Challenges of a type error slicer for SML

Vincent Rahli and J. B. Wells and Fairouz Kamareddine

ULTRA group, Heriot-Watt University, http://www.macs.hw.ac.uk/ultra/

Abstract. Existing SML implementations have confusing type error messages. *Type error slicing* (TES) gives the programmer more help: (1) It identifies all program points that contribute to a type error rather than blaming just one point. (2) It exhibits a slice of the original program's syntax instead of showing an internal representation of some program subtree which may have been substantially transformed. (3) It avoids showing internal details of inferred types which were not written by the programmer. Unfortunately, TES was initially done for a very tiny toy language. Extending TES to a full programming language is extremely challenging, and for SML we needed a number of innovations. Some issues would be faced for any language, and some are SML-specific but representative of the complexity of language-specific issues likely to be faced for other languages.

1 Introduction

Type inference for SML. SML is a higher-order function-oriented imperative programming language. SML (and similar languages like OCaml, Haskell, etc.) has polymorphic types allowing considerable flexibility, and almost fully automatic type inference, which frees the programmer from writing explicit types. We say "almost fully" because some explicit types are required in SML, e.g., as part of datatype definitions, module types, and type annotations sometimes needed in special circumstances. Milner's W algorithm [4] is the original type-checking algorithm of the functional core of ML (variables, abstractions, applications and polymorphic let-expressions). W implementations generally give error messages relative to the syntax tree node the algorithm was visiting when unification failed, and this is often unsatisfactory.

Moving the error spot. Following W, other algorithms try to get better locations by arranging that untypability will be discovered when visiting a different syntax tree node. For example, Lee and Yi proved that the folklore algorithm M [12] finds errors "earlier" than W and claimed that their combination "can generate strictly more informative type-error messages than either of the two algorithms alone can". Similar claims are made for W$'$ [14] and UAE [25]. McAdam observes that W suffers a left-to-right bias and tries to eliminate it using "unification of substitutions". Yang claims that UAE's primary advantage is that it also eliminates this left-to-right bias. However, all the algorithms mentioned above retain a left-to-right bias in handling of let-bindings and they all blame only one syntax tree node for the failure of type inference when a node set is at fault.

Choosing only one node to report as the error site often identifies error locations far away from the actual programming error locations. The situation is made worse because the node targeted for blame depends on internal implementation details, i.e., the order in which tree nodes are visited and which constraints are accumulated and solved at different points of the traversal. The confusion is further worsened because these algorithms generally exhibit in error messages (1) an internal representation of the program subtree at the identified error location which often has been transformed substantially from what the programmer wrote, and (2) details of inferred types which were not written by the programmer and which are anyway erroneous and therefore confusing.

Other improved error reporting systems. Attempting to solve the problem, constraint-based type inference algorithms [17–19] separate the two following processes: the generation of type constraints for a given term and the unification of the generated constraints. Many works are based on this idea to improve error reporting (a probably incomplete list includes [10, 6, 7, 5, 20–22]). Independently from this separation, there exist many different approaches toward improving error reporting [27]: error explanation systems [2, 26] and error reporting systems [23]. Another approach to type error reporting is the one of Lerner et al. [13] or Hage and Heeren [8] suggesting changes to perform in the untypable code to solve type errors.

Type error slicing. Haack and Wells [7] noted that "*Identifying only one node or subtree of the program as the error location makes it difficult for programmers to understand type errors. To choose the correct place to fix a type error, the programmer must find all of the other program points that participate in the error.*" They proposed locating type errors at *program slices* which include all parts of an untypable piece of code where changes can be made to fix the type error and exclude the parts where changes cannot fix the error.

Haack and Wells gave their method of *type error slicing* (TES) for a tiny subset of SML barely larger than the λ-calculus. The method was given in three main steps: assigning constraints to program points, finding minimal unsolvable subsets in the set of all constraints, and computing type error slices. This method first generates a type constraint set before using a unification algorithm on the set. This method meets all the criteria listed in [27] for error reports to be "good": it reports only errors for ill-typed pieces of code (*correct*), it reports no more than the conflicting portions of code (*precise*), it reports short messages (*succinct*), it does not report internal information such as internal types generated during type inference (*a-mechanical*), it reports only code written by the programmer which has not been transformed as happens with existing SML implementations (*source-based*), it does not privilege any location over the others (*unbiased*), and it reports all the conflicting portions of code (*comprehensive*).

Their TES enumerates minimal unsatisfiable subsets of the constraint set generated for a given piece of code. Each type constraint is labelled by the location responsible for its generation. These labels are used to identify minimal sets of locations corresponding to type errors. Computing slices is based on these label sets. Slices are pieces of code where all sub-terms not involved in an error

are replaced by some visually convenient symbols such as dots. These slices are intended to contain all and only what the programmer needs to solve the type errors. Slices can also be shown by highlighting in the original code.

Slicing for a full language. Our goal is a TES that (1) covers the full SML language and (2) is practical on real programs. As would happen for any programming language, while doing this we encountered challenges.

A large challenge has been giving reasonable slices for each different untypable SML syntactic form. Unexpectedly, we faced many special cases which needed unique care. Some of these cases are specific to SML such as: functions defined with the `fun` feature, value polymorphism, binding of explicit type variables, etc. Some of them are generic such as: representing if it is important for a piece of code to be a direct subtree of its context (which we do when highlighting by careful handling of white space), indicating when a syntax tree node which contributes no symbols is part of an error (e.g., for function application we highlight the space and put a box around the argument), etc. For each of these cases we had to modify or introduce new ground rules for our TES to generate useful slices and at the same time stay as general as possible for later reuse for other languages. Some of these cases are illustrated in the next section.

Another challenge was to avoid a combinatorial explosion of the number of constraints during constraint generation. A naive approach to constraint generation might involve duplicating the environment of a polymorphic binding such as in a let-expression. Our solution is inspired by a constraint system by Pottier and Rémy [19, 18]. The most interesting constraints in their constraint system are "let-constraints" (generated for let-bindings). They are to some extent, inspired by constraint-based type systems such as the one by Odersky, Sulzmann and Wehr [17] (and mainly by the type schemes used in this system). As explained by Pottier such constraints "allow building a constraint of linear size" [18]. We have generalised the structure of these constraints to deal with diverse identifiers (polymorphic as well as monomorphic bindings): values as well as type/structure/signature names. We faced significant challenges in making a fast and compatible minimisation algorithm for let-constraints.

Another challenge is SML's value identifier statuses. In SML, a value identifier can be a value variable (the only status considered by Haack and Wells), a datatype constructor, or an exception constructor (due to space limit, we omit the treatment of exceptions). For example, assuming that unknown free identifiers have value variable status, `fn c => (c 1, c ())` has a unique minimal error which is that `c` has a monomorphic type but it is applied to two expressions with different types: `int` and `unit`. This minimal error would not exist if the code was preceded, for example, by `datatype t = c` (the `fn`-binding would not constrain `c`'s type to be monomorphic) but then another minimal error would be that `c` is declared as a datatype constructor without argument and it is applied to an argument (`c 1` for example). To compute correct minimal error slices, we annotate constraints by context dependencies. For the `fn`-binding presented above we generate constraints relating the occurrences of `c` annotated by the dependency that `c` is a value variable and not a datatype constructor. These constraints are

discarded if a context confirms that `c` must be a datatype constructor. The dependency is discarded and the constraints are kept if a context confirms that `c` cannot be a datatype constructor. Considering that most identifiers bound by a function are value variables, for incomplete programs we report conditional errors that assume the dependencies are true. Doing otherwise would cause a great increase in unhelpful reported slices.

In addition to overcoming challenges like those above, we also use slicing for all context-sensitive syntax errors. This comes naturally from handling identifier statuses and doing context-independent type checking. For example, in `fn (x, x) => x`, the fact that `x` occurs twice in the pattern is an error only if `x` is a value variable but not if `x` is a datatype constructor.

2 Type error slices for SML

Some of the examples here are handled only by our implementation but not by the simplified formalism in Sec. 4.

Due to space limit, we usually present here partial reports from our TES. A complete report contains: an highlighting of the reported type error in the original code, a slice, a list of context dependencies, a verbal explanation of the kind of the error and a legend of the colours used by highlightings for the new users not yet familiar with our colour convention described below. This section compares some reports to some SML compilers' reports. These compilers are SML/NJ (v110.52), HaMLet (v1.3.0) and Moscow ML (v2.01).

The colour convention we use is that a piece of code is highlighted in red (or very light grey) if it is involved in the reported error; a piece of code is highlighted in purple or blue (or dark grey) if the error reported is a clash (between two type constructors for example) and the code is an end point in the error (one of the clashing type constructors for example: `1` and `true` in `fn x => (x 1, x true)`); finally a piece of code is highlighted in green (or light grey) if it is a record field that occurs in more than one minimal slice of a same record clash.

We define "programming error" to mean a location where the programmer will eventually make some changes to solve a type error.

2.1 Output formats. In this section we briefly present our syntax for slices and we introduce the use of boxes in syntax like function application. We start with two examples involving datatype definitions leading to type constructor clashes (the error locations reported by SML/NJ are in ovals):

```
fun ex1 z = let datatype X = C1 of int in (C1 true) end
fun ex2 z = let datatype Y = C2 | C3 of int in (C2 z) end
```

For both these untypable functions SML/NJ reports only one location as the error location (the circled code). In `ex1`, the datatype constructor `C1` is applied to a `bool` but is defined as containing an `int`. We obtain a clash between the `int` and `bool` types. Note that one can solve the type error replacing `int` by `bool` but this location is not reported. In `ex2`, the datatype constructor `C2` is applied to an argument but is not defined as taking an argument. Again, SML/NJ, HaMLet

and Moscow ML only report `C2` (or `C2 z`) when `C2`'s definition might be the programming error location. Our TES exhibits the two following highlightings:

```
fun ex1 z = let datatype X = C1 of int in C1 true end
fun ex2 z = let datatype Y = C2 | C3 of int in C2 z end
```

We print a box around `z`'s last occurrence because highlighting the space between `C2` and `z` makes sense only because `z` is the argument of `C2`. This is one of our uses of boxes: around arguments whose content does not matter but whose presence matters. Note that the highlighting of the white spaces between a function and its argument is important because they are one of the end points of our second highlighting. This application constrains `C2` to have an arrow type.

As explained above, our reports also contain slices intended to contain all and only the parts involved in the reported errors. Dots are used to show that some irrelevant information for an error to occur has been removed. Given a term and a type error in this term, the irrelevant information for this error is precisely the set of nodes and edges in the abstract syntax tree of the term not involved in the error. The angle brackets with more than .. inside are used to indicate nesting at unfixed depth. The two generated slices are as follows:

```
⟨..datatype ⟨..⟨..⟩ = ⟨..C1 of ⟨..⟩ int..⟩..⟩..C1 true..⟩
⟨..datatype ⟨..⟨..⟩ = ⟨..C2..⟩..⟩..C2 ⟨..⟩..⟩
```

Many parts of the original terms are sliced out, e.g., as the datatype names which are not involved in the errors. Our slices aim to contain all and only the information needed to solve type errors. For example in the first slice, ⟨..`C1 of` ⟨..⟩ `int`..⟩ stresses that the number of datatype constructors defined along `C1` and the arity of `int` are irrelevant. It is not important that `C1` is the only type constructor in a datatype definition and that the arity of `int` is 0.

2.2 Multiple slices. The next examples shows some of the details stressed by our slices for recursive functions defined using the `fun` feature. We present here a small piece of code, but the reader can easily imagine the functions `g` and `f` being far apart in a larger program. Let us consider the two following error highlightings for the same piece of code:

```
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)
```

This code becomes well-typed if the last occurrence of `y` in the third line (the programming error) is replaced by a `bool`. The programming error is in both slices. Both errors are due to type constructor clashes. The first slice is due to the fact that `f` returns a `bool` (it returns its second argument and it calls `g` where its second argument is used as a `bool`) and also returns an `int` (it returns a call to `g` which returns an `int`). The end points of the second slice are the same as for the first slice, namely: `if then else` and `+`. However, in the second slice we do not need to know, e.g., that `g` returns an `int`: there is no highlighting of the white spaces between `=` and `if`, between `then` and `x` and between `y` and `else`. What we need to know however is that there is a call to `g` where the types of its second and third arguments are constrained to be equal (being a parameter of a function,

`y` has a monomorphic type) and that in `g`'s definition, its second argument has to be an `int` (because `+` has the type it is given by the standard basis) while its third argument has to be a `bool` (because of the conditional). We can observe that from a unique problem, different slices can be derived.

The slices also contain the same information. The part of the first slice corresponding to `g`'s body is `if z then` ⟨..⟩ `+` ⟨..⟩ `else` ⟨..⟩ while the same part for the second slice is ⟨..`if z then` ⟨..⟨..⟩ `+ y`..⟩ `else` ⟨..⟩..⟩.

The dots in slices do not necessarily replace explicit syntax tokens. Some of them replace only syntax tree edges. In the last partial slice above, we removed the edge between the conditional and its `then` branch as explained above. Another interesting example is that the part of the second slice corresponding to the third code line is ⟨..⟩..⟨..`y`..⟩.. `=` ⟨..`g` ⟨..⟩ `y y`..⟩, because the function name is unimportant and it only matters that there is a parameter in which `y` occurs. The circled dots left of the equal symbol mean it is irrelevant if this function has more arguments. These dots do not match anything because this function has in fact no more arguments after `y`.

2.3 Context dependencies. The function `fun f x y = if y then y else 0.` is untypable because `y` is constrained to be an `int` and a `bool`. We obtain the highlighting `fun f x y = if y then y else 0`. Our programming error is that we wrote `if y` instead of `if x`. This error is obtained under the context dependency that `y` is a value variable. However, if `datatype t = y` is before this function, instead of this highlighting we would obtain, e.g., a clash between the types of the branches of the conditional (one returns a `t`, the other returns an `int`): ⟨..`datatype` ⟨..⟨..⟩ `t = y`..⟩..⟩..`if` ⟨..⟩ `then y else 0`..⟩. Now, if `fun y () = ()`, which forces `y` to be a value variable, was preceding the function then `y` would definitely be a value variable, confirming the context dependence. The error would be context independent and augmented with `y`'s declaration.

Context dependencies can also affect the polymorphism of an identifier's type. For example `val (u, v) = (g (), fn x => x); v 1; v true;` is untypable if `g` is a value variable but may be typable if `g` is a value constructor. This is due to SML's value polymorphism restriction which forces `v`'s type to be monomorphic if `g` is a value variable. However, `v`'s type is polymorphic if `g` is a constructor. We obtain the highlighting `val (u, v) = (g (), fn x => x); v 1; v true;` with the context dependency that `g` is a value variable. Preceded by `val g () = ()` our code is untypable but preceded by `datatype 'a t = g of 'a` it is typable.

2.4 Context-sensitive syntax errors. We do slicing for all context-sensitive syntax errors, because it is cleaner and also this needs the analysis of identifiers statuses. For example, see these two syntactic errors in the same code: `fn (f, f y) => y + 1` and `fn (f, f y) => y + 1`.

The first (context-dependent) highlighting reports that if `f` is a value variable then it should not occur twice in a pattern. The context sensitivity would go away if the context had confirmed `f` is not a constructor. For example if `fun f _ = ()` preceded the code, this declaration of `f` would be in the error because it constrains `f` to be a value variable.

The second highlighting reports that two pattern occurrences of `f` are constrained to have a functional (applied) and non-functional (not applied) type. This error would be the same if a declaration such as `fun f _ = ()` or `datatype t = f` was preceding the code because if `f` was a value variable then the second highlighted occurrence of `f` would constrain it to be a datatype constructor. If `f` was a datatype constructor, as explained above it would be used both at locations constraining its type to be functional and non-functional.

2.5 Where our type error slicer becomes vital. Consider a more complicated example (displayed on the right) which shows how TES is important for more complicated errors. The code declares a datatype and a function to deal with colours. This function is then used on an instance of a colour (the first element in the pair `x`). Assume that our programming error is that we wrote `'b` instead of `'c` in `Green`'s definition (at location ①).

```
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  ①
                        | 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)  ②
  | trans (Green  (x, y, z)) = Yellow (y, x, z)  ③
  | 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)  ⑤
val y : (int, bool) u = (trans (#1 x), #2 x)
                          ④
```

SML/NJ reports a type constructor mismatch error at location ④ (displayed on the right). The reported code does not look like our code and is far away from the programming error location. It is then hard to find the error site, e.g., we would see the same error message if instead of the error described above, we had written `x` instead of `z` in the right-hand-side of any of `trans`'s branch. With SML/NJ's report we need to check the entire piece of code.

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

Moscow ML reports also a type constructor clash (displayed on the right) and blames the same portions as SML (and HaMLet).

```
File "test-prog.sml", line 15, characters 25-37:
! val y : (int, bool) u = (trans (#1 x), #2 x)
!                         ^^^^^^^^^^^^
! Type clash: expression of type
!   (int, int, int) t
! cannot have type
!   (int, int, bool) t
```

This error is context-dependent: it assumes that `y` and `z` are value variables. The programming error location is in the slice. We can search for our error looking only at the highlighted portions of code starting with the end points. Note that the type `(int, bool) u` constrains the type of the application of `trans` to an argument and that the highlighted portion of `trans` is when applied to `Green`. At ①, `Green`'s second argument is constrained to be of the same type as its third argument. At ②, `y` is incidentally constrained to be of the same type as `z`. At ③, because `y` and `z` are respectively the first and third arguments of `Yellow` and using the definition of `Yellow`, we can deduce that the type of the application of `Yellow` to its three arguments and so the result type of `trans` is `t` where its first (`'a`) and third (`'c`) parameters have to be equal. Using the code at ④ and ⑤ we can see that the result type of `trans` is constrained to be of type `t` where its first (`int`) and third (`bool`) parameters are different.

We can see that without the highlighting it would be even harder to guess why we obtain a conflict between `(int,int,int) t` and `(int,int,bool) t`.

2.6 Merging of minimal slices. We have found cases needing the display of many minimal errors at once. One important case is in record field name clashes where, e.g., the highlighting `val {foo,bar} = {fool=0,bar=1}` is obtained from two minimal errors reporting that `fool` is not in $\{\texttt{foo}, \texttt{bar}\}$ and `foo` is not in $\{\texttt{fool}, \texttt{bar}\}$. This merging must be done due to the explosion in the number of minimal slices. Light grey is used to highlight the fields that are common to different minimal slices. For merged slices minimality is understood as follows: retain a single dark grey label in one of the two clashing records and all labels in the other.

Here SML/NJ reports a type constructor mismatch (displayed on the right). This report contains type variables that have not been written by the programmer and the reported code is confusing because it differs from what the programmer wrote. HaMLet reports only a "type mismatch between pattern and expression" and Moscow ML reports that `foo` is missing in the type of the expression `{fool=0,bar=1}`.

```
pattern: {bar:'Z, foo:'Y}
expression: {bar:int, fool:int}
in declaration:
  {bar=bar,foo=foo} =
    (case {fool=0,bar=1}
      of {bar=bar,foo=foo} => (bar,foo))
```

3 Mathematical definitions and notations

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers and let i, j, k, n, m, p, q be metavariables ranging over $\mathbb{N}$. If a metavariable v ranges over a class C, then the metavariables v_x (where x can be anything) and the metavariables v', v'', etc., also range over C. Let s range over sets. The power set of a set is defined as usual: $\mathbb{P}(s) = \{s' \mid s' \subseteq s\}$. If v ranges over s, $\overline{v}$ is defined to range over $\mathbb{P}(s)$.

A binary relation is a set of pairs. Let R range over binary relations. Let $\mathrm{dom}(R) = \{x \mid (\!|x, y|\!) \in R\}$ and $\mathrm{ran}(R) = \{y \mid (\!|x, y|\!) \in R\}$ (note that the pair $(\!|x, y|\!)$ differs from the tuple (defined below) of length 2, $\langle x, y\rangle$). A function is a binary relation f such that if $\{(\!|x, y|\!), (\!|x, z|\!)\} \subseteq f$ then $y = z$. Let f range over functions. Let $s \to s' = \{f \mid \mathrm{dom}(f) \subseteq s \wedge \mathrm{ran}(f) \subseteq s'\}$. Let $x \mapsto y$ be an alternative notation for $(\!|x, y|\!)$ used when writing functions.

A tuple t is a function such that $\mathrm{dom}(t) \subset \mathbb{N}$ and if $1 \leq k \in \mathrm{dom}(t)$ then $k-1 \in \mathrm{dom}(t)$. Let t range over tuples. We write the tuple $\{0 \mapsto x_1, \ldots, n-1 \mapsto x_n\}$ as $\langle x_1, \ldots, x_n\rangle$. We say that $\langle x_1, \ldots, x_n\rangle$ is a tuple of length n. We define the appending $\langle x_1, \ldots, x_n\rangle @ \langle y_1, \ldots, y_m\rangle$, of two tuples $\langle x_1, \ldots, x_n\rangle$ and $\langle y_1, \ldots, y_m\rangle$ as the tuple $\langle x, \ldots, x_n, y_1, \ldots, y_m\rangle$. Given a set s, let $\mathrm{tuple}(s) = \{t \mid \mathrm{ran}(t) \subseteq s\}$. If v ranges over s, $\overrightarrow{v}$ is defined to range over $\mathrm{tuple}(s)$.

4 Technicalities

This section presents a restricted version of our TES implementation. Our method consists of three main steps: constraint generation, enumeration and minimisation of label sets representing errors, and slicing. The enumeration and minimisation algorithms both make an extensive use of a unification algorithm.

Fig. 1. Restricted version of the labelled syntax of our implementation

Let lx be a term of the form x^l for any x. In this table let $n \geq 1$ and $m \geq 2$. In the case of *match* and *conbindseq* only, the vertical bars are terminals of the grammar and do not perform their usual function of separating grammar alternatives.

$$
\begin{array}{rlrl}
\iota \in \mathsf{Int} & \text{(the set of integers)} & \alpha, tv \in \mathsf{TyVar} & \text{(set of type variables)} \\
l \in \mathsf{Label} & \text{(set of labels)} & tc \in \mathsf{TyCon} & \text{(set of type constructors)} \\
vid \in \mathsf{VId} & \text{(set of identifiers)} & id \in \mathsf{Id} & = \mathsf{VId} \cup \mathsf{TyCon}
\end{array}
$$

$$
\begin{array}{rl}
ty ::= & tv^l \mid \mathrm{int}^l \mid \mathrm{unit}^l \mid (lty)^l \mid lty_1 \xrightarrow{l} lty_2 \mid \lceil lty_1 \times \cdots \times lty_m \rceil^l \mid \lceil tyseq\ ltc \rceil^l \\
tvseq ::= & ltv^l \mid \epsilon^l \mid (ltv_1{}^{l_1}, \ldots, ltv_n{}^{l_n})^l \qquad tyseq ::= ty^l \mid \epsilon^l \mid (lty_1, \ldots, lty_n)^l \\
conbind ::= & vid^l \mid lvid^{l_1}\ \mathrm{of}^{l_2}\ lty \qquad conbindseq ::= conbind_1 \mid \cdots \mid conbind_n \\
datbind ::= & tvseq\ ltc \stackrel{l}{=} conbindseq \\
dec ::= & \mathrm{val\ rec}\ lpat \stackrel{l}{=} lexp \mid \mathrm{datatype}\ datbind_1\ \mathrm{and}\ \cdots\ \mathrm{and}\ datbind_n \\
atexp ::= & vid^l \mid \iota^l \mid ()^l \mid (lexp)^l \mid (lexp_1, \ldots, lexp_m)^l \mid \mathrm{let}^l\, dec\ \mathrm{in}\ lexp\ \mathrm{end} \\
atpat ::= & \_ \mid vid^l \mid \iota^l \mid ()^l \mid (lpat)^l \mid (lpat_1, \ldots, lpat_m)^l \\
exp ::= & atexp \mid \mathrm{fn}^l\ match \mid \lceil exp\ atexp \rceil^l \qquad pat ::= atpat \mid \lceil lvid\ atpat \rceil^l \\
match ::= & mrule_1 \mid \cdots \mid mrule_n \qquad mrule ::= lpat \stackrel{l}{\Rightarrow} lexp
\end{array}
$$

Fig. 2. Environments and constraints

$$
\begin{array}{rlrl}
\upsilon \in \mathsf{CsId} & ::= \langle \alpha_1, \alpha_2, l \rangle & tenv \in \mathsf{TyConEnv} & = \mathsf{TyCon} \to \mathbb{P}(\mathsf{CsId}) \\
fenv \in \mathsf{VIdEnv} & = \mathsf{VId} \to \mathbb{P}(\mathsf{CsId}) & env \in \mathsf{Env} & ::= \langle tenv, fenv_1, fenv_2, fenv_3 \rangle
\end{array}
$$

$$
\begin{array}{rlrl}
\gamma \in \mathsf{ITyName} & \text{(type names)} & \omega \in \mathsf{LabSeq} & ::= \beta \mid \langle \overline{\alpha}, l \rangle \\
\delta \in \mathsf{ITyNameVar} & \text{(type name variables)} & \mu \in \mathsf{LabName} & ::= \langle itcv, l \rangle \\
\beta \in \mathsf{ISeqVar} & \text{(sequence variables)} & \tau \in \mathsf{LabTy} & ::= \alpha \mid \langle \mu, \omega \rangle \\
itcb \in \mathsf{ITyConBase} & = \{\to, \times\} & poly \in \{\mathrm{poly}, \mathrm{mono}\} & \\
itcv \in \mathsf{ITyConVar} & ::= \delta \mid \gamma \mid itcb & csb \in \mathsf{CsBind} & ::= \langle id, \alpha_1, \alpha_2, poly, l \rangle
\end{array}
$$

$$
\begin{array}{rl}
semc \in \mathsf{CsSem} & ::= \tau_1 \overset{\overline{l},\overline{vid}}{=\!=} \tau_2 \mid \mu_1 \overset{\overline{l},\overline{vid}}{=\!=} \mu_2 \mid \omega_1 \overset{\overline{l},\overline{vid}}{=\!=} \omega_2 \mid \mathbf{let}\ \overline{csb}_1, \overline{csb}_1(\overline{c}_1)\ \mathbf{in}\ \overline{c}_2 \\
sync \in \mathsf{CsSyn} & ::= \mathrm{mul}(\overline{l}, \overline{vid}) \mid \mathrm{con}(\overline{l}, \overline{vid}) \mid \mathrm{inc}(\overline{l}) \mid \mathrm{app}(\overline{l}) \\
c \in \mathsf{Constraint} & ::= semc \mid sync
\end{array}
$$

4.1 Syntax. Fig. 1 presents a portion of our syntax. To provide a visually convenient place for labels some terms are surrounded by $\lceil$ and $\rceil$. We use ϵ for an empty sequence. These symbols are not part of the syntax seen by programmers but are part of an internal representation. We use $\lceil\ \rceil$ to avoid confusion with ( ) as part of SML syntax.

All the labels such as l_1 and l_2 in $\lceil ty_1{}^{l_1} \times ty_2{}^{l_2} \rceil^l$ are associated to context information. For instance, the label l_1 is associated to the context of the type ty_1, meaning the space between ty_1 and $\times$. A singleton type variable sequence is ltv^l (i.e., $tv^{l'\,l}$) because the inner label (l') is associated to the type variable occurring in ltv itself and l is associated to the singleton type variable sequence.

The portion of SML we implement contains additional features such as SML's `fun` feature, exceptions, records, and many others. As explained before, the context-sensitive syntactic restrictions [15] on the language (e.g., no pattern may contain the same variable twice) are dealt with using slices (reporting syntactic errors using slices comes almost for free in our TES).

4.2 Constraints.

Syntactic forms used in our constraints. Fig. 2 presents the syntax of type environments used by our constraint generator (top of Fig. 2) and constraints (bottom of Fig. 2, where the prefixing I in set names stands for Internal). An important feature of our syntax is the type sequences (ω), corresponding to, e.g., `(int,bool)` from the type `(int,bool)t`. These types help catch arity clashes. The ω and μ that are not variables contain labels used to distinguish end points in clashes (for more details see the unification section in our technical report (TR)).

Another important syntactic form is the constraints on identifiers (υ) used in type environments. A υ contains two type variables in order to record extra information on some identifiers. The type of an identifier in a pattern might differ depending on whether it is a value variable or a datatype constructor. The first type variable constrains the type of the identifier in both cases and the second one is complementary information in case the identifier is a datatype constructor. For instance, in `fn c => c`, if `c` is a datatype constructor then `c`'s type cannot be a basic type constructor ($\rightarrow$ or $\times$) and as a result `c` cannot be defined as taking an argument. The second type variable holds this information. For example if $\langle \alpha_1, \alpha_2, l \rangle$ constrains `c` then α_2 is constrained to be equal to $\langle \langle \delta, l \rangle, \beta \rangle$ where δ is always considered to be different from any basic type constructor such as $\rightarrow$. If it turned out that `c` is defined as a datatype constructor with an argument (as in `datatype t = c of int`) then we would obtain a conflict between δ and $\rightarrow$. This second variable is also used to constrain the arity of a type constructor. For example in `('a, 'b) t`, if $\langle \alpha_1, \alpha_2, l \rangle$ constrains `t` then α_2 is constrained to be equal to $\langle \langle \delta, l \rangle, \langle \alpha, \alpha' \rangle \rangle$ where $\langle \alpha, \alpha' \rangle$ constrains the arity of `t` to be two.

Type environments. An environment (*env*) for our constraint generator is composed of four environments to which we associate four different meanings. The first environment is used for type names. The second one for confirmed datatype constructors (applied identifiers in patterns). The third one for the bound identifiers that are considered as value variables but might as well turn out to be datatype constructors. The fourth one for the free identifiers. In any of these environments, each υ in $\mathit{fenv}(\mathit{vid})$ (resp. $\mathit{tenv}(\mathit{tc})$) is associated to a distinct occurrence of *vid* (resp. *tc*) in the code.

Constraints. We formally present in this paper some of the type errors handled by our implementation (the ones relevant to our simplified syntax). We present four syntactic errors, each of them is described below and corresponds to one kind of syntactic constraint (CsSyn for syntactic constraints). We present four semantic constraints (CsSem for semantic constraints) used to record the bindings in the studied piece of code (let-constraints) and to record the constraints between internal types (LabTy), type sequences (LabSeq) and type names (LabName).

Semantic equality constraints. Our three kinds of equality constraint are annotated by labels corresponding to the program locations responsible for their generation. When initially generated by the constraint generator, such a constraint is annotated by a unique label (the unique location responsible for its generation) and no context dependency. Equality constraints generated during unification can then be annotated by a label set and a context dependency set.

For example, $\alpha_1 \overset{\{l_1\},\varnothing}{=\!=\!=} \alpha_2$ and $\alpha_2 \overset{\{l_2\},\varnothing}{=\!=\!=} \alpha_3$ can generate $\alpha_1 \overset{\{l_1,l_2\},\varnothing}{=\!=\!=\!=} \alpha_3$ during unification. Context dependencies can only be generated during unification, while dealing with let-constraints.

Semantic let-constraints. In a let-constraint **let** $\overline{csb}_1, \overline{csb}_2(\overline{c}_1)$ **in** $\overline{c}_2$, the bindings $\overline{csb}_1$ and $\overline{csb}_2$ are generated from the bindings in the studied piece of code. A binding can either be polymorphic (poly) or monomorphic (mono). The first binding set $\overline{csb}_1$ is used for the identifiers with unknown status, while $\overline{csb}_2$ is used for identifiers with known status (which might be variable or datatype constructor). For the simple language presented here (but not in our implementation), $\overline{csb}_1$ only contains monomorphic bindings. This is because this paper does not present non-recursive value declarations (while our implementation does).

Both the polymorphic and monomorphic cases are illustrated with the code `let val rec f = (fn x => x) in f 1 end`. The first occurrence of `f` binds the second one. If $\overline{c}_1$ is the constraint set generated for `f`'s declaration and $\overline{c}_2$ is the constraint set generated for `f 1` then **let** $\varnothing, \{\langle \texttt{f}, \alpha_2, \alpha_1, \text{poly}, l\rangle\}(\overline{c}_1)$ **in** $\overline{c}_2$ is generated where α_1 and α_2 are the type variables associated to the first and second occurrences of `f` respectively, poly marks this binding as polymorphic and l is the label of `f`'s first occurrence. The first binding set in the let-constraint generated for `fn x => x` is $\{\langle \texttt{x}, \alpha_4, \alpha_3, \text{mono}, l'\rangle\}$, where α_3 and α_4 are the type variables associated to the first and second occurrences of `x` respectively, mono marks this binding as monomorphic and l' is the label of `x`'s first occurrence. This binding is discarded if it turns out that `x` is a datatype constructor. This binding is moved to the second binding set if it turns out that `x` is a value variable.

Syntactic constraints. We introduce four kinds of syntactic constraints. A $\text{mul}(\overline{l}, \overline{vid})$ is generated when it is syntactically invalid for an identifier to occur more than once. A $\text{con}(\overline{l}, \overline{vid})$ is generated when a value variable occurs at a function position in a pattern and thus must be a datatype constructor. An $\text{inc}(\overline{l})$ is generated when a free explicit type variable occurs in a datatype declaration. An $\text{app}(\overline{l})$ is generated when an identifier is constrained to be both a value variable and a datatype constructor in patterns. Consider the first kind of constraints. For example, for `datatype t = c | c` we generate $\text{mul}(\{l_1, l_2\}, \varnothing)$ where l_1 and l_2 are the labels of the two occurrences of `c`. The set $\overline{vid}$ is empty because `c`'s status is known: datatype constructor. Here is an example with context dependencies. For `val (c, c) = (1, 1)` we generate $\text{mul}(\{l_1, l_2\}, \{\texttt{c}\})$ where l_1 and l_2 are as before and `c` is a context dependency because without any context, `c`'s status cannot be inferred. The constraint means that there is a multi-occurrence error only if `c` is a value variable. If we had a declaration such as `fun c () = ()` (which fixes `c`'s status) in the context we would discard the context dependency and add the label of the new occurrence of `c` in the label set $\overline{l}$. If we had a declaration such as `datatype t = c` we would discard the constraint.

4.3 Constraint generator. Most of the judgements in our constraint generator are similar to the ones for expressions: $exp \Downarrow \langle env, \alpha, \overline{c}\rangle$. It expresses that the triple $\langle env, \alpha, \overline{c}\rangle$ is associated to exp using the relation $\Downarrow$. Meaning, in the environment env constrained by the constraint set $\overline{c}$, the type α also constrained

Fig. 3. Error results returned by enumeration algorithm

$$
\begin{array}{rl}
err \in \mathsf{Error} & ::= \langle \overline{l}, \overline{vid}, ek \rangle \\
ek \in \mathsf{ErrorKind} & ::= \mathrm{tyConsClash}(\langle l_1, itcv_1 \rangle, \langle l_2, itcv_2 \rangle) \mid \mathrm{arityClash}(\langle l_1, n_1 \rangle, \langle l_2, n_2 \rangle) \\
& \quad \mid \mathrm{circularity} \mid \mathrm{multiOcc} \mid \mathrm{nonConsApp} \mid \mathrm{inclusion} \mid \mathrm{appNotApp}
\end{array}
$$

by $\overline{c}$ is associated to *exp*. For example, the rule for match rules (e.g., the match rule is "`x => x + 1`" in the `fn`-binding "`fn x => x + 1`") is as follows:

$$\frac{lpat \Downarrow \langle env_1, \alpha_1, \overline{c}_1 \rangle \quad lexp \Downarrow \langle env_1, \alpha_2, \overline{c}_2 \rangle}{lpat \overset{l}{\Rightarrow} lexp \Downarrow \langle env, \alpha, \overline{c} \rangle}$$

where $\overline{c}_1$, $\overline{c}_2$ and a new constraint $\alpha \overset{l,\varnothing}{=\!=} \alpha_1 \overset{l}{\rightarrow} \alpha_2$ occur in $\overline{c}$ ($\alpha_1 \overset{l}{\rightarrow} \alpha_2$ is $\langle\langle \rightarrow, l\rangle, \langle\langle \alpha_1, \alpha_2\rangle, l\rangle\rangle$). The relation $\Downarrow$ is used by another relation genCs which generates a constraint set from an *exp*:

$$exp \xrightarrow{\mathrm{genCs}} \overline{c} \cup \overline{c}' \iff exp \Downarrow \langle env, \alpha, \overline{c} \rangle$$

where $\overline{c}'$ constrains, e.g., each free type constructor's arity to be unique. (For a complete definition of our constraint generator see our TR.)

4.4 Unification. Our unification algorithm extends the one by Haack and Wells [7] to cope with constraints on type names and sequences and let-constraints. It takes as parameter a constraint set, builds three unifiers for our three kinds of variables, keeps track of the monomorphic bindings and either returns these unifiers if it succeeds or returns one of the following type errors (see Fig. 3): a constructor clash, a type constructor arity clash or a circularity error. Our unification algorithm is divided into three parts: the driving rules, the simplification rules and a circularity test. The driving rules are as follows (where *uni* is a pack of unifiers and the constant *uniInit* is a an initial pack of unifiers):

$$
\begin{array}{lll}
\mathrm{unify}(\overline{c}) & \rightarrow \mathrm{unify}(uniInit, \overline{c}) & \text{(starting rule)} \\
\mathrm{unify}(uni, \varnothing) & \rightarrow \mathrm{success}(uni) & \text{(initial constraint set is solvable)} \\
\mathrm{unify}(uni, \overline{c}) & \rightarrow \mathrm{unify}(uni, \overline{c}', \overline{c} \setminus \overline{c}') & \text{(selection of constraints to unify)} \\
\mathrm{unify}(uni, \overline{c}, \varnothing) & \rightarrow \mathrm{unify}(uni, \overline{c}) & \text{(repeat main loop)}
\end{array}
$$

The simplification rules are of three different sorts. They can either trigger a circularity test or be one of fourteen rules of these two shapes depending on the form of c (see our unification algorithm in our TR):

$$
\begin{array}{ll}
\mathrm{unify}(uni, \overline{c}, \overline{c}' \uplus \{c\}) \rightarrow \mathrm{error}(\overline{l}, \overline{vid}, ek) & \text{(failure of unification)} \\
\mathrm{unify}(uni, \overline{c}, \overline{c}' \uplus \{c\}) \rightarrow \mathrm{unify}(uni, \overline{c}, \overline{c}' \cup \overline{c}'') & \text{(generation of } \overline{c}'' \text{ from } c\text{)}
\end{array}
$$

The unification algorithm always triggers the circularity test (which is composed of five rewriting rules) before updating one of the unifiers. The general scheme for this (in the success case) proceeds like

$$
\begin{array}{l}
\mathrm{unify}(uni, \overline{c}, \overline{c}' \uplus \{c\}) \rightarrow \mathrm{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho}, n) \rightarrow^* \\
\qquad\qquad \mathrm{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho}', m) \rightarrow \mathrm{unify}(uni', \overline{c}, \overline{c}')
\end{array}
$$

where $\overline{\rho}$ and n record information while traversing a type structure. The updating takes place only if the circularity test succeeds, meaning that the updating will not lead to a cycle.

Let us sketch our solution to handle let-constraints using the untypable expression $exp =$ `(fn x => let val rec g = (fn h => x h) in (g 1, g true) end)`.

If $exp \xrightarrow{\text{genCs}} \overline{c}_0$ then the first state of the unification of $\overline{c}_0$ is unify($\overline{c}_0$). The first important state reached during unification is when dealing with the constraint for the top-level function:

$$(1)\ \text{unify}(\overline{c}_0) \rightarrow^* \text{unify}(uni_1, \overline{c}_1, \overline{c}_2 \uplus \{\textbf{let}\ \{\langle x, \alpha_1, \alpha_2, \text{mono}, l_1\rangle\}, \varnothing(\varnothing)\ \textbf{in}\ \overline{c}_3\})$$

where α_1 and α_2 are the types of the second and first occurrences of `x` respectively. At this stage, the unification algorithm records the equality between α_1 and α_2, annotated by the context dependency `x` and the label l_1 (`x`'s label in the pattern). It also records that α_2 cannot be generalised until the unification of $\overline{c}_3$ is done. We obtain uni_2 from uni_1.

The constraints set $\overline{c}_3$ is then unified before $\overline{c}_1$ and $\overline{c}_2$. We obtain the following partial computation where we single out the constraint for the let-declaration:

$$(2)\ \text{unify}(uni_2, \overline{c}_3) \rightarrow^* \text{unify}(uni_3, \overline{c}_4, \overline{c}_5 \uplus \{\textbf{let}\ \varnothing, \{csb_1, csb_2\}(\overline{c}_6)\ \textbf{in}\ \overline{c}_7\})$$

where $csb_1 = \langle g, \alpha_3, \alpha_5, \text{poly}, l_2\rangle$, $csb_2 = \langle g, \alpha_4, \alpha_5, \text{poly}, l_2\rangle$, and α_3, α_4 and α_5 are the types of `g`'s second, third and first occurrences respectively.

The unification algorithm carries on unifying $\overline{c}_6$ which is the constraint set associated to `g`'s declaration. We obtain the following computation where the we single out the constraint for the inner function:

$$(3)\ \text{unify}(uni_4, \overline{c}_6) \rightarrow^* \text{unify}(uni_5, \overline{c}_8, \overline{c}_9 \uplus \{\textbf{let}\ \{\langle h, \alpha_6, \alpha_7, \text{mono}, l_3\rangle\}, \varnothing(\varnothing)\ \textbf{in}\ \overline{c}_{10}\})$$

where α_7 and α_6 are the types of `h`'s first and second occurrences respectively. The unifier uni_5 is then updated recording that α_6 has to be equal to α_7 if `h` is a value variable. When dealing with $\overline{c}_{10}$, the unification algorithm also constrains α_1 to be equal to $\alpha_6 \xrightarrow{l_4} \alpha_8$ (which is $\langle\langle \rightarrow, l_4\rangle, \langle\langle \alpha_6, \alpha_8\rangle, l_4\rangle\rangle$) where α_8 and l_4 are the type and label of the application `x h`. When $\overline{c}_6$ has been unified, no error has been found yet: the partial computation displayed in (3) ends in success(uni_6).

Going back to (2), the unification algorithm builds `g`'s type τ from α_5 and uni_6 and before generalising τ to constrain α_3 and α_4, it recomputes the type variables that cannot be generalised. In fact, α_2 cannot be generalised but the unification algorithm does not know yet that α_6, α_7 and α_8 cannot be generalised either because they are related to α_2 by some type equalities. Hence, the unification algorithm builds the types equal to the type variables which currently cannot be generalised. Using these types it then recomputes the set of type variables that cannot be generalised. This set is discarded after the generalisation of τ. Thanks to this intermediate step, the unification algorithm knows that τ cannot be generalised. It finally discovers an error when unifying $\overline{c}_7$. (For a complete definition of our unification algorithm see our TR.)

4.5 Minimisation and enumeration algorithms.

Enumeration. Our enumeration runs on the constraint set generated by our constraint generator. This algorithm makes an extensive use of our minimisation and unification algorithms. Our enumeration algorithm is a modified version of the one defined by Haack and Wells [7] to cope with context dependencies and to incorporate the minimisation process. Let us outline our enumeration algorithm:

$$
\begin{array}{ll}
\text{(R1) } \mathrm{enum}(\overline{c}) & \rightarrow \mathrm{enumSyn}(\mathrm{getSyn}(\overline{c}), \mathrm{getSem}(\overline{c}), \varnothing, \varnothing) \\
\text{(R2) } \mathrm{enumSyn}(\overline{c} \uplus \{\mathit{sync}\}, \overline{c'}, \overline{\mathit{err}}, \overline{\overline{l}}) & \rightarrow \mathrm{enumSyn}(\overline{c}, \overline{c'}, \overline{\mathit{err}'}, \overline{\overline{l}}') \\
\text{(R3) } \mathrm{enumSyn}(\varnothing, \overline{c}, \overline{\mathit{err}}, \overline{\overline{l}}) & \rightarrow \mathrm{enumSem}(\overline{c}, \overline{\mathit{err}'}, \overline{\overline{l}}') \\
\text{(R4) } \mathrm{enumSem}(\overline{c}, \overline{\mathit{err}}, \overline{\overline{l}} \uplus \{\overline{l}\}) & \rightarrow \mathrm{enumSem}(\overline{c}, \overline{\mathit{err}'}, \overline{\overline{l}}') \\
\text{(R5) } \mathrm{enumSem}(\overline{c}, \overline{\mathit{err}}, \varnothing) & \rightarrow \mathrm{errorSet}(\overline{\mathit{err}})
\end{array}
$$

The initial state is $\mathrm{enum}(\overline{c})$ where $\overline{c}$ is generated from some expression *exp* by $\mathit{exp} \xrightarrow{\mathrm{genCs}} \overline{c}$. First, the syntactic constraints are extracted from the initial constraint set (R1). Then the syntactic errors are enumerated (R2) and once all of these have been enumerated (R3) it enumerates the semantic errors (R4) using the semantic constraints and making an extensive use of our unification and minimisation algorithms. The enumeration of the syntactic errors (R2) consists of the transformation of the syntactic constraints into errors (see Fig. 3). During this first phase we also build an initial set of filters (set of labels used to restrict the constraint set to unify) used to restrict the search space (the $\overline{\overline{l}}$s).

These five rules are extracted from the enumeration algorithm in our TR.

Theoretically, the enumeration algorithm would be sufficient because it is designed to find all minimal errors in a piece of code. Unfortunately, because of the exponential number of potential minimal type errors, an algorithm is needed to minimise an error before searching for another one. In practise our enumeration algorithm reports as many minimal errors as it can find in the amount of time it is allowed to run (the process can be stopped whenever wanted).

Haack and Wells's enumeration and minimisation algorithms were designed to work consecutively. Their enumeration algorithm was allowed to run for a certain amount of time before returning the found errors to the minimisation algorithm which would minimise them. (The found errors would already be minimal if the time limit was not exceeded.) There are many advantages in having the enumeration algorithm call the minimisation algorithm as soon as it finds an error such as: all errors have to be minimised at some point, so we might as well minimise them as soon as we find them; using Haack and Wells's method, two different errors can be enumerated and then minimised to the same minimal error (this cannot happen if the minimisation algorithm is called as soon as an error has been found); the search space is reduced more quickly.

Minimisation. We designed a new minimisation algorithm because let-constraints interfered with the one by Haack and Wells. The fact that we have to solve the constraints associated to the left-hand-side of a let-binding before any other constraints interferes with a naive approach for a minimisation algorithm that

consists of starting from a type error and building a unifier containing the type information about the locations that *have to be* in the currently built minimal error. These locations cannot be picked as the last locations treated before the failure of unification. For example, for: `let val (x, y) = (true, 1) in x + y end`, with the let-constraints we first unify the constraints associated to the declaration. Fresh instances of the types of `x` and `y` are then created and constrained to be equal to the types of the occurrences of these identifiers in their scope. If an error is not restricted to the body of only one declaration, it can only be discovered when dealing with a let-constraint. It becomes then complicated to blame a specific label for the failure of the unification (in Haack and Wells's approach such labels are used to build a minimal error).

Our algorithm is in two steps. During these two steps, the algorithm removes from a location set all the unnecessary locations for an error still to occur. From the constraint point of view, it consists in removing as many constraints as possible from an unsolvable constraint set while keeping the constraint set unsolvable. First, it tries to remove complete declarations from a found error, making use of the bindings occurring in let-constraints:

$$\begin{array}{l}\text{unbind}(\langle\overline{l}_0\rangle@\,\overrightarrow{\overline{l}}\,,\overline{l},\overline{c}) \rightarrow \text{unbind}(\,\overrightarrow{\overline{l}}\,,\overline{l}',\overline{c})\\ \text{if unify}(\text{projLabCs}(\overline{c},\overline{l}_1\setminus\overline{l})) \rightarrow^* \text{success}(\mathit{uni}) \text{ then } \overline{l}' = \overline{l} \text{ else } \overline{l}' = \overline{l}\setminus\overline{l}_0\end{array}$$

where $\text{projLabCs}(\overline{c},\overline{l})$ are the constraints from $\overline{c}$ annotated by $\overline{l}$ only. The initial state is $\text{unbind}(\,\overrightarrow{\overline{l}}\,,\overline{l}_0,\overline{c})$ where $\overrightarrow{\overline{l}}$ is a sequence of label sets occurring in the let-constraints of $\overline{c}$. In `let val x = let val y = 1 in y end in x end` if l_1 is the label of `x`'s first occurrence and l_2 is the label of `y`'s first occurrence then $\overrightarrow{\overline{l}}$ would be $\langle\{l_1\},\{l_2\}\rangle$. At any stage of the first phase, the second component $\overline{l}\subseteq\overline{l}_0$ is the label set being minimised such that the constraint set annotated by $\overline{l}$ only is still unsolvable. In a second phase it removes any other label that can be removed:

$$\begin{array}{l}\text{reduce}(\overline{c},\overline{l}_1,\{l\}\cup\overline{l}_2) \rightarrow \text{reduce}(\overline{c},\overline{l}'_1,\overline{l}_2)\\ \text{if unify}(\text{projLabCs}(\overline{c},\overline{l}_1\cup\overline{l}_2)) \rightarrow^* \text{success}(\mathit{uni}) \text{ then } \overline{l}'_1 = \overline{l}_1\cup\{l\} \text{ else } \overline{l}'_1 = \overline{l}_1\end{array}$$

Initially, the first label set is empty and the second one is the label set being minimised. At each step these two sets are such that the constraint set annotated by their union only is still unsolvable. Finally the minimisation of a label set is as follows (from $\overline{l}_1$ to $\overline{l}_3$):

$$\text{unbind}(\,\overrightarrow{\overline{l}}\,,\overline{l}_1,\overline{c}) \rightarrow^* \text{unbind}(\varnothing,\overline{l}_2,\overline{c}) \rightarrow \text{reduce}(\overline{c},\varnothing,\overline{l}_2) \rightarrow^* \text{reduce}(\overline{c},\overline{l}_3,\varnothing)$$

(For a complete definition of our minimisation and enumeration algorithms see our TR.)

4.6 Slicing algorithm. From a minimal label set our slicing algorithm computes a slice by removing nodes and edges from an abstract syntax tree. In order to represent such a partial abstract syntax tree we extend the syntax presented in Sec. 4.1 with "dot" terms which are terms of the syntax presented in Fig. 1 in which some nodes and edges have been removed. For example, if we remove the

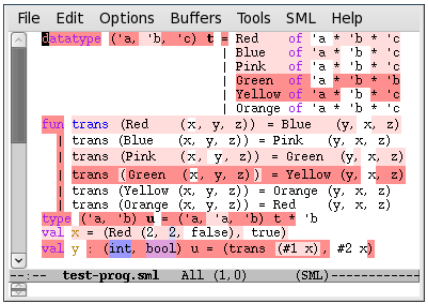
node associated to the label l_2 (the unit atomic expression) in $\lceil 1^{l_1}\ ()^{l_2} \rceil^{l_3}$ then we obtain $\lceil 1^{l_1}\ \text{dots}(\varnothing) \rceil^{l_3}$. In our implementation, such a term (slice) is displayed as: `1 ⟨..⟩`.

Even though the internal syntax for pieces of code and slices is the same, the printing of a term differs depending if it is a piece of code or a slice. For example, in a match the number of match rules being irrelevant, as a slice $\text{fn}^{l}\ 1 \overset{l_1}{\Rightarrow} 1 \mid 2 \overset{l_2}{\Rightarrow} 2$ is displayed as `fn ⟨..1 => 1..2 => 2..⟩`. (For the definition of the extension of the syntax presented in Sec. 4.1 with "dot" terms, see our TR.)

The slicing algorithm takes as parameters an expression and a label set and slices out all these labels in the expression as follows: if $exp \xrightarrow{\text{genCs}} \overline{c}$ for some expression exp and $\text{enum}(\overline{c}) \rightarrow^* \text{errorSet}(\overline{err})$ then each error in $\overline{err}$ contains a label set which is used to generate a slice from exp. (For a complete definition of the slicing algorithm, see our TR.)

5 Implementation details

User interface. We have an Emacs interface to highlight slices directly into the source code as it is being edited. We provide a screenshot of the type error presented in Sec. 2.5 on the right.



The Standard ML basis library. Our examples have used operators like `::` and `+`. For now, we allow defining the Standard ML basis in a file, and we provide a file declaring a portion of the basis. For the future, we have begun implementing a way to use library types extracted from a running instance of SML/NJ, but there are still technical challenges to overcome. Some types and structures appear to be hidden in such running instances. One challenge is to get information about type equality. For example, we discover that `+` is overloaded and one of its implementations is `I32.+` at type `?.int32 * ?.int32 -> ?.int32`, but unfortunately there is no structure named `I32` and the type that is supposedly named `?.int32` is probably in fact `Int32.int`. More work is needed, including negotiation with compiler implementers on a standard way of getting this information.

6 Related work

Methods making use of slices. After the first version of TES presented by Haack and Wells, many researchers began to present type errors as program slices obtained from unsolvable sets of constraints.

Neubauer and Thiemann [16] use flow analysis to compute type dependencies for a small ML-like language to then produce type error reports. Based on discriminative sum types, their system can type any term. Type errors are produced from the analysis of type derivations. Their method is in two steps. The first step ("collecting phase") labels the studied term and infers type information. This analysis generates a set of program point sets. These program points are directly

stored in the discriminative sum types. A conflicting type ("multivocal") is then paired with the locations responsible for its generation. The second step ("reporting phase") consists in generating error reports from the conflicts generated during the first phase. Slices are built from which highlighting are produced. An interesting detail is that a type derivation can be viewed as the description of all type errors in an untypable piece of code, from which another step is needed to compute error reports.

A similar approach to ours is the one by Stuckey, Sulzmann and Wazny [22, 24] (based on earlier work without slices [20, 21]). They do type inference, type checking and report type errors for the Chameleon language (derived from a Haskell subset). This language includes algebraic data types, type-class overloading, and functional dependencies. They code the typing problem into a constraint problem. They use labels attached to constraints to track program locations and highlight parts of untypable pieces of code. First they compute a minimal unsatisfiable set of generated constraints from which they select one of the type error locations to provide their type explanation. They finally provide a highlighting and an error message depending on the selected location. They provide slice highlighting but using a different strategy from ours. They focus on explaining conflicts in the inferred types at one program point inside the error location set. It is not completely clear, but they do not seem to worry much about whether the program text they are highlighting is exactly (no more and no less) a complete explanation of the type error. For example, they do not highlight applications because "they have no explicit tokens in the source code". It is then stated: "We leave it to the user to understand when we highlight a function position we may also refer to its application". This differs from our strategy in the sense that we think it is preferable to highlight all the program locations responsible for an error even if these are white spaces. Moreover, they do not appear to highlight the parts of datatype declarations relevant to type errors.

When running on a translation of the code presented in Sec. 2.5 into Haskell, ChameleonGecko outputs an error report containing the text presented on the right (the rest of the output seems to be internal information computed during unification only). This highlighting identifies the same error location as SML/NJ does and would not help solve the error.

```
ERROR: Type error; conflicting sites:
y = (trans x1, x2)
```

Significantly, because they handle a Haskell-like language, they face challenges for accurate type error location that are different from the ones for SML.

Gast [5] generates "detailed explanations of ML type errors in terms of data flows". His method is in three steps: generation of subtyping constraints annotated by reasons for their generation; gathering of reasons during constraint unification; transformation of the gathered reasons into explanations by data flows. He provides a visually convenient display of the data flows with arrows in XEmacs. Gast's method (which seems to be designed only for a small portion of OCaml) can be considered as a slicing method with data flow explanations.

Braßel [3] presents a generic approach (implemented for the language Curry) for type error reporting that consists in two different procedures. The first one

tries to replace portions of code by dummy terms that can be assigned any type. If an untypable piece of code becomes typable when one of its subtrees has been replaced by a dummy term then the process goes on to apply the same strategy inside the subtree. The second procedure consists in the use of a heuristic to guide the search of type errors. The heuristic is based on two principles: it will always "prefer an inner correction point to an outer one" and will always "prefer the point which is located in a function farther away in the call graph from the function which was reported by the type checker as the error location". Braßel's method does not seem to compute proper slices but instead singles out different locations that might be the cause of a type error inside a piece of code.

Significant non-slicing type explanation methods. Heeren et al. designed a method used in the Helium project [10, 9, 11, 8] to provide error messages for the Haskell language relying on a constraint-based type inference. First, a constraint graph is generated from a piece of code. For an ill-typed piece of code, a conflicting path called an inconsistency is extracted from the constraint graph. Such a conflicting path is a structured unsolvable set of type constraints. Heuristics are used to remove inconsistencies. To each type constraint is associated a trust value and depending on these values and the defined heuristics, some constraints are discarded until the inconsistency is removed. They also propose some "program correcting heuristics" used to search for a typable piece of code from an untypable one. Such a heuristic is for example the permutation of parameters which is a common mistake in programming. Their approach has been used with students learning functional programming. Using pieces of code written by students and their expertise of the language they refined their heuristics. This approach differs from ours by privileging locations over others by the use of some heuristics. They do not compute minimal slices and highlightings.

We present on the right the most interesting part of the error report obtained using Helium on a translation of the code presented in Sec. 2.5 into Haskell. It is reported that `x1` and `trans` don't have the expected types. The application, which is at the end of the code, is then blamed when our programming error is at the very beginning of the code.

```
(16,6): Type error in application
expression       : trans x1
term             : trans
  type           : T a   a   a    -> T a   a   a
  does not match : T Int Int Bool -> T Int Int Bool

Compilation failed with 1 error
```

Lerner, Flower, Grossman and Chambers [13] present a new approach for type error messages. The targeted language is Caml, but they also developed a prototype for C++. As for Heeren et al. [8], their method consists in the construction of a well-typed program from an ill-typed one using different techniques such as switching two parameters. The generated typable code is presented as a possible code that the programmer intended to write. Each time the ill-typed piece of code is modified, an unchanged type checker is called to check if the modified code is now typable. Even though it is a completely different approach from the slicing approach it could be interesting to study their combination.

7 Conclusion

Summary of contributions. Our contributions in this paper are as follows:

1. We have carefully defined unambiguous slices for many different kinds of SML type errors. They provide as close as possible to exactly the information needed by the programmer to solve its errors. Our slices and highlightings provide specific details concerning each feature of SML that we implement.

2. We solve a previous efficiency problem (combinatorial explosion of the number of generated constraints) by adopting a new system of constraints in the style of Pottier and Rémy [19, 18]. We design a minimisation algorithm that can handle the new system of constraints.

3. We deal with the ambiguity of distinct identifier statuses in SML while also computing minimal type error slices. We solve this ambiguity by generating type constraints with context dependencies on the status of identifiers. Every dependency has the potential to at any time either be confirmed or rejected.

4. We extend TES to handle context-sensitive syntactic errors. These include errors that are related to the ambiguity of identifier status, as well as other context-sensitive syntax errors.

5. We report technical details on the mechanisms of our TES and discuss implementation issues encountered during the creation of our TES.

6. We have an implementation that handles more features than reported in this paper. We report in the paper's formalism features such as datatype declarations but we implement many more features such as records, exceptions, type definitions, explicit types, many module features, etc.

7. A web demo of our TES as well as a downloadable package are available at `http://www.macs.hw.ac.uk/ultra/compositional-analysis/type-error-slicing` .

Future work. We have already implemented some merging of minimal slices and are currently extending this idea to other kind of errors than record clashes. Merging errors can sometimes be useful to emphasis the locations shared by different slices for the same programming error. It might involve using different colour shades depending on the percentage of minimal slices to which a location contributes. (If a location is shared by many slices it does not always mean that it is more likely to be a programming error location.)

We plan extra constraint annotations to enhance error reports. Such annotations would, e.g., explain that a constraint comes from a monomorphic binding or because of the value polymorphism restriction. The messages would give more insight when non-obvious features might be responsible for an error.

In the near future, we plan to finish extending our TES to the full SML language. This includes finishing our implementation of SML's structures and functors (i.e., SML's module system). It also includes features such as flexible records or equality types which are not yet implemented.

Finally, we have begun user evaluations that will help us design proper experiments comparing the effectiveness in improving the productivity of real users of TES vs. more traditional type error messages.

References

1. *16th Int'l Workshop, IFL 2004*, vol. 3474 of *LNCS*. Springer, 2005.
2. M. Beaven, R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4), 1993.
3. B. Braßel. TypeHope - there is hope for your type errors. In *16th Int'l Workshop, IFL 2004* [1].
4. L. Damas, R. Milner. Principal type-schemes for functional programs. New York, NY, USA, 1982. ACM.
5. H. Gast. Explaining ML type errors by data flows. In *16th Int'l Workshop, IFL 2004* [1].
6. C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *ESOP*, vol. 2618 of *LNCS*. Springer, 2003.
7. C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3), 2004.
8. J. Hage, B. Heeren. Heuristics for type error discovery and recovery. In *18th Int'l Symp., IFL 2006*, vol. 4449 of *LNCS*. Springer, 2007.
9. B. Heeren, J. Hage. Type class directives. In *7th Int'l Symp., PADL 2005*, vol. 3350 of *LNCS*. Springer, 2005.
10. B. Heeren, J. Jeuring, D. Swierstra, P. A. Alcocer. Improving type-error messages in functional languages. Technical report, Utrecht University, 2002.
11. B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005.
12. O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4), 1998.
13. B. S. Lerner, M. Flower, D. Grossman, C. Chambers. Searching for type-error messages. In *ACM SIGPLAN 2007 Conference PLDI*. ACM, 2007.
14. B. J. McAdam. On the unification of substitutions in type inference. In *10th Int'l Workshop, IFL'98*, vol. 1595 of *LNCS*. Springer, 1999.
15. R. Milner, M. Tofte, R. Harper, D. Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.
16. M. Neubauer, P. Thiemann. Discriminative sum types locate the source of type errors. In *8th ACM SIGPLAN Int'l Conference, ICFP 2003*. ACM, 2003.
17. M. Odersky, M. Sulzmann, M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1), 1999.
18. F. Pottier. A modern eye on ML type inference: old techniques and recent developments. Lecture notes for the APPSEM Summer School, 2005.
19. F. Pottier, D. Rémy. The essence of ML type inference. In B. C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, chapter 10. MIT Press, 2005.
20. P. J. Stuckey, M. Sulzmann, J. Wazny. Interactive type debugging in haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2003. ACM.
21. P. J. Stuckey, M. Sulzmann, J. Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2004. ACM.
22. P. J. Stuckey, M. Sulzmann, J. Wazny. Type processing by constraint reasoning. In *4th Asian Symp., APLAS 2006*, vol. 4279 of *LNCS*. Springer, 2006.
23. M. Wand. Finding the source of type errors. In *13th ACM SIGACT-SIGPLAN Symp., POPL'86*, New York, NY, USA, 1986. ACM.
24. J. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, University of Melbourne, Australia, 2006.
25. J. Yang. Explaining type errors by finding the source of a type conflict. In *1st Workshop, SFP'99*, Exeter, UK, 2000. Intellect Books.
26. J. Yang, G. Michaelson, P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45, 2002.
27. J. Yang, J. Wells, P. Trinder, G. Michaelson. Improved type error reporting. In *12th Int'l Workshop, IFL 2000*, vol. 2011 of *LNCS*. Springer, 2001.