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} <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.

In languages using the ML type discipline, the option type

\[ \text{type } \alpha \text{ option } = \text{None } | \text{Some } \alpha \]

injects regular values and the nullary data constructor None, into a single type that could be thought of 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 hides the partiality of functions. In OCaml, None and Some are automatically inserted by the compiler for dealing with optional arguments.

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(\_\_) is represented as \_\_ itself, that is, without allocating a block tagged as Some, we need a special representation for None, distinct from the representation of \_\_, for any \_\_. Of course, when \_\_ 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 Some(\_\_) values, for \_\_ >= 0 needs a special representation such that Some(\_\_) cannot be confused with Some(None) when \_\_ != j.

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

Now, the compilation of Some(expr) needs to generate a test, in order to use the special representation of Some(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 \( i \) 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 \( \text{NULL} \) \([3, 2, 11, 1]\), 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 \( \tau \) (that are assigned to expressions \( e \)) are pairs \((t, v)\) of a usual type \( t \) and a “nullability” type information \( v \). A type \((t, ?)\) corresponds to values that may be NULL, whereas \((t, \Delta)\) denotes values that cannot be NULL. Nullability variables are written \( \delta \). 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 naïve type system.

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

---

\[^1\]Although polymorphic recursion theoretically allows for unbounded depth of Some(None) while this representation allows for representing only a finite number of \( n \), this limitation should never be met in practice.
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 $\Phi, \Gamma \vdash e : (t, v) \triangleright \Phi'$, where $\Phi$ and $\Phi'$ are substitutions, $\Gamma$ is a type environment that maps program identifiers to type schemes (types with a prefix universal quantification of type and nullability variables). Such a judgement should be read as: given $\Phi$, under assumption $\Gamma$, the expression $e$ has type $\tau$ with substitution $\Phi'$.

The rules should be read as the different cases of an algorithm that, given $\Phi$, $\Gamma$, $e$, and $\tau$, computes substitution $\Phi'$ which, when applied to $\tau$ and $\gamma$, assigns 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 instantiation 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 3. The only interesting resolution rules are the ones that introduce (EQNEW) or merge (EQMERE) variable bindings in the $\Phi$ substitution.

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

EQMERE($\alpha$) merges a new constraint $\alpha = \tau'$ to a previous $\alpha = t$ occurring in $\Phi$. Here, the resulting substitution $\Phi'$ is obtained by resolving the constraint $t = \tau'$. EQMERE(δ) 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 handle operations such as being called as a function, tested or handle operations such as being called as a function, tested
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 bk =  
    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 (int, α) in order to be sent to + and (int,?) in order to have the same type as NULL.

## 4 Encoding nullability as variants

### 4.1 Encoding variants

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

```plaintext
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 inequalities are clearly not satisfiable, such as those produced when typing a conditional if NULL then ... else ..., where one fails to prove α ∈ bool, or an application NULL(...), failing to prove α ∈ τ₁ → τ₂. 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 10. The essential change that we bring to our initial system is in the TAPP
null
(a) Main comparison rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEQ</td>
<td>$\Phi \vdash \tau_2 \leq \tau_1 \Rightarrow \Phi'$</td>
</tr>
<tr>
<td>EQ</td>
<td>$\Phi \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi'$ $\Phi' \vdash \tau_1 \geq \tau_2 \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>LEQNEW</td>
<td>when $\tau_1 \leq \tau_2 \not\in \Phi$ $\Phi, \tau_1 \leq \tau_2 \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi'$</td>
</tr>
</tbody>
</table>

(b) Standard comparison rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQBASETy</td>
<td>$\Phi \vdash \ell_b \leq \ell_b \Rightarrow \Phi$</td>
</tr>
<tr>
<td>LEQARROW</td>
<td>$\Phi \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi'$ $\Phi' \vdash \tau_2 \leq \tau_1 \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>LEQBASENULL</td>
<td>$\Phi \vdash \ell_b \leq \ell_b \not\in \Phi$</td>
</tr>
<tr>
<td>LEQARROWnull</td>
<td>$\Phi \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi'$ $\Phi' \vdash \tau_2 \leq \tau_1 \Rightarrow \Phi''$</td>
</tr>
</tbody>
</table>

(c) Type-variable comparison rule

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQVARLEQTy</td>
<td>when $\tau' \neq \alpha$ and $\tau \approx \tau' \not\in \Phi$ $\Phi, \alpha \leq \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \Rightarrow \Phi'$ $\Phi' \vdash \tau \approx \tau' \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>EQGVARLEQTy</td>
<td>when $\tau' \neq \alpha$ and $\tau \leq \tau' \not\in \Phi$ $\Phi, \alpha \leq \tau, \tau \approx \tau' \leq \alpha \Rightarrow \Phi'$ $\Phi' \vdash \tau \approx \tau' \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>LEQTYLEQVAR</td>
<td>when $\tau' \neq \alpha$ and $\tau \leq \tau' \not\in \Phi$ $\Phi, \alpha \leq \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \Rightarrow \Phi'$ $\Phi' \vdash \tau \approx \tau' \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>GEGTYLEQVAR</td>
<td>when $\tau' \neq \alpha$ and $\tau \approx \tau' \not\in \Phi$ $\Phi, \alpha \leq \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \Rightarrow \Phi'$ $\Phi' \vdash \tau \approx \tau' \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>LEQVARCPTTy</td>
<td>when $\tau' \neq \alpha$ and $\tau \approx \tau' \not\in \Phi$ $\Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \Rightarrow \Phi'$ $\Phi' \vdash \tau \approx \tau' \Rightarrow \Phi''$</td>
</tr>
<tr>
<td>GEGVARCPTTy</td>
<td>when $\tau' \neq \alpha$ and $\tau \approx \tau' \not\in \Phi$ $\Phi, \alpha \approx \tau, \tau \approx \tau' \vdash \alpha \leq \tau' \Rightarrow \Phi'$ $\Phi' \vdash \tau \approx \tau' \Rightarrow \Phi''$</td>
</tr>
</tbody>
</table>

(continued)

Figure 7: Comparison rules
(a) Main compatibility rules

\[
\begin{align*}
\text{CPTNEW} & \quad \text{when } \tau_1 = \tau_2 \not\in \Phi \quad \Phi, \tau_1 \vdash \tau_2 \implies \Phi' \\
\Phi \vdash \tau_1 \approx \tau_2 \implies \Phi'
\end{align*}
\]

(b) Standard compatibility rules

\[
\begin{align*}
\text{CPTBASETY} & \quad \Phi, \tau_3 \vdash \Phi' \\
\Phi \vdash \tau_3 \approx \tau_3 \implies \Phi'
\end{align*}
\]

\[
\begin{align*}
\text{CPTARROW} & \quad \Phi, \tau_1 \vdash \Phi' \\
\Phi \vdash \tau_1 \rightarrow \tau_2 \approx \tau_1' \rightarrow \tau_2' \implies \Phi''
\end{align*}
\]

\[
\begin{align*}
\text{CPTARROWNULL} & \quad \Phi, \tau_1 \vdash \Phi' \\
\Phi \vdash \tau_1 \rightarrow \tau_2 \approx (\tau_1' \rightarrow \tau_2') \implies \Phi''
\end{align*}
\]

\[
\begin{align*}
\text{CPTBASENULL} & \quad \Phi, \tau_3 \vdash \Phi' \\
\Phi \vdash \tau_3 \approx \tau_3 \implies \Phi'
\end{align*}
\]

(c) Variable compatibility rules

\[
\begin{align*}
\text{CPTVARSAMEVAR} & \quad \Phi \vdash \alpha \equiv \beta \implies \Phi
\end{align*}
\]

\[
\begin{align*}
\text{CPTVARLEQTY} & \quad \text{when } \tau' \not\equiv \alpha \text{ and } \tau \equiv \tau' \not\in \Phi \\
\Phi, \alpha \leq \tau, \tau \equiv \tau' \implies \alpha \equiv \tau' \implies \Phi' \quad \Phi \vdash \tau \equiv \tau' \implies \Phi''
\end{align*}
\]

\[
\begin{align*}
\text{CPTTYLEQVAR} & \quad \text{when } \tau' \not\equiv \alpha \text{ and } \tau \equiv \tau' \not\in \Phi \\
\Phi, \alpha \leq \tau, \tau \equiv \tau' \implies \alpha \equiv \tau' \implies \Phi' \quad \Phi \vdash \tau \equiv \tau' \implies \Phi''
\end{align*}
\]

\[
\begin{align*}
\text{CPTVARCPTY} & \quad \text{when } \tau' \not\equiv \alpha \text{ and } \tau \equiv \tau' \not\in \Phi \\
\Phi, \alpha \equiv \tau, \tau \equiv \tau' \implies \alpha \equiv \tau' \implies \Phi' \quad \Phi \vdash \tau \equiv \tau' \implies \Phi''
\end{align*}
\]

\[
\begin{align*}
\text{CPTVAREND} & \quad \text{when } \tau' \not\equiv \alpha \\
\Phi \vdash \alpha \equiv \tau' \implies \Phi
\end{align*}
\]

Figure 8: Compatibility rules

References