Nullable Type Inference

Michel Mauny Benoît Vaugon

Unité d'Informatique et d'Ingénierie des Systèmes (U2IS)
ENSTA-ParisTech
France

{michel.mauny,benoit.vaugon} <at> ensta-paristech.fr

We present type inference algorithms for nullable types in ML-like programming languages. Starting with a simple system, presented as an algorithm, whose only interest is to introduce the formalism that we use, we replace unification by subtyping constraints and obtain a more interesting system. We state the usual properties for both systems. This is work in progress.

1 Nullable *vs.* option types

Imperative programming languages, such as C or Java derivatives, make abundant use of `NULL` either as a value for unknown or invalid references, or as failure return values. Using `NULL` is rather practical, since the `if` statement suffices for checking `NULL`-ity. Of course, the downside of having `NULL` as a possible value is that, without further support, it could accidentally be confused with a legal value, leading to execution errors [7].

In languages using the ML type discipline, the `option` type

```
type α option = None | Some of α
```

injects regular values and the nullary data constructor `None`, into a single type that could be thought as a nullable type. The type system guarantees that options cannot be confused with regular values, and pattern-matching is used to check and extract regular values from options. In Haskell, the "`maybe`" data type is heavily used for representing successes and failures. The JaneStreet Core library [9] uses the `option` type to show possible failures in function types: it purposely avoids exceptions, because their non-appearance in OCaml types hides the partiality of functions. In OCaml, `None` and `Some` are automatically inserted by the compiler for dealing with optional arguments [5].

Using the `option` data type presents the disadvantage of allocating memory blocks for representing `Some(_)` values, which may refrain experienced programmers from heavily using options. Avoiding those memory allocations is not really difficult: if `Some`(v) is represented as v itself, that is, without allocating a block tagged as `Some`, we need a special representation for `None`, distinct from the representation of v, for any v. Of course, when v is `None` itself, it then becomes impossible to distinguish `Some(None)` from `None`. Therefore, `None` is not the only special case that needs a special treatment: the whole range of $\texttt{Some}^n(\texttt{None})$ values, for $n \geq 0$ needs a special representation such that $\texttt{Some}^i(\texttt{None})$ cannot be confused with $\texttt{Some}^j(\texttt{None})$ when $i \neq j$.

For instance, unaligned addresses on 64-bits architectures, or a statically pre-allocated array of a sufficient size could do the job[1].

[1] Although polymorphic recursion theoretically allows for unbounded depth of $\texttt{Some}^n(\texttt{None})$ while this representation allows for representing only a finite number of n, this limitation should never be met in practice.

Now, the compilation of `Some`(*expr*) needs to generate a test, in order to use the special representation of $\texttt{Some}^n(\texttt{None})$ when *expr* evaluates to `None`, and pattern-matching against `Some`/`None` also needs to be adjusted.

This paper does not aim at opposing nullable types to option types. Options, in the Hindley-Milner type discipline, offer not only type safety, but also precision by distinguishing `Some(None)` from `None`, but at the price of a memory allocation or a dynamic test for `Some`. On the other hand, nullable types extend any classical type t into $t?$, to include `NULL`. Such "nullable values" are easier to represent and compile than options, but offer less precision since it makes no sense to extend further $t?$. Also, their static inference haven't received much attention, so far. Indeed, although quite a few recent programming laguages statically check the safety of `NULL` [3, 2, 11, 1], none of them really performs type inference in the ML sense, but rather local inference, propagating mandatory type annotations of function parameters inside the function bodies.

2 Nullable type inference

The purpose of this work is to study type inference of nullable types, by adding them as a feature in a small functional language. The language that we consider, given in figure 1, is a classical mini-ML, extended with `NULL` test and creation.

Section 3 starts with a naïve approach, where the types τ

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

Figure 1: The language

(that are assigned to expressions e) are pairs (t, ν) of a usual type t and a "nullability" type information ν. A type $(t, ?)$ corresponds to values that *may* be `NULL`, whereas (t, Δ) denotes values that *cannot* be `NULL`. Nullability variables are written δ. This system is mainly used to introduce the formalism that we use for writing our algorithms.

Section 4 shows a translation algorithm that encode nullable values with polymorphic variants. Typing the translated programs with a unification-based mechanism suffers the same weakness as our naive type system.

Section 5 presents a more sophisticated typing mechanism, where unification is replaced by subtyping constraints.

$$\frac{\Phi \vdash \tau = T(c) \rhd \Phi'}{\Phi, \Gamma \vdash c : \tau \rhd \Phi'}\ \textsc{TConst} \qquad \frac{\Phi \vdash \tau' = \tau \rhd \Phi'}{\Phi, \Gamma \oplus \mathtt{x} : \tau' \vdash \mathtt{x} : \tau \rhd \Phi'}\ \textsc{TInstVar} \qquad \frac{\mathtt{let}\ \alpha'\ \mathit{fresh} \quad \Phi, \Gamma \oplus \mathtt{x} : \sigma[\alpha \to \alpha'] \vdash \mathtt{x} : \tau \rhd \Phi'}{\Phi, \Gamma \oplus \mathtt{x} : \forall \alpha.\sigma \vdash \mathtt{x} : \tau \rhd \Phi'}\ \textsc{TInst}(\alpha)$$

$$\frac{\mathtt{let}\ \delta'\ \mathit{fresh} \quad \Phi, \Gamma \oplus \mathtt{x} : \sigma[\delta \to \delta'] \vdash \mathtt{x} : \tau \rhd \Phi'}{\Phi, \Gamma \oplus \mathtt{x} : \forall \delta.\sigma \vdash \mathtt{x} : \tau \rhd \Phi'}\ \textsc{TInst}(\delta)$$

$$\frac{\mathtt{let}\ \alpha_1, \delta_1, \alpha_2, \delta_2\ \mathit{fresh} \quad \Phi, \Gamma \oplus x : (\alpha_1, \delta_1) \vdash e : (\alpha_2, \delta_2) \rhd \Phi' \quad \Phi' \vdash \tau = (\alpha_1, \delta_1) \to (\alpha_2, \delta_2) \rhd \Phi''}{\Phi, \Gamma \vdash \lambda x.e : \tau \rhd \Phi''}\ \textsc{TLambda}$$

$$\frac{\mathtt{let}\ \alpha, \delta\ \mathit{fresh} \quad \Phi, \Gamma \vdash e_1 : ((\alpha, \delta) \to \tau), \Delta) \rhd \Phi' \quad \Phi', \Gamma \vdash e_2 : (\alpha, \delta) \rhd \Phi''}{\Phi, \Gamma \vdash e_1\ e_2 : \tau \rhd \Phi''}\ \textsc{TApp}$$

$$\frac{\mathtt{let}\ \alpha, \delta\ \mathit{fresh} \quad \Phi, \Gamma \vdash e_1 : (\alpha, \delta) \rhd \Phi' \quad \Phi', \Gamma \oplus x : \mathit{gen}(\Phi', \Gamma, (\alpha, \delta)) \vdash e_2 : \tau \rhd \Phi''}{\Phi, \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau \rhd \Phi''}\ \textsc{TLet}$$

$$\frac{\Phi, \Gamma \vdash e_1 : (\mathit{bool}, \Delta) \rhd \Phi' \quad \Phi', \Gamma \vdash e_2 : \tau \rhd \Phi'' \quad \Phi'', \Gamma \vdash e_3 : \tau \rhd \Phi'''}{\Phi, \Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau \rhd \Phi'''}\ \textsc{TIfThenElse} \qquad \frac{\mathtt{let}\ \alpha\ \mathit{fresh} \quad \Phi \vdash \tau = (\alpha, ?) \rhd \Phi'}{\Phi, \Gamma \vdash \mathtt{NULL} : \tau \rhd \Phi'}\ \textsc{TNull}$$

$$\frac{\mathtt{let}\ \delta\ \mathit{fresh} \quad \Phi, \Gamma \vdash e_1 : (t, ?) \rhd \Phi' \quad \Phi', \Gamma \vdash e_2 : \tau \rhd \Phi'' \quad \Phi'', \Gamma \oplus x : \forall \delta.(t, \delta) \vdash e_3 : \tau \rhd \Phi'''}{\Phi, \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ \mathtt{NULL} \to e_2 \parallel x \to e_3 : \tau \rhd \Phi'''}\ \textsc{TCase}$$

Figure 3: Typing rules

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

Figure 2: The types

$$\mathit{gen}(\Phi, \Gamma, \tau) = \mathit{gen}'(\Phi(\Gamma), \Phi(\tau))$$

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

Figure 4: Generalisation

3 A simple type system

We first present a rather simple type system, where the types carry a “nullability information” saying whether the value of an expression may be NULL or not.

The judgements of our language’s type system are of the form $\Phi, \Gamma \vdash e : (t, \nu) \rhd \Phi'$, where Φ and Φ' are substitutions, Γ is a type environment that maps program identifiers to type schemes (types with a prenex universal quantification of type and nullability variables). Such a judgement should be read as: *given Φ, under assumption Γ, the expression e has type τ with substitution Φ'*.

The rules should be read as the different cases of an algorithm that, given Φ, Γ, e, and τ, computes substitution Φ' which, when applied to τ and Γ, assign the type $\Phi'(\tau)$ to e. In other words, under assumptions $\Phi'(\Gamma)$, e has type $\Phi'(\tau)$.

Because we have two kinds of variables, we have two instanciation mechanisms, in distinct rules: TINST(α) for universally quantified type variables, and TINST(δ) for universally quantified nullability variables.

Type equality constraints, introduced by typing rules, are solved using a set of rules displayed in figure 5. The only interesting resolution rules are the ones that introduce (EQNEW) or merge (EQMERGE) variable bindings in the Φ substitution.

EQNEW(α), installs a binding $\alpha = t'$ in the substitution $\Phi[\alpha \to t']$ (that is, Φ in which free occurences of α are replaced by t') when there is no previous binding about α in Φ. Here, t' is either $\mu\alpha.t$ or t, depending on whether α occurs or not in t. EQNEW(δ) does the same, in a simpler context, on nullability variables.

EQMERGE(α) merges a new constraint $\alpha = t'$ to a previous $\alpha = t$ occuring in Φ. Here, the resulting substitution Φ' is obtained by resolving the constraint $t = t'$. EQMERGE(δ) does the same for nullability variables.

Correctness. The operational semantics needed for stating the correction of the type system is also classical. NULL cannot handle operations such as being called as a function, tested as a boolean, added as an integer, *etc*. The typing of values, procuded by correct executions, is also slightly changed: the type $(t, ?)$ includes NULL as well as regular values. The correctness property can be stated as follows:

> If $\Phi, \Gamma \vdash e : \tau \rhd \Phi'$, then the evaluation of e in an execution environment compatible with $\Phi'(\Gamma)$ produces a value typable with $\Phi'(\tau)$, if evaluation terminates.

Although this type system is simple and may seem practical,

$$\textsc{EqSplit}\ \frac{\Phi \vdash t_1 = t_2 \rhd \Phi' \qquad \Phi' \vdash \nu_1 = \nu_2 \rhd \Phi''}{\Phi \vdash (t_1,\nu_1) = (t_2,\nu_2) \rhd \Phi''} \qquad \textsc{EqBase}\ \frac{}{\Phi \vdash t_b = t_b \rhd \Phi}$$

$$\textsc{EqArrow}\ \frac{\Phi \vdash \tau_1 = \tau_1' \rhd \Phi' \qquad \Phi' \vdash \tau_2 = \tau_2' \rhd \Phi''}{\Phi \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' \rhd \Phi''} \qquad \textsc{EqTrivial}(\alpha)\ \frac{}{\Phi \vdash \alpha = \alpha \rhd \Phi}$$

$$\textsc{EqNew}(\alpha)\ \frac{\textit{when } t \neq \alpha \wedge \alpha = \_ \notin \Phi \qquad \texttt{let } t' = \mathit{clos}(\alpha, \Phi(t))}{\Phi \vdash \alpha = t \rhd \Phi[\alpha \to t'] \oplus \alpha = t'} \qquad \textsc{EqMerge}(\alpha)\ \frac{\texttt{when } t \neq \alpha \qquad \Phi \oplus \alpha = t' \vdash t = t' \rhd \Phi'}{\Phi \oplus \alpha = t' \vdash \alpha = t \rhd \Phi'} \qquad \textsc{EqTrivial}(\delta)\ \frac{}{\Phi \vdash \delta = \delta \rhd \Phi}$$

$$\textsc{EqNew}(\delta)\ \frac{\texttt{when } \nu \neq \delta \wedge \delta = \_ \notin \Phi}{\Phi \vdash \delta = \nu \rhd \Phi[\delta \to \nu] \oplus \delta = \nu} \qquad \textsc{EqMerge}(\delta)\ \frac{\texttt{when } \nu \neq \delta \qquad \Phi \oplus \delta = \nu' \vdash \nu = \nu' \rhd \Phi'}{\Phi \oplus \delta = \nu' \vdash \delta = \nu \rhd \Phi'} \qquad \begin{array}{lcll} \mathit{clos}(\alpha, t) & = & t & \textit{when } \alpha \notin \mathit{FTV}(t) \\ & = & \mu\alpha.t & \textit{otherwise} \end{array}$$

Figure 5: Resolution rules

it imposes a certain style of programming where `NULL` values are as much isolated as possible from other parts of the program. In a ML-like language, where `NULL` (or similar construct such as `None`) are not as commonly used as in C or Java, this might be acceptable. Still, one might want the following example to be typable:

$$\begin{array}{l} \texttt{let}\ f\ b\ k\ = \\ \quad \texttt{let}\ p\ =\ k+1\ \texttt{in} \\ \quad \texttt{if}\ b\ \texttt{then}\ k\ \texttt{else}\ \texttt{NULL}\ \texttt{in} \\ \dots \end{array}$$

This program cannot typecheck since the k parameter must at the same time have type (`int`,Δ) in order to be sent to + and (`int`,?) in order to have the same type as `NULL`.

4 Encoding nullability as variants

$$\begin{array}{lcl} \llbracket\, \texttt{NULL} \,\rrbracket & = & \texttt{`None} \\ \llbracket\, x \,\rrbracket & = & x \\ \llbracket\, c \,\rrbracket & = & \texttt{`Some}(c) \\ \multicolumn{3}{l}{\llbracket\, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\rrbracket} \\ & = & \texttt{if match}\ \llbracket\, e1 \,\rrbracket\ \texttt{with `Some}(b) \to b \\ & & \texttt{then}\ \llbracket\, e_2 \,\rrbracket\ \texttt{else}\ \llbracket\, e_3 \,\rrbracket \\ \llbracket\, \lambda x.e \,\rrbracket & = & \texttt{`Some}(\lambda x.\llbracket\, e \,\rrbracket) \\ \multicolumn{3}{l}{\llbracket\, \texttt{case}\ e_1\ \texttt{of NULL} \to e_2 \parallel x \to e_3 \,\rrbracket} \\ & = & \texttt{match}\ \llbracket\, e_1 \,\rrbracket\ \texttt{with} \\ & & \texttt{| `None} \ \to \llbracket\, e_2 \,\rrbracket \\ & & \texttt{| `Some}(x) \to (\lambda x.\llbracket\, e_3 \,\rrbracket)(\texttt{`Some}(x)) \\ \llbracket\, e_1\ e_2 \,\rrbracket & = & (\texttt{match}\ \llbracket\, e_1 \,\rrbracket\ \texttt{with `Some}(f) \to f)\ \llbracket\, e_2 \,\rrbracket \end{array}$$

Figure 9: Encoding of `NULL` as variants

Typing nullability of our language can be easily performed by a typechecker for polymorphic variants such as those provided by OCaml. The trick is to encode `NULL` as `` `None``, other values as `` `Some(_)``, and computations must assume that only definite functions, that is, with shape `` `Some(_)``, can be applied without further tests. In the same vein, primitive operations, such as +, accept only definite arguments. The translation, given in figure 9, shows that the typability of nullable values can trivially be reduced to the typing of polymorphic variants. Not a big deal, but improving the latter improves also the former.

As expected, the translation of the above example does not pass OCaml typechecking:

```
let (+) = `Some (fun x y ->
  match x, y with
  | `Some x, `Some y -> `Some (x+y))

let f = `Some (fun b k ->
  let p = (match (+) with
    | `Some f_0 -> f_0) k (`Some 1) in
  if match b with `Some b_0 -> b_0
  then k else `None) in ...
                  ^^^^^
Error: this expression has type [> `None ] but an
expression was expected of type [< `Some of int ].
```

This is due to the fact that the type inference of OCaml polymorphic variants use unification, even though it emulates some form of subtyping with a rich type algebra [6].

We have extended the work of Garrigue in order to have a more flexible and powerful type system for polymorphic variants. Although this is still work in progress, we show here how to apply this result to nullable types.

5 A subtyping approach

We saw in the example above that the propagation of information "backwards", by unification in the typing environment, prevents typing some programs that could be perfectly acceptable.

Replacing unification, which comes from type equality constraints, by *inequality* constraints, that is, by subtyping, relaxes the programming style imposed by using type unification. While this is clearly more permissive, the resolution of inequality constraints may still fail. On the one hand, some unification constraints remain hidden as double inequalities (*e.g.* when trying to type `1 + "hello"`). On the other hand, some inequalites are clearly not satisfiable, such as those produced when typing a conditional **if** `NULL` **then** … **else** …, where one fails to prove $\alpha? \leqslant \mathit{bool}$, or an application `NULL(...)`, failing to prove $\alpha? \leqslant \tau_1 \to \tau_2$. Also, many primitive operations (like `(+)`) won't accept nullable arguments.

We start to change the type algebra of our language and introduce the syntax "t?" for nullable types, figure 10.

The new set of typing rules is given in figure 6. The essential change that we bring to our initial system is in the TApp

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

TInst
$$\frac{\texttt{let}\ \{\ \alpha'_k\ \}_1^n\ \mathit{fresh} \qquad \Phi, \{\ \alpha_k \leqslant \{\ \tau_{k,i}^{\leqslant}\ \}, \alpha_k \geqslant \{\ \tau_{k,i}^{\geqslant}\ \}, \alpha_k \approx \{\ \tau_{k,i}^{\approx}\ \}\ \}_1^n[\alpha_k \to \alpha'_k]_1^n, \vdash \tau[\alpha_k \to \alpha'_k]_1^n \leqslant \tau' \rhd \Phi'}{\Phi, \Gamma \oplus x : \forall\{\ \alpha_k[\leqslant \{\ \tau_{k,i}^{\leqslant}\ \}][\geqslant \{\ \tau_{k,i}^{\geqslant}\ \}][\approx \{\ \tau_{k,i}^{\approx}\ \}]\ \}_1^n.\tau \vdash x : \tau' \rhd \Phi'}$$

TLambda
$$\frac{\texttt{let}\ \alpha_1, \alpha_2\ \mathit{fresh} \qquad \Phi, \Gamma \oplus x : \alpha_1 \vdash e : \alpha_2 \rhd \Phi' \qquad \Phi' \vdash \alpha_1 \to \alpha_2 \leqslant \tau \rhd \Phi''}{\Phi, \Gamma \vdash \lambda x.e : \tau \rhd \Phi''}$$

TApp
$$\frac{\texttt{let}\ \alpha\ \mathit{fresh} \qquad \Phi, \Gamma \vdash e_1 : \alpha \to \tau \rhd \Phi' \qquad \Phi', \Gamma \vdash e_2 : \alpha \rhd \Phi''}{\Phi, \Gamma \vdash e_1\ e_2 : \tau \rhd \Phi''}$$

TLet
$$\frac{\texttt{let}\ \alpha\ \mathit{fresh} \qquad \Phi, \Gamma \vdash e_1 : \alpha \rhd \Phi' \qquad \Phi', \Gamma \oplus x : \mathit{gen}(\Phi', \Gamma, \alpha) \vdash e_2 : \tau \rhd \Phi''}{\Phi, \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \rhd \Phi''}$$

TIfThenElse
$$\frac{\Phi_0, \Gamma \vdash e_1 : \mathit{bool} \rhd \Phi_1 \qquad \Phi_1, \Gamma \vdash e_2 : \tau \rhd \Phi_2 \qquad \Phi_2, \Gamma \vdash e_3 : \tau \rhd \Phi_3}{\Phi, \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \rhd \Phi_3}$$

TNull
$$\frac{\texttt{let}\ \alpha\ \mathit{fresh} \qquad \Phi \vdash \alpha? \leqslant \tau \rhd \Phi'}{\Phi, \Gamma \vdash \texttt{NULL} : \tau \rhd \Phi'}$$

TCase
$$\frac{\texttt{let}\ \alpha\ \mathit{fresh} \qquad \Phi_0, \Gamma \vdash e_1 : \alpha? \rhd \Phi_1 \qquad \Phi_1, \Gamma \vdash e_2 : \tau \rhd \Phi_2 \qquad \Phi_2, \Gamma \oplus x : \alpha_1 \vdash e_3 : \tau \rhd \Phi_3}{\Phi_0, \Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ \texttt{NULL} \to e_2 \parallel x \to e_3 : \tau \rhd \Phi_3}$$

Figure 6: Subtyping rules

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

Figure 10: Nullable types

$$\begin{array}{ll}
\mathit{gen}(\Phi, \Gamma, \tau) = \forall\{\ \alpha[\Phi|_\alpha] \mid \alpha \in \mathit{gen_vars}(\Phi, \Gamma, \tau)\ \}.\tau & \\
\mathit{gen_vars}(\Phi, \Gamma, \tau) = \bigcup_{\alpha \in \mathit{gen_FTV}(\Phi,\Gamma,\tau)} \mathit{deps}^*(\Phi, \{\ \alpha\ \}) & \\
\mathit{gen_FTV}(\Phi, \Gamma, \tau) = & \\
\qquad \{\ \alpha \mid \alpha \in \mathit{FTV}(\tau) \land \mathit{deps}^*(\Phi, \{\ \alpha\ \}) \cap \mathit{FTV}(\Gamma) = \emptyset\ \} & \\
\mathit{deps}^*(\Phi, A) = A & \mathit{when}\ \mathit{deps}(\Phi, A) = A \\
\mathit{deps}^*(\Phi, A) = \mathit{deps}^*(\Phi, \mathit{deps}(\Phi, A)) & \mathit{otherwise} \\
\mathit{deps}(\Phi, A) = \bigcup_{\alpha \in A} \mathit{dep_alphas}(\Phi, \alpha) & \\
\mathit{dep_alphas}(\Phi, \alpha) = \bigcup_{\tau \in \mathit{dep_tys}(\Phi,\alpha)} \mathit{FTV}(\tau) & \\
\mathit{dep_tys}(\Phi, \alpha) = \{\ \tau \mid \alpha \leqslant \tau \in \Phi\ \lor\ \tau \leqslant \alpha \in \Phi\ \lor\ \alpha \approx \tau \in \Phi\ \} &
\end{array}$$

Figure 11: Generalization

rule, where the domain type of the function is constrained to be “larger” than the type of the argument in order to accept it. Intuitively, a function accepting a possibly null value as argument, accepts also a provably non-null argument.

The inequality constraints, written $\tau_1 \geqslant \tau_2$, are integrated in the Φ component of the typing rules by the set of resolution rules given in figure 7. At resolution time, newly integrated constraints are checked to be consistent with constraints existing in Φ. In particular, resolution performs the basic subtyping checks through rules LeqBaseNull and LeqArrowNull, on figure 7(b).

When a type variable α has to be “smaller” (resp. “greater”) than two types τ_1 and τ_2, the τ_i become constrained to be “compatible” (see figure 8), that is to differ only in their (possibly internal) “?” annotations.

Generalization (figure 11) universally quantifies type variables α together with its associated constraints, written $\Phi|_\alpha$, when the set $\{\ \alpha\ \} \cup \mathit{FTV}(\Phi|_\alpha)$ does not intersect the set of free type variables occurring in Γ.

At instanciation time, a fresh instance of constraints is reinjected in Φ.

Another important change in the new system concerns the conditional, case selection (and, more generally pattern-matching constructs). Instead of unifying the types of all branches, each of them is contrained to be “smaller” than the type of the construct itself. This forces all types of the branches to be compatible, but no more. See for instance rules TIfThenElse and TCase on figure 6. This is precisely the reason why the example given at the end of section 3 is accepted by the new system.

6 Properties

We have a prototype implementation of this typing algorithm, and the proof of correctness of the extension of Garrigue’s typing of polymorphic variants, in which this algorithm can easily be translated. The proof is available online at `https://github.com/bvaugon/variants/`.

7 Conclusion

We have presented two type systems and a translation algorithm aiming at inferring nullable types in ML-like languages. The first type system, rather naive and interesting by its simplicity, is probably too restrictive to be usable by daily programmers. The translation technique using standard polymorphic variants has the same weakness. However, exchanging unification against subtyping provides us with a more expressive type system. Soundness and termination properties have been checked.

(a) Main comparison rules

GEQ
$$\frac{\Phi \vdash \tau_2 \leqslant \tau_1 \rhd \Phi'}{\Phi \vdash \tau_1 \geqslant \tau_2 \rhd \Phi'}$$

EQ
$$\frac{\Phi \vdash \tau_1 \leqslant \tau_2 \rhd \Phi' \qquad \Phi' \vdash \tau_1 \geqslant \tau_2 \rhd \Phi''}{\Phi \vdash \tau_1 = \tau_2 \rhd \Phi''}$$

LEQNEW
$$\frac{\textit{when } \tau_1 \leqslant \tau_2 \notin \Phi \qquad \Phi, \tau_1 \leqslant \tau_2 \vdash \tau_1 \leq \tau_2 \rhd \Phi'}{\Phi \vdash \tau_1 \leqslant \tau_2 \rhd \Phi'}$$

LEQALREADYPROVED
$$\frac{}{\Phi, \tau_1 \leqslant \tau_2 \vdash \tau_1 \leqslant \tau_2 \rhd \Phi, \tau_1 \leqslant \tau_2}$$

(b) Standard comparison rules

LEQBASETY
$$\frac{}{\Phi \vdash t_b \leq t_b \rhd \Phi}$$

LEQARROW
$$\frac{\Phi \vdash \tau_1' \leqslant \tau_1 \rhd \Phi' \qquad \Phi' \vdash \tau_2 \leqslant \tau_2' \rhd \Phi''}{\Phi \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \rhd \Phi''}$$

LEQBASENULL
$$\frac{}{\Phi \vdash t_b \leq t_b? \rhd \Phi}$$

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

(c) Type-variable comparison rule

LEQSAMEVAR
$$\frac{}{\Phi \vdash \alpha \leq \alpha \rhd \Phi}$$

LEQVARLEQTY
$$\frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \leqslant \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \leqslant \tau \vdash \alpha \leq \tau' \rhd \Phi''}$$

GEQVARLEQTY
$$\frac{\textit{when } \tau' \neq \alpha \text{ and } \tau' \leqslant \tau \notin \Phi \qquad \Phi, \alpha \leqslant \tau, \tau' \leqslant \tau \vdash \tau' \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau' \leq \tau \rhd \Phi''}{\Phi, \alpha \leqslant \tau \vdash \tau' \leq \alpha \rhd \Phi''}$$

LEQTYLEQVAR
$$\frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \leqslant \tau' \notin \Phi \qquad \Phi, \tau \leqslant \alpha, \tau \leqslant \tau' \vdash \alpha \leq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \leq \tau' \rhd \Phi''}{\Phi, \tau \leqslant \alpha \vdash \alpha \leq \tau' \rhd \Phi''}$$

GEQTYLEQVAR
$$\frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \tau \leqslant \alpha, \tau \approx \tau' \vdash \tau' \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \tau \leqslant \alpha \vdash \tau' \leq \alpha \rhd \Phi''}$$

LEQVARCPTTY
$$\frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \approx \tau \vdash \alpha \leq \tau' \rhd \Phi''}$$

GEQVARCPTTY
$$\frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \tau' \leq \alpha \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \approx \tau \vdash \tau' \leq \alpha \rhd \Phi''}$$

LEQVAREND
$$\frac{\textit{when } \tau' \neq \alpha \qquad \textit{when } (\forall \tau \mid \alpha \leqslant \tau \in \Phi \implies \tau \approx \tau' \in \Phi) \qquad \textit{when } (\forall \tau \mid \tau \leqslant \alpha \in \Phi \implies \tau \leqslant \tau' \in \Phi) \qquad \textit{when } (\forall \tau \mid \alpha \approx \tau \in \Phi \implies \tau \approx \tau' \in \Phi)}{\Phi \vdash \alpha \leq \tau' \rhd \Phi}$$

GEQVAREND
$$\frac{\textit{when } \tau' \neq \alpha \qquad \textit{when } (\forall \tau \mid \alpha \leqslant \tau \in \Phi \implies \tau' \leqslant \tau \in \Phi) \qquad \textit{when } (\forall \tau \mid \tau \leqslant \alpha \in \Phi \implies \tau \approx \tau' \in \Phi) \qquad \textit{when } (\forall \tau \mid \alpha \approx \tau \in \Phi \implies \tau \approx \tau' \in \Phi)}{\Phi \vdash \tau' \leq \alpha \rhd \Phi}$$

Figure 7: Comparison rules

(a) Main compatibility rules

$$\textsc{CPTNew}\quad \frac{\textit{when } \tau_1 \approx \tau_2 \notin \Phi \qquad \Phi, \tau_1 \approx \tau_2 \vdash \tau_1 \simeq \tau_2 \rhd \Phi'}{\Phi \vdash \tau_1 \approx \tau_2 \rhd \Phi'}$$

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

(b) Standard compatibility rules

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

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

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

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

(c) Type-variable compatibility rules

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

$$\textsc{CptVarLeqTy}\quad \frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \leqslant \tau, \tau \approx \tau' \vdash \alpha \simeq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \leqslant \tau \vdash \alpha \simeq \tau' \rhd \Phi''}$$

$$\textsc{CptTyLeqVar}\quad \frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \tau \leqslant \alpha, \tau \approx \tau' \vdash \alpha \simeq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \tau \leqslant \alpha \vdash \alpha \simeq \tau' \rhd \Phi''}$$

$$\textsc{CptVarCptTy}\quad \frac{\textit{when } \tau' \neq \alpha \text{ and } \tau \approx \tau' \notin \Phi \qquad \Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \alpha \simeq \tau' \rhd \Phi' \qquad \Phi' \vdash \tau \simeq \tau' \rhd \Phi''}{\Phi, \alpha \approx \tau \vdash \alpha \simeq \tau' \rhd \Phi''}$$

$$\textsc{CptVarEnd}\quad \frac{\begin{array}{c}\textit{when } \tau' \neq \alpha \\ \textit{when } (\forall \tau \mid \alpha \leqslant \tau \in \Phi \;\Rightarrow\; \tau \approx \tau' \in \Phi) \qquad \textit{when } (\forall \tau \mid \tau \leqslant \alpha \in \Phi \;\Rightarrow\; \tau \approx \tau' \in \Phi) \qquad \textit{when } (\forall \tau \mid \alpha \approx \tau' \in \Phi \;\Rightarrow\; \tau \approx \tau' \in \Phi)\end{array}}{\Phi \vdash \alpha \simeq \tau' \rhd \Phi}$$

Figure 8: Compatibility rules

References

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

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

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

[6] 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.

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

[8] Laurent Hubert, Thomas Jensen & David Pichardie (2008): *Semantic Foundations and Inference of Non-null Annotations*. Available at http://hal.inria.fr/inria-00332356/en/.

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

[10] 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/.

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

[12] 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.

[13] 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.