

Type Error Slicing in Implicitly Typed Higher-Order Languages*

Christian Haack and J. B. Wells

Heriot-Watt University
<http://www.cee.hw.ac.uk/ultra/>

Abstract. Previous methods have generally identified the location of a type error as a particular program point or the program subtree rooted at that point. We present a new approach that identifies the location of a type error as a set of program points (a *slice*) all of which are necessary for the type error. We describe algorithms for finding minimal type error slices for implicitly typed higher-order languages like Standard ML.

1 Introduction

1.1 Previous Approaches to Identifying Type Error Locations. There has been a large body of work on explaining type errors in implicitly typed, higher-order languages with let-polymorphism (Haskell, Miranda, O’Caml, Standard ML (SML), etc.) [26, 19, 18, 30, 28, 2, 3, 9, 1, 15, 8, 20, 29]. This is much harder than in monomorphic, explicitly typed, first-order languages. None of the previous work on this is entirely satisfactory. In particular, the previous approaches do a poor job of identifying the *location* of type errors.

As an example, consider the following SML program fragment:

```
val f = fn x => fn y => let val w = x + 1 in w::y end
```

This defines a function `f` such that the function call `(f 1 [2])` should compute the list `[2,2]`. Suppose the programmer erroneously typed this instead, making an error at the indicated spot:

```
val f = fn x => fn y => let val w = y + 1 in w::y end
```

When the \mathcal{W} [6], \mathcal{M} [18], or the \mathcal{UAE} [28, 30] type inference algorithms are used to identify the error location, the type inference algorithm traverses the program’s abstract syntax tree and when it fails, the node of the tree currently being visited is blamed. The algorithms differ in how eagerly they check the various type constraints, so they may fail at different nodes. When using either \mathcal{W} or \mathcal{UAE} for the example, this error location is identified:

```
val f = fn x => fn y => let val w = y + 1 in w::y end
```

* This work was partially supported by EPSRC grant GR/R 41545/01, EC FP5 grant IST-2001-33477, NATO grant CRG 971607, NSF grants CCR 9988529 and ITR 0113193, Sun Microsystems equipment grant EDUD-7826-990410-US.

Although \mathcal{UAE} was designed with the intention that unlike \mathcal{W} it would blame a location containing the error, it handles `let`-bindings in the same way as \mathcal{W} so it fails in the same way on this error. It has been proposed to use \mathcal{M} instead of \mathcal{W} because this would yield more “accurate” error locations. For the example, \mathcal{M} identifies this error location:

```
val f = fn x => fn y => let val w = y + 1 in w::y end
```

This example illustrates the general fact that \mathcal{W} , \mathcal{M} , and \mathcal{UAE} often fail to identify the real location of the error. They identify *one* node of the program tree which *participates* in the type error, but will often be the wrong node to *blame*. These approaches also often identify program subtrees that include many locations that do *not* participate in the type error, e.g., in the example both \mathcal{W} and \mathcal{UAE} include the occurrence of `w` in the blamed subtree. This problem can also happen for \mathcal{M} in some cases, although it does not happen as often. For \mathcal{W} and \mathcal{M} , this is not necessarily wrong because only the root of the subtree is being blamed, not necessarily all of the other nodes in the subtree, but the programmer will often not understand this distinction.

1.2 A New Notion of Type Error Location. In contrast, this paper locates errors not at single nodes or subtrees of the abstract syntax tree, but at program *slices*. For the example, our implementation finds this error location:

```
val f = fn x => fn y => let val w = y + 1 in w::y end
```

This correctly includes all of the parts of the program where changes can be made to fix the type error. Importantly, it also correctly *excludes* all of the parts of the program where changes can not fix the type error. The occurrences of `+` and `::` are highlighted differently to show they are the “endpoints” of a clash between the `int` and `list` type constructors. As an alternative, the erroneous slice of the program can be presented separately by displaying a very small program that contains the same type error as the source program, and nothing but this type error. In many cases, this will make it easier for the programmer to understand the error, especially when the error spans multiple source files. Here is actual output from our implementation in this style:¹

```
Type of error: type constructor clash, endpoints int vs. list.
(.. y => (.. y + (..) .. (..) ::y ..) ..)
```

Formally, a *type error slice* is a set of program points. It is *complete* if these program points together “form a type error”. It is *accurate* if none these program points is irrelevant for the type error. Examples of incomplete type error slices include the locations that are returned in most error messages of, for example, the SML/NJ compiler. They consist of a single program point, namely the point where the type inference algorithm detects a failure. This program point by

¹ The `fn` keyword is missing because SML has the *match* syntax. That `x` is bound in a *fn-match* as opposed to a *case-match* is irrelevant for the error.

itself does not form a type error. As an example of an inaccurate type error slice, one could take the entire program if it contains a type error. If the type error locations produced by the \mathcal{W} , \mathcal{M} , or \mathcal{UAE} algorithms are taken to be identifying a program *subtree*, then they will usually be inaccurate.

1.3 Related Work. Dinesh and Tip have applied slicing techniques for locating sources of type errors [8]. Their techniques are applicable to *explicitly* typed languages. Their approach depends on the fact that the type system can be expressed as a rewrite system, and they use techniques for origin and dependency tracking in rewrite systems to find error locations. Although type inference algorithms for implicitly typed languages can be phrased as rewrite systems, a large part of the rewrite rules would concern auxiliary functions, i.e., unification and constraint solving. For this reason, we do not believe that a direct application of Dinesh and Tip’s methods results in accurate location of type error sources in languages with type inference.

Our work is based on Damas’ type inference system [7]. This system differs from the more widely known type *scheme* inference system (i.e., the Hindley/Milner system) in the typing rule for let-expressions, but admits the same set of well-typed closed expressions. It can be seen as a restriction of a system of rank-2 intersection types. Jim [14] has proposed using rank-2 intersection types for accurate type error location. Bernstein and Stark [2] use Damas’s type inference system for type error debugging of open terms.

Wand has presented an algorithm for finding the source of type errors in implicitly typed languages [26]. Similar methods have been used by Duggan and Bent [9]. Wand’s algorithm uses a modified unification procedure that keeps track of constraint sets that have been used in the derivation of unsolvable constraints. However, there is no attempt to present the corresponding program slices and these constraint sets need not be minimally unsolvable. We use a similar method as a subroutine, but in addition, we minimize constraint sets and present the resulting minimal type error slices. Our slices are minimal in the sense that the omission of further program points yields a non-error. Johnson and Walz have a method which attempts to choose the location to blame by counting the number of sites which prefer one type over another [15].

Chopella and Haynes study type error diagnosis in a simply typed language [5, 4]. Unlike our work, they do not actually treat let-polymorphism. They propose to present type error locations as program slices, but have no notation for slices. Moreover, they present a graph-based unification framework, based on work by Port [23], which could be used for finding minimal unsolvable constraint sets. However, the diagnostic unification algorithm that is eventually presented in [4] only computes a single unsolvable constraint set that is not necessarily minimal. In contrast, our algorithms are not graph-based but based on running a unification algorithm multiple times. A big advantage of our approach is simplicity of presentation and implementation. Unlike Chopella and Haynes, we give a detailed presentation of an algorithm that enumerates minimal unsolvable constraint sets. On the other hand, while our algorithm enumerates *some* minimal unsolvable subsets of a given constraint set, the algorithm is impractical for

exhaustively enumerating *all* such sets. In the worst case, enumerating all such sets is intractable [27]. In some cases an algorithm based on Port’s idea may find all minimal unsolvable subsets, whereas ours does not. In the future, we may adopt the algorithm that is sketched by Port.

Heeren and others propose constraint-based type inference for improved type error messages [13, 12, 11]. They treat let-polymorphism, and their type system is between a type inference system and a type scheme inference system. In addition to equality constraints, their inference algorithm generates type scheme instance constraints. As a result, the constraint solving order is restricted. We believe that a type inference system without type schemes would simplify their system and sometimes permit more accurate error messages. They do not attempt to compute type error slices.

MrSpidey is a static debugger for Scheme that is distributed with some versions of the DrScheme programming environment [10]. It is based on set-based flow analysis, constructs and, on demand, displays parts of flow graphs, and highlights critical program points at which runtime errors may occur.

Much related work on type error analysis has spent a great deal of effort on sophisticated ways for automatically generating type error *explanations* [3, 9, 20, 26, 29, 1, 19]. Such explanations tend to be complicated and lengthy. We believe that it is most important to *accurately locate* type errors, and display type error *locations* in a user-friendly way. For *understanding* errors, programmers typically use additional semantic knowledge that cannot be provided automatically anyways. Our work is intended to be a step into this direction.

1.4 Outline of Paper. Section 3 gives an overview of Damas’ type inference system. The methods for type error slicing proceed in three steps. The first step consists of assigning constraints to program points. In order to obtain accurate type error slices, it is important to follow a certain strategy. This strategy is described in section 4. The second step consist of finding minimal unsolvable subsets in the set of all constraints. Section 5 describes algorithms for doing this. Finally, section 6 describes how type error slices are computed from the results obtained in the previous step.

For concreteness, we describe our methodology in detail for the small model language shown in figure 1. The labels that superscript expressions mark program points. The labeled expression language is a sublanguage of Standard ML (SML) [21]. We have an implementation for a larger sublanguage of SML.²

1.5 Acknowledgments We thank Sébastien Carrier for his help in making the web demonstration interface and Greg Michaelson, Phil Trinder, and Jun Yang for stimulating discussions.

2 Some Definitions and Notations

The symbols \langle and \rangle denote tuple braces. We use the terms “tuple” and “list” interchangeably. For a list $xs = \langle x_1, \dots, x_n \rangle$, the expression $y :: xs$ denotes the

² <http://www.cee.hw.ac.uk/ultra/compositional-analysis/type-error-slicing>

| | |
|-------------------------|---|
| $l \in \text{Label}$ | a fixed infinite set of labels |
| $L \in \text{LabelSet}$ | all finite subsets of Label |
| $x \in \text{Var}$ | a fixed infinite set of variables |
| $n \in \text{Int}$ | the set of integers |
| $lexp \in \text{LExp}$ | $::= x^l \mid n^l \mid (lexp + lexp)^l \mid (\text{fn } x^l \Rightarrow lexp)^l$ $\mid (lexp \ lexp)^l \mid (\text{let val } x^l = lexp \ \text{in } lexp \ \text{end})^l$ |

Restriction: The labels that occur in a labeled expression must be distinct.

Fig. 1. Labeled expressions

list $\langle y, x_1, \dots, x_n \rangle$. For each natural number i , the symbol π_i denotes the i -th projection operator, i.e., if $xs = \langle x_1, \dots, x_n \rangle$ and $i \in \{1, \dots, n\}$, then $\pi_i(xs) = x_i$. If f is a function, then $f[x \mapsto y]$ denotes the function $(f \setminus \{x, f(x)\}) \cup \{x, y\}$. If X is a set and \rightarrow is a subset of $X \times X$, then \rightarrow^* denotes its reflexive and transitive closure. An element x is called *irreducible* with respect to \rightarrow , if there is no element y such that $x \rightarrow y$. If X is a set of sets, then $\min(X)$ denotes the set of all elements of X that are minimal with respect to set inclusion. Two sets are called *incomparable* if neither of them is a subset of the other one. In definitions of rewrite systems, we use a form of pattern matching. The symbol \cdot denotes a *wildcard* and is matched by any element of the appropriate domain. A *disjoint union pattern* is of the form $pat_1 \uplus pat_2$ and is matched by a set X , iff there are sets X_1, X_2 such that $X_1 \cup X_2 = X$, $X_1 \cap X_2 = \emptyset$, X_1 matches pat_1 and X_2 matches pat_2 . Usually, X matches $pat_1 \uplus pat_2$ in more than one way.

3 Damas's Type Inference System

Types are defined as follows:

| | |
|---|--|
| $ty \in \text{Ty} ::= a \mid \text{int} \mid ty \rightarrow ty$ | $ity \in \text{IntTy} ::= \wedge S$ |
| $a \in \text{TyVar}$ | a fixed infinite set of type variables |
| $S \in \text{TySet}$ | the set of all finite subsets of Ty |

The elements of IntTy are called *intersection types*. The symbol \wedge is syntax. For example, $\wedge\{a \rightarrow \text{int}, \text{int} \rightarrow a\} \in \text{IntTy}$. A *type environment* is a total function from Var to IntTy . Let Γ range over Env , the set of all type environments. Let empty be the type environment that maps all variables to $\wedge\{\}$.

Damas's type inference system is defined in figure 2. We will call it Damas's System T because it is used with Damas's algorithm T. It differs in the rule for let-expressions from the usual system for SML, which Damas called the type *scheme* inference system. Whereas the type *scheme* inference system requires the types of all occurrences of a let-bound variable to be substitution instances of a common type scheme, System T does not require this. However, Damas showed that the two approaches accept the same expressions. The following fact is a variation of proposition 2 in Damas's Ph.D. thesis [7, p. 85].

Fact 3.1 *For closed $lexp$, $(\text{empty} \vdash lexp : ty)$ iff $lexp$ has type ty in SML.*³ \square

³ Formally, some minor syntactic adjustments (omitted here) are needed to translate $lexp$ into an exp of the SML definition [21].

$$\begin{array}{l}
\Gamma[x \mapsto \wedge\{ty, \dots\}] \vdash x^l : ty \\
\Gamma \vdash n : \mathbf{int} \\
(\Gamma \vdash lexp_1 : \mathbf{int}) \text{ and } (\Gamma \vdash lexp_2 : \mathbf{int}) \quad \Rightarrow \Gamma \vdash (lexp_1 + lexp_2)^l : \mathbf{int} \\
\Gamma[x \mapsto \wedge\{ty\}] \vdash lexp : ty' \quad \Rightarrow \Gamma \vdash (\mathbf{fn } x^l \Rightarrow lexp)^l : ty \rightarrow ty' \\
(\Gamma \vdash lexp_1 : ty' \rightarrow ty) \text{ and } (\Gamma \vdash lexp_2 : ty') \quad \Rightarrow \Gamma \vdash (lexp_1 lexp_2)^l : ty \\
(n \geq 1) \text{ and } (\forall i \in \{1, \dots, n\}. \Gamma \vdash lexp : ty_i) \text{ and } (\Gamma[x \mapsto \wedge\{ty_1, \dots, ty_n\}] \vdash lexp' : ty) \\
\Rightarrow \Gamma \vdash (\mathbf{let val } x^l = lexp \mathbf{ in } lexp' \mathbf{ end})^l : ty
\end{array}$$

Fig. 2. Damas's typing rules

We use the system, because it is good for accurately locating sources of type errors. The use of closely related systems has been proposed previously for type error analysis [2, 14] as well as separate compilation [24, 14].

4 Assigning Constraints to Program Points

This section explains how constraints are assigned to program points. We will define a function that maps labeled expressions to finite sets of constraints associated with program points. An expression is typable if and only if the associated constraint set is solvable. The association between constraints and particular program points is important for an untypable expression $lexp$. All program points in $lexp$ associated with a minimal unsolvable subset of the set of constraints generated for $lexp$ jointly cause a type error, and we display these program points as the location of the type error.

A *labeled constraint* is a triple $\langle ty, ty', L \rangle$, which will be written as $ty \stackrel{L}{=} ty'$. Such a labeled constraint is called *atomically labeled*, if L is a one-element set. Let $ty \stackrel{l}{=} ty'$ stand for $ty \stackrel{\{l\}}{=} ty'$. Let C range over $\mathbf{AtConstraintSet}$, the set of all finite sets of atomically labeled constraints. Let D range over $\mathbf{ConstraintSet}$, the set of all finite sets of labeled constraints. A *type substitution* is a function from \mathbf{TyVar} to \mathbf{Ty} . If s is a type substitution and ty a type, then $s(ty)$ denotes the type that results from ty by replacing each type variable occurrence a in ty by $s(a)$. A *solution* to a constraint $ty \stackrel{L}{=} ty'$ is a type substitution s such that $s(ty)$ and $s(ty')$ are syntactically equal. A solution to a set of constraints is a type substitution that solves all constraints in the constraint set simultaneously. The projection operator Π_L is defined by $\Pi_L(C) = \{(ty \stackrel{l}{=} ty') \in C \mid l \in L\}$. Let Π_l stand for $\Pi_{\{l\}}$.

The total function \Downarrow from \mathbf{LExp} to $\mathbf{Env} \times \mathbf{Ty} \times \mathbf{AtConstraintSet}$ is defined as the least relation that satisfies the rules in figure 3. This function is a variation of Damas's type assignment algorithm T. We use the term "fresh variant" of an object involving type variables to denote the result of renaming the type variables occurring in it by fresh type variables. We define $(\wedge S) \wedge (\wedge S') = \wedge(S \cup S')$. The operation \wedge on type environments is defined by $(\Gamma \wedge \Gamma')(x) = \Gamma(x) \wedge \Gamma'(x)$. We define $(\wedge S) \blacktriangleleft (\wedge S')$, iff $S \subseteq S'$, and $\Gamma \blacktriangleleft \Gamma'$, iff $\Gamma(x) \blacktriangleleft \Gamma'(x)$ for all x in \mathbf{Var} . The following facts are variations of propositions 7 and 8 on pages 39 and 44 in Damas's Ph.D. thesis [7].

$$\frac{}{x^l \Downarrow \langle \text{empty}[x \mapsto \wedge\{a_x\}], a, \{a_x \stackrel{l}{=} a\} \rangle} \quad \text{where } a_x, a \text{ fresh}$$

$$\frac{}{n^l \Downarrow \langle \text{empty}, a, \{\text{int} \stackrel{l}{=} a\} \rangle} \quad \text{where } a \text{ fresh}$$

$$\frac{\text{lexp}_1 \Downarrow \langle \Gamma_1, ty_1, C_1 \rangle; \quad \text{lexp}_2 \Downarrow \langle \Gamma_2, ty_2, C_2 \rangle}{(\text{lexp}_1 + \text{lexp}_2)^l \Downarrow \langle \Gamma_1 \wedge \Gamma_2, a, C_{\text{new}} \cup C_1 \cup C_2 \rangle}$$

where a fresh, $C_{\text{new}} = \{ty_1 \stackrel{l}{=} \text{int}, ty_2 \stackrel{l}{=} \text{int}, \text{int} \stackrel{l}{=} a\}$

$$\frac{\text{lexp} \Downarrow \langle \Gamma[x \mapsto \wedge\{ty_1, \dots, ty_n\}], ty, C \rangle}{(\text{fn } x^l \Rightarrow \text{lexp})^{l'} \Downarrow \langle \Gamma[x \mapsto \wedge\{\}], a, C_{\text{new}} \cup C \rangle}$$

where a_x, a fresh, $C_{\text{new}} = \{a_x \stackrel{l}{=} ty_1, \dots, a_x \stackrel{l}{=} ty_n, a_x \rightarrow ty \stackrel{l'}{=} a\}$

$$\frac{\text{lexp}_1 \Downarrow \langle \Gamma_1, ty_1, C_1 \rangle; \quad \text{lexp}_2 \Downarrow \langle \Gamma_2, ty_2, C_2 \rangle}{(\text{lexp}_1 \text{ lexp}_2)^l \Downarrow \langle \Gamma_1 \wedge \Gamma_2, a, C_{\text{new}} \cup C_1 \cup C_2 \rangle}$$

where a, a_1, a_2 fresh, $C_{\text{new}} = \{ty_1 \stackrel{l}{=} a_1 \rightarrow a_2, ty_2 \stackrel{l}{=} a_1, a \stackrel{l}{=} a_2\}$

$$\frac{\text{lexp}_1 \Downarrow \langle \Gamma_1, ty_1, C_1 \rangle; \quad \text{lexp}_2 \Downarrow \langle \Gamma_2[x \mapsto \wedge\{ty'_1, \dots, ty'_n\}], ty_2, C_2 \rangle}{(\text{let val } x^l = \text{lexp}_1 \text{ in } \text{lexp}_2 \text{ end})^{l'} \Downarrow \langle \Gamma'_1 \wedge \Gamma_2[x \mapsto \wedge\{\}], a, C_{\text{new}} \cup C'_1 \cup C_2 \rangle}$$

where $\langle \Gamma_{1,1}, ty_{1,1}, C_{1,1} \rangle, \dots, \langle \Gamma_{1,k}, ty_{1,k}, C_{1,k} \rangle$ are fresh variants of $\langle \Gamma_1, ty_1, C_1 \rangle$,
 $\Gamma'_1 = \Gamma_{1,1} \wedge \dots \wedge \Gamma_{1,k}$, $C'_1 = C_{1,1} \cup \dots \cup C_{1,k}$, $C = \{ty_{1,1} \stackrel{l}{=} ty'_1, \dots, ty_{1,n} \stackrel{l}{=} ty'_n\}$,
 a fresh, $C_{\text{new}} = \{a \stackrel{l'}{=} ty_2\} \cup C$, $k = \max(n, 1)$

Fig. 3. Algorithm T

Fact 4.1 Suppose $(\text{lexp} \Downarrow \langle \Gamma, ty, C \rangle)$.

1. If s is a solution of C , then $(s(\Gamma) \vdash \text{lexp} : s(ty))$.
2. If $(\Gamma' \vdash \text{lexp} : ty')$, then there is a solution s of C such that $s(\Gamma) \blacktriangleleft \Gamma'$ and $s(ty) = ty'$. □

Example 4.2. Consider the following partially labeled expression. (We have omitted all labels that are irrelevant for this example.)

$$\text{lexp} = (\text{fn } x^{l_1} \Rightarrow f(x^{l_2} \ 0)^{l_3} (x^{l_4} + 0)^{l_5})$$

Note that this expression has an obvious type error. The bound variable x is used both as a function and as an integer. Formally, it is the case that $(\text{lexp} \Downarrow \langle \text{empty}[f \mapsto a], a', C \rangle)$ for some type variables a, a' and some constraint set C that has the following subset C' .

$$C' = \left\{ a_1 \stackrel{l_2}{=} a_2, a_2 \stackrel{l_3}{=} a_3 \rightarrow a_4, a_5 \stackrel{l_4}{=} a_6, a_6 \stackrel{l_5}{=} \text{int}, a_7 \stackrel{l_1}{=} a_1, a_7 \stackrel{l_1}{=} a_5 \right\}$$

It is not hard to see that C' is unsolvable. Moreover, it is *minimally* unsolvable, i.e., every proper subset of C' is solvable. As a type error message, our implementation displays a program slice that contains all program points that are

associated with C' . When applied to the declaration

```
val _ = fn x => f (x 0) (x + 0)
```

it displays a message like this one:

```
type constructor clash, endpoints: function vs. int
(.. fn x => (.. x (..) .. x + (..) ..) ..) □
```

Unlike Damas's original algorithm, in our variation of algorithm T every expression's result type is a fresh type variable a equated to a type ty by a separate constraint. The additional constraints and type variables are vital for obtaining complete type error slices. For example, if the variable rule were replaced by

$$\frac{}{x^l \Downarrow \langle \text{empty}[x \mapsto \wedge\{a_x\}], a_x, \emptyset \rangle} \quad \text{where } a_x \text{ fresh}$$

then in example 4.2 the generated constraint set would not mention the labels l_2 or l_4 . Thus, these relevant program points would be wrongly omitted from the type error location. The resulting type error slice would be incomplete:

```
(.. fn x => (.. (..) (..) .. (..) + (..) ..) ..)
```

The let-expression rule copies the constraint set C_1 for $lexp_1$ for each use of the variable x in $lexp_2$. In bad cases, the number of copies of a constraint set can be exponential in the size of the program. Consider, this example program:

```
let val x1 = lexp in
let val x2 = f x1 x1 in
...
let val xn = f xn-1 xn-1 in f xn xn end ... end
```

The resulting constraint set contains 2^n variants of $lexp$'s constraint set. Note, however, that this family of expressions is notorious also for algorithm \mathcal{W} : If $lexp = (\text{fn } x \Rightarrow x)$ and f 's type scheme is assumed to be $(\forall a. \forall b. a \rightarrow b \rightarrow a \rightarrow b)$, then the principal type scheme of the entire expression contains $2^{(n+1)}$ distinct type variables. Remember also that Hindley/Milner (SML) typability in our small expression language is exponential time complete [16, 17].

5 Finding Minimal Unsolvable Constraint Sets

We define a function that maps sets of atomically labeled constraints to sets of associated labels by $\text{labels}(C) = \{ l \mid (\exists ty, ty')((ty \stackrel{l}{=} ty') \in C) \}$. A set of labels L is called an *error* with respect to C , if C has an unsolvable subset C' such that $L = \text{labels}(C')$. We denote the set of all such errors by $\text{errors}(C)$. Moreover, $\text{minErrors}(C)$ denotes the set of all those elements of $\text{errors}(C)$ that are minimal with respect to set inclusion. This section shows how to find minimal errors in an unsolvable constraint set. We will present a greedy minimization algorithm that, given an unsolvable constraint set C , finds a *single* element of $\text{minErrors}(C)$. This algorithm is reasonably efficient for practical purposes. It

is not practical to always exhaustively enumerate *all* elements of $\text{minErrors}(C)$, because this set has a worst-case size exponential in the size of C [27]. However, our simple enumeration algorithm seems to always find a few good candidates for some (but not all) minimal errors. These candidates are close to minimal and can be minimized with the minimization algorithm.

5.1 Labeled Unification. Unification can be viewed as a rewrite system on constraints. Our algorithms label each derived constraint with the labels of constraints used in deriving it. Our unification algorithm is similar to the one in [26]. Our *labeled unification algorithm* is a set of state transformation rules given in figure 4 which define the state transformation relation \rightarrow . Initial states are of the form $\text{unify}(C)$ and final states of the form $\text{Success}(E)$ or $\text{Error}(L, l)$. Intermediate states are of the form $\text{unify}(C, E)$ or $\text{unify}(C, E, D, l)$ where the state components are as follows:

| | |
|--|--|
| $C \in \text{AtConstraintSet}$ | initial constraints not yet considered |
| $E \in \text{TyVar} \rightarrow ((\text{Ty} \times \text{LabelSet}) \cup \{\perp\})$ | <i>environment</i> , contains derived bindings |
| $D \in \text{ConstraintSet}$ | <i>derived constraints</i> that are not bindings yet |
| $l \in \text{Label}$ | the label whose constraints are currently under inspection |

Proposition 5.1 (Termination of unify). *Each state transformation sequence terminates. A state is irreducible iff it is a final state.* \square

We define a function app that maps environments to partial functions from TyVar to Ty . For every fixed E , the binary relation $\text{app}(E)(\cdot) = \cdot$ is defined inductively as the least relation that satisfies the following conditions:

$$\begin{aligned}
(E(a) = \perp) &\Rightarrow (\text{app}(E)(a) = a) \\
(E(a) = \langle ty, L \rangle) &\Rightarrow (\text{app}(E)(a) = \text{app}(E)(ty)) \\
(\text{app}(E)(ty_1) = ty'_1) \wedge (\text{app}(E)(ty_2) = ty'_2) &\Rightarrow (\text{app}(E)(ty_1 \rightarrow ty_2) = ty'_1 \rightarrow ty'_2) \\
&(\text{app}(E)(\text{int}) = \text{int})
\end{aligned}$$

$\text{app}(E)$ is a partial function for every E . Although $\text{app}(E)$ is not always total, because the second equation (for variables) is not size decreasing, it is only used in defined cases. For type substitutions s and s' , their *composition* $s' \circ s$ is the type substitution that satisfies $(s' \circ s)(a) = s'(s(a))$ for all type variables a . A type substitution s is called a *most general unifier* of C , iff for every solution s' of C there exists a type substitution s'' such that $s' = s'' \circ s$.

Theorem 5.2 (Correctness of unify).

1. If $\text{unify}(C) \rightarrow^* \text{Success}(E)$, then $\text{app}(E)$ is a total function and a most general unifier of C .
2. If $\text{unify}(C) \rightarrow^* \text{Error}(L, l)$, then $L \in \text{errors}(C)$ and $L \setminus \{l\} \notin \text{errors}(C)$. \square

If one ignores the labels, the labeled unification algorithm looks very much like standard presentations of unification. Note that the transformation system in figure 4 is non-deterministic. Arbitrary choices can be used for the label l in

dummy is some arbitrarily chosen fixed label

$$\begin{aligned}
&\text{unify}(C) && \rightarrow \text{unify}(C, (\lambda a \in \text{TyVar.}\perp)) \\
&\text{unify}(C, E) && \rightarrow \text{unify}(C, E, \emptyset, \text{dummy}) \\
&\text{unify}(\emptyset, E, \emptyset, l) && \rightarrow \text{Success}(E) \\
&\text{unify}(C, E, \emptyset, l') && \rightarrow \text{unify}(C \setminus \Pi_l(C), E, \Pi_l(C), l), \\
&\quad \text{if } \Pi_l(C) \neq \emptyset \\
&\text{unify}(C, E, \{ty \stackrel{L}{=} ty\} \uplus D, l) && \rightarrow \text{unify}(C, E, D, l) \\
&\text{unify}(C, E, \{ty_1 \rightarrow ty_2 \stackrel{L}{=} \text{int}\} \uplus D, l) && \rightarrow \text{Error}(L, l) \\
&\text{unify}(C, E, \{\text{int} \stackrel{L}{=} ty_1 \rightarrow ty_2\} \uplus D, l) && \rightarrow \text{Error}(L, l) \\
&\text{unify}(C, E, \{\text{int} \stackrel{L}{=} a\} \uplus D, l) && \rightarrow \text{unify}(C, E, \{a \stackrel{L}{=} \text{int}\} \cup D, l) \\
&\text{unify}(C, E, \{ty_1 \rightarrow ty_2 \stackrel{L}{=} a\} \uplus D, l) && \rightarrow \text{unify}(C, E, \{a \stackrel{L}{=} ty_1 \rightarrow ty_2\} \cup D, l) \\
&\text{unify}(C, E, \{ty_1 \rightarrow ty_2 \stackrel{L}{=} ty'_1 \rightarrow ty'_2\} \uplus D, l) \\
&\quad \rightarrow \text{unify}(C, E, \{ty'_1 \stackrel{L}{=} ty_1, ty_2 \stackrel{L}{=} ty'_2\} \cup D, l) \\
&\text{unify}(C, E[a \mapsto \langle ty', L' \rangle], \{a \stackrel{L}{=} ty\} \uplus D, l) \\
&\quad \rightarrow \text{unify}(C, E[a \mapsto \langle ty', L' \rangle], \{ty' \stackrel{L \cup L'}{=} ty\} \cup D, l) \\
&\text{unify}(C, E[a \mapsto \perp], \{a \stackrel{L}{=} ty\} \uplus D, l) \\
&\quad \rightarrow \begin{cases} \text{unify}(C, E[a \mapsto \langle ty, L \rangle], D, l) & \text{if } \text{occurs}(E, L, a, ty, 0) = \emptyset \\ \text{Error}(L', l) & \text{if } \langle L', n \rangle \in \text{occurs}(E, L, a, ty, 0) \text{ and } n \geq 1 \\ \text{unify}(C, E[a \mapsto \perp], D, l) & \text{otherwise} \end{cases} \\
&\text{occurs}(E[a' \mapsto \langle ty, L' \rangle], L, a, a', i) && = \text{occurs}(E[a' \mapsto \langle ty, L' \rangle], L \cup L', a, ty, i) \\
&\text{occurs}(E[a' \mapsto \perp], L, a, a', i) && = \{\langle L, i \rangle\} \\
&\text{occurs}(E[a' \mapsto \perp], L, a, a', i) && = \emptyset \quad \text{if } a \neq a' \\
&\text{occurs}(E, L, a, \text{int}, i) && = \emptyset \\
&\text{occurs}(E, L, a, ty_1 \rightarrow ty_2, i) \\
&\quad = \text{occurs}(E, L, a, ty_1, i + 1) \cup \text{occurs}(E, L, a, ty_2, i + 1)
\end{aligned}$$

Fig. 4. A non-deterministic labeled unification algorithm

the fourth unify rule, the labeled constraint $(ty \stackrel{L}{=} ty')$ in the last ten unify rules, and the label set associated with an occurs-check failure in the last unify rule. Different choices may yield different final results. This is not a surprise, because the label sets that get returned in case of failure record parts of the histories of transformation sequences.

Example 5.3.

$$C = \{ a_1 \stackrel{l_1}{=} a_2 \rightarrow a_3, a_2 \stackrel{l_2}{=} \text{int} \rightarrow a_4, a_1 \stackrel{l_3}{=} (a_5 \rightarrow (a_6 \rightarrow a_7)) \rightarrow \text{int}, \\ a_2 \stackrel{l_4}{=} a_8 \rightarrow \text{int} \}$$

Both $\text{unify}(C) \rightarrow^* \text{Error}(\{l_1, l_2, l_3, l_4\}, l_4)$ and $\text{unify}(C) \rightarrow^* \text{Error}(\{l_1, l_3, l_4\}, l_4)$. The first result is obtained, for instance, if the constraints are inspected in the order l_1, l_2, l_3, l_4 ; the second result is obtained, for instance, if they are inspected in the order l_1, l_3, l_4 . Note that this example shows that $\text{unify}(C) \rightarrow^* \text{Error}(L, l)$ does *not* imply that L is minimal. \square

Example 5.4.

$$C = \{ a_1 \stackrel{l_1}{=} a_2 \rightarrow a_3, a_1 \stackrel{l_2}{=} (a_4 \rightarrow (a_5 \rightarrow a_6)) \rightarrow \text{int}, \\ a_1 \stackrel{l_3}{=} (a_7 \rightarrow (a_8 \rightarrow a_9)) \rightarrow \text{int}, a_2 \stackrel{l_4}{=} \text{int} \rightarrow \text{int} \}$$

Then, $\text{unify}(C) \rightarrow^* \text{Error}(\{l_1, l_2, l_4\}, l_4)$. The result is obtained, for instance, if the constraints are inspected in the order l_1, l_2, l_3, l_4 . Note that, although l_3 is inspected before the error is discovered, l_3 is not an element of the return set. This is so, because the constraint that is labeled by l_3 does not increment the knowledge that has already been accumulated as a result of inspecting l_1 and l_2 .

It is also the case that $\text{unify}(C) \rightarrow^* \text{Error}(\{l_1, l_3, l_4\}, l_4)$. This result is obtained, for instance, if the constraints are inspected in the order l_1, l_3, l_2, l_4 . It happens to be the case that $\text{minErrors}(C) = \{\{l_1, l_2, l_4\}, \{l_1, l_3, l_4\}\}$ \square

5.2 Error Minimization. Both our minimization and enumeration algorithms are based on the labeled unification algorithm; they execute it multiple times on different subsets of the initial constraint set. The minimization algorithm is based on the following idea: Remember that if $\text{unify}(C) \rightarrow^* \text{Error}(L, l)$, then L is an error and $L \setminus \{l\}$ is not an error. It follows that l is an element of every minimal error that is contained in L . The minimization algorithm exploits this fact repetitively, and iteratively builds a minimal error.

In figure 5, the algorithm is presented as a set of state transformation rules. Initial states are of the form $\text{minimize}(C, L, l)$ and final states of form $\text{MinError}(L)$. Intermediate states are of the form $\text{minimize}(C, E, L, l, L')$. The intention is that, if, initially, $L \in \text{errors}(C)$ and $L \setminus \{l\} \notin \text{errors}(C)$, and if $\text{minimize}(C, L, l) \rightarrow^* \text{MinError}(L')$, then L' is a minimal error that is contained in L .

Proposition 5.5. *Suppose $L_{in} \in \text{errors}(C_{in})$, $L_{in} \setminus \{l_{in}\} \notin \text{errors}(C_{in})$ and $\text{minimize}(C_{in}, L_{in}, l_{in}) \rightarrow^* \text{minimize}(C, E, L, l, L')$. Then all of these hold:*

1. $C = C_{in}$, $l \in L$, $l_{in} \in L'$, $L \cap L' = \emptyset$ and $L \cup L' \subseteq L_{in}$.
2. $\text{app}(E)$ is a most general unifier of $\Pi_{L'}(C)$.
3. $\text{app}(E)(\Pi_L(C))$ is not solvable.
4. $\text{app}(E)(\Pi_{L \setminus \{l\}}(C))$ is solvable. \square

Proposition 5.6 (Termination of minimize). *Let $L \in \text{errors}(C)$ and $L \setminus \{l\} \notin \text{errors}(C)$. Every transformation sequence starting from $\text{minimize}(C, L, l)$ terminates. If $\text{minimize}(C, L, l) \rightarrow^* s$, then s is irreducible iff it is a final state. \square*

Lemma 5.7. *Suppose $L_{in} \in \text{errors}(C)$, $L_{in} \setminus \{l_{in}\} \notin \text{errors}(C)$ and $\text{minimize}(C, L_{in}, l_{in}) \rightarrow^* \text{minimize}(C', E, L, l, L')$. Then:*

$$\forall L_0 \in \text{errors}(C). ((L_0 \subseteq L \cup L') \Rightarrow (L' \cup \{l\} \subseteq L_0)) \quad \square$$

Theorem 5.8 (Correctness of minimize). *If $L \in \text{errors}(C)$, $L \setminus \{l\} \notin \text{errors}(C)$ and $\text{minimize}(C, L, l) \rightarrow^* \text{MinError}(L')$, then $L' \in \text{minErrors}(C)$ and $L' \subseteq L$. \square*

$$\begin{array}{c}
\text{minimize}(C, L, l) \rightarrow \text{minimize}(C, \lambda a \in \text{TyVar}.\perp, L, l, \emptyset) \\
\\
\frac{\text{unify}(\Pi_l(C), E) \rightarrow^* \text{Error}(\cdot, \cdot)}{\text{minimize}(C, E, L, l, L') \rightarrow \text{MinError}(L' \cup \{l\})} \\
\\
\frac{\text{unify}(\Pi_l(C), E) \rightarrow^* \text{Success}(E_{new}); \\ \text{unify}(\Pi_{L \setminus \{l\}}(C), E_{new}) \rightarrow^* \text{Error}(L_{new}, l_{new})}{\text{minimize}(C, E, L, l, L') \rightarrow \text{minimize}(C, E_{new}, L_{new}, l_{new}, L' \cup \{l\})}
\end{array}$$

Fig. 5. A non-deterministic error slice minimization algorithm

The transformation sequence $\text{minimize}(C, L, l) \rightarrow^* \text{MinError}(L')$ requires at most $2n$ calls to the labeled unification algorithm, where n is the size of $\Pi_L(C)$. In the worst case, our labeled unification algorithm takes exponential time in the size of the constraint set, but linear time unification algorithms exist. Using a linear time unification algorithm, minimization would take quadratic time in the size of $\Pi_L(C)$. We apply the minimization algorithm only to label sets L returned by an initial run of labeled unification. Even for large input programs we expect these label sets, and also $\Pi_L(C)$, to be small.

5.3 Error Enumeration. Enumerating all minimal errors is harder than finding just one. In the worst case, the number of minimal errors is exponential in the size of the constraint set [27]. We use a simple algorithm that quickly finds a number of different errors that are close to minimal. In principle (but not in practice), this algorithm eventually returns the set of all minimal errors. However, we interrupt its execution after a short time. The interrupted algorithm returns an intermediate state that contains a list of candidates. These candidates are errors that are not guaranteed to be minimal yet. However, they are close to minimal and the minimization algorithm can be used to minimize them. Our algorithm has the property that it finds a few minimal errors fast, at the expense of behaving badly in the hypothetical limit case.⁴ We do not think that, in practice, it is a great disadvantage that our algorithms only find some, but not all, minimal error slices of a program at once. Many of today’s compilers report only a few type errors at a time. Even if they do report many type errors at once, most programmers correct only few of the reported errors before they try to recompile.

The (previously defined) function minErrors satisfies the following equations:

$$\begin{array}{l}
\text{If } \text{unify}(C) \rightarrow^* \text{Success}(\cdot): \quad \text{minErrors}(C) = \emptyset \\
\text{If } \text{unify}(C) \rightarrow^* \text{Error}(L, \cdot): \\
\quad \text{minErrors}(C) = \min(\bigcup \{ \text{minErrors}(\Pi_{\text{labels}(C) \setminus \{l\}}(C)) \mid l \in L \} \cup \{L\})
\end{array}$$

A recursive implementation of these equations rediscovers identical errors many times. For instance, if $\text{unify}(C) \rightarrow^* \text{Error}(L, \cdot)$ and L' is a minimal error of C

⁴ An example of an algorithm that “behaves well” in the hypothetical limit case, but may often not even find a single minimal error in reality because of time or space limits, is a breadth-first exploration of all possible transformation sequences of labeled unification.

$$\text{enum}(C) \rightarrow \text{enum}(C, \emptyset, \{\emptyset\}); \quad \text{enum}(C, \text{found}, \emptyset) \rightarrow \text{MinErrors}(\text{found})$$

$$\frac{\text{unify}(\Pi_{\text{labels}(C) \setminus L}(C)) \rightarrow^* \text{Success}(\cdot)}{\text{enum}(C, \text{found}, \{L\} \uplus \text{todo}) \rightarrow \text{enum}(C, \text{found}, \text{todo})}$$

$$\frac{\text{unify}(\Pi_{\text{labels}(C) \setminus L}(C)) \rightarrow^* \text{Error}(L', \cdot); \quad \text{insertError}(L', \text{found}) = \text{found}_1; \quad \text{insertTodos}(\text{distribute}(L', L), \text{todo}) = \text{todo}_1}{\text{enum}(C, \text{found}, \{L\} \uplus \text{todo}) \rightarrow \text{enum}(C, \text{found}_1, \text{todo}_1)}$$

$$\text{insertError}(L, \text{found}) \stackrel{\text{def}}{=} \begin{cases} \text{found}, & \text{if } (\exists L' \in \text{found})(L' \subseteq L) \\ \{ L' \in \text{found} \mid L \not\subseteq L' \} \cup \{L\}, & \text{otherwise} \end{cases}$$

$$\text{insertTodos}(Ls, \text{todo}) \stackrel{\text{def}}{=} \text{todo} \cup \{ L \in Ls \mid (\forall L' \in \text{todo})(L' \not\subseteq L) \}$$

$$\text{distribute}(L', L) \stackrel{\text{def}}{=} \{ \{l'\} \cup L \mid l' \in L' \}$$

Fig. 6. A non-deterministic minimal error slice enumeration algorithm

that is contained in $(\text{labels}(C) \setminus L)$, then L' gets returned by each one of the recursive calls. Our enumeration algorithm suffers from such recomputations. For that reason, the algorithm is impractical for exhaustively enumerating all minimal errors, even in cases where $\text{minErrors}(C)$ is small.

The algorithm in figure 6 is essentially an iterative version of the above recurrences presented as a set of state transformation rules. Initial states are of the form $\text{enum}(C)$ and final states of the form $\text{MinErrors}(Ls)$, where Ls is a set of pairwise incomparable label sets. Intermediate states are of the form $\text{enum}(C, \text{found}, \text{todo})$ where both found and todo are sets of pairwise incomparable label sets. At each state, the set found contains close approximations of some minimal errors of C (“candidate set”). Members of the set todo represent work items that still need to be done (“to-do set”). Specifically, for each label set L in the to-do set, the minimal errors that are contained in $(\text{labels}(C) \setminus L)$ still need to be found. We usually interrupt the execution of $\text{enum}(C)$ before it terminates but after it has found at least one error. In this case, the elements of the current found -set get minimized and then returned.

Proposition 5.9 (Termination of enum). *Each state transformation sequence terminates. A state is irreducible iff it is a final state.* \square

Theorem 5.10 (Correctness of enum). *If $\text{enum}(C) \rightarrow^* \text{MinErrors}(Ls)$, then $Ls = \text{minErrors}(C)$.* \square

6 Slicing the Program

Figure 7 defines the abstract syntax class of *slices*. The grammar extends the labeled expression grammar by the additional phrase

$$sl ::= \dots \mid \text{dots}(sl_1, \dots, sl_k) \mid \dots$$

A `dots`-node in a slice’s abstract syntax tree represents an irrelevant segment of the corresponding program’s abstract syntax tree. Our experimental implementation displays `dots(sl1, sl2, sl3)` as:

$$\begin{aligned}
k &\in \{0, 1, 2, \dots\} \\
vsl &\in \text{VarSlice} \quad ::= x^l \mid \text{dots}() \\
sl &\in \text{Slice} \quad ::= x^l \mid n^l \mid (sl + sl)^l \mid (\text{fn } vsl \Rightarrow sl)^l \mid \\
&\quad (sl \ sl)^l \mid (\text{let val } vsl = sl \text{ in } sl \text{ end})^l \mid \text{dots}(sl_1, \dots, sl_k)
\end{aligned}$$

Typing rules

$$\begin{aligned}
(\forall i \in \{1, \dots, k\}. \Gamma \vdash sl_i : ty_i) &\Rightarrow (\Gamma \vdash \text{dots}(sl_1, \dots, sl_k) : ty) \\
(\Gamma \vdash sl : ty') &\Rightarrow (\Gamma \vdash (\text{fn dots}() \Rightarrow sl)^l : ty \rightarrow ty') \\
(\Gamma \vdash sl' : ty') \text{ and } (\Gamma \vdash sl : ty) &\Rightarrow (\Gamma \vdash (\text{let val dots}() = sl' \text{ in } sl \text{ end})^l : ty)
\end{aligned}$$

Algorithm T

$$\begin{aligned}
&\frac{sl_i \Downarrow \langle \Gamma_i, ty_i, C_i \rangle \text{ for } i \text{ in } \{1, \dots, k\}; \quad a \text{ fresh}}{\text{dots}(sl_1, \dots, sl_k) \Downarrow \langle \Gamma_1 \wedge \dots \wedge \Gamma_k, a, C_1 \cup \dots \cup C_k \rangle} \\
&\frac{sl \Downarrow \langle \Gamma, ty, C \rangle; \quad a, a' \text{ fresh}}{(\text{fn dots}() \Rightarrow sl)^l \Downarrow \langle \Gamma, a, \{a' \rightarrow ty \stackrel{l}{=} a\} \cup C \rangle} \\
&\frac{sl_1 \Downarrow \langle \Gamma_1, ty_1, C_1 \rangle; \quad sl_2 \Downarrow \langle \Gamma_2, ty_2, C_2 \rangle; \quad a \text{ fresh}}{(\text{let val dots}() = sl_1 \text{ in } sl_2 \text{ end})^l \Downarrow \langle \Gamma_1 \wedge \Gamma_2, a, \{a \stackrel{l}{=} ty_2\} \cup C_1 \cup C_2 \rangle}
\end{aligned}$$

Fig. 7. Additional rules for slices

$$(\dots sl_1 \dots sl_2 \dots sl_3 \dots)$$

For instance, the type error slice

$$\text{fn } x^{l_1} \Rightarrow \text{dots}((x^{l_2} \text{ dots}())^{l_3}, (x^{l_4} + \text{dots}())^{l_5})$$

computed for the erroneous program from example 4.2 is displayed as:

$$\text{fn } x \Rightarrow (\dots x (\dots) \dots x + (\dots) \dots)$$

Figure 7 defines additional typing rules for slices. A slice of the form $\text{dots}(sl_1, \dots, sl_k)$ is well-typed iff sl_1 through sl_k are. In this case, it has all types. The typing rules for other phrases are omitted, because they are the same as for expressions (see figure 2). Figure 7 also extends algorithm T. We need this extension, in order to formulate a statement that relates erroneous programs to their type error slices. The rule for dots -phrases does not generate any additional constraints. It merely propagates recursively computed results. The rules for other phrases are omitted, because they are exactly as in figure 3.

Figure 8 defines the function `slice` which takes a label set L and a labeled expression $lexp$ and returns a slice. This function replaces each node of $lexp$'s syntax tree by dots , if its node label is not in L . It also flattens nested dots . As a result of flattening, $\text{slice}(L, lexp)$ does not have immediately nested dots .

Theorem 6.1 (Faithfulness). *If $(lexp \Downarrow \langle \cdot, \cdot, C \rangle)$, $L \in \text{errors}(C)$ and $(\text{slice}(L, lexp) \Downarrow \langle \cdot, \cdot, C' \rangle)$, then $L \in \text{errors}(C')$.* \square

$$\begin{array}{c}
\frac{\text{lexp} \downarrow^L \text{sl}}{\text{slice}(L, \text{lexp}) = \text{sl}} \quad \frac{l \in L}{x^l \downarrow^L x^l} \quad \frac{l \notin L}{x^l \downarrow^L \text{dots}()} \quad \frac{l \in L}{n^l \downarrow^L n^l} \quad \frac{l \notin L}{n^l \downarrow^L \text{dots}()} \\
\frac{\text{lexp}_1 \downarrow^L \text{sl}_1; \text{lexp}_2 \downarrow^L \text{sl}_2; l \in L}{(\text{lexp}_1 + \text{lexp}_2)^l \downarrow^L (\text{sl}_1 + \text{sl}_2)^l} \quad \frac{\text{lexp}_1 \downarrow^L \text{sl}_1; \text{lexp}_2 \downarrow^L \text{sl}_2; l \notin L}{(\text{lexp}_1 + \text{lexp}_2)^l \downarrow^L \text{merge}(\langle \text{sl}_1, \text{sl}_2 \rangle)} \\
\frac{x^{l_1} \downarrow^L \text{vsl}; \text{lexp} \downarrow^L \text{sl}; l_1 \in L \text{ or } l_2 \in L}{(\text{fn } x^{l_1} \Rightarrow \text{lexp})^{l_2} \downarrow^L (\text{fn } \text{vsl} \Rightarrow \text{sl})^{l_2}} \quad \frac{\text{lexp} \downarrow^L \text{sl}; l_1 \notin L \text{ and } l_2 \notin L}{(\text{fn } x^{l_1} \Rightarrow \text{lexp})^{l_2} \downarrow^L \text{merge}(\langle \text{sl} \rangle)} \\
\frac{\text{lexp}_1 \downarrow^L \text{sl}_1; \text{lexp}_2 \downarrow^L \text{sl}_2; l \in L}{(\text{lexp}_1 \text{lexp}_2)^l \downarrow^L (\text{sl}_1 \text{sl}_2)^l} \quad \frac{\text{lexp}_1 \downarrow^L \text{sl}_1; \text{lexp}_2 \downarrow^L \text{sl}_2; l \notin L}{(\text{lexp}_1 \text{lexp}_2)^l \downarrow^L \text{merge}(\langle \text{sl}_1, \text{sl}_2 \rangle)} \\
\frac{x^{l_1} \downarrow^L \text{vsl}; \text{lexp}_1 \downarrow^L \text{sl}_1; \text{lexp}_2 \downarrow^L \text{sl}_2; l_1 \in L \text{ or } l_2 \in L}{(\text{let val } x^{l_1} = \text{lexp}_1 \text{ in } \text{lexp}_2 \text{ end})^{l_2} \downarrow^L (\text{let val } \text{vsl} = \text{sl}_1 \text{ in } \text{sl}_2 \text{ end})^{l_2}} \\
\frac{\text{lexp}_1 \downarrow^L \text{sl}_1; \text{lexp}_2 \downarrow^L \text{sl}_2; l_1 \notin L \text{ and } l_2 \notin L}{(\text{let val } x^{l_1} = \text{lexp}_1 \text{ in } \text{lexp}_2 \text{ end})^{l_2} \downarrow^L \text{merge}(\langle \text{sl}_1, \text{sl}_2 \rangle)} \\
\text{.....} \\
\frac{}{\text{merge}(\langle \rangle) = \text{dots}()} \quad \frac{\text{merge}(\text{sls}) = \text{dots}(\text{sl}_1, \dots, \text{sl}_n); \text{sl} \neq \text{dots}(\cdot)}{\text{merge}(\text{sl} :: \text{sls}) = \text{dots}(\text{sl}, \text{sl}_1, \dots, \text{sl}_n)} \\
\frac{\text{merge}(\text{sls}) = \text{dots}(\text{sl}'_1, \dots, \text{sl}'_k)}{\text{merge}(\text{dots}(\text{sl}_1, \dots, \text{sl}_n) :: \text{sls}) = \text{dots}(\text{sl}_1, \dots, \text{sl}_n, \text{sl}'_1, \dots, \text{sl}'_k)}
\end{array}$$

Fig. 8. Slicing

Let \sqsubset be the least contextually closed and transitive relation on slices satisfying the following:

$$\left. \begin{array}{l}
\text{dots}() \sqsubset x^l; \quad \text{dots}(\text{sl}) \sqsubset (\text{fn } \text{dots}() \Rightarrow \text{sl})^l \\
\text{dots}() \sqsubset n^l; \quad \text{dots}(\text{sl}_1, \text{sl}_2) \sqsubset (\text{sl}_1 \text{sl}_2)^l \\
\text{dots}(\text{sl}_1, \text{sl}_2) \sqsubset (\text{sl}_1 + \text{sl}_2)^l; \quad \text{dots}(\text{sl}_1, \text{sl}_2) \sqsubset (\text{let val } \text{dots}() = \text{sl}_1 \text{ in } \text{sl}_2 \text{ end})^l \\
\left. \begin{array}{l}
(\text{sl} = \text{dots}(\text{sl}_1, \dots, \text{sl}_i, \dots, \text{sl}_n)) \\
\text{and } (\text{sl}_i = \text{dots}(\text{sl}'_1, \dots, \text{sl}'_k))
\end{array} \right\} \Rightarrow \left\{ \begin{array}{l}
\text{dots}(\text{sl}_1, \dots, \text{sl}_{i-1}, \\
\text{sl}'_1, \dots, \text{sl}'_k, \text{sl}_{i+1}, \dots, \text{sl}_n) \sqsubset \text{sl}
\end{array} \right.
\end{array}$$

Theorem 6.2 (Accuracy). *If $(\text{lexp} \Downarrow \langle \cdot, \cdot, C \rangle)$, $L \in \text{minErrors}(C)$ and $\text{sl} \sqsubset \text{slice}(L, \text{lexp})$, then sl is well-typed.* \square

7 Conclusion

We have presented algorithms for type error slicing in an implicitly typed λ -calculus with let-polymorphism. These algorithms first generate type equality constraints using a version of Damas's type inference algorithm \mathbb{T} , and then find minimal unsolvable subsets of the set of generated constraints. Type error slices are programs where irrelevant program points are masked.

In the future, we want to extend our implementation of type error slicing to full SML and improve its user interface. The user interface will both highlight program points in the source code and display separate type error slices. The separate slices will be especially useful, if relevant program points are far apart, possibly in multiple files. Hyperlinks will relate program points in the separate slice to the corresponding points in the source. The extension to full

SML will require the treatment of additional issues. For instance, the presence of equality types and overloaded built-in operations requires an additional sort of constraints: kind constraints for type variables. Another important issue are explicit type and signature annotations. These will put natural boundaries on type error slices. For instance, if library modules are always annotated with explicit signatures, then type error slices for programs that use the library will never contain parts of the library implementation.

References

- [1] M. Beaven, R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2, 1993.
- [2] K. L. Bernstein, E. W. Stark. Debugging type errors (full version). Technical report, State University of New York, Stony Brook, 1995.
- [3] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proc. 6th Int'l Conf. Functional Programming*. ACM Press, 2001.
- [4] V. Chopella. *Unification source-tracking with application to diagnosis of type inference*. PhD thesis, Indiana University, 2002.
- [5] V. Chopella, C. T. Haynes. Diagnosis of ill-typed programs. Technical Report 426, Indiana University, 1995.
- [6] L. Damas, R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, 1982.
- [7] L. M. M. Damas. *Type assignment in Programming Languages*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1985.
- [8] T. B. Dinesh, F. Tip. A slicing-based approach for locating type errors. In *Proceedings of the USENIX conference on Domain-Specific Languages*, Santa Barbara, California, 1997.
- [9] D. Duggan, F. Bent. Explaining type inference. *Sci. Comput. Programming*, 27, 1996.
- [10] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, M. Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design & Impl.*, 1996.
- [11] B. Heeren, J. Hage. Parametric type inferencing for helium. Technical Report UU-CS-2002-035, University Utrecht, 2002.
- [12] B. Heeren, J. Hage, D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, University Utrecht, 2002.
- [13] B. Heeren, J. Jeuring, D. Swierstra, P. A. Alcocer. Improving type-error messages in functional languages. Technical Report UU-CS-2002-009, University Utrecht, 2002.
- [14] T. Jim. What are principal typings and what are they good for? In *POPL '96* [22].
- [15] G. F. Johnson, J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *POPL '96* [22].
- [16] P. Kanellakis, H. Mairson, J. C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez, G. Plotkin, eds., *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.
- [17] A. J. Kfoury, J. Tiuryn, P. Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2), 1994.
- [18] O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. on Prog. Langs. & Sys.*, 20(4), 1998.
- [19] B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, A. J. T. Davie, C. Clack, eds., *Implementation of Functional Languages (IFL'98)*, vol. 1595 of *LNCS*, London, UK, 1998. Springer-Verlag.
- [20] B. J. McAdam. Generalising techniques for type debugging. In Trinder et al. [25].
- [21] R. Milner, M. Tofte, R. Harper, D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [22] *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.
- [23] G. S. Port. A simple approach to finding the cause of non-unifiability. In *Proc. Fifth International Conference on Logic Programming*. MIT Press, 1988.
- [24] Z. Shao, A. Appel. Smartest recompilation. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, 1993.
- [25] P. Trinder, G. Michaelson, H.-W. Loidl, eds. *Trends in Func. Programming*. Intellect, 2000.
- [26] M. Wand. Finding the source of type errors. In *Conf. Rec. 13th Ann. ACM Symp. Princ. of Prog. Langs.*, 1986.
- [27] D. A. Wolfram. Intractable unifiability problems and backtracking. In *Proc. Third International Conference on Logic Programming*, vol. 225 of *LNCS*, 1986.
- [28] J. YANG. Explaining type errors by finding the source of a type conflict. In Trinder et al. [25].
- [29] J. YANG, G. Michaelson, P. Trinder. Explaining polymorphic types. *Computer Journal*, 200X. to appear.
- [30] J. YANG, G. Michaelson, P. Trinder, J. B. Wells. Improved type error reporting. In *[Draft] Proc. 12th Int'l Workshop Implementation Functional Languages*, Aachen, Germany, 2000.