Nullable Type Inference
Michel Mauny, Benoît Vaugon

To cite this version:
Michel Mauny, Benoît Vaugon. Nullable Type Inference. OCaml 2014 - The OCaml Users and Developers Workshop, Sep 2014, Gothenbourg, Sweden. hal-01413294

HAL Id: hal-01413294
https://inria.hal.science/hal-01413294
Submitted on 9 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Nullable Type Inference

Michel Mauny
Benoît Vaugon

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

{michel.mauny,benoit.vaugon}@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.

In languages using the ML discipline, the option type

\[
type \alpha option = None | Some \ of \ \alpha
\]

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 uses the option type to show possible failures in function types: it purposely avoids exceptions, because their non-appearance in OCaml type to show possible failures in function types: it purposely avoids exceptions, because their non-appearance in OCaml, 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 \( \tau \)

\[
c ::= \{ \text{true} \} | false | 0 | 1 | \ldots + | \ldots
\]

\[
e ::= c | x | \lambda x.e | e_1 e_2 | \text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

\[
\begin{aligned}
&\text{let } x = e_1 \text{ in } e_2 \\
&\text{NULL } | \text{case } e_1 \text{ of } \text{NULL } \rightarrow e_2 | x \rightarrow e_3
\end{aligned}
\]

Figure 1: The language

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

Now, the compilation of \(\text{Some}(expr)\) needs to generate a test, in order to use the special representation of \(\text{Some}(\text{None})\) when \(expr\) evaluates to \(\text{None}\), and pattern-matching against \(\text{Some}/\text{None}\) 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 \(\text{Some}(\text{None})\) from \(\text{None}\), but at the price of a memory allocation or a dynamic test for \(\text{Some}\). On the other hand, nullable types extend any classical type \(t\) into \(?\), 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 \(?\). Also, their static inference haven’t received much attention, so far. Indeed, although quite a few recent programming languages statically check the safety of NULL \([8,2,11,1]\), 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

...
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 \( \Phi, \Gamma \triangleright e : (t, \nu) \triangleright \Phi' \), where \( \Phi \) and \( \Phi' \) are substitutions, \( \Gamma \) 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: \text{given } \Phi, \text{ under assumption } \Gamma, \text{ the expression } e \text{ has type } t \text{ with substitution } \Phi'.

The rules should be read as the different cases of an algorithm that, given \( \Phi, \Gamma, e, \) and \( t \), computes substitution \( \Phi' \) which, when applied to \( t \) and \( \Gamma \), assign the type \( \Phi'(t) \) to \( e \). In other words, under assumptions \( \Phi'(\Gamma) \), \( e \) has type \( \Phi'(t) \).

Because we have two kinds of variables, we have two instantiation mechanisms, in distinct rules: \text{TInst}(a) for universally quantified type variables, and \text{TInst}(\delta) 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 (EQMerge) variable bindings in the \( \Phi \) substitution. EQNew(a), installs a binding \( a \rightarrow t' \) in the substitution \( \Phi[a \rightarrow t'] \) (that is, \( \Phi \) in which free occurrences of \( a \) are replaced by \( t' \)) when there is no previous binding about \( a \) in \( \Phi \). Here, \( t' \) is either \( a.t \) or \( t \), depending on whether \( a \) occurs or not in \( t \).

EONew(\delta) does the same, in a simpler context, on nullability variables.

EQMerge(a) merges a new constraint \( a = t' \) to a previous \( a = t \) occurring in \( \Phi \). Here, the resulting substitution \( \Phi' \) is obtained by resolving the constraint \( t = t' \). EQMerge(\delta) does the same for nullability variables.

Correctness. The operational semantics needed for stating the correctness of the type system is also classical. NULL cannot handle operations such as being called as a function, tested or merged (EQMerge) variable bindings in the \( \Phi \) substitution.

Although this type system is simple and may seem practical,
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 constructs 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:

```plaintext
let f = \( k \) + 
  let \( p = k + 1 \) in 
  if \( b \) then \( k \) else NULL in 
...
```

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

### 4 Encoding nullability as variants

In a more flexible and powerful type system for polymorphic variants, 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.

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

```plaintext
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 [\( \tau \) but an expression was expected of type [\( \tau \).]
```

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 \( 1 + "\text{hello}" \). On the other hand, some inequalities are clearly not satisfiable, such as those produced when typing a conditional if \( \text{NULL} \) then \( \ldots \) else \( \ldots \), where one fails to prove \( a \neq \text{bool} \), or an application \( \text{NULL}(\ldots) \), failing to prove \( a \ll \tau_1 \to \tau_2 \). Also, many primitive operations (like \( (+) \)) won’t accept nullable arguments.

We start to change the type algebra of our language and introduce the syntax “?!” for nullable types, figure [10].

The new set of typing rules is given in figure [9]. The essential change that we bring to our initial system is in the TAPP
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 \( t_1 \geq t_2 \), are integrated in the \( \Phi \) 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 \( \Phi \). In particular, resolution performs the basic subtyping checks through rules \texttt{LEQBASENULL} and \texttt{LEQARROWNULL}, on figure 7(b).

When a type variable \( a \) has to be “smaller” (resp. “greater”) than two types \( t_1 \) and \( t_2 \), the \( t_i \) 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 \( a \) together with its associated constraints, written \( \Phi_{a_i} \), when the set \( \{ a \} \cup \text{FTV}(\Phi_{a_i}) \) does not intersect the set of free type variables occurring in \( \Gamma \).

At instantiation time, a fresh instance of constraints is re-injected in \( \Phi \).

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 constrained 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 \texttt{TIFTHENELSE} and \texttt{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/
\[
\begin{align*}
\text{GEQ} & : \Phi \vdash t_2 \leq t_1 \Rightarrow \Phi' \\
\text{EQ} & : \Phi \vdash t_1 \leq t_2 \Rightarrow \Phi' \\
\text{LEQNEW} & : \text{when } t_1 \leq t_2 \Rightarrow \text{when } t_1 \leq t_2 \Rightarrow \Phi' \\
\text{LEQALREADYPROVED} & : \Phi, t_1 \leq t_2 \Rightarrow \Phi, t_1 \leq t_2
\end{align*}
\]

(a) Main comparison rules

\[
\begin{align*}
\text{LEQBASETY} & : \Phi \vdash t \leq t' \Rightarrow \Phi \\
\text{LEQARROW} & : \Phi \vdash t_1 \leq t_2 \Rightarrow \Phi' \\
\text{LEQBASENULL} & : \Phi \vdash t_1 \leq t_2 \Rightarrow \Phi' \\
\text{LEQARROWNULL} & : \Phi \vdash t_1 \leq t_2 \Rightarrow \Phi'
\end{align*}
\]

(b) Standard comparison rules

\[
\begin{align*}
\text{LEQVARLEQTY} & : \text{when } \tau \neq \alpha \text{ and } \tau \equiv \tau' \Rightarrow \Phi \Rightarrow \Phi' \\
\text{GEQVARLEQTY} & : \text{when } \tau \neq \alpha \text{ and } \tau \equiv \tau' \Rightarrow \Phi \Rightarrow \Phi' \\
\text{LEQTYLEQVAR} & : \text{when } \tau \neq \alpha \text{ and } \tau \equiv \tau' \Rightarrow \Phi \Rightarrow \Phi' \\
\text{GEQTYLEQVAR} & : \text{when } \tau \neq \alpha \text{ and } \tau \equiv \tau' \Rightarrow \Phi \Rightarrow \Phi' \\
\text{LEQVARCPTTY} & : \text{when } \tau \neq \alpha \text{ and } \tau \equiv \tau' \Rightarrow \Phi \Rightarrow \Phi' \\
\text{GEQVARCPTTY} & : \text{when } \tau \neq \alpha \text{ and } \tau \equiv \tau' \Rightarrow \Phi \Rightarrow \Phi'
\end{align*}
\]

(c) Type-variable comparison rule

\[
\begin{align*}
\text{LEQSAMEVAR} & : \Phi \vdash \alpha \leq \alpha > \Phi \\
\text{LEQVAREND} & : \text{when } \tau \neq \alpha \Rightarrow \text{when } \tau \equiv \tau' \Rightarrow \Phi \\
\text{GEQVAREND} & : \text{when } \tau \neq \alpha \Rightarrow \text{when } \tau \equiv \tau' \Rightarrow \Phi
\end{align*}
\]

Figure 7: Comparison rules
(a) Main compatibility rules

\[ \begin{array}{c}
\text{CPTNEW} \\
\text{when } \tau_1 \equiv \tau_2 \not\in \Phi \\
\Phi \vdash \tau_1 \vdash \tau_2 \not\in \Phi' \\
\Phi \vdash \tau_1 \equiv \tau_2 \not\in \Phi' \end{array} \]

\[ \begin{array}{c}
\Phi \vdash \tau_1 \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTALREADYPROVED} \\
\Phi, \tau_1 \not\in \Phi \vdash \tau_1 \equiv \tau_2 \not\in \Phi, \tau_1 \equiv \tau_2 \not\in \Phi \end{array} \]

(b) Standard compatibility rules

\[ \begin{array}{c}
\text{CPTBASETY} \\
\Phi \vdash \tau \equiv \tau \not\in \Phi \\
\Phi \vdash \tau \not\in \Phi \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTARROW} \\
\Phi \vdash \tau_1 \equiv \tau_1' \not\in \Phi' \\
\Phi \vdash \tau_2 \equiv \tau_2' \not\in \Phi'' \\
\Phi \vdash \tau_1 \equiv \tau_2 \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTARROWNULL} \\
\Phi \vdash \tau_1 \equiv \tau_1' \not\in \Phi' \\
\Phi \vdash \tau_2 \equiv \tau_2' \not\in \Phi'' \\
\Phi \vdash \tau_1 \equiv \tau_2 \not\in \Phi' \\
\end{array} \]

(c) Type-variable compatibility rules

\[ \begin{array}{c}
\text{CPTSAMEVAR} \\
\Phi \vdash \tau \equiv \tau \not\in \Phi \\
\Phi \vdash \tau \not\in \Phi \\
\end{array} \]

\[ \begin{array}{c}
\Phi, \alpha \equiv \tau \not\in \Phi' \\
\Phi, \tau \equiv \tau \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTVARLEQTY} \\
\text{when } \tau \not\in \alpha \text{ and } \tau \equiv \tau \not\in \Phi \\
\Phi, \alpha \equiv \tau \not\in \Phi' \\
\Phi, \tau \equiv \tau \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTTYLEQVAR} \\
\text{when } \tau \not\in \alpha \text{ and } \tau \equiv \tau \not\in \Phi \\
\Phi, \alpha \equiv \tau \not\in \Phi' \\
\Phi, \tau \equiv \tau \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTVARCPTTY} \\
\text{when } \tau \not\in \alpha \text{ and } \tau \equiv \tau \not\in \Phi \\
\Phi, \alpha \equiv \tau \not\in \Phi' \\
\Phi, \tau \equiv \tau \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\Phi, \alpha \equiv \tau \not\in \Phi' \\
\Phi, \tau \equiv \tau \not\in \Phi' \\
\end{array} \]

\[ \begin{array}{c}
\text{CPTVAREND} \\
\text{when } \tau \not\in \alpha \\
\text{when } (\forall \tau \mid \alpha \equiv \tau \not\in \Phi \ldots) \\
\text{when } (\forall \tau \mid \alpha \equiv \tau \not\in \Phi \ldots) \\
\end{array} \]

\[ \begin{array}{c}
\Phi \vdash \alpha \equiv \tau \not\in \Phi' \\
\end{array} \]

Figure 8: Compatibility rules

References


