SML-TES, a type error slicer for SML: from constraint generation to minimisation

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

ULTRA group, MACS, Heriot-Watt University

January 21, 2010

The SML-TES team

Current Implementation (for full SML)

- Joe Wells: Theory and Implementation Consulting Advice and Emacs Interface
- Fairouz Kamareddine: Theory Consulting
- Steven Shiells: Web Interface Implementation, Emacs Interface, packaging
- David Dunsmore: Testing of the effectiveness of the Slicer
- Vincent Rahli: Type Error Slicer Theory and Implementation

Earlier Implementation (for a tiny subset of SML)

- Joe Wells: Theory and Implementation Consulting Advice
- Christian Haack: Type Error Slicer Theory and Implementation
- Sébastien Carlier: Web Interface Implementation

The SML programming language

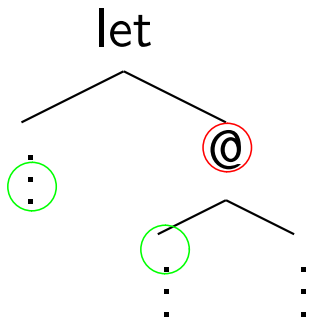
- SML is a higher-order function-oriented imperative programming language.
- It has polymorphic types.
- It has a safe (all program behavior is guaranteed to be well defined) type system.
- It has a definition.

Syntax:

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

W algorithm

- Most of SML's implementation (e.g., SML/NJ) use type inference algorithms based on the well known **W algorithm**.
- W uses a unification algorithm to infer the type of every application in a term. W fails when unification fails. The only node blamed by W is the node where unification failed.
- Because W blames only one node when failing and because of its traversal of the abstract syntax trees, the type error reports can sometimes be confusing and far away from the real programming error locations.



W algorithm

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

The infix constructor `::` is one of the two list constructors (`nil` and `::`).

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

W fails when trying to infer a type for `1 :: x` and its implementations blame this location but not `x = 0`.

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

Other algorithms such as M or UAE try to report different locations but suffer from the same problems.

All these algorithms try to report different locations but all suffer from the same problem: **they report only one location when a location set is 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

```
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, the programmer's error is not too far away from the reported location. 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

SML/NJ: an implementation of W

We reported that 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 variables (`'z`).
- Non source-based: the reported expression does not match the source code. The code gets transformed 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

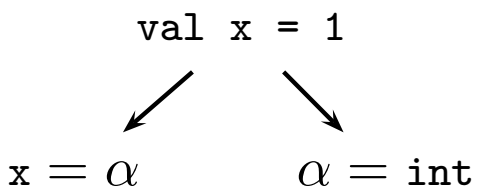
How to obtain all the locations participating in an error?

Earlier inference algorithms use a unification algorithm during their process.

New algorithms [HW04, PR05, SSW06], split the two processes:

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

  Let us consider the following declaration d: `val x = 1`.
  At constraint generation (where α is a type variable):



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

Substitutes for W (2)

New approaches to type error reporting:

- 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 slicer developed by Haack and Wells [HW04] relies on a constraint-based type inference algorithm.
- As for similar projects [Wan86, HJSA02, SSW06], "**reasons**" are associated to the generated constraints to keep track of the locations responsible for the type deductions.
  A label is associated to (almost) each term:

$$\begin{array}{c} \mathtt{1}^{l} : \alpha \\ \downarrow \\ \alpha \overset{\{l\}}{=} \mathtt{int} \end{array}$$

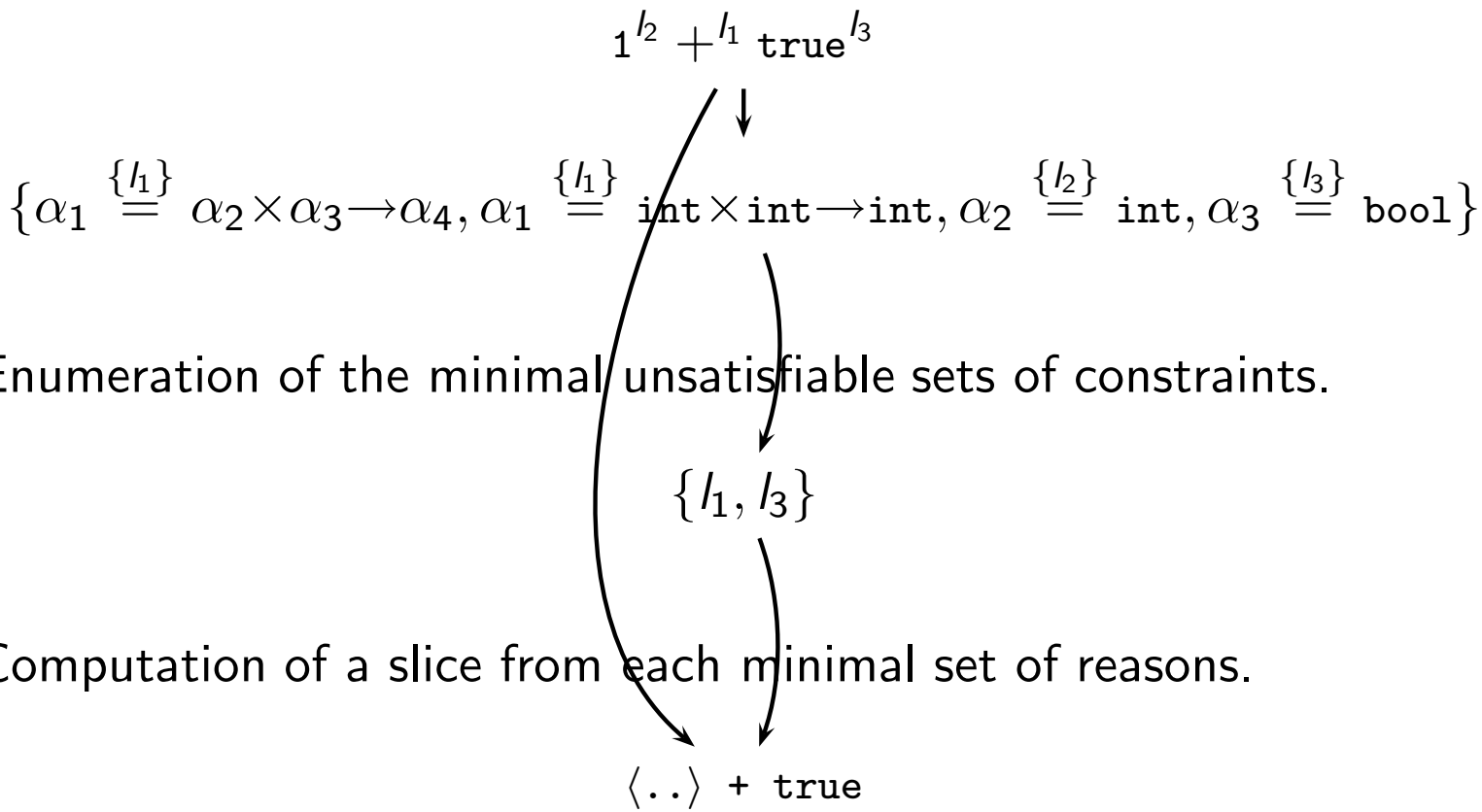
  ($\mathtt{1}^{l} : \alpha$ means that the type α is associated to the labelled term $\mathtt{1}^{l}$.)
- A type error is identified as a (minimal) set of reasons.

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

- A slice is a program in which some nodes have been discarded (e.g., from `1 + true`, one can generate $\langle . . \rangle$ `+ true`).
- A type error slice is minimal if it is untypable and any smaller slice is typable (e.g., $\langle . . \rangle$ `+ true` is a minimal type error slice).
- 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 type constraints for a given term.



- Enumeration of the minimal unsatisfiable sets of constraints.
- Computation of a slice from each minimal set of reasons.

A label set called filters (search space) is maintained during enumeration.

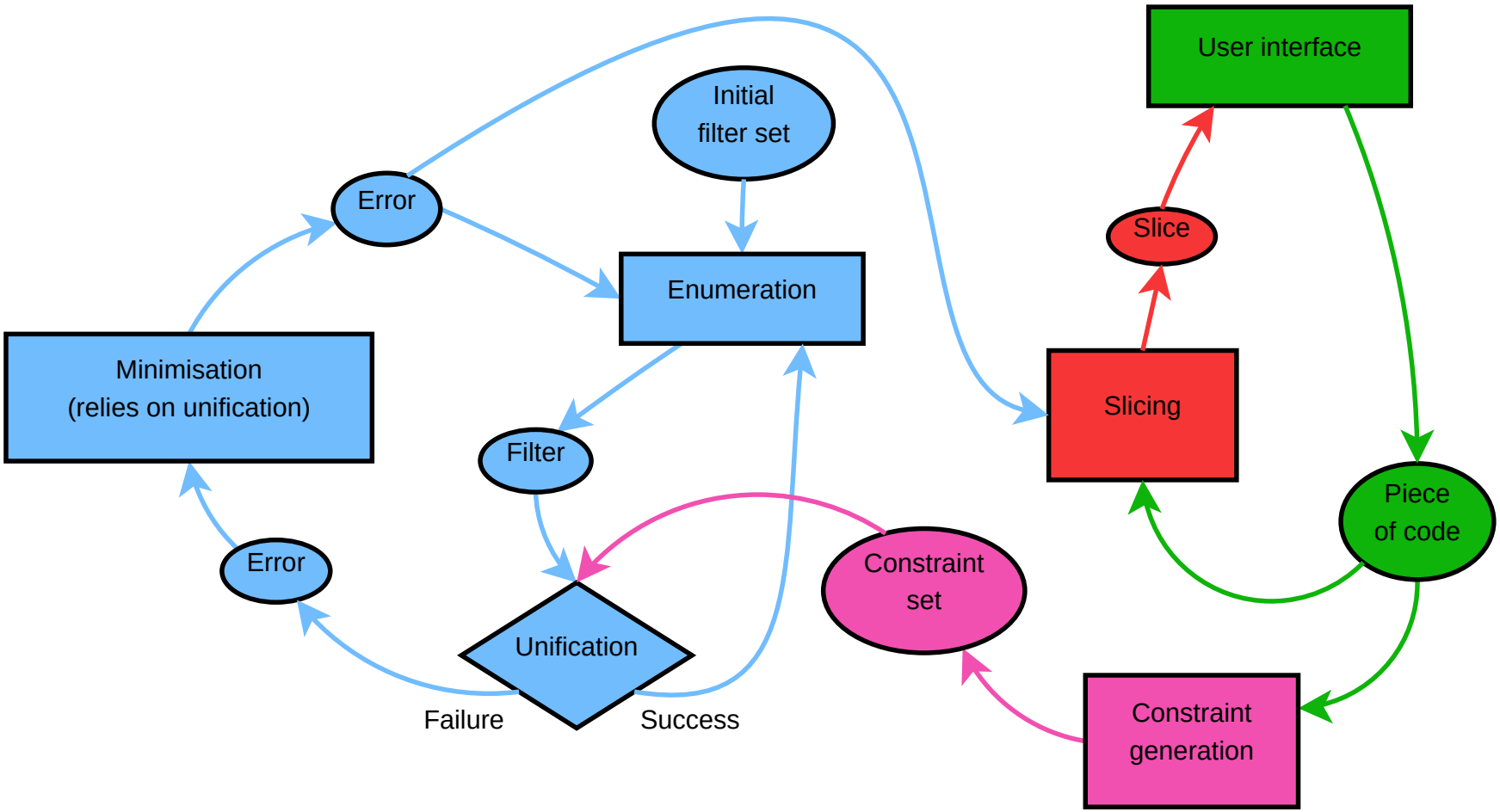
When enumeration starts, the filter set is $\{\varnothing\}$.

An enumeration step is as follows:

- We select a filter *filter* from the filter set.
- We unify the constraints which are not labelled by *filter*.
  - If the constraint set is solvable then *filter* is discarded.
  - Otherwise, we obtain a type error labelled by a label set $\overline{l}$.
    - We minimise this error and obtain a smaller error labelled by $\overline{l}' \subseteq \overline{l}$.
    - We create new filters as follows: $\{\{l\} \cup \mathit{filter} \mid l \in \overline{l}'\}$.
    - We discard *filter* and augment our filter set with the new ones. (This insure that the enumerated errors are disjoint from each others.)

The type error slicing project

The steps of our type error slicer



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.

The main differences with similar approaches (e.g., Stuckey, Sulzmann and Wazny's approach [SSW06]) is that:

- We provide detailed slices where every location participating in errors is present in our slices and highlighting. For example we highlight parts of datatype declarations when participating in type errors.
- Another example is white spaces, such as the white spaces between the two terms of an application when the first term is not a function.
- We deal with SML so, as with any programming language, we have to deal with its particularities: non-lexical distinction between datatype constructors and value variables or value polymorphism restriction.

Extension of Haack and Wells's type error slicer
Example

Example of a datatype declaration:

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

Two explanations:

The datatype constructor `APP` is applied to two arguments but takes only one.

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

The datatype constructor `VAR` is a `LC` constructor and not a tuple constructor.

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

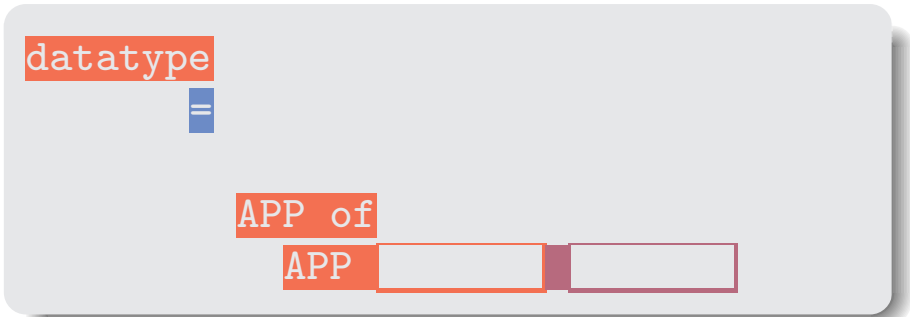
Extension of Haack and Wells's type error slicer
Example

Example of a datatype declaration:

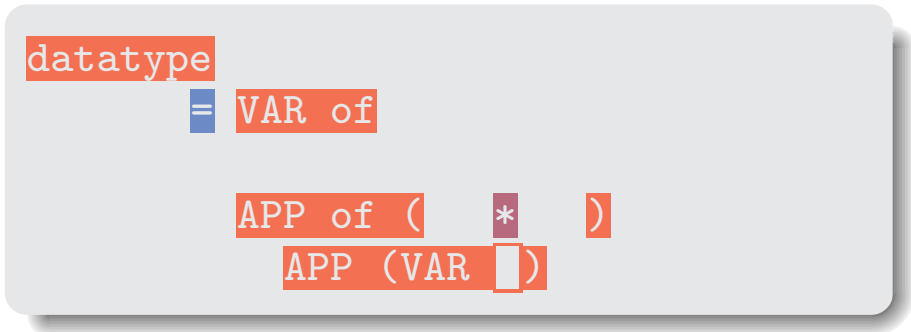
```
datatype Nat = Z | S of Nat
and LC = VAR of Nat
       | ABS of LC
       | APP of (LC * LC)
val term = APP (VAR Z) (VAR Z)
```

Two explanations:

The datatype constructor `APP` is applied to two arguments but takes only one.



The datatype constructor `VAR` is a `LC` constructor and not a tuple constructor.



Let us consider the following function:

```
fun f x =
  let val g = fn h => x h
  in (g 1, g true)
  end
```

Let us assume that the constraint generator generates:

- $\langle env_1, \alpha_1, \overline{c}_1 \rangle$ for `val g = fn h => x h`
- $\langle env \cup \{\mathtt{g} \overset{\{l\}}{=} \alpha, \mathtt{g} \overset{\{l'\}}{=} \alpha'\}, \alpha_0, \overline{c} \rangle$ for `(g 1, g true)`,

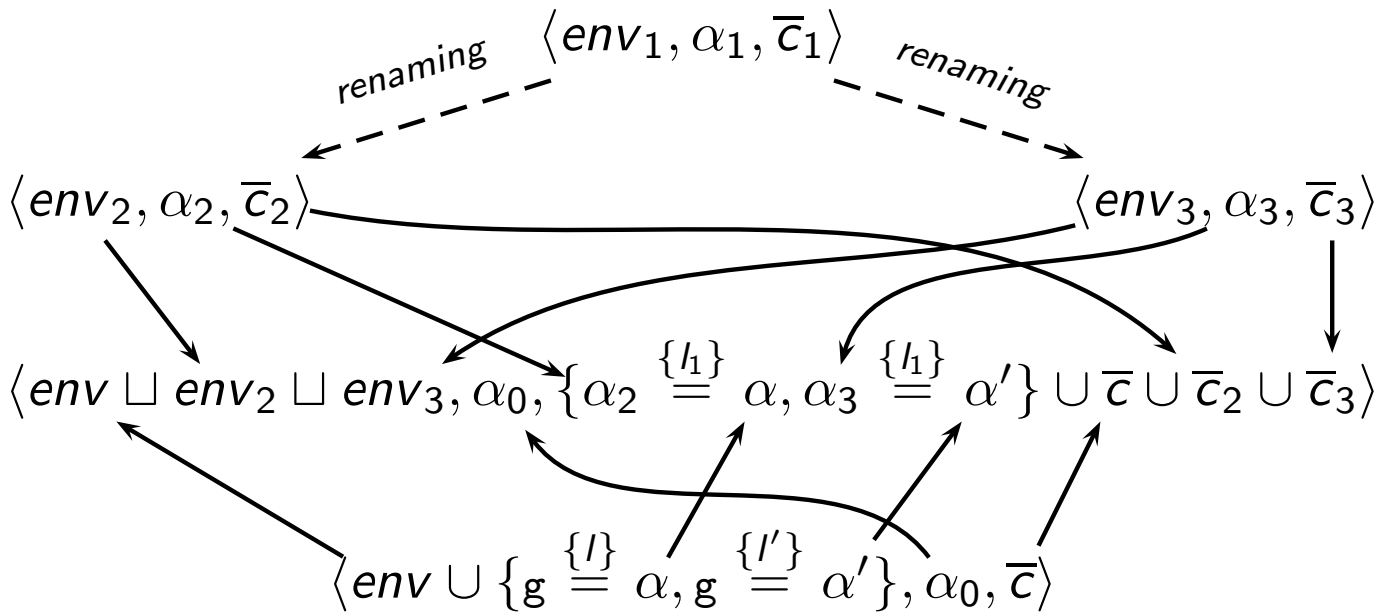
The environments are for free identifiers, e.g., env_1 contains a constraint of the form $\mathtt{x} \overset{\{l_2\}}{=} \alpha_4$.

Extension of Haack and Wells's type error slicer

Old constraint generation

```
fun f x =
  let val g = fn h => x h
  in (g 1, g true)
  end
```

The old approach to constraint generation would duplicate and rename the type variables of $\overline{c}_1$ twice for the two occurrences of g in g's scope.



Combinatorial explosion of the number of constraints.

Extension of Haack and Wells's type error slicer
New constraint generation

```
fun f x =
  let val g = fn h => x h
  in (g 1, g true)
  end
```

We want to do better.

We introduce binders (*bind*) of the form: $\langle \alpha_1, \alpha_2, l \rangle$ and new constraints: **bind** $\overline{bind}(\overline{c}_1)$ **in** $\overline{c}_2$ (where $\overline{bind}$ is a binding set).

For the piece of code above, we generate two binders: $\langle \alpha, \alpha_1, l_1 \rangle$ and $\langle \alpha', \alpha_1, l_1 \rangle$ because g's first occurrence binds its two other occurrences.

The constraint set generated for the let-expression would then be:

$$\{\textbf{bind } \{\langle \alpha, \alpha_1, l_1 \rangle, \langle \alpha', \alpha_1, l_1 \rangle\}(\overline{c}_1) \textbf{ in } \overline{c}\}$$

We do not duplicate constraints at constraint generation but we duplicate types when unifying the generated constraints. It is more cost-effective.

Extension of Haack and Wells's type error slicer
Any inconvenience?

```
fun f x =
  let val g = fn h => x h
  in (g 1, g true)
  end
```

Back to the old approach...

We keep track of the free identifiers' types in type environments.

- When generating the constraint set for the let-expression we have in our type environment before duplication: $\texttt{x} \overset{\{l_2\}}{=} \alpha_4$
- After duplication: $\texttt{x} \overset{\{l_2\}}{=} \alpha_5, \texttt{x} \overset{\{l_2\}}{=} \alpha_6$
- The type variables α_5 and α_6 are then constrained to be equal.

Old approach: ordinary constraints used to handle the monomorphism.
New approach: not enough anymore.

Extension of Haack and Wells's type error slicer

Inconvenience in our new approach during unification

```
fun f x =
  let val g = fn h => x h
  in (g 1, g true)
  end
```

In our new approach, when solving:

$$\textbf{bind } \{\langle \alpha, \alpha_1, l_1 \rangle, \langle \alpha', \alpha_1, l_1 \rangle\}(\overline{c}_1) \textbf{ in } \overline{c}$$

- We first solve $\overline{c}_1$.
- Using the generated unifier we build up a type equivalent to α_1.
  In our example it would be a type τ of the form: $\alpha_7 \rightarrow \alpha_8$.
  It also happens that `x`'s type is equivalent to $\alpha_7 \rightarrow \alpha_8$.
- We generalise `g`'s type τ twice, to get two types τ_1 and τ_2 and we generate two constraints from our two bindings: $\alpha \stackrel{\overline{l}}{=} \tau_1$ and $\alpha' \stackrel{\overline{l}}{=} \tau_2$ (where $\overline{l}$ is generated along with τ).
- And we continue the process by solving $\overline{c}$.

Extension of Haack and Wells's type error slicer

Inconvenience in our new approach during unification

```
fun f x =
  let val g = fn h => x h
  in (g 1, g true)
  end
```

- The problem is that when generalising τ we now have to take care not to generalise monomorphic type variables.
- In our example we have to mark `x`'s type as being not generalisable when unifying the bind-constraint, but we also have to mark α_7 and α_8 has being not generalisable because `x`'s type depends on both of them. `g`'s type is monomorphic because equivalent to `x`'s type
- We have to have some mechanism to keep track of the non generalisable type variables.

In our approach α_7 and α_8 are computed as non generalisable and marked as so before generalising τ and unmarked after generalisation. Didier Rémy [DR92] uses ranks to distinguish generalisable and non generalisable type variables. We might want to adopt this solution.

The old minimisation algorithm consists in building up a unifier along with a smaller error.

It takes as inputs:

- an error (label set) $\overline{l}$ and
- a constraint set $\overline{c}$.

Because $\overline{l}$ is an error, the constraint set from $\overline{c}$ labelled by $\overline{l}$ only is unsolvable.

We start with an empty unifier *uni* and an empty $\overline{l}_0$.

The old minimisation algorithm works as follows.

- Using *uni*, we run the unification algorithm on the constraint set labelled by $\bar{l} \setminus \bar{l}_0$ and we single out the label l of the last constraint unified before failure. (When generated constraints are labelled by a unique label.)
- This label has the particularity to be in all the errors smaller than the error returned by the unification.
- Using *uni*, if the unification of the constraints labelled by l only leads to a failure then we have our new error: $\{l\} \cup \bar{l}_0$. Otherwise, it is a success and the unification algorithm returns a unifier. We then start again from the first step with this new unifier. We also move l from $\bar{l}$ to $\bar{l}_0$.

Extension of Haack and Wells's type error slicer

Problem with old minimisation/new constraints?

Let us consider the following piece of code:

```
val x = true
val y = x + 1
```

The old minimisation algorithm would:

- single out the label associated to `x`'s second occurrence and update the unifier,
- single out the label associated to `+` and update the unifier,
- single out the label associated to `x`'s first occurrence and update the unifier.

At this point `x`'s type has been generalised and because it is not yet related to `true`, it is a new type variable which is not related to any type variable in the initial constraint set.

The algorithm would then fail because at the next step the constraint set would be typable given the current unifier.

Unsuitable minimisation algorithm.

We then designed a new minimisation algorithm.

Instead of building up a new error by adding labels to an empty label set as in the old algorithm, we minimise the current error by removing labels from it.

The algorithm works in two phases:

- The **unbind** phase which consists in removing labels at binding position.
- The **reduce** phase which consists in removing labels at any position.

Extension of Haack and Wells's type error slicer
New minimisation

Let us consider the following piece of code:

```
fun not true  = false
  | not false = true
fun toTrue _ = true
fun f bool g = if g bool
               then not bool
               else toTrue bool
val _ = f true (fn x => x + 1)
```

The enumeration algorithm enumerates:

```
⟨..fun ⟨..⟩ ⟨..⟨..toTrue ⟨..⟩ = true..⟩..⟩
 ..fun ⟨..⟩ ⟨..⟨..f bool g = if g bool
                         then ⟨..⟩
                         else toTrue ⟨..⟩..⟩..⟩
        ..f true (fn ⟨..x => ⟨..x + ⟨..⟩..⟩..⟩)..⟩
```

The enumeration algorithm enumerates:

```
⟨..fun ⟨..⟩ ⟨..⟨..toTrue ⟨..⟩ = true..⟩..⟩
 ..fun ⟨..⟩ ⟨..⟨..f bool g = if g bool
                         then ⟨..⟩
                         else toTrue ⟨..⟩..⟩..⟩
        ..f true (fn ⟨..x => ⟨..x + ⟨..⟩..⟩..⟩)..⟩
```

`toTrue`'s declaration is discarded at unbind phase:

```
⟨..fun ⟨..⟩ ⟨..⟨..f bool g = if g bool
                         then ⟨..⟩
                         else ⟨..⟩..⟩..⟩
        ..f true (fn ⟨..x => ⟨..x + ⟨..⟩..⟩..⟩)..⟩
```

Extension of Haack and Wells's type error slicer
New minimisation

`toTrue`'s declaration is discarded at unbind phase:

```
⟨..fun ⟨..⟩ ⟨..⟨..f bool g = if g bool
                         then ⟨..⟩
                         else ⟨..⟩..⟩..⟩
        ..f true (fn ⟨..x => ⟨..x + ⟨..⟩..⟩..⟩)..⟩
```

The conditional is discarded at reduce phase:

```
⟨..fun ⟨..⟩ ⟨..⟨..f bool g..= ⟨..g bool..⟩..⟩..⟩
        ..f true (fn ⟨..x => ⟨..x + ⟨..⟩..⟩..⟩)..⟩
```

Extension of Haack and Wells's type error slicer
New minimisation

Initial piece of code:

```
fun not true  = false
  | not false = true
fun toTrue _ = true
fun f bool g = if g bool
               then not bool
               else toTrue bool
val _ = f true (fn x => x + 1)
```

Final slice:

```
⟨..fun ⟨..⟩ ⟨..⟨..f bool g..= ⟨..g bool..⟩..⟩..⟩
         ..f true (fn ⟨..x => ⟨..x + ⟨..⟩..⟩..⟩)..⟩
```

Extension of Haack and Wells's type error slicer
Minimality

Aim: provide minimal slices for untypable pieces of code.

A slice is minimal if it is untypable and if every smaller slice is typable.

A slice is smaller than another slice if it contains less nodes and if the bindings are not changed. For example

```
fn x => (⟨..x..⟩, x)
```

is smaller than

```
fn x => (x, x)
```

but is not smaller than

```
fn x => (fn x => x, x)
```

Issues:

- We need constraint generation rules for dot terms (⟨..x..⟩) as well.
- We do not prove that our slices are actually minimal.
- We do not actually check this condition in our slicer.
- Minimality does not always seems to be the correct answer.

Extension of Haack and Wells's type error slicer

Minimality: merging, free identifiers

Merging of slices for field record clashes.

We want:

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

and not:

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

Free identifiers.

We prefer:

```
type t = int
val y :  t = 1
val x = z + y
```

instead of something like:

```
type t = int
val y : t = 1
val x = z + y
```

Conclusion

We saw:

- SML-TES's steps
- A challenge w.r.t. constraint generation.
- A challenge w.r.t. unification.
- A challenge w.r.t. minimisation.
- Challenges w.r.t. minimality.

What's next:

- Tests on real users.
- Implement support for important features of SML such as **open**.
- Develop a faster minimisation algorithm.

Didier Remy.

Extension of ML type system with a sorted equation theory on types.
Technical Report RR-1766, INRIA, 1992.
Projet FORMEL.

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 *ACM SIGPLAN 2007 Conference PLDI*. ACM, 2007.

Matthias Neubauer and Peter Thiemann.

Discriminative sum types locate the source of type errors.
In *8th ACM SIGPLAN Int'l Conference, ICFP 2003*, pages 15–26. ACM, 2003.

Franois Pottier and Didier Rémy.

The essence of ML type inference.
In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny.

Type processing by constraint reasoning.
In *4th Asian Symp., APLAS 2006*, volume 4279 of *LNCS*, pages 1–25. Springer, 2006.

Mitchell Wand.

Finding the source of type errors.
In *13th ACM SIGACT-SIGPLAN Symp., POPL'86*, pages 38–43, New York, NY, USA, 1986. ACM.

J. Yang, J. Wells, P. Trinder, and G. Michaelson.

Improved type error reporting.
In *12th Int'l Workshop, IFL 2000*, volume 2011 of *LNCS*, pages 71–86. Springer, 2001.