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.
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

Impressive 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

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

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 [9] uses the option type to show possible failures in function types: it purposely 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(\_\_) 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 \_\_ ≠ 0.

For instance, unaligned addresses on 64-bits architectures, or a statically pre-allocated array of a sufficient size could do the job.\footnote{Although polymorphic recursion theoretically allows for unbounded depth of Some(\_\_) while this representation allows for representing only a finite number of \_\_, 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 Some(\_\_) 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 \_\_ 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 [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 naive approach, where the types \(\tau\)

\[
c ::= \text{true} \mid \text{false} \mid 0 \mid 1 \mid 2 \mid \ldots \mid + \mid \ldots
c ::= e \mid \lambda x. e_1 \mid e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3
\mid \text{let } x = e_1 \text{ in } e_2
\mid \text{NULL } \mid \text{case } e_1 \text{ of } \text{NULL } \rightarrow e_2 \parallel x \rightarrow e_3
\]

Figure 1: The language (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 \((t, \text{\_})\) 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 naive type system.

Section 5 presents a more sophisticated mechanism, where unification is replaced by subtyping constraints.
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 $\Phi \vdash e : \tau \triangleright \Phi'$, where $\Phi$ and $\Phi'$ are substitutions, $\Gamma$ is a type environment that maps program identifiers to type variables, 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 $\Gamma$, 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 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 2. The only interesting resolution rules are the ones that introduce (EQNEW) or merge (EQMERGE) variable bindings in the $\Phi$ substitution. $\text{EQNEW}(a)$, installs a binding $\alpha = \alpha'$ in the substitution $\Phi[\alpha \rightarrow \alpha']$ (that is, $\Phi$ in which free occurrences of $\alpha$ are replaced by $\alpha'$) when there is no previous binding about $\alpha$ in $\Phi$. Here, $\alpha'$ is either $\mu.\cdot, t$, depending on whether $\alpha$ occurs or not in $t$. $\text{EQNEW}(\delta)$ does the same, in a simpler context, on nullability variables.

$\text{EQMERGE}(a)$ merges a new constraint $\alpha = \alpha'$ to a previous $\alpha = t$ occurring in $\Phi$. Here, the resulting substitution $\Phi'$ is obtained by resolving the constraint $\alpha = t$. $\text{EQMERGE}(\delta)$ 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 for null, etc. The typing of values, procured 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 $\Phi, \Gamma \vdash e : \tau \triangleright \Phi'$, 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,
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:

```ml
let f b 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 `(int, α)` in order to be sent to `+` and `(int, ?)` in order to have the same type as `NULL`.

### 4 Encoding nullability as variants

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

```ml
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 elve '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` ? `τ_1 -> τ_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 6. The essential change that we bring to our initial system is in the TAPP
The inequality constraints, written $\Phi, \gamma \vdash x : \tau \to \Phi'$, $\Phi', \alpha \vdash e : \tau \to \Phi''$, $\Phi', \alpha \vdash \gamma : \tau \to \Phi'$, and $\Phi, \gamma \vdash x : \tau \to \Phi'$,

when the set $\{ \alpha \} \cup FTV(\Phi_\alpha)$ 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 $\text{TIMThenElse}$ and $\text{TCas}$ 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/](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

<table>
<thead>
<tr>
<th>GEQ</th>
<th>EQ</th>
<th>LEQNEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi \vdash \tau_2 \leq \tau_1 \Rightarrow \Phi')</td>
<td>(\Phi \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi')</td>
<td>(\Phi \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi')</td>
</tr>
<tr>
<td>(\Phi \vdash \tau_1 \geq \tau_2 \Rightarrow \Phi')</td>
<td>(\Phi \vdash \tau_1 = \tau_2 \Rightarrow \Phi'')</td>
<td>(\Phi \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi')</td>
</tr>
</tbody>
</table>

**LEQAlreadyProved**

\(\Phi, \tau_1 \leq \tau_2 \vdash \tau_1 \leq \tau_2 \Rightarrow \Phi, \tau_1 \leq \tau_2\)

(b) Standard comparison rules

<table>
<thead>
<tr>
<th>LEQBaseTy</th>
<th>LEQArrow</th>
<th>LEQBaseNull</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi \vdash I_1 \leq I_2 \Rightarrow \Phi)</td>
<td>(\Phi \vdash \tau_1 \leq \tau_2 \leq \tau_1 \Rightarrow \tau_2 \Rightarrow \Phi'')</td>
<td>(\Phi \vdash I_1 \leq I_2 \Rightarrow \Phi)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEQArrowNull</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi \vdash \tau_1 \leq \tau_2 \leq \tau_1 \Rightarrow \tau_2 \Rightarrow \Phi'')</td>
</tr>
</tbody>
</table>

(c) Type-variable comparison rule

<table>
<thead>
<tr>
<th>LEQSameVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Phi \vdash \alpha \leq \alpha &gt; \Phi)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEQVarEqTy</th>
<th>GEQVarEqTy</th>
<th>LEQTyEqVar</th>
<th>GEQTyEqVar</th>
<th>LEQVarCptTy</th>
<th>GEQVarCptTy</th>
<th>LEQVarEnd</th>
<th>GEQVarEnd</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\phi, \alpha \leq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \geq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \leq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \geq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \geq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \geq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \geq \alpha \leq \tau' \Rightarrow \Phi'')</td>
<td>(\phi, \alpha \geq \alpha \leq \tau' \Rightarrow \Phi'')</td>
</tr>
</tbody>
</table>

**Figure 7: Comparison rules**
### References


