \Phi_0$. En appliquant une seconde fois l'hypothèse d'induction, nous obtenons $\Phi, \Gamma \vdash e_2 : \tau \rhd \Phi_2$ avec $\Phi_0 \geqslant_{R_2} \Phi_2$. D'après la transitivité de $\geqslant_R$, nous obtenons $\Phi_1 \geqslant_{R_1 \circ R_2} \Phi_2$.

Lemme 9 (Les expressions bloquées sont non-typables) *Si* e *est une expression bloquée, alors il n'existe aucun couple* (τ, Φ) *tel que* $\emptyset, \emptyset \vdash e : \tau \rhd \Phi$.

Preuve par induction et analyse de cas sur les différentes formes d'expressions bloquées :

- Cas $c\ v$. Les règles de typage TApp et TConst nous obligent à construire, pour un α frais, un Φ' tel que $\emptyset \vdash T(c) \leqslant \alpha \to \tau \rhd \Phi'$. Aucune règle de résolution ne permet de prouver une comparaison entre un type de base et un type flèche.
- Cas `NULL` v. Les règles TApp et TNull nous imposent de construire, pour un α et un α' frais, un Φ' tel que $\emptyset \vdash \alpha'? \leqslant \alpha \to \tau \rhd \Phi'$. Aucune règle de résolution ne permet de prouver la compraison entre un type _? et un type flèche.
- Cas $e_1\ e_2$ avec e_1 est bloquée. La règle TApp nous oblige à construire un Φ' tel que $\emptyset, \emptyset \vdash e_1 : \alpha \to \tau \rhd \Phi'$. Impossible par hypothèse d'induction sur e_1.
- Cas $v\ e'$ avec e' est bloquée. Idem au premier cas en remplaçant TTuple par TApp.
- Cas `let` $x = e_1$ `in` e_2 avec e_1 bloquée. La règle TLet nous oblige à construire un Φ' tel que $\emptyset, \emptyset \vdash e_1 : \alpha \rhd \Phi'$. Impossible par hypothèse d'induction sur e_1.
- Cas `if` e_1 `then` e_2 `else` e_3 avec e_1 est bloquée. La règle TIfThenElse nous oblige à construire un Φ' tel que $\emptyset, \emptyset \vdash e_1 : \alpha \rhd \Phi'$. Impossible par hypothèse d'induction sur e_1.
- Cas `if` c `then` e_1 `else` e_2 avec $c \notin \{ true, false \}$. La règle TIfThenElse nous impose de typer c avec le type *bool*, ce qui est impossible pour une constante différente de *true* et $false$.
- Cas `if NULL then` e_1 `else` e_2. Les règles TIfThenElse et TNull nous imposent de construire, pour un α frais, un Φ' tel que $\emptyset \vdash \alpha? \leqslant bool \rhd \Phi'$. Aucune règle de résolution ne permet de prouver la comparaison entre un type _? et un type de base.
- Cas `if` $\lambda x.e$ `then` e_1 `else` e_2. Les règles TIfThenElse et TLambda nous obligent pour un Φ', un α_1 et un α_2 frais, à construire un Φ'' tel que $\Phi' \vdash \alpha_1 \to \alpha_2 \leqslant bool : \Phi''$. Aucune règle de résolution ne permet de prouver la comparaison entre un type flèche et un type de base.
- Cas `case` e_1 `of NULL` $\to e_2 \parallel x \to e_3$ avec e_1 bloqué. La règle TCase nous impose de construire, pour un α frais, un Φ' tel que $\emptyset, \emptyset \vdash e_1 : \alpha? \vdash \Phi'$. Impossible par hypothèse d'induction sur e_1.

Lemme 10 (Évaluation uniforme) *Une expression e vérifie une et une seule des propriété suivantes :*

- *L'évaluation de e boucle indéfiniment ($e \Uparrow$)*
- *Il existe une valeur v telle que $e \longmapsto\!\!\!\!\to v$*
- *Il existe une expression bloquée e' telle que $e \longmapsto\!\!\!\!\to e'$*

Preuve Si nous ne sommes pas dans le premier cas (ie. e ne boucle pas indéfiniment) alors il existe une expression e' telle que $e \longmapsto\!\!\!\!\to e'$ et aucune expression e'' telle que $e' \longmapsto e''$. Si e' est une valeur, nous sommes dans le second cas. Si e' n'est pas une valeur, par définition, e' est bloquée, et nous sommes alors dans le dernier cas.

Théorème 11 (Validité) *Si* $\emptyset, \emptyset \vdash e : \tau \rhd \Phi$ *alors e vérifie une et une seule des propriétés suivantes :*

- *L'évaluation de e boucle indéfiniment ($e \Uparrow$)*
- *Il existe une valeur v telle que $e \longmapsto\!\!\!\!\to v$*

Preuve D'après le lemme 10 (évaluation uniforme), soit l'évaluation de l'expression e boucle indéfiniment, soit il existe une valeur v telle que $e \longmapsto\!\!\!\!\to v$, soit il existe e' bloquée telle que $e \longmapsto\!\!\!\!\to e'$. Prouvons par l'absurde que le troisième cas est impossible en supposant qu'il existe un e' tel que $e \longmapsto\!\!\!\!\to e'$ et e' est bloquée. Alors, d'après le lemme 8 (préservation du typage par "$\longmapsto\!\!\!\!\to$"), nous savons que $\emptyset, \emptyset \vdash e' : \tau \rhd \Phi'$. L'expression e' est donc typable mais son évaluation est bloquée, cw qui contradedit le lemme 9 qui dit que toute expression bloquée est non-typable.

6. Travaux connexes

Parmi les langages qui disposent de types nullables, peu utilisent une inférence de type complète. Le langage Hack utilise une inférence partielle : les types des paramètres et retours de fonctions doivent être explicitement donnés. Localement à un bloc, donc, Hack utilise un typage « progressif[3] » [18], qui permet de mixer typage statique et typage dynamique. Pour ce qui est de la partie purement statique, Hack permet d'avoir une forme de sous-typage au sens où une variable de type `int?`, peut avoir des occurrences de type `int` après qu'on ait vérifié que sa valeur était bien définie. Cependant, Hack ne gère ni le polymorphisme, ni l'inférence complète. De plus, nous ne connaissons pas à l'heure actuelle de description complète du typage de Hack.

Typed Racket [19] utilise l'information contenue dans la partie test des conditionnelles pour obtenir des informations plus fines au sujet des valeurs des variables testées et les utiliser dans le typage des branches des conditionnelles. En pratique, les tests produisent des propositions logiques qui sont utilisées dans le typage des expressions qui dépendent de ces tests. Un solveur externe est utilisé par le typeur pour déterminer les types des occurrences des variables à partir de ces propositions.

Le langage Swift utilise lui aussi une technique d'inférence de types. Nous n'avons pas non plus pu accéder à une description complète du typage de Swift et les quelques exemples que nous avons pu tester n'ont malheureusement pas été réellement concluants[4].

Il existe de nombreux travaux sur le sous-typage avec polymorphisme, inférence de types et fonctions d'ordre supérieur, mais aucun à notre connaissance n'a été spécialisé au cas des types nullables. Les techniques de sous-typage sémantique [4] et les extensions du système de Hindley-Milner au sous-typage donnent des cadres généraux de conception de systèmes de types. Les travaux sur les présentations de Hindley-Milner sous la forme HM(X) [14, 16] présentent l'avantage de pouvoir réutiliser une partie des preuves des propriétés de ces systèmes. La présentation que nous donnons ici pourrait être reformulée comme une instance de HM(X), voire même comme une instance du cadre donné par Pottier dans [16]. Cependant, notre présentation diffère de celle de Pottier : non seulement notre système est directement adapté aux types nullables, mais il est aussi directement implémentable puisqu'il précise exactement chacune des étapes de calcul (génération de variables fraîches, généralisation, algorithme de saturation). Naturellement, le prix à payer pour cette présentation directe est la perte de la modularité qu'aurait apportée une instanciation de HM(X).

7. Conclusion

Nous avons présenté deux systèmes de types pour un langage avec types nullables, et une traduction de ce langage en variants polymorphes à la OCaml. Le premier système de types, assez naïf et intéressant par sa simplicité, est probablement trop restrictif pour être utilisable dans un langage de programmation réel. La technique de traduction en variants polymorphes souffre des mêmes faiblesses, à cause de l'usage de contraintes d'unification entre types. Le second système que nous présentons, en changeant les contraintes d'unification en contraintes de sous-typage, fournit une souplesse accrue et autorise des styles de programmation plus réalistes. Enfin, nous avons prouvé la correction sémantique de cet algorithme.

Références

[1] Apple (2014) : *Swift, a new programming language for iOS and OS X*. Available at https://developer.apple.com/swift.

[2] Facebook (2014) : *HHVM/Hack/Nullable*. Available at http://docs.hhvm.com/manual/en/hack.nullable.php.

3. *Gradual typing*.
4. En particulier, les compilateurs en ligne que nous avons testés (http://www.runswiftlang.com/ et http://swiftstub.com/), tout comme la version Mac de Swift, n'acceptent pas l'expression Swift « `{ (b : Bool, s) in b ? "s = " + s : nil }` » équivalente à « `fun b s -> if b then s else NULL` ».

[3] Facebook (2014) : *Programming productivity without breaking things*. Available at http://hacklang.org/.

[4] Alain Frisch, Giuseppe Castagna & Véronique Benzaken (2008) : *Semantic Subtyping : Dealing Set-theoretically with Function, Union, Intersection, and Negation Types*. *J. ACM* 55(4), pp. 19 :1–19 :64. Available at http://doi.acm.org/10.1145/1391289.1391293.

[5] Jacques Garrigue (1998) : *Programming with polymorphic variants*. In : *ML Workshop*. Available at http://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf.

[6] Jacques Garrigue (2001) : *Labeled and optional arguments for Objective Caml*. In : *JSSST Workshop on Programming and Programming Languages*. Available at http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/ppl2001.ps.gz.

[7] Jacques Garrigue (2002) : *Simple type inference for structural polymorphism*. In : *The Ninth International Workshop on Foundations of Object-Oriented Languages*. Available at http://www.math.nagoya-u.ac.jp/~garrigue/papers/structural-inf.pdf. Revised on 2002/12/11.

[8] C. A. R. Hoare (2009) : *Null References : The Billion Dollar Mistake*. In : *Proceedings of QCon*, Historically Bad Ideas, London, UK.

[9] Laurent Hubert, Thomas Jensen & David Pichardie (2008) : *Semantic Foundations and Inference of Non-null Annotations*. In Gilles Barthe & Frank. de Boer, editors : *Formal Methods for Open Object-Based Distributed Systems*, *Lecture Notes in Computer Science* 5051, Springer Berlin Heidelberg, pp. 132–149, doi :10.1007/978-3-540-68863-1_9. Available at http://hal.inria.fr/inria-00332356/en/.

[10] JaneStreet : *Package core*. Available at https://ocaml.janestreet.com/ocaml-core/latest/doc/core.

[11] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2008) : *The Objective Caml system (release 4.01) : Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique. Available at http://caml.inria.fr/pub/docs/manual-ocaml/.

[12] Michel Mauny & Benoît Vaugon (2014) : *Inférence de types nullables (version longue)*. Available at http://michel.mauny.net/data/papers/mauny-vaugon-nullables-french-long.pdf.

[13] Microsoft (2014) : *Nullable Types (C# Programming Guide)*. Available at http://msdn.microsoft.com/en-US/library/1t3y8s4s.aspx.

[14] Martin Odersky, Martin Sulzmann & Martin Wehr (1997) : *Type inference with constrained types*. In : *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*.

[15] Atsushi Ohori (1995) : *A Polymorphic Record Calculus and Its Compilation*. *ACM Transactions on Programming Languages and Systems* 17(6), pp. 844–895. Available at http://www.riec.tohoku.ac.jp/~ohori/research/toplas95.pdf.

[16] François Pottier (1998) : *A Framework for Type Inference with Subtyping*. *SIGPLAN Not.* 34(1), pp. 228–238. Available at http://doi.acm.org/10.1145/291251.289448.

[17] Didier Rémy (1989) : *Typechecking Records and Variants in a Natural Extension of ML*. In : *POPL : 16th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Available at http://gallium.inria.fr/~remy/ftp/taoop1.pdf.

[18] Jeremy G. Siek & Walid Taha (2006) : *Gradual Typing for Functional Languages*. In : *Scheme and functional programming workshop*, pp. 81–92.

[19] Sam Tobin-Hochstadt & Matthias Felleisen (2010) : *Logical Types for Untyped Languages*. In : *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, ACM, New York, NY, USA, pp. 117–128. Available at http://doi.acm.org/10.1145/1863543.1863561.