Challenges of a type error slicer for the SML language

Vincent Rahli and J. B. Wells and Fairouz Kamareddine

ULTRA group, MACS, Heriot-Watt University

May 8, 2009

The SML programming language

- SML is a higher-order function-oriented imperative programming language.
- It has polymorphic types.
- It has a sophisticated, flexible and safe (all program behavior is guaranteed to be well defined) type system.

Syntax:

```
fun factorial 0 = 1 | factorial 1 = 1 | factorial n = n * factorial (n - 1)
```

W algorithm

- Most of the implementations of SML use type inference algorithms based on the well known **W algorithm**.
  (The type inference algorithm used by the SML/NJ compiler is based on the W algorithm.)

- W uses a unification algorithm to infer the type of every application in a term. W fails when the unification fails. The node blamed by W is only the node where the unification failed.

- Because W blames only one node when failing and because of its traversal of the abstract syntax trees, the type errors reported can sometimes be far away from the real error locations.

W algorithm

W takes as input: a set of type assumptions and an expression
W returns: a modified set of type assumptions and a type

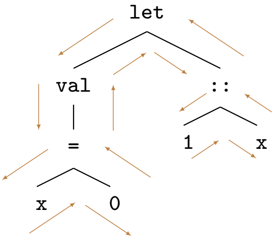
(Remark: if $\mathrm{W}(A, e) = (S, \tau)$ then (SA, τ) is a typing of e.)

W algorithm

W takes as input: a set of type assumptions and an expression
W returns: a modified set of type assumptions and a type

(Remark: if $W(A, e) = (S, \tau)$ then (SA, τ) is a typing of e.)

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

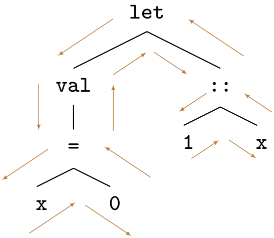


W algorithm

W takes as input: a set of type assumptions and an expression
W returns: a modified set of type assumptions and a type

(Remark: if $W(A, e) = (S, \tau)$ then (SA, τ) 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 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
```

Substitutes for W (1)

To summarise, W:

- identifies only one location as the error
- often identifies a location far away from the real error location
- often identifies locations which do not participate in the error

Earlier algorithms:

- M tries to do better than W by sometimes reporting smaller subtrees than W.
- There are many other algorithms trying to improve W.
  UAE uses another unification leading to better error report than W but still retains a bias in handling of let-bindings.

All these algorithms try to report different locations but all suffer from the same problem: **they report only one location when sets of locations are usually involved in a type error**.

Confusing error messages

An example using the SML/NJ compiler

(1) We intended to write:

```
fun g x y =
  let
      val f = if x
              then fn _ => fn z => z
              else fn z => z
      val u = (f, true)
  in (#1 u) y
  end
```

(2) We wrote:

```
fun g x y =
  let
      val f = if y
              then fn _ => fn z => z
              else fn z => z
      val u = (f, true)
  in (#1 u) y
  end
```

(1) for example `(g true (fn x => x + 1)) 2` evaluates to `2` and `(g false (fn x => x + 1)) 2` evaluates to `3`.

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

```
Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z -> 'Z
  operand:         bool
  in expression:
    ((fn {1=<pat>,...} => 1) u) y
```

Confusing error messages

An example using the SML/NJ compiler

Recall:

```
fun g x y =
  let
      val f = if y
              then fn _ => fn z => z
              else fn z => z
      val u = (f, true)
  in (#1 u) y
  end
```

```
Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z -> 'Z
  operand:         bool
  in expression:
    ((fn {1=<pat>,...} => 1) u) y
```

In this example, programmer's error is not far away from the reported error. It is not always the case: the real error location might even be in another file.

Problems:

- SML/NJ reports only one location
- the reported location is far from the real error location
- `'Z -> 'Z` is an internal type made up by SML/NJ
- the reported expression does not match the source code

An implementation of W

We reported that the type SML/NJ's inference algorithm is based on W.

We saw that SML/NJ's reports are:

- Biased: it reports only one location far from the real error location.
- Mechanical: it reports internal type variable (`'Z`).
- Non source-based: the reported expression does not match the source code. The code goes through some transformations before being reported.

As reported by Yang et al. [YWTM01], a "good" report should be:

correct	(reports errors only for pieces of code that are ill-typed)
precise	(reports no more than the conflicting portions of code)
succinct	(short reports)
non-mechanical	(no internal mechanical details)
source-based	(reports only portions of source code)
unbiased	(no location is privileged over the others in an error)
comprehensive	(reports all the conflicting portions of code)

New type inference algorithms

How to obtain all the locations participating in an error?

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

Some new algorithms [SSW06, HW04], split the two processes:

- **Generation of type constraints** for a given expression.

  Let us consider the following declaration *d*: `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` (the one in the type environment) is not actually `int`.

- Application of a **unification algorithm** to the generated constraints.

Substitutes for W (2)

New approaches:

- Haack and Wells's type error slicer [HW04] for SML.
- Neubauer and Thiemann's type error slicer [NT03] based on flow analysis and union types.
- Stuckey, Sulzmann and Wazny's type error slicer [SSW06] for Haskell implemented in their Chameleon framework.
- Lerner, Flower, Grossman and Chambers's approach [LFGC07] consisting in using different heuristics to build a well-typed program from an ill-typed one.

The type error slicing project
Haack and Wells's type error slicer

- The type error slicer developed by Haack and Wells [HW04] uses this new kind of algorithm (generation of type constraints then unification).
- It is inspired by **intersection types** instead of "for all" types (it allows compositional analysis).
- As for similar projects [Wan86, HJSA02, SSW06], "**reasons**" 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^l.

  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 reasons.

- Haack and Wells's type error slicer computes a **minimal slice** from a minimal set of reasons.
- They also highlight the slices in the source code.
- These minimal slices 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 [YWTM01].

3 main steps:

- Generations of the type constraints for to a given term.

  $\{\mathrm{int} \stackrel{l_1}{=} \alpha_1, \alpha_1 \stackrel{l_2}{=} \mathrm{bool}, \alpha_1 \stackrel{l_3}{=} \alpha_2\}$

- Enumeration of the minimal unsatisfiable sets of constraints. The enumerator makes an extensive use of a unification algorithm.

  $\{\mathrm{int} \stackrel{l_1}{=} \alpha_1, \alpha_1 \stackrel{l_2}{=} \mathrm{bool}\}$

- Computation of a slice from each minimal set of reasons (extracted from a minimal unsatisfiable set of constraints).

  $\{l_1, l_2\}$

Here is the highlighting we obtain for the code presented before:

```
fun g x y =
  let
      val f = if y then fn _ => fn z => z else fn z => z
      val u = (f, true)
  in (#1 u) y
  end
```

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

We can also solve the error by replacing the last occurrence of `z` by `fn z => z`.

Here is the highlighting we obtain for the code presented before:

```
fun g x y =
  let
      val f = if y then fn _ => fn z => z else fn z => z
      val u = (f, true)
  in (#1 u) y
  end
```

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

We can also solve the error by replacing the last occurrence of `z` by `fn z => z`.

We aim to:

- Extend Haack and Wells's type error slicer to the **full** SML language.
- Provide **detailed** highlighting and slices for every SML feature.

Our approach is close to Stuckey, Sulzmann and Wazny's approach [SSW06].

Some differences between our type error slicers are:

- SML vs. Haskell.
- Recall: we want to provide detailed slices where every location participating in errors is present in our slices and highlighting.
- One important difference is that Stuckey, Sulzmann and Wazny don't "burden" the user, for example, by highlighting the white spaces between a function and its arguments when this is crucial in our approach as we will see later on in the talk.
- They don't seem to highlight parts of datatype declarations.

A first step consists in adding the following features:

- datatype declarations
- records
- exceptions
- type declarations
- explicit types
- unrestricted value declarations
- mutually recursive functions
- value polymorphism
- scope of explicit type variables
- tuples
- list
- while loops
- case expressions
- sequencing of expressions
- conditional
- `fun` syntax

Extension of Haack and Wells's type error slicer
Datatypes

- Example of a datatype declaration:

```
datatype Nat = z | s of Nat
and LC = var of Nat | abs of LC | app of LC * LC
```

- This feature raises the issue of the distinction between value variables and value constructors in SML.
- In `fun f x = D true`, `D` can be a **value variable** or a **value constructor**.
- We shouldn't make assumptions over the status of identifiers:

This is a minimal slice only if `c` is a value variable: `fn c => (c 1, c true)`
It does not exist if `c` is a value constructor:
`datatype t = c; fn c => (c 1, c true)`

The errors we catch are:

Semantic errors:

- clashes between type constructors
- different arities for the same type name
- circularity errors (SML forbids recursive types)
- clashes between labels of records

Context-sensitive syntactic errors:

- multi-occurrences of identifiers
- application of value variable in a pattern
- identifier occurring in a pattern both applied and not applied
- free explicit type variables in datatype/type declarations
- definition of a function with different names
- free explicit type variable at top level
- value constructor occurring in a pattern on the left of a "as".

The value constructor `C2` is applied but defined without argument. (application as an end point)

```
fun ex2 z = let datatype Y = C2 | C3 of int in C2 z end
```

`u` and `v` occur with one and two parameters.

```
datatype 'a t = U of (bool -> ('a, 'a) u) u | V of ('a, 'a) v v
```

```
datatype 'a t = U of (bool -> ('a, 'a) u) u | V of ('a, 'a) v v
```

The value constructor `C2` is applied but defined without argument. (application as an end point)

```
datatype   = C2          C2
```

`u` and `v` occur with one and two parameters.

```
( , ) u u
```

```
( , ) v v
```

There is circularity problem when trying to infer a type for `f` because of the conflict between the definition of the function and its use.

```
fun f () = f () 0
```

Conflicting record labels.

```
val {foo,bar} = {fool=0,bar=1}
val {foo,bar} = {fool=0,bar=1}
```

A function is defined with names `f` and `g`.

```
fun f 0 = 1
  | g n = n + 1
```

There is circularity problem when trying to infer a type for `f` because of the conflict between the definition of the function and its use.

```
fun f [] = f [] []
```

Conflicting record labels.

```
val {foo,bar} = {fool= ,     }
val {foo,   } = {fool= ,bar= }
```

A function is defined with names `f` and `g`.

```
fun f   =
  | g   =
```

If `f` is a value variable, it shouldn't occur twice in a pattern.

```
fn fn (f, f y, g x) => x + y
```

If `g` is a value variable, it shouldn't be applied in a pattern.

```
fn fn (f, f y, g x) => x + y
```

`f` occurs both applied and not applied in a pattern.

```
fn fn (f, f y, g x) => x + y
```

If `f` is a value variable, it shouldn't occur twice in a pattern.

```
fn  f  f        =>
```

If `g` is a value variable, it shouldn't be applied in a pattern.

```
fn          g ▯  =>
```

`f` occurs both applied and not applied in a pattern.

```
fn  f  f ▯      =>
```

Extension of Haack and Wells's type error slicer

Free explicit type variables in datatype/type declarations
Free explicit type variable at top level (context-sensitive error)
Value constructor occurring in a pattern on the left of a "as"

`'b` is free in the datatype declaration.

```
datatype 'a t = T of (bool -> (('a, 'b) w)) w
```

If `'a` is at top level then it is free.

```
exception e of 'a
```

The value constructor `c` occurs directly on the left of a "as".

```
datatype t = c; val c as (x, y) = (1, true)
```

Extension of Haack and Wells's type error slicer

Free explicit type variables in datatype/type declarations
Free explicit type variable at top level (context-sensitive error)
Value constructor occurring in a pattern on the left of a "as"

'b is free in the datatype declaration.

```
datatype 'a = 'b
```

If 'a is at top level then it is free.

```
exception of 'a
```

The value constructor c occurs directly on the left of a "as".

```
datatype = c val c as =
```

Extension of Haack and Wells's type error slicer

How useful is our type error slicer?

```
datatype ('a, 'b, 'c) t = Red of 'a * 'b * 'c
                        | Blue of 'a * 'b * 'c
                        | Pink of 'a * 'b * 'c
                        | Green of 'a * 'b * 'b
                        | Yellow of 'a * 'b * 'c
                        | Orange of 'a * 'b * 'c
fun trans (Red (x, y, z)) = Blue (y, x, z)
  | trans (Blue (x, y, z)) = Pink (y, x, z)
  | trans (Pink (x, y, z)) = Green (y, x, z)
  | trans (Green (x, y, z)) = Yellow (y, x, z)
  | trans (Yellow (x, y, z)) = Orange (y, x, z)
  | trans (Orange (x, y, z)) = Red (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t * 'b
val x = (Red (2, 2, false), true)
val y : (int, bool) u = (trans (#1 x), #2 x)
```

should be 'c

(the error is context-sensitive: only obtained if y and z are value variables)

SML/NJ reports:

```
operator domain:  (int,int,int) t
operand:  (int,int,bool) t
in expression:
  trans ((fn 1=<pat>,...  => 1) x)
```

Extension of Haack and Wells's type error slicer

How useful is our type error slicer?

```
datatype ('a,   , 'c) t =


                         Green of    * 'b * 'b
                         Yellow of 'a *    * 'c

fun
  |
  |
  | trans  Green ( , y, z)  = Yellow (y,   , z)
  |
  |
type ('a, 'b) u = ('a,   , 'b) t *

val   : (int, bool) u = (trans        ,     )
```

(the error is context-sensitive: only obtained if y and z are value variables)

SML/NJ reports:

```
operator domain:  (int,int,int) t
operand:  (int,int,bool) t
in expression:
  trans ((fn 1=<pat>,...  => 1) x)
```

Extension of Haack and Wells's type error slicer

Recursive functions

We often obtain more than one explanation for a same error:

should be `g x y true`

```
fun g x y z = if z then x + y else y
fun f [] y = y
  | f [x] y = g x y y
  | f (x :: xs) y = x + (f xs y)
```

```
fun g x y z = if z then x + y else y
fun f [] y = y
  | f [x] y = g x y y
  | f (x :: xs) y = x + (f xs y)
```

We use ⟦..⟧ to replace irrelevant portions of code for an error to occur.

(1) `if z then ⟦..⟧ + ⟦..⟧ else ⟦..⟧`

(2) `⟦..if z then ⟦..⟦..⟧ + y..⟧ else ⟦..⟧..⟧`

(3) `⟦..⟧..⟦..y..⟦..= ⟦..g ⟦..⟧ y y..⟧`

(1): it matters that g returns an integer
(2): it doesn't matter if g returns an integer
(3): it doesn't matter if the function has more arguments

One of our example was:
$\texttt{fool} \notin \{\texttt{foo}, \texttt{bar}\}$ $\texttt{foo} \notin \{\texttt{fool}, \texttt{bar}\}$.

```
val {foo,bar} = {fool=0,bar=1}
val {foo,bar} = {fool=0,bar=1}
```

In this case it might be better to present the two slices together as follows:

```
val {foo,bar} = {fool=0,bar=1}
```

where orange would be used for common end points.

Minimality would be: green (resp. blue) in one record and blue (resp. green) and orange in the other record.

We developed two ways to deal with the standard basis so far:

- A built-in subset of the standard basis is implemented in our type error slicer.
- Joe Wells developed a tool extracting from a running SML/NJ session the predefined environments, containing the standard basis but also the own declarations of the user of the session.

  It has problems and needs a better compiler support.

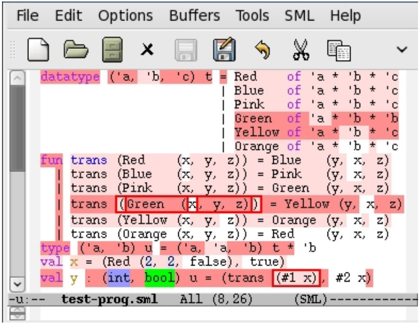
  Example of problem to face: the numerous presence of hidden structures and types (`?.int32`).

Extension of Haack and Wells's type error slicer

Interaction with a developing environment

Joe Wells developed a highlighting mode of SML type errors for emacs.

```
datatype ('a, 'b, 'c) t = Red of 'a * 'b * 'c
                        | Blue of 'a * 'b * 'c
                        | Pink of 'a * 'b * 'c
                        | Green of 'a * 'b * 'b
                        | Yellow of 'a * 'b * 'c
                        | Orange of 'a * 'b * 'c
fun trans (Red (x, y, z)) = Blue (y, x, z)
  | trans (Blue (x, y, z)) = Pink (y, x, z)
  | trans (Pink (x, y, z)) = Green (y, x, z)
  | trans (Green (x, y, z)) = Yellow (y, x, z)
  | trans (Yellow (x, y, z)) = Orange (y, x, z)
  | trans (Orange (x, y, z)) = Red (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t * 'b
val x = (Red (2, 2, false), true)
val y : (int, bool) u = (trans (#1 x), #2 x)
```



The light red areas are the ones participating in other slices.

Conclusion and Future work

Conclusion:

- We formalised a restricted version of our type error slicer implementation.
- Our type error slicer is implemented in SML.
- It provides detailed error reports: in-place highlighting and separate slices.
- Our slicer is nearing usability on full programs.

Near future work:

- Finishing implementing support for structures.
- Solve efficiency problem (constraints set size).
- Test with real users.

Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer.
Improving type-error messages in functional languages.
Technical report, Utrecht University, 2002.

Christian Haack and J. B. Wells.
Type error slicing in implicitly typed higher-order languages.
Science of Computer Programming, 50(1-3):189–224, 2004.

Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers.
Searching for type-error messages.
In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 2007.

Matthias Neubauer and Peter Thiemann.
Discriminative sum types locate the source of type errors.
In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 15–26. ACM, 2003.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny.
Type processing by constraint reasoning.
In Naoki Kobayashi, editor, *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2006.

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

J. Yang, J. Wells, P. Trinder, and G. Michaelson.
Improved type error reporting.
In Markus Mohnen and Pieter W. M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, volume 2011 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2001.