An extended type error slicer

Vincent Rahli
supervisors: Professor Fairouz Kamareddine and Doctor J. B. Wells

ULTRA group, MACS, Heriot-Watt University

January 20, 2009
Programming languages are languages designed to instruct computers to do computations.

A language is usually defined by its 2 levels: syntactic and semantic.

The static semantics of a language is given by types. For example, usually the expression 1 is of type int (integer). Typing rules are used to associate types to syntactic expressions.

Well-typed programs can be guaranteed not to “go wrong” ([SSW06, Mil78])
(Most of) the implementations of SML use type inference algorithms based on the well known $\mathcal{W}$ algorithm (or its variants such as $\mathcal{M}$ or $\mathcal{UAE}$).
(The type inference algorithm used by the SML/NJ compiler is based on the $\mathcal{W}$ algorithm.)

All of these algorithms suffer a left-to-right bias.

A consequence of this bias is that the type errors reported by these algorithms can sometimes be far away from the real error locations.
Algorithm $\mathcal{W}$

$\mathcal{W}(A, e) = (S, \tau)$ where $A$ is a set of type assumptions, $e$ is an expression, $S$ is a substitution and $\tau$ is a type.

(If $\mathcal{W}(A, e) = (S, \tau)$ then $(SA, \tau)$ is a typing of $e$.)
Algorithm $\mathcal{W}$

$\mathcal{W}(A, e) = (S, \tau)$ where $A$ is a set of type assumptions, $e$ is an expression, $S$ is a substitution and $\tau$ is a type.

(If $\mathcal{W}(A, e) = (S, \tau)$ then $(SA, \tau)$ is a typing of $e$.)

Example: let the expression $f$ be \texttt{let val x = 0 in 1 :: x end}

Using Standard ML of New Jersey v110.52 we obtain the following error message:

```
Error: operator and operand don't agree
operator domain: int * int list
operand: int * int
in expression:
1 :: x
```
Algorithm $\mathcal{W}$

$\mathcal{W}(A, e) = (S, \tau)$ where $A$ is a set of type assumptions, $e$ is an expression, $S$ is a substitution and $\tau$ is a type.  
(If $\mathcal{W}(A, e) = (S, \tau)$ then $(SA, \tau)$ is a typing of $e$.)

Example: let the expression $f$ be `let val x = 0 in 1 :: x end`

If we assume that the type of 0 is different from the type of a list then the expression $f$ is not typable.

The $\mathcal{W}$ algorithm fails when trying to infer a type for $1 :: x$.

Using Standard ML of New Jersey v110.52 we obtain the following error message:

```
Error: operator and operand don’t agree [literal]
operator domain: int * int list
operand:       int * int
in expression: 1 :: x
```
Left-to-right bias
An example using the SML/NJ compiler

(1) We intended to write:

```sml
val g = fn x =>
    fn y =>
        let
            val f = if x
                then fn z => z + 1
                else fn z => z
        in
            f y
        end
```

(2) We wrote:

```sml
val g = fn x =>
    fn y =>
        let
            val f = if y
                then fn z => z + 1
                else fn z => z
        in
            f y
        end
```

(1) for example `g true 2` evaluates to 3 and `g false 2` evaluates to 2.

(2) Using Standard ML of New Jersey v110.52 we obtain the following error message:

```
Error: operator and operand don't agree [literal]
operator domain: int
operand: bool
in expression:
    f y
```
Left-to-right bias
An example using the SML/NJ compiler

Recall:

```sml
val g = fn x =>
  fn y =>
    let
      val f = if y
        then fn z => z + 1
        else fn z => z
    in f y
  end
```

In our simple example the programmer’s error is not far away from the reported error.

But it can happen that `y` is constrained to be of type `bool` because of some code located far away from the location proposed by SML/NJ (or in another file).
How to overcome this left-to-right bias?

The earlier inference algorithms use a unification algorithm during their process.

In some new algorithms [SSW06, HW04], the two processes are split:

- First, the type inference algorithm generates type constraints for a given expression. Let us consider the following declaration $d$: \texttt{val \ x = 1}.
  One of the constraints generated for $d$ is that the type of $x$ has to be equal to the type of 1, but the type inferred for $x$ is not actually int.

- Then, it applies a unification algorithm to the generated set of constraints.
The type error slicing project

- The type error slicer developed by Haack and Wells [HW04] is based on this new kind of algorithm (generation of type constraints then unification).

- It uses intersection types instead of for all types (it allows compositional analysis).

- As for similar projects [Wan86, HJSA02, SSW06], justifications are associated to the generated constraints to keep track of the type deductions. A label is associated to (almost) each term:

  The label \( l \) is associated to the expression \( 1: 1' \).

  At this point a constraint labelled by \( l \) is generated specifying that the type of \( 1 \) is equal to the integer type.

- A type error is identified to a (minimal) set of justifications.
The type error slicing project

- The slicer developed by Haack and Wells goes further by computing a **minimal slice** from a minimal set of justifications.

- These minimal slices are designed so that they present all and only the information needed by the programmer to repair its errors.

- Their slicer handles a small extension of the terms typable by HM.

Haack and Wells’s slicer meet the criteria listed in [YWTM00]: **correct**, **precise**, **succinct**, **non-mechanical** (for example, no artificial type variable), **source-based** (this is almost true, the slices actually contain some extra parentheses and dots), **unbiased**, **comprehensive** (every location needed by the programmer to solve his error is reported).
Type error slicing
The steps of Haack and Wells’s slicer

3 main steps:
▶ Generations of the type constraints for to a given term.
▶ Enumeration of the minimal unsatisfiable sets of constraints. The enumerator makes an extensive use of a unification algorithm.
▶ Computation of a slice from each minimal set of justifications (extracted from a minimal unsatisfiable set of constraints).
Type error slicing

Example

Recall:

```ocaml
val g = fn x =>
  fn y => let
    val f = if y
      then fn z => z + 1
      else fn z => z
    in f y
  end
```

We can solve the errors by replacing `y` by `x`.

We can also solve the errors by replacing `z + 1` by `not z`.

Haack and Wells’s slicer computes two slices (two minimal type errors):

```
( . . y = > ). . f y . . )
( . . y = > )
```

Vincent Rahli
An extended type error slicer
January 20, 2009 13/22
Recall:

```
val g = fn x =>
  fn y =>
    let
      val f = if y then fn z => z + 1 else fn z => z
    in f y
  end
```

Haack and Wells’s slicer computes two slices (two minimal type errors):

1. 

   (. . y => (. . val f = if y then fn z => (z + ( . . )) else ( . . ) . . f y . . ) . . )

2. 

   (. . y =>
     (. . val f = if y then fn ( . . ) => (( . . ) + ( . . )) else fn z => z
     .. f y ..) .. )
Type error slicing
Example

Recall:

\[
\begin{align*}
\text{val } g &= \\
\text{fn } x &\Rightarrow \\
\text{fn } y &\Rightarrow \\
\text{let } \text{val } f &= \text{if } y \\
&\text{then } \text{fn } z &\Rightarrow z + 1 \\
&\text{else } \text{fn } z &\Rightarrow z \\
\text{in } f y \\
\text{end }
\end{align*}
\]

We can solve the errors by replacing \( y \) by \( x \).
We can also solve the errors by replacing \( z + 1 \) by \( \text{not } z \).

Haack and Wells’s slicer computes two slices (two minimal type errors):

\[
(\ldots \ y \Rightarrow (\ldots \ \text{val } f = \text{if } y \text{ then } \text{fn } z \Rightarrow (z + (\ldots)) \text{ else } (\ldots) \ldots f y \ldots) \ldots)
\]

\[
(\ldots \ y \Rightarrow (\ldots \ \text{val } f = \text{if } y \text{ then } \text{fn } (\ldots) \Rightarrow ((\ldots) + (\ldots)) \text{ else } \text{fn } z \Rightarrow z \ldots f y \ldots) \ldots)
\]
An extended slicer

- We consider new programming features such as data types. Example:

```
datatype Nat = Z | S of Nat
and LC = VAR of Nat | ABS of LC | APP of (LC * LC)
```

- In `val x = D true`, `D` can be a value variable or a value constructor.

- We don’t want to make assumptions over the status of the identifiers and we want our slicer to be compositional.
Recall:

```plaintext
val g = fn x => fn y =>
  let
    val f = if y then fn z => z + 1
             else fn z => z
  in  f y
end
```

Here is one of the slice computed by Haack and Wells's slicer:

```plaintext
( . . y ⇒ (. . val f = if y then fn z ⇒ (z + (. .)) else (. .) .. f y ..) ..)
```

This slice wouldn’t be a slice if y was a value constructor.

In our extended slicer we specify some **constraints** for the slice to exist: y, f and z have to be value variables and not value constructors.
For example if we have a declaration such as \texttt{datatype} \( t = y \), we wouldn’t obtain the slice presented above.

Using Standard ML of New Jersey v110.52 we obtain the following error message:

\begin{verbatim}
Error: test expression in if
    is not of type bool [tycon mismatch]
    test expression: ?.t
    in expression:
        if y then (fn z => z + 1) else (fn z => z)
\end{verbatim}

Programmer’s error might be in the \texttt{datatype} declaration. One of the slice we obtain is:

\begin{verbatim}
(.
  \texttt{datatype} (.,.) = (.,y,.)
  .
  val f = if (.,) then fn (.,(.,) => (.,) + (.,)) else fn (.,z => z,.,) f y

Vincent Rahli
An extended type error slicer
January 20, 2009 18/22
\end{verbatim}
An extended slicer

Example

For example if we have a declaration such as \textbf{datatype} \( t = y \), we wouldn’t obtain the slice presented above.

```
let
  datatype t = y
  val g = fn x =>
    fn y =>
      let
        val f = if y
          then fn z => z + 1
          else fn z => z
        in f y
      end
  in ()
end
```

Using Standard ML of New Jersey v110.52 we obtain the following error message:

```
Error: test expression in if
  is not of type bool [tycon mismatch]
  test expression: ?.t
  in expression:
    if y then (fn z => z + 1) else (fn z => z)
```

Programmer’s error might be in the \textbf{datatype} declaration. One of the slice we obtain is:

```
(\ldots\textbf{datatype} \ldots) = (\ldots y \ldots)\ldots\textbf{val} f = \textbf{if} (\ldots) \textbf{then} \textbf{fn} (\ldots z \Rightarrow z + (\ldots)\ldots) \textbf{else} (\ldots)\ldots f y\ldots)
```
For example if we have a declaration such as `datatype t = y`, we wouldn’t obtain the slice presented above.

Using Standard ML of New Jersey v110.52 we obtain the following error message:

```
Error: test expression in if
is not of type bool [tycon mismatch]
test expression: ?.t
in expression:
  if y then (fn z => z + 1) else (fn z => z)
```

Programmer’s error might be in the `datatype` declaration. One of the slice we obtain is:

```
( . . datatype ( . . ) = ( . . y . . ) . . if y then ( . . ) else ( . . ) . . )
```
Remark: As for Haack and Wells we also highlight the slices into the original piece of code.

- Finish the technical parts of our slicer (finish the implementation, test the implementation, improve the syntax of the slices, improve the highlighting).

- Prove the different properties (termination, correctness, minimisation, ...) of the different modules of our slicer.

- Extend the framework to a bigger language.
Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer.
Improving type-error messages in functional languages.

Christian Haack and J. B. Wells.
Type error slicing in implicitly typed higher-order languages.

Robin Milner.
A theory of type polymorphism in programming.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny.
Type processing by constraint reasoning.

Mitchell Wand.
Finding the source of type errors.
In In Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’86), pages 38–43, New York, NY, USA, 1986. ACM.

J. Yang, J. Wells, P. Trinder, and G. Michaelson.
Improved type error reporting.