Type Error Slicing in Implicitly Typed Higher-Order Languages ⋆

Christian Haack

DePaul University, Chicago, USA

J. B. Wells

Heriot-Watt University, Edinburgh, UK

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 identify the criteria of *completeness* and *minimality* for type error slices. We discuss the advantages of complete and minimal type error slices over previous methods of presenting type errors. We present and prove the correctness of algorithms for finding complete and minimal type error slices for implicitly typed higher-order languages like Standard ML.

Key words: type error location, type inference, intersection types

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, OCaml, Standard ML (SML), etc.) [29,22,21,32,31,3,5,11,2,17,10,23,33]. This is harder than in monomorphic, explicitly typed, first-order languages. None

⋆ 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, and Sun Microsystems equipment grant EDUD-7826-990410-US.

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.

When the $\mathcal{W}$ [8], $\mathcal{M}$ [21], or the $\mathcal{UAE}$ [31,32] 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. In addition to the confusion caused by blaming just one program node, user interfaces using the results of these algorithms typically print the entire program subtree under the node at which inference failed, so programmers may believe the entire program subtree is being blamed rather than the root of the subtree. [1]

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 the following instead, making the error of typing `y` instead of `x` at the highlighted spot:

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

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
```

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

[1] Adding to the confusion, some user interfaces will, somewhat arbitrarily, identify a node a bit higher in the program tree. For example, the SML/NJ compiler does this in the numbers it emits for use in source code highlighting, because it does not maintain source code location information for every node in the abstract syntax tree that it manipulates internally. This appears to be because the human programmer writes in a syntax containing "derived forms" and SML/NJ internally translates this into the "bare language" before running type checking.

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 $\mathcal{M}$, in some sense this might not be wrong, because the intention may be that only the root of the subtree is being blamed rather that all of the nodes in the subtree, but the programmer may not always understand this distinction.

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. To find these program points, the programmer must reconstruct the state of the type inference algorithm at the time it failed, and then run the type inference algorithm *backward*. The programmer must understand the type inference process and be able to run it in their mind. Obviously, this can be mentally taxing, so it would be a good idea to do this for the programmer and save them the effort.

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 incomplete 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 for the example: [2]

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

Formally, a *type error slice* is a set of program points. It is a *complete* representation of a type error if these program points and the relationships between the program points together guarantee that the program will have a type error. It is a *minimal* representation if none of 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 itself does not form a type error. As an example of an non-minimal 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 viewed as identifying a program *subtree* rather than merely a node in the program tree (a view encouraged by the way the location is typically presented to the user), then they will usually be non-minimal.

1.3 Related Work.

Dinesh and Tip have applied slicing techniques for locating sources of type errors [10]. 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 depending on significant amounts of type inference.

Our work is based on what Damas called his "type inference system" [9]. Damas did not name this type system, so we call it Damas's System T because it is used with Damas's algorithm T. This system has the same set of typable expressions as the more widely known Hindley/Milner system, but instead

[2] The output does not match what would be expected from the formalism presented later in this paper, because our implementation is for a slightly richer language that is closer to SML. 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.

of using $\forall$-quantified types, allows multiple types in the type environments for each free variable. This can be seen as using intersection types for free variables and Damas's System T can be seen as a restriction of a system of rank-2 intersection types. Jim [16] has proposed using rank-2 intersection types for accurate type error location. Bernstein and Stark [3] use Damas's System T for type error debugging of open terms.

Wand has presented an algorithm for finding the source of type errors in implicitly typed languages [29]. Similar methods have been used by Duggan and Bent [11]. 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 related but more carefully designed method as a subroutine. 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 [17].

Choppella and Haynes study type error diagnosis in a simply typed language [7,6]. Unlike our work, they do not actually treat let-polymorphism.[3] They propose to present type error locations as program slices, but have no notation for slices. They present a graph-based unification framework, based on work by Port [26], which could be used for finding minimal unsolvable constraint sets. However, the diagnostic unification algorithm that is actually presented in [6] 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. An advantage of our approach is simplicity of presentation and implementation. Unlike Choppella and Haynes, we give a detailed presentation of an algorithm that enumerates minimal unsolvable constraint sets. Our algorithm quickly enumerates *some* minimal unsolvable subsets of a given constraint set and is then cut off by a time limit. Our algorithm is too expensive in practice for exhaustively enumerating *all* such sets; solving this for practical cases will be difficult because the worst-case time complexity for enumerating all such sets is intractable [30]. In some cases an algorithm based on Port's idea may find in a feasible time all minimal unsolvable subsets for cases that arise in practice, whereas ours does not. In the future, we may adopt an algorithm influenced by the one sketched by Port.

Heeren and others propose constraint-based type inference for improved type

[3] Recent unpublished work by Choppella (mentioned by Choppella in verbal communication) treats let-polymorphism using an approach that alternates between generating and solving constraints.

error messages [15,14,13]. They treat let-polymorphism, and their type system has features both in the style of the Hindley/Milner system and Damas's System T. 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 they could simplify their system and sometimes permit more accurate error messages by removing the Hindley/Milner-style features from their type system. 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 [12]. The debugger is based on set-based flow analysis. It constructs and, on demand, displays parts of flow graphs, and highlights critical program points at which runtime errors may occur.

Much of the related work on type error analysis has been on sophisticated ways for automatically generating type error *explanations* [5,11,23,29,33,2,22]. 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 anyway. Our work is intended as a step in this direction.

1.4 Outline of Paper.

Section 2 informally discusses two larger examples. The remainder of the paper is technical. Section 3 introduces some terminology. Section 4 gives an overview of Damas's System T. The methods for type error slicing proceed in three steps. The first step consists of assigning constraints to program points. This is described in section 5. The second step consist of finding minimal unsolvable subsets in the set of all constraints. Section 6 describes algorithms for doing this. It also contains an example that in the worst case the number of minimal type error slices grows exponentially in the size of the program, which gives support to our choice to only enumerate some of the error slices in a program. Finally, section 7 describes how type error slices are computed from the results obtained in the previous steps, and states a completeness and a minimality theorem. These theorems are proved in appendix A. The completeness proof is less straightforward than one might expect, because constraints that are associated with variable binders may get lost as a result of slicing.

```
val average = fn weight => fn list =>
  let val iterator = fn (x,(sum,length)) =>
                        (sum + weight x, length + 1)
      val (sum,length) = foldl iterator (0,0) list
  in sum div length end

val find_best = fn weight => fn lists =>
  let val average = average weight
      val iterator = fn (list,(best,max)) =>
                        let val avg_list = average list
                        in if avg_list > max then
                             (list,avg_list)
                           else
                             (best,max)
                        end
      val (best,_) = foldl iterator (nil,0) lists
  in best end

val find_best_simple = find_best 1
```

Fig. 1. An SML program with type error

1.5 Acknowledgments

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

2 Examples to Illustrate the Important Concepts

This section uses example erroneous SML programs together with the output from our prototype type error slicing implementation to further explain important concepts.

2.1 Complete and Minimal Error Regions and Slices

Consider the (erroneous) SML program in figure 1. It defines the three functions `average`, `find_best` and `find_best_simple`. The function `average` takes a weight and a list, scales each list element by the weight and then computes the average over the scaled list elements. The function `find_best` uses the `average`-function to find the list with the highest average in a list of lists. Finally, the function `find_best_simple` specializes the function `find_best` by applying it to the identity weight. Scaling a list element by the identity weight leaves the element fixed. Thus, `find_best_simple` simply finds the list with the highest average in a list of lists. However, this program has a type error. A

traditional compiler that uses algorithm $\mathcal{W}$ would identify the following error region: [4]

```
val average = fn weight => fn list =>
  let val iterator = fn (x,(sum,length)) =>
                       (sum + weight x, length + 1)
      val (sum,length) = foldl iterator (0,0) list
  in sum div length end

val find_best = fn weight => fn lists =>
  let val average = average weight
      val iterator = fn (list,(best,max)) =>
                       let val avg_list = average list
                       in if avg_list > max then
                            (list,avg_list)
                          else
                            (best,max)
                       end
      val (best,_) = foldl iterator (nil,0) lists
  in best end

val find_best_simple = find_best 1
```

This region is an incomplete representation of the actual type error, i.e., the error cannot be explained by pointing to this region without referring to the context. As a result, the error may have to be fixed somewhere outside the highlighted region; the actual fix may leave the highlighted region unchanged.

The trouble with the program is that there is confusion whether the weight is represented as an integer or as a function. In the body of `average`, the parameter `weight` is applied to variable `x` and, thus, used as a function. On the other hand, in the last line `find_best` is applied to `1`; an integer is passed to its `weight`-parameter. Our prototype implementation highlights the following error region:

```
val average = fn weight => fn list =>
  let val iterator = fn (x,(sum,length)) =>
                       (sum + weight x, length + 1)
      val (sum,length) = foldl iterator (0,0) list
  in sum div length end

val find_best = fn weight => fn lists =>
  let val average = average weight
      val iterator = fn (list,(best,max)) =>
                       let val avg_list = average list
                       in if avg_list > max then
                            (list,avg_list)
                          else
                            (best,max)
                       end
      val (best,_) = foldl iterator (nil,0) lists
  in best end

val find_best_simple = find_best 1
```

[4] Algorithms $\mathcal{M}$ would identify only the `1` in the last line as the error region. Algorithm $\mathcal{UAE}$ would identify the same location as algorithm $\mathcal{W}$.

Technically, the type error is a type constructor clash between a function type constructor and the integer type constructor. The *endpoints* of this type constructor clash are highlighted in a darker color. Our prototype also displays an alternative representation of the type error location as a *program slice*, where all irrelevant program points are omitted (sliced away):

```
type constructor clash, endpoints: function vs. int

(.. val average = fn weight =>
      (.. weight   (..) ..)

 .. val find_best = fn weight =>
      (.. average weight ..)

 .. find_best 1 ..)
```

The type error can be completely explained just by looking at the program slice. The programmer can easily read the following explanation directly from the slice:

> The `weight` parameter of `average` is a function, because it is applied to some argument. The `weight` parameter of `find_best` must also be a function, because it is passed to `average`. But, in the last line of the slice, `find_best` is applied to the integer `1`, which is not a function.

Because this type error slice permits an independent explanation of the type error without needing to refer to any other part of the program, we call it a *complete* error representation. The slice is also a *minimal* error representation because omitting additional program points would break the explanation.

2.2 Fix Location Depends on Semantics

If it is a goal that compilers report error regions that always include the location that must be fixed (the *fix location*), then compilers should always report complete error regions. Omitting program points from a complete region may result in omitting the fix location. The fix location depends on the intended semantics of a program, i.e., on what the programmer has in mind when designing the program. Clearly, a compiler cannot read programmers' minds. Therefore, identifying complete error regions is the best a compiler can do. To illustrate this point, let us consider possible fix locations in the example. One possibility is that the programmer intended the weight to be an integer, not a function, and, in the body of `average`, forgot a multiplication sign. The fixed

slice would look like this: [5]

```
(.. val average = fn weight =>
      (.. weight * (..) ..)

 .. val find_best = fn weight =>
      (.. average weight ..)

 .. find_best 1 ..)
```

We have highlighted the inserted multiplication sign. Another possibility is that the programmer intended the weight to be a function and forgot about it in the last line. In that case, the fix would replace the integer `1` in the last line by the identity function:

```
(.. val average = fn weight =>
      (.. weight (..) ..)

 .. val find_best = fn weight =>
      (.. average weight ..)

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

Finally, it is possible that the programmer intended the `weight` parameter for `average` to be a function, but the `weight` parameter for `find_best` to be an integer. This gives rise to another possible fix location:

```
(.. val average = fn weight =>
      (.. weight (..) ..)

 .. val find_best = fn weight =>
      (.. average (fn x => weight * x) ..)

 .. find_best 1 ..)
```

2.3 Overlapping Error Regions

It is often the case that several complete error regions overlap. A single fix in the overlapping region may fix all of the error regions at once. As an example, consider the (erroneous) SML program in figure 2. In this example, it is likely that in line 4 the programmer has forgotten to turn the element `x` into a one-element list. Thus, in line 4, `list @ x` should be replaced by `list @ [x]`. [6] However, a traditional compiler that uses algorithm $\mathcal{W}$ identifies the following error region, which does not contain this likely fix location, and the region

[5] For space reasons, we argue in terms of the slice instead of the complete program.
[6] In SML, `@` is an infix operator that appends two lists.

```
val mapActL = fn iterator => fn (list,state) =>
  let val iterator' = fn (x,(list,state)) =>
                        let val (x,state) = iterator (x,state)
                        in (list @ x, state) end
  in foldl iterator' (nil,state) list end

val isEven = fn n => n mod 2 = 0

val doubleOdds = fn list =>
  let val iterator = fn (n,inc) => if isEven n then
                                     (n, inc)
                                   else
                                     (2 * n, inc + n)
  in mapActL iterator (list,0) end
```

Fig. 2. Another SML program with type error

identified by algorithm $\mathcal{M}$ is contained within the identified by $\mathcal{W}$:

```
val mapActL = fn iterator => fn (list,state) =>
  let val iterator' = fn (x,(list,state)) =>
                        let val (x,state) = iterator (x,state)
                        in (list @ x, state) end
  in foldl iterator' (nil,state) list end

val isEven = fn n => n mod 2 = 0

val doubleOdds = fn list =>
  let val iterator = fn (n,inc) => if isEven n then
                                     (n, inc)
                                   else
                                     (2 * n, inc + n)
  in mapActL iterator (list,0) end
```

In contrast, figure 3 shows two error regions produced by our prototype implementation. The likely fix location is contained in both of the regions. Here is a display of both regions in a single picture with the overlapping region highlighted darker:

```
val mapActL = fn iterator => fn (list,state) =>
  let val iterator' = fn (x,(list,state)) =>
                        let val (x,state) = iterator (x,state)
                        in (list @ x, state) end
  in foldl iterator' (nil,state) list end

val isEven = fn n => n mod 2 = 0

val doubleOdds = fn list =>
  let val iterator = fn (n,inc) => if isEven n then
                                     (n, inc)
                                   else
                                     (2 * n, inc + n)
  in mapActL iterator (list,0) end
```

```
val mapActL = fn iterator => fn (list,state) =>
  let val iterator' = fn (x,(list,state)) =>
                        let val (x,state) = iterator (x,state)
                        in (list @ x, state) end
  in foldl iterator' (nil,state) list end

val isEven = fn n => n mod 2 = 0

val doubleOdds = fn list =>
  let val iterator = fn (n,inc) => if isEven n then
                                     (n, inc)
                                   else
                                     (2 * n, inc + n)
  in mapActL iterator (list,0) end
```

.....

```
val mapActL = fn iterator => fn (list,state) =>
  let val iterator' = fn (x,(list,state)) =>
                        let val (x,state) = iterator (x,state)
                        in (list @ x, state) end
  in foldl iterator' (nil,state) list end

val isEven = fn n => n mod 2 = 0

val doubleOdds = fn list =>
  let val iterator = fn (n,inc) => if isEven n then
                                     (n, inc)
                                   else
                                     (2 * n, inc + n)
  in mapActL iterator (list,0) end
```

Fig. 3. Two overlapping error regions

Actually, there are more than just two error regions in this example; there are four complete error regions altogether. The reader is invited to find the other two regions using our web demonstration tool [4]. The likely fix location in line 4 is contained in all of these regions and the fix in line 4 fixes all regions at once. Informing the programmer of overlapping regions often helps to find the fix location.

However, do not jump to the conclusion that the correct fix will always be in the overlap. There is at least one common case where this does not hold: when the programmer changed a data representation and failed to fix all of the locations creating or using the data representation.

$$
\begin{array}{rcll}
l & \in & \mathsf{Label} & \text{a fixed infinite set of labels} \\
L & \in & \mathsf{LabelSet} & \text{all finite subsets of } \mathsf{Label} \\
x & \in & \mathsf{Var} & \text{a fixed infinite set of variables} \\
n & \in & \mathsf{Int} & \text{the set of integers} \\
\mathit{lexp} & \in & \mathsf{LExp} & ::= \; x^l \mid n^l \mid (\mathit{lexp}\ \texttt{+}\ \mathit{lexp})^l \mid (\texttt{fn}\ x^l\ \texttt{=>}\ \mathit{lexp})^l \\
 & & & \quad \mid (\mathit{lexp}\ \mathit{lexp})^l \mid (\texttt{let val}\ x^l\ \texttt{=}\ \mathit{lexp}\ \texttt{in}\ \mathit{lexp}\ \texttt{end})^l
\end{array}
$$

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

Fig. 4. Labeled expressions

3 Some Definitions and Notations

This section defines some basic mathematical notions and notations. For each natural number i, the symbol π_i denotes the i-th projection operator, i.e., if $xs = \langle x_1, \ldots, x_n \rangle$ and $i \in \{1, \ldots, n\}$, then $\pi_i(xs) = x_i$. If f is a function, then $f[x \mapsto y]$ denotes the function $(f \setminus \{\langle x, f(x)\rangle\}) \cup \{\langle x, y\rangle\}$. If X is a set and $\rightarrow$ is a subset of $X \times X$, then $\rightarrow^*$ denotes its reflexive (w.r.t. X) and transitive closure. An element x is called *irreducible* with respect to $\rightarrow$ iff there is no element y such that $x \rightarrow y$. If X is a set of sets, then $\mathsf{min}(X)$ denotes the set of all elements of X that are minimal with respect to set inclusion. Two sets are called *incomparable* iff 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.

4 Damas's Type Inference System

For concreteness, we describe our methodology in detail for the small model language shown in figure 4. The labels that superscript expressions mark program points. The labeled expression language is a sublanguage of Standard ML (SML) [24]. We have an implementation for a larger sublanguage of SML [4].

$$\Gamma[x \mapsto \wedge\{ty, \ldots\}] \vdash x^l : ty$$

$$\Gamma \vdash n : \texttt{int}$$

$$(\Gamma \vdash lexp_1 : \texttt{int}) \text{ and } (\Gamma \vdash lexp_2 : \texttt{int}) \Rightarrow \Gamma \vdash (lexp_1 \texttt{ + } lexp_2)^l : \texttt{int}$$

$$\Gamma[x \mapsto \wedge\{ty\}] \vdash lexp : ty' \Rightarrow \Gamma \vdash (\texttt{fn } x^l \texttt{ => } lexp)^{l'} : ty \texttt{ -> } ty'$$

$$(\Gamma \vdash lexp_1 : ty' \texttt{ -> } ty) \text{ and } (\Gamma \vdash lexp_2 : ty') \Rightarrow \Gamma \vdash (lexp_1\ lexp_2)^l : ty$$

$$(S \neq \emptyset) \text{ and } (\forall ty \in S.\ \Gamma \vdash lexp : ty) \text{ and } (\Gamma[x \mapsto \wedge S] \vdash lexp' : ty')$$
$$\Rightarrow \Gamma \vdash (\texttt{let val } x^l \texttt{ = } lexp \texttt{ in } lexp' \texttt{ end})^{l'} : ty'$$

Fig. 5. Damas's typing rules

Types are defined as follows:

$$ty \in \mathsf{Ty} ::= a \mid \texttt{int} \mid ty \texttt{ -> } ty \qquad ity \in \mathsf{IntTy} ::= \wedge S$$

$$a \in \mathsf{TyVar} \quad \text{a fixed infinite set of type variables}$$

$$S \in \mathsf{TySet} \quad \text{the set of all finite subsets of } \mathsf{Ty}$$

The elements of IntTy are called *intersection types*. The symbol $\wedge$ is syntax. For example, $\wedge\{a \texttt{ -> int}, \texttt{int -> } a\} \in \mathsf{IntTy}$. 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 5. 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 Hindley/Milner system, which Damas called the "type *scheme* inference system". Whereas the Hindley/Milner 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. 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 [9, p. 85].

Fact 1 *For closed lexp,* $(\mathsf{empty} \vdash lexp : ty)$ *iff lexp has type ty in SML.*[7]

We use System T, 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 [3,16] as well as separate compilation [27,16].

[7] Formally, some minor syntactic adjustments (omitted here) are needed to translate *lexp* into an *exp* of the SML definition [24].

5 Assigning Constraints to Program Points

This section explains how type constraints are assigned to program points. We will define a function that maps labeled expressions to finite sets of type constraints. An expression is typable iff the associated constraint set is solvable. The function also keeps track of the program point that imposes a particular type constraint. This association between type constraints and program points is important for locating type errors.

A *labeled constraint* is a triple $\langle ty, ty', L\rangle$, which will be written as $ty \stackrel{L}{=} ty'$. It expresses that the types ty and ty' need to be equal for the program to be well-typed, and that this constraint has been jointly imposed by the program points contained in L. A labeled constraint is called *atomically labeled*, iff L is a one-element set. Initially, all constraints are atomically labeled, but during constraint solving arbitrarily labeled constraints get generated. Let $ty \stackrel{l}{=} ty'$ stand for $ty \stackrel{\{l\}}{=} ty'$. Let C range over $\mathsf{AtConstraintSet}$, the set of all finite sets of atomically labeled constraints. Let D range over $\mathsf{ConstraintSet}$, the set of all finite sets of labeled constraints. A *type substitution* is a function from TyVar to Ty.

Whenever a type substitution s is used in a position expecting a function from Ty to Ty, then s is implicitly *lifted* (coerced) to be a function from Ty to Ty such that for any type ty the function application $s(ty)$ yields the result of modifying 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 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 LExp to $\mathsf{Env} \times \mathsf{Ty} \times \mathsf{AtConstraintSet}$ is defined as the least relation that satisfies the rules in figure 6. 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 $(\wedge S) \geq (\wedge S')$ iff $S \subseteq S'$, and $\Gamma \geq \Gamma'$, iff $\Gamma(x) \geq \Gamma'(x)$ for all x in Var. The following facts are variations of propositions 7 and 8 on pages 39 and 44 in Damas's Ph.D. thesis [9].

Fact 2 *Suppose* $(lexp \Downarrow \langle \Gamma, ty, C\rangle)$.

(1) If s is a solution of C, then $(s(\Gamma) \vdash lexp : s(ty))$.

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

$$\frac{}{n^l \;\Downarrow\; \langle\; \mathsf{empty},\; a,\; C_0 \;\rangle} \quad \text{where } a \text{ fresh, } C_0 = \{\mathtt{int} \overset{l}{=} a\}$$

$$\frac{lexp_1 \Downarrow \langle \Gamma_1, ty_1, C_1\rangle; \qquad lexp_2 \Downarrow \langle \Gamma_2, ty_2, C_2\rangle}{(lexp_1 \;\mathtt{+}\; lexp_2)^l \;\Downarrow\; \langle\; \Gamma_1 \wedge \Gamma_2,\; a,\; C_0 \cup C_1 \cup C_2 \;\rangle}$$
where a fresh, $C_0 = \{ty_1 \overset{l}{=} \mathtt{int},\; ty_2 \overset{l}{=} \mathtt{int},\; \mathtt{int} \overset{l}{=} a\}$

$$\frac{lexp \Downarrow \langle \Gamma[x \mapsto \wedge S],\; ty,\; C\rangle}{(\mathtt{fn}\; x^l \;\mathtt{=>}\; lexp)^{l'} \;\Downarrow\; \langle\; \Gamma[x \mapsto \wedge\{\}],\; a,\; C_0 \cup C \;\rangle}$$
where a_x, a fresh, $C_0 = \{a_x \overset{l}{=} ty' \mid ty' \in S\} \cup \{\; a_x \;\mathtt{->}\; ty \;\overset{l'}{=}\; a \;\}$

$$\frac{lexp_1 \Downarrow \langle \Gamma_1, ty_1, C_1\rangle; \qquad lexp_2 \Downarrow \langle \Gamma_2, ty_2, C_2\rangle}{(lexp_1 \;\; lexp_2)^l \;\Downarrow\; \langle\; \Gamma_1 \wedge \Gamma_2,\; a,\; C_0 \cup C_1 \cup C_2 \;\rangle}$$
where a, a_1, a_2 fresh, $C_0 = \{\; ty_1 \;\overset{l}{=}\; a_1 \;\mathtt{->}\; a_2,\; ty_2 \overset{l}{=} a_1,\; a \overset{l}{=} a_2 \;\}$

$$\frac{lexp_1 \Downarrow \langle \Gamma_1, ty_1, C_1\rangle; \qquad lexp_2 \Downarrow \langle \Gamma_2[x \mapsto \wedge\{ty'_1, \ldots, ty'_n\}],\; ty_2,\; C_2\rangle}{(\mathtt{let\ val}\; x^l \;\mathtt{=}\; lexp_1 \;\mathtt{in}\; lexp_2 \;\mathtt{end})^{l'} \;\Downarrow\; \langle\; \Gamma'_1 \wedge \Gamma_2[x \mapsto \wedge\{\}],\; a,\; C_0 \cup C'_1 \cup C_2 \;\rangle}$$
where $\langle \Gamma_{1,1}, ty_{1,1}, C_{1,1}\rangle, \ldots, \langle \Gamma_{1,k}, ty_{1,k}, C_{1,k}\rangle$ are fresh variants of $\langle \Gamma_1, ty_1, C_1\rangle$,
$k = \mathsf{max}(n, 1),\; \Gamma'_1 = \Gamma_{1,1} \wedge \ldots \wedge \Gamma_{1,k},\; C'_1 = C_{1,1} \cup \ldots \cup C_{1,k}$,
$C = \{ty_{1,1} \overset{l}{=} ty'_1, \ldots, ty_{1,n} \overset{l}{=} ty'_n\},\; a$ fresh, $C_0 = \{a \overset{l'}{=} ty_2\} \cup C$

Fig. 6. Algorithm T

(2) If $(\Gamma' \vdash lexp : ty')$*, then there is a solution* s *of* C *such that* $s(\Gamma) \geq \Gamma'$ *and* $s(ty) = ty'$*.*

As an example, consider the following partially labeled expression. (We have omitted all labels that are irrelevant for this example.)

$$lexp \quad = \quad (\mathtt{fn}\; \mathtt{x}^{l_1} \;\mathtt{=>}\; \mathtt{f}\; (\mathtt{x}^{l_2}\; \mathtt{0})^{l_3}\; (\mathtt{x}^{l_4} \;\mathtt{+}\; \mathtt{0})^{l_5})$$

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 $(lexp \Downarrow \langle \mathsf{empty}[\mathtt{f} \mapsto a],\; a',\; C\rangle)$ for some type variables a, a' and some constraint set C

that has the following subset C':

$$C' = \{\ a_1 \overset{l_2}{=\!=} a_2,\ a_2 \overset{l_3}{=\!=} a_3\ \texttt{->}\ a_4,\ a_5 \overset{l_4}{=\!=} a_6,\ a_6 \overset{l_5}{=\!=} \texttt{int},\ a_7 \overset{l_1}{=\!=} a_1,\ a_7 \overset{l_1}{=\!=} a_5\ \}$$

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

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

then in the example 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* = (`fn x => x`) and `f`'s type scheme is assumed to be $(\forall a.\forall b.\ a \texttt{ -> } b \texttt{ -> } a \texttt{ -> } b)$ [8], then the principal type scheme of the entire expression contains $2^{(n+1)}$ distinct type variables. [9] Remember also that Hindley/Milner (SML) typability in our small expression language is DEXPTIME-complete [18,20]. The bad example above fortunately involves deep let-nesting that is rare in practice.

6 Finding Minimal Unsolvable Constraint Sets

We define a function that maps sets of atomically labeled constraints to sets of associated labels by $\mathsf{labels}(C) = \{\ l \mid (\exists ty, ty')((ty \stackrel{l}{=} ty') \in C)\ \}$. A set of labels L is called an *error* with respect to C iff C has an unsolvable subset C' such that $L = \mathsf{labels}(C')$. We denote the set of all such errors by $\mathsf{errors}(C)$. Moreover, $\mathsf{minErrors}(C)$ denotes the set of all those elements of $\mathsf{errors}(C)$ that are minimal with respect to set inclusion.

This section shows how to find minimal errors in an unsolvable constraint set. First, we present labeled unification, a vital tool used in this task. Then, we present a greedy minimization algorithm that, given an unsolvable constraint set C, finds *a single* element of $\mathsf{minErrors}(C)$. This algorithm is reasonably efficient for practical purposes. Finally, we show how to enumerate the elements of $\mathsf{minErrors}(C)$. Unfortunately, it is not practical to always exhaustively enumerate *all* elements of $\mathsf{minErrors}(C)$, because this set has a worst-case size exponential in the size of C [30]. However, we present a simple enumeration algorithm that 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.

6.1 Labeled Unification.

Our *labeled unification algorithm* is presented as a set of state transformation rules in figure 7. These rules define the state transformation relation $\rightarrow$. The algorithm is similar to Wand's algorithm [29]. Initial states are of the form $\mathsf{unify}(C)$ and final states of the form $\mathsf{Success}(E)$ or $\mathsf{Error}(L, l)$. Intermediate states are of the form $\mathsf{unify}(C, E)$ or $\mathsf{unify}(C, E, D, l)$ where the state compo-

[8] E.g., `fn x => fn y => fn z => (fn v => y) (fn u => (u x) (u z z))`.

[9] This example does not work in SML because of its value polymorphism restriction which only allows generalizing the types of syntactic values. To make this example work in SML, η-expand by replacing each occurrence of `f` $\mathtt{x}_i$ $\mathtt{x}_i$ by `fn z => f` $\mathtt{x}_i$ $\mathtt{x}_i$ `z`.

nents are as follows:

$C \in \mathsf{AtConstraintSet}$	initial constraints not yet considered
$E \in \mathsf{TyVar} \to ((\mathsf{Ty} \times \mathsf{LabelSet}) \cup \{\bot\})$	*environment* of derived bindings
$D \in \mathsf{ConstraintSet}$	*derived constraints*, not yet bindings
$l \in \mathsf{Label}$	the label whose constraints are currently the focus of attention

If one ignores the labels, the labeled unification algorithm is just a variation of transformation-based syntactic unification as presented, for instance, in [1], chapter 4.6. The following proposition is a consequence of lemma 4.6.5 in [1].

Proposition 3 (Termination of unify) *Each state transformation sequence terminates. A state is irreducible iff it is a final state.*

We define a function app that maps environments to partial functions from Ty to Ty. Let the function $\mathsf{app}(E)$ be the least defined function such that:

$$\mathsf{app}(E)(\texttt{int}) = \texttt{int} \tag{1}$$

$$(E(a) = \bot) \Rightarrow (\mathsf{app}(E)(a) = a) \tag{2}$$

$$(E(a) = \langle ty, L\rangle) \wedge (\mathsf{app}(E)(ty) = ty') \Rightarrow (\mathsf{app}(E)(a) = ty') \tag{3}$$

$$(\mathsf{app}(E)(ty_i) = ty_i' \text{ for } i = 1, 2) \Rightarrow (\mathsf{app}(E)(ty_1 \texttt{->} ty_2) = ty_1' \texttt{->} ty_2') \tag{4}$$

The function $\mathsf{app}(E)$ is a partial function for every E. Note that $\mathsf{app}(E)$ is not always total, because rule 3 is not size decreasing — the variable a in this rule may, for instance, occur in type ty. Environments E for which $\mathsf{app}(E)$ is not total are not generated by our algorithms, so this issue is unimportant. When $\mathsf{app}(E)$ is total, in fact its behavior as a function from Ty to Ty is the same as the lifting to Ty to Ty of the substitution that results from $\mathsf{app}(E)$ by restricting it to the domain TyVar. So in this case, we will implicitly treat $\mathsf{app}(E)$ as though it were the substitution that results from restricting its domain.

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. The *identity* substitution is denoted by id and is defined by $\mathsf{id}(a) = a$ for all type variables a. A type substitution s is called a *most general unifier* (mgu) of C iff for every solution s' of C there exists a type substitution s'' such that $s' = s'' \circ s$. Part 1 of the following theorem is a consequence of lemma 4.6.7 in [1]. Part 2 of the theorem can be derived from lemmas 4.6.7 and 4.6.10 in [1].

Theorem 4 (Correctness of unify)

(1) If $\mathsf{unify}(C) \to^* \mathsf{Success}(E)$*, then* $\mathsf{app}(E)$ *is a total function and a most general unifier of* C*.*

(2) If $\mathsf{unify}(C) \to^* \mathsf{Error}(L, l)$*, then* $L \in \mathsf{errors}(C)$ *and* $L \setminus \{l\} \notin \mathsf{errors}(C)$*.*

dummy is some arbitrarily chosen fixed label

Driver rules:

$$\begin{aligned}
\mathsf{unify}(C) &\to \mathsf{unify}(C, (\lambda a \in \mathsf{TyVar}.\bot)) && (1)\\
\mathsf{unify}(C, E) &\to \mathsf{unify}(C, E, \emptyset, \mathsf{dummy}) && (2)\\
\mathsf{unify}(\emptyset, E, \emptyset, l) &\to \mathsf{Success}(E) && (3)\\
\mathsf{unify}(C, E, \emptyset, l') &\to \mathsf{unify}(C \setminus \Pi_l(C), E, \Pi_l(C), l) \quad \text{if } \Pi_l(C) \neq \emptyset && (4)
\end{aligned}$$

Unification rules:

$$\begin{aligned}
&\mathsf{unify}(C, E, \{ty \overset{L}{=} ty\} \uplus D, l) \to \mathsf{unify}(C, E, D, l)\\
&\mathsf{unify}(C, E, \{ty_1 \texttt{ -> } ty_2 \overset{L}{=} \texttt{int}\} \uplus D, l) \to \mathsf{Error}(L, l)\\
&\mathsf{unify}(C, E, \{\texttt{int} \overset{L}{=} ty_1 \texttt{ -> } ty_2\} \uplus D, l) \to \mathsf{Error}(L, l)\\
&\mathsf{unify}(C, E, \{\texttt{int} \overset{L}{=} a\} \uplus D, l) \to \mathsf{unify}(C, E, \{a \overset{L}{=} \texttt{int}\} \cup D, l)\\
&\mathsf{unify}(C, E, \{ty_1 \texttt{ -> } ty_2 \overset{L}{=} a\} \uplus D, l) \to \mathsf{unify}(C, E, \{a \overset{L}{=} ty_1 \texttt{ -> } ty_2\} \cup D, l)\\
&\mathsf{unify}(C, E, \{ty_1 \texttt{ -> } ty_2 \overset{L}{=} ty_1' \texttt{ -> } ty_2'\} \uplus D, l)\\
&\quad \to \mathsf{unify}(C, E, \{ty_1' \overset{L}{=} ty_1, ty_2 \overset{L}{=} ty_2'\} \cup D, l)\\
&\mathsf{unify}(C, E[a \mapsto \langle ty', L'\rangle], \{a \overset{L}{=} ty\} \uplus D, l)\\
&\quad \to \mathsf{unify}(C, E[a \mapsto \langle ty', L'\rangle], \{ty' \overset{L \cup L'}{=\!=} ty\} \cup D, l)
\end{aligned}$$

$$\mathsf{unify}(C, E[a \mapsto \bot], \{a \overset{L}{=} ty\} \uplus D, l) \to \begin{cases} \mathsf{unify}(C, E[a \mapsto \langle ty, L\rangle], D, l) & \text{if } \mathsf{occurs}(E, L, a, ty, 0) = \emptyset \\ \mathsf{Error}(L', l) & \text{if } \langle L', n\rangle \in \mathsf{occurs}(E, L, a, ty, 0) \text{ and } n \geq 1 \\ \mathsf{unify}(C, E[a \mapsto \bot], D, l) & \text{otherwise} \end{cases}$$

Occurs check:

$$\begin{aligned}
\mathsf{occurs}(E[a' \mapsto \langle ty, L'\rangle], L, a, a', n) &= \mathsf{occurs}(E[a' \mapsto \langle ty, L'\rangle], L \cup L', a, ty, n)\\
\mathsf{occurs}(E, L, a, a, n) &= \{\langle L, n\rangle\}\\
\mathsf{occurs}(E[a' \mapsto \bot], L, a, a', n) &= \emptyset \qquad \text{if } a \neq a'\\
\mathsf{occurs}(E, L, a, \texttt{int}, n) &= \emptyset\\
\mathsf{occurs}(E, L, a, ty_1 \texttt{ -> } ty_2, n) &= \bigcup_{i=1,2} \mathsf{occurs}(E, L, a, ty_i, n+1)
\end{aligned}$$

Fig. 7. A non-deterministic labeled unification algorithm

If one ignores the labels, the labeled unification algorithm looks very much like standard presentations of unification. Our version of the occurs check may look a bit unfamiliar. Here is an explanation: $\mathsf{occurs}(E, L, a, ty, 0)$ returns a set of pairs of the form $\langle L', n\rangle$. If $\langle L', n\rangle \in \mathsf{occurs}(E, L, a, ty, 0)$, then there

is an occurrence of type variable a in $\mathsf{app}(E)(ty)$ "under" n function type constructors (and remember that this is the only type constructor in our small model language). The occurs check succeeds iff either $\mathsf{occurs}(E, L, a, ty, 0)$ is empty (a does not occur in $\mathsf{app}(E)(ty)$) or it only contains pairs of the form $\langle L, 0\rangle$ (a is equal to $\mathsf{app}(E)(ty)$).

Note that the transformation system in figure 7 is non-deterministic. Arbitrary choices can be used for the label l in driver rule 4, the constraint in each of the unification rules and the label set L' associated with the occurs check in the error case of the last unification 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 $C = \{\ a_1 \overset{l_1}{=\!=} a_2 \texttt{ -> } a_3,\ a_2 \overset{l_2}{=\!=} \texttt{int -> } a_4,\ a_1 \overset{l_3}{=\!=} (a_5 \texttt{ -> } (a_6 \texttt{ -> } a_7)) \texttt{ -> int},\ a_2 \overset{l_4}{=\!=} a_8 \texttt{ -> int}\ \}$

Both $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(\{l_1, l_2, l_3, l_4\}, l_4)$ *and* $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(\{l_1, l_3, l_4\}, l_4)$. *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* $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(L, l)$ *does* not *imply that* L *is minimal.*

Example 6 $C = \{\ a_1 \overset{l_1}{=\!=} a_2 \texttt{ -> } a_3,\ a_1 \overset{l_2}{=\!=} (a_4 \texttt{ -> } (a_5 \texttt{ -> } a_6)) \texttt{ -> int},\ a_1 \overset{l_3}{=\!=} (a_7 \texttt{ -> } (a_8 \texttt{ -> } a_9)) \texttt{ -> int},\ a_2 \overset{l_4}{=\!=} \texttt{int -> int}\ \}$

Then $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(\{l_1, l_2, l_4\}, l_4)$. *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. It is also the case that* $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(\{l_1, l_3, l_4\}, l_4)$. *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* $\mathsf{minErrors}(C) = \{\{l_1, l_2, l_4\}, \{l_1, l_3, l_4\}\}$

6.2 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: If $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(L, l)$, 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 repeatedly to build a minimal error.

In figure 8, the algorithm is presented as a set of state transformation rules. Initial states are of the form $\mathsf{minimize}(C, L, l)$. An initial state of this form is called *nice* iff $L \in \mathsf{errors}(C)$ and $L \setminus \{l\} \notin \mathsf{errors}(C)$. Final states are of the form $\mathsf{MinError}(L)$. Intermediate states are of the form $\mathsf{minimize}(C, s, L, l, L')$, where s ranges over type substitutions. The intention is that, if $\mathsf{minimize}(C, L, l)$ is nice and $\mathsf{minimize}(C, L, l) \rightarrow^* \mathsf{MinError}(L')$, then L' is a minimal error of C that is contained in L.

Lemma 7 *If* $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(L, l)$*, then* $l \in \bigcap\{\, L' \in \mathsf{errors}(C) \mid L' \subseteq L \,\}$.

Proof. Suppose, towards a contradiction, that $L' \subseteq L$, $L' \in \mathsf{errors}(C)$ and $l \notin L'$. Then $L \setminus \{l\} \in \mathsf{errors}(C)$, because $L' \subseteq L \setminus \{l\}$. But also $L \setminus \{l\} \notin \mathsf{errors}(C)$, by theorem 4. □

We will make use of the following standard property of most general unifiers.

Proposition 8 *Suppose s is a mgu of C. Then the following statements hold:*

(1) If s' is a mgu of $s(C')$, then $(s' \circ s)$ is a mgu of $C \cup C'$.
(2) If $s(C')$ is unsolvable, then so is $C \cup C'$.

Proof. Suppose s is a mgu of C.

Part 2: Let s_0 be a solution of $C \cup C'$. We prove that $s(C')$ is solvable, thus, establishing part 2 of the lemma. Certainly, s_0 is a solution of C. Thus, by definition of mgu, there exists s_1 such that $s_0 = s_1 \circ s$. Now, s_1 solves $s(C')$, because $s_1 \circ s$ solves C'.

Part 1: Let s' be a mgu of $s(C')$. Let s_0 be a solution of $C \cup C'$. We need to find s'' such that $s_0 = s'' \circ (s' \circ s)$. By the same argumentation as in the proof of part (2), there exists s_1 such that $s_0 = s_1 \circ s$ and s_1 solves $s(C')$. Because s_1 solves $s(C')$ and s' is a mgu of $s(C')$, there exists s'' such that $s_1 = s'' \circ s'$. Then $s_0 = s_1 \circ s = (s'' \circ s') \circ s = s'' \circ (s' \circ s)$, by associativity. □

Lemma 9 *Suppose* $\mathsf{minimize}(C_{in}, L_{in}, l_{in})$ *is nice and* $\mathsf{minimize}(C_{in}, L_{in}, l_{in}) \rightarrow^* \mathsf{minimize}(C, s, L, l, L')$*. 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) s is a most general unifier of $\Pi_{L'}(C)$.
(3) $s(\Pi_L(C))$ is not solvable.
(4) $s(\Pi_{L \setminus \{l\}}(C))$ is solvable.

Proof. Each statement from part 1 is proved by induction on the length of the transformation sequence. Part 2 is proved by induction on the length of the transformation sequence, using theorem 4(1) and proposition 8(1). Parts 3

$$\mathsf{minimize}(C, L, l) \to \mathsf{minimize}(C, \mathsf{id}, L, l, \emptyset)$$

$$\frac{\mathsf{unify}(\ s(\Pi_l(C))\ )\ \to^*\ \mathsf{Error}(\cdot,\cdot)}{\mathsf{minimize}(C, s, L, l, L')\ \to\ \mathsf{MinError}(L' \cup \{l\})}$$

$$\frac{\begin{array}{c}\mathsf{unify}(\ s(\Pi_l(C))\ )\ \to^*\ \mathsf{Success}(E_0); \quad s_0 = \mathsf{app}(E_0) \circ s; \\ \mathsf{unify}(\ s_0(\Pi_{L\setminus\{l\}}(C))\ )\ \to^*\ \mathsf{Error}(L_0, l_0)\end{array}}{\mathsf{minimize}(C, s, L, l, L')\ \to\ \mathsf{minimize}(C, s_0, L_0, l_0, L' \cup \{l\})}$$

Fig. 8. A non-deterministic error slice minimization algorithm

and 4 are proved by inspection of the last transformation rule, using theorem 4(2). □

Proposition 10 (Termination of minimize) *Let* $\mathsf{minimize}(C, L_{in}, l_{in})$ *be nice. Every transformation sequence starting from* $\mathsf{minimize}(C, L_{in}, l_{in})$ *terminates. If* $\mathsf{minimize}(C, L_{in}, l_{in}) \to^* st$*, then* st *is irreducible iff it is a final state.*

Proof. Transformation sequences terminate, because after the first step each subsequent step decrements the size of the label set L. When considering arbitrary states of the form $\mathsf{minimize}(C, s, L, l, L')$, including those not reachable from nice initial states, the rules are non-exhaustive. Specifically, in the third rule, it is conceivable that $\mathsf{unify}(s_0(\Pi_{L\setminus\{l\}}(C))) \to^* \mathsf{Success}(\cdot)$. We now show that this is impossible for states reachable from nice initial states. First, we assume that from the initial state we have reached the state $\mathsf{minimize}(C, s, L, l, L')$:

(1) $\mathsf{minimize}(C, L_{in}, l_{in}) \to^* \mathsf{minimize}(C, s, L, l, L')$ assumption

Next, we assume that the two first premises of rule 3 hold:

(2) $\mathsf{unify}(s(\Pi_l(C))) \to^* \mathsf{Success}(E_0)$ assumption

(3) $s_0 = \mathsf{app}(E_0) \circ s$ assumption

Then, we make the following assumption, towards a contradiction:

(4) $\mathsf{unify}(s_0(\Pi_{L\setminus\{l\}}(C))) \to^* \mathsf{Success}(\cdot)$ assumption

Now, by (2) and theorem 4(1), $\mathsf{app}(E_0)$ is a mgu of $s(\Pi_l(C))$. By (4) and theorem 4(1), $s_0(\Pi_{L\setminus\{l\}}(C))$ is solvable. Then, by proposition 8(1), $s(\Pi_L(C))$ is solvable. But this contradicts lemma 9(3). □

The following lemma is the key for the correctness of minimize.

Lemma 11 *Suppose* $\mathsf{minimize}(C, L_{in}, l_{in})$ *is nice and* $\mathsf{minimize}(C, L_{in}, l_{in}) \to^*$

$\mathsf{minimize}(C', s, L, l, L')$. *Then:*

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

Proof. By induction on the length of the transformation sequence. Suppose $\mathsf{minimize}(C, L_{in}, l_{in})$ is nice.

Case, transformation sequence is of length 1: In this case, the transformation sequence only uses rule 1:

(1) $\mathsf{minimize}(C, L_{in}, l_{in}) \rightarrow \mathsf{minimize}(C, \mathsf{id}, L_{in}, l_{in}, \emptyset)$ assumption

(2) $\forall L_0 \in \mathsf{errors}(C).\ ((L_0 \subseteq L_{in}) \Rightarrow (\{l_{in}\} \subseteq L_0))$ goal

But (2) holds, because $(L_{in} \setminus \{l_{in}\}) \notin \mathsf{errors}(C)$, because we assumed that $\mathsf{minimize}(C, L_{in}, l_{in})$ is nice.

Case, transformation sequence has a length of at least 2: In this case, the last rule of the transformation sequence is rule 3. First, we assume that we have reached a state $\mathsf{minimize}(C, s, L, l, L')$:

(1) $\mathsf{minimize}(C, L_{in}, l_{in}) \rightarrow^* \mathsf{minimize}(C, s, L, l, L')$ assumption

Next, we assume that the premises of rule 3 hold:

(2) $\mathsf{unify}(s(\Pi_l(C))) \rightarrow^* \mathsf{Success}(E_0)$ assumption

(3) $s_0 = \mathsf{app}(E_0) \circ s$ assumption

(4) $\mathsf{unify}(s_0(\Pi_{L \setminus \{l\}}(C))) \rightarrow^* \mathsf{Error}(L_0, l_0)$ assumption

We need to show the following statement:

$$\forall L'' \in \mathsf{errors}(C).\ ((L'' \subseteq L_0 \cup (L' \cup \{l\})) \Rightarrow ((L' \cup \{l\}) \cup \{l_0\} \subseteq L''))$$

To this end, we pick an arbitrary label set L'' that satisfies the premise of the implication:

(5) $L'' \in \mathsf{errors}(C)$ assumption

(6) $L'' \subseteq L_0 \cup (L' \cup \{l\})$ assumption

(7) $L' \cup \{l, l_0\} \subseteq L''$ goal

(8) $L' \cup \{l\} \subseteq L''$ by ind. hyp. on (1)

(9) s_0 is a mgu of $(\Pi_{L' \cup \{l\}}(C))$ by lemma 9(2)

(10) $L'' \setminus (L' \cup \{l\}) \in \mathsf{errors}(s_0(\Pi_{L \setminus \{l\}}(C)))$ by subproof below

Now, by lemma 7 and (4), l_0 is an element of *every* error of $s_0(\Pi_{L \setminus \{l\}}(C))$ that is contained in L_0. By (10) and (6), $(L'' \setminus (L' \cup \{l\}))$ is such an error. Thus, $l_0 \in L''$. From this and (8), it follows that $L' \cup \{l, l_0\} \subseteq L''$.

Subproof of (10): Suppose, towards a contradiction, that (10) does not hold, i.e., $\Pi_{L''\setminus(L'\cup\{l\})}(s_0(\Pi_{L\setminus\{l\}}(C)))$ is solvable. Note first that the following subset inclusion holds:

(11) $L'' \setminus (L' \cup \{l\}) \subseteq L_0 \subseteq L \setminus \{l\}$ by (6), (4)

Now, the following chain of equations holds:

$$\begin{aligned}\Pi_{L''\setminus(L'\cup\{l\})}(s_0(\Pi_{L\setminus\{l\}}(C))) &= s_0(\Pi_{(L''\setminus(L'\cup\{l\}))\cap(L\setminus\{l\})}(C)) \\ &= s_0(\Pi_{L''\setminus(L'\cup\{l\})}(C))\end{aligned}$$

The first of these equations follows from the definition of Π, the second one holds by (11). Now, by proposition 8(1) and (9), it follows that $\Pi_{L''}(C)$ is solvable. That contradicts (5). □

Theorem 12 (Correctness of minimize**)** *If* $\mathsf{minimize}(C, L_{in}, l_{in})$ *is nice and* $\mathsf{minimize}(C, L_{in}, l_{in}) \to^* \mathsf{MinError}(L_{out})$*, then* $L_{out} \in \mathsf{minErrors}(C)$ *and* $L_{out} \subseteq L_{in}$*.*

Proof. Suppose $\mathsf{minimize}(C, L_{in}, l_{in})$ is nice and suppose $\mathsf{minimize}(C, L_{in}, l_{in}) \to^* \mathsf{MinError}(L_{out})$. Then the last step of the transformation sequence must be an instance of rule 2. Therefore, there are s, L, l, L' such that:

(1) $L_{out} = L' \cup \{l\}$

(2) $\mathsf{minimize}(C, L_{in}, l_{in}) \to^* \mathsf{minimize}(C, s, L, l, L')$

(3) $\mathsf{minimize}(C, s, L, l, L') \to \mathsf{MinError}(L_{out})$

(4) $\mathsf{unify}(s(\Pi_l(C))) \to^* \mathsf{Error}(\cdot, \cdot)$

By (2) and lemma 9(1), $L_{out} = L' \cup \{l\} \subseteq L_{in}$. We show that $L_{out} \in \mathsf{errors}(C)$:

(5) s is a mgu of $(\Pi_{L'}(C))$ by (2), lemma 9(2)

(6) $\{l\} \in \mathsf{errors}(s(C))$ by (4)

(7) $L_{out} = L' \cup \{l\} \in \mathsf{errors}(C)$ by (5), (6), proposition 8(2)

It remains to show that L_{out} is minimal. To this end, let $L_0 \in \mathsf{errors}(C)$ and $L_0 \subseteq L_{out}$. Then $L_{out} = L' \cup \{l\} \subseteq L_0$, by lemma 11 and (2). □

The transformation sequence $\mathsf{minimize}(C, L, l) \to^* \mathsf{MinError}(L')$ 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 that can be adapted to perform the same role. 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 that are 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.

6.3 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. Wolfram has shown this for arbitrary constraint sets [30]. The following example shows that this worst case behavior comes up for constraint sets that have been generated by algorithm T. Note that the example does not use type polymorphism, i.e., there are no let-expressions.

Example 13 (An exponentially sized set of minimal errors)
The following expression has 2^n distinct minimal errors.

$$\begin{array}{l}\texttt{fn}\ \mathtt{x}_0\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{x}_n\ \texttt{=>}\ \texttt{fn}\ \mathtt{f}_1\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{f}_n\ \texttt{=>}\\ \quad \texttt{fn}\ \mathtt{g}_1\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{g}_n\ \texttt{=>}\ \texttt{fn}\ \mathtt{y}_1\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{y}_{2n}\ \texttt{=>}\\ \quad\quad \mathtt{u}\ \mathit{lexp}_1\ \ldots\ \mathit{lexp}_n\ (\mathtt{x}_n\ \mathtt{x}_0)\end{array}$$

where, for each k in $\{1,\ldots,n\}$, $lexp_k$ is defined by

$$\mathit{lexp}_k\ =\ \mathtt{z}_k\ (\mathtt{f}_k\ \mathtt{x}_{k-1})\ (\mathtt{g}_k\ \mathtt{x}_k)\ (\mathtt{f}_k\ \mathtt{y}_{2k-1})\ (\mathtt{g}_k\ \mathtt{y}_{2k-1})\ (\mathtt{f}_k\ \mathtt{y}_{2k})\ (\mathtt{g}_k\ \mathtt{y}_{2k})$$

Each minimal error contains all program points that are associated with the following program slice. (This program slice itself is not an error, though.)

$$\begin{array}{l}\texttt{fn}\ \mathtt{x}_0\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{x}_n\ \texttt{=>}\ \texttt{fn}\ \mathtt{f}_1\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{f}_n\ \texttt{=>}\\ \quad \texttt{fn}\ \mathtt{g}_1\ \texttt{=>}\ \ldots\ \texttt{fn}\ \mathtt{g}_n\ \texttt{=>}\\ \quad\quad (\ldots\ \mathit{sl}_1\ \ldots\ \mathit{sl}_n\ \ldots\ (\mathtt{x}_n\ \mathtt{x}_0)\ \ldots)\end{array}$$

where, for each k in $\{1,\ldots,n\}$, sl_k is

$$\mathit{sl}_k\ =\ (\ldots\ (\mathtt{f}_k\ \mathtt{x}_{k-1})\ \ldots\ (\mathtt{g}_k\ \mathtt{x}_k)\ \ldots.)$$

These program points impose the following type constraints for each k in $\{1,\ldots,n\}$.

(1) argument type of $\mathtt{f}_k$ = type of $\mathtt{x}_{k-1}$
(2) argument type of $\mathtt{g}_k$ = type of $\mathtt{x}_k$

In addition, for each k in $\{1,\ldots,n\}$, each minimal type error contains exactly

one of the following two sets of program points:

$$\texttt{..\ fn}\ \texttt{y}_{2k-1}\ \texttt{=>}\ (\texttt{..}\ (\texttt{f}_k\ \texttt{y}_{2k-1})\ \texttt{..}\ (\texttt{g}_k\ \texttt{y}_{2k-1})\ \texttt{..})\ \texttt{..}$$

or

$$\texttt{..\ fn}\ \texttt{y}_{2k}\ \texttt{=>}\ (\texttt{..}\ (\texttt{f}_k\ \texttt{y}_{2k})\ \texttt{..}\ (\texttt{g}_k\ \texttt{y}_{2k})\ \texttt{..})\ \texttt{..}$$

Each one of these forces the following type constraint:

(3) argument type of $\texttt{f}_k$ *= argument type of* $\texttt{g}_k$

Note, that there are 2^n *possibilities for picking these* n *additional sets of program points. From (1), (2) and (3), it follows that* $\texttt{x}_0$ *and* $\texttt{x}_n$ *must have identical types. But then* $(\texttt{x}_n\ \texttt{x}_0)$ *is not well-typed.*

For error enumeration, 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.[10] We think that in practice it is not a problem that our algorithms find only some of the minimal error slices of a program. 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 $\mathsf{minErrors}$ satisfies the following equations:

If $\mathsf{unify}(C) \rightarrow^* \mathsf{Success}(\cdot)$: $\quad \mathsf{minErrors}(C) \;=\; \emptyset$

If $\mathsf{unify}(C) \rightarrow^* \mathsf{Error}(L, \cdot)$:

$$\mathsf{minErrors}(C) \;=\; \mathsf{min}(\bigcup \{\ \mathsf{minErrors}(\Pi_{\mathsf{labels}(C)\setminus\{l\}}(C)) \mid l \in L\ \} \;\cup\; \{L\})$$

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

[10] 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.

$$\mathsf{enum}(C) \;\rightarrow\; \mathsf{enum}(C, \emptyset, \{\emptyset\}); \qquad \mathsf{enum}(C, \mathit{found}, \emptyset) \;\rightarrow\; \mathsf{MinErrors}(\mathit{found})$$

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

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

$$\begin{array}{lcl}
\mathsf{insertError}(L, \mathit{found}) & \stackrel{\text{def}}{=\!=} & \begin{cases} \mathit{found}, & \text{if } (\exists L' \in \mathit{found})(L' \subseteq L) \\ \{\; L' \in \mathit{found} \mid L \not\subseteq L' \;\} \;\cup\; \{L\}, & \text{otherwise} \end{cases} \\
\mathsf{insertTodos}(\mathit{Ls}, \mathit{todo}) & \stackrel{\text{def}}{=\!=} & \mathit{todo} \;\cup\; \{\; L \in \mathit{Ls} \mid (\forall L' \in \mathit{todo})(L' \not\subseteq L) \;\} \\
\mathsf{distribute}(L', L) & \stackrel{\text{def}}{=\!=} & \{\; \{l'\} \cup L \mid l' \in L' \;\}
\end{array}$$

Fig. 9. A non-deterministic error slice enumeration algorithm

C that is contained in $(\mathsf{labels}(C) \setminus L)$, 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 $\mathsf{minErrors}(C)$ is small. The algorithm in figure 9 is essentially an iterative version of the above recurrences presented as a set of state transformation rules. Initial states are of the form $\mathsf{enum}(C)$ and final states of the form $\mathsf{MinErrors}(\mathit{Ls})$, where Ls is a set of pairwise incomparable label sets. Intermediate states are of the form $\mathsf{enum}(C, \mathit{found}, \mathit{todo})$ 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 $(\mathsf{labels}(C) \setminus L)$ still need to be found. We usually interrupt the execution of $\mathsf{enum}(C)$ before it terminates, but only after it has found at least one error. In this case, the elements of the current *found*-set get minimized and then returned.

Proposition 14 (Termination of enum) *Each state transformation sequence terminates. A state is irreducible iff it is a final state.*

Proof. First, one proves the following by induction on the length of the transformation sequence: *If* $\mathsf{enum}(C) \rightarrow^* \mathsf{enum}(C, \mathit{found}, \mathit{todo})$*, then the elements of todo are pairwise incomparable with respect to subset inclusion.* Let $\downarrow(\mathit{todo}) = \{L \mid (\exists L' \in \mathit{todo})(L \subseteq L')\}$. Fix C and let $\mathcal{P}$ be the powerset of $\mathsf{labels}(C)$.

Let $f(todo) = \mathcal{P}\setminus \downarrow(todo)$. Suppose that elements of *todo* are pairwise incomparable with respect to subset inclusion. Then the following statements hold.

(1) If $f(todo) = \emptyset$, then every transformation sequence starting from state $\mathsf{enum}(C, found, todo)$ terminates in $\mathsf{minErrors}(found)$.
(2) If $f(todo) \neq \emptyset$ and $\mathsf{enum}(C, found, todo) \rightarrow \mathsf{enum}(C, found', todo')$, then $f(todo')$ is a proper subset of $f(todo)$. □

Theorem 15 (Correctness of enum) *If* $\mathsf{enum}(C) \rightarrow^* \mathsf{MinErrors}(Ls)$, *then* $Ls = \mathsf{minErrors}(C)$.

Proof. Let $\mathsf{enum}(C) \rightarrow^* \mathsf{enum}(C, found, todo)$. One shows the following statements, separately, by induction on the length of the transformation sequence:

(1) Elements of *found* are pairwise incomparable with respect to subset inclusion.
(2) $\mathsf{minErrors}(C) = \mathsf{min}(found \cup \bigcup\{\mathsf{minErrors}(\Pi_{\mathsf{labels}(C)\setminus L}(C)) \mid L \in todo\})$

Correctness now follows by inspection of the last transformation rule. □

7 Slicing the Program

Figure 10 defines the abstract syntax class of *slices*. The grammar for *sl* in figure 10 extends the labeled expression grammar for *lexp* in figure 4 by the additional phrase `dots`(*sls*), where *sls* is a (possibly empty) finite sequence of slices. 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`(sl_1, sl_2, sl_3) as:

$$\texttt{(..}\ sl_1\ \texttt{..}\ sl_2\ \texttt{..}\ sl_3\ \texttt{..)}$$

For instance, the type error slice

$$\texttt{fn x}^{l_1}\ \texttt{=>}\ \texttt{dots(}\ (\texttt{x}^{l_2}\ \texttt{dots()})^{l_3},\ (\texttt{x}^{l_4}\ \texttt{+ dots()})^{l_5}\ \texttt{)}$$

computed for the erroneous program from section 5 is displayed as:

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

Figure 10 defines additional typing rules for slices. A slice of the form `dots`(sl_1, ..., sl_k) is typable using type assumptions Γ with any result type iff sl_1 through

$$
\begin{array}{rcll}
sls & \in & \text{set of finite sequences of slices} & \\
vsl & \in & \mathsf{VarSlice} & ::= \; x^l \mid \texttt{dots}() \\
sl & \in & \mathsf{Slice} & ::= \; x^l \mid n^l \mid (sl \texttt{ + } sl)^l \mid (\texttt{fn } vsl \texttt{ => } sl)^l \;\; (sl \; sl)^l \mid \\
 & & & \quad (\texttt{let val } vsl \texttt{ = } sl \texttt{ in } sl \texttt{ end})^l \mid \texttt{dots}(sls)
\end{array}
$$

Typing rules

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

Algorithm T

$$
\frac{sl_i \Downarrow \langle \Gamma_i, ty_i, C_i \rangle \quad \text{for } i \text{ in } \{1,\ldots,k\}; \qquad a \text{ fresh}}{\texttt{dots}(sl_1,\ldots,sl_k) \;\Downarrow\; \langle\; \Gamma_1 \wedge \ldots \wedge \Gamma_k,\; a,\; C_1 \cup \ldots \cup C_k \;\rangle}
$$

$$
\frac{sl \Downarrow \langle \Gamma, ty, C \rangle; \qquad a, a' \text{ fresh}}{(\texttt{fn dots() => } sl)^l \;\Downarrow\; \langle \Gamma,\; a,\; \{a' \texttt{ -> } ty \stackrel{l}{=} a\} \cup C \;\rangle}
$$

$$
\frac{sl_1 \Downarrow \langle \Gamma_1, ty_1, C_1 \rangle; \qquad sl_2 \Downarrow \langle \Gamma_2, ty_2, C_2 \rangle; \qquad a \text{ fresh}; \qquad C_0 = \{a \stackrel{l}{=} ty_2\}}{(\texttt{let val dots() = } sl_1 \texttt{ in } sl_2 \texttt{ end})^l \;\Downarrow\; \langle\; \Gamma_1 \wedge \Gamma_2,\; a,\; C_0 \cup C_1 \cup C_2 \;\rangle}
$$

Fig. 10. Additional rules for slices

sl_k are typable using Γ. The typing rules for other phrases are omitted, because they are the same as for expressions (see figure 5). Figure 10 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 6. Figure 11 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 uses the auxiliary function mask(*sls*), which, roughly speaking, returns `dots`(*sls*) but also flattens immediately nested `dots` on the fly. As a result of flattening, slice(L, *lexp*) does not have immediately nested `dots`.

$$\frac{lexp \;\downarrow^L\; sl}{\mathsf{slice}(L, lexp) = sl}$$

$$\frac{l \in L}{x^l \;\downarrow^L\; x^l} \qquad \frac{l \notin L}{x^l \;\downarrow^L\; \mathsf{mask}()} \qquad \frac{l \in L}{n^l \;\downarrow^L\; n^l} \qquad \frac{l \notin L}{n^l \;\downarrow^L\; \mathsf{mask}()}$$

$$\frac{lexp_1 \;\downarrow^L\; sl_1; \quad lexp_2 \;\downarrow^L\; sl_2; \quad l \in L}{(lexp_1 \texttt{ + } lexp_2)^l \;\downarrow^L\; (sl_1 \texttt{ + } sl_2)^l}$$

$$\frac{lexp_1 \;\downarrow^L\; sl_1; \quad lexp_2 \;\downarrow^L\; sl_2; \quad l \notin L}{(lexp_1 \texttt{ + } lexp_2)^l \;\downarrow^L\; \mathsf{mask}(sl_1, sl_2)}$$

$$\frac{x^{l_1} \;\downarrow^L\; vsl; \quad lexp \;\downarrow^L\; sl; \quad l_1 \in L \text{ or } l_2 \in L}{(\texttt{fn } x^{l_1} \texttt{ => } lexp)^{l_2} \;\downarrow^L\; (\texttt{fn } vsl \texttt{ => } sl)^{l_2}}$$

$$\frac{lexp \;\downarrow^L\; sl; \quad l_1 \notin L \text{ and } l_2 \notin L}{(\texttt{fn } x^{l_1} \texttt{ => } lexp)^{l_2} \;\downarrow^L\; \mathsf{mask}(sl)}$$

$$\frac{lexp_1 \;\downarrow^L\; sl_1; \quad lexp_2 \;\downarrow^L\; sl_2; \quad l \in L}{(lexp_1 \;\; lexp_2)^l \;\downarrow^L\; (sl_1 \;\; sl_2)^l}$$

$$\frac{lexp_1 \;\downarrow^L\; sl_1; \quad lexp_2 \;\downarrow^L\; sl_2; \quad l \notin L}{(lexp_1 \;\; lexp_2)^l \;\downarrow^L\; \mathsf{mask}(sl_1, sl_2)}$$

$$\frac{x^{l_1} \;\downarrow^L\; vsl; \quad lexp_1 \;\downarrow^L\; sl_1; \quad lexp_2 \;\downarrow^L\; sl_2; \qquad l_1 \in L \text{ or } l_2 \in L}{(\texttt{let val } x^{l_1} \texttt{ = } lexp_1 \texttt{ in } lexp_2 \texttt{ end})^{l_2} \;\downarrow^L\; (\texttt{let val } vsl \texttt{ = } sl_1 \texttt{ in } sl_2 \texttt{ end})^{l_2}}$$

$$\frac{lexp_1 \;\downarrow^L\; sl_1; \quad lexp_2 \;\downarrow^L\; sl_2; \qquad l_1 \notin L \text{ and } l_2 \notin L}{(\texttt{let val } x^{l_1} \texttt{ = } lexp_1 \texttt{ in } lexp_2 \texttt{ end})^{l_2} \;\downarrow^L\; \mathsf{mask}(sl_1, sl_2)}$$

. . . . . .

$$\frac{}{\mathsf{mask}() \;=\; \texttt{dots}()} \qquad \frac{(\forall sls')(sl \neq \texttt{dots}(sls')); \quad \mathsf{mask}(sls) \;=\; \texttt{dots}(sls'')}{\mathsf{mask}(sl, sls) \;=\; \texttt{dots}(sl, sls'')}$$

$$\frac{\mathsf{mask}(sls) \;=\; \texttt{dots}(sls'')}{\mathsf{mask}(\texttt{dots}(sls'), sls) \;=\; \texttt{dots}(sls', sls'')}$$

Fig. 11. Slicing

The function slice constitutes the last phase of our type error slicing method. To summarize, our method consists of the following three phases:

(1) Compute a type constraint set C for the input program *lexp* using algorithm T from figure 6.
(2) Find minimal error sets L of the constraint set C, using a combination of the algorithms from figures 9 and 8, as described in the beginning of section 6.
(3) Use the function from figure 11 to compute type error slices $\mathsf{slice}(L, \mathit{lexp})$.

It is a consequence of the following completeness theorem that slices that are computed in this way are untypable.

Theorem 16 (Completeness) *If* $(\mathit{lexp} \Downarrow \langle \cdot, \cdot, C \rangle)$, $L \in \mathsf{minErrors}(C)$ *and* $(\mathsf{slice}(L, \mathit{lexp}) \Downarrow \langle \cdot, \cdot, C' \rangle)$, *then* $L \in \mathsf{errors}(C')$.

Let $\sqsubset$ be the least contextually closed and transitive relation on slices satisfying the axioms below. Informally, $sl_1 \sqsubset sl_2$ iff sl_1 is obtained from sl_2 by masking some of sl_2's syntax nodes. We say that sl_1 is a *proper slice* of sl_2 iff $sl_1 \sqsubset sl_2$.

$$\texttt{dots}() \sqsubset x^l; \qquad \texttt{dots}(sl) \sqsubset (\texttt{fn dots() => } sl)^l;$$

$$\texttt{dots}() \sqsubset n^l; \qquad \texttt{dots}(sl_1, sl_2) \sqsubset (sl_1\ sl_2)^l;$$

$$\texttt{dots}(sl_1, sl_2) \sqsubset (sl_1 \texttt{ + } sl_2)^l;$$

$$\texttt{dots}(sl_1, sl_2) \sqsubset (\texttt{let val dots() = } sl_1 \texttt{ in } sl_2 \texttt{ end})^l;$$

$$\texttt{dots}(sls_1, sls_2, sls_3) \sqsubset \texttt{dots}(sls_1, \texttt{dots}(sls_2), sls_3)$$

The axiom $\texttt{dots}() \sqsubset x^l$ may be applied both if x^l is an expression and if x^l is a variable binder. For instance, the following three slices

$$\begin{aligned} sl_1 &= \texttt{fn x => (.. x (..) .. (..) + (..) ..)} \\ sl_2 &= \texttt{fn x => (.. x (..) ..)} \\ sl_3 &= \texttt{fn (..) => (.. x (..) .. x + (..) ..)} \end{aligned}$$

are all proper slices of sl_4.

$$sl_4 = \texttt{fn x => (.. x (..) .. x + (..) ..)}$$

Note that all of sl_1, sl_2 and sl_3 are typable, whereas sl_4 is not. In fact, *all* proper slices of sl_4 are typable — sl_4 is a minimal untypable slice. It is a consequence of the following minimality theorem that all slices that are computed by our type error slicing method are minimally untypable in this way.

Theorem 17 (Minimality) *If* $(lexp \Downarrow \langle \cdot, \cdot, C \rangle)$, $L \in \mathsf{minErrors}(C)$, *all bound variables in* $\mathsf{slice}(L, lexp)$ *are distinct and* $sl \sqsubset \mathsf{slice}(L, lexp)$, *then* sl *is typable.*

In the minimality theorem, the condition that all bound variables are distinct is needed. To see this, consider the following expression:

```
fn x => ((fn x => x 1) x (x + 1))
```

Our methods compute the following type error slice:

sl_5 = `fn x => (.. (fn x => x (..)) x .. x + (..) ..)`

However, sl_5 is not minimally untypable, because $sl_6 \sqsubset sl_5$ and sl_6 is untypable.

sl_6 = `fn x => (.. (fn (..) => x (..)) x .. x + (..) ..)`

sl_6 differs from sl_5 only because the inner variable binder has been masked. This causes the occurrence of `x` in the inner function body to now be bound to the outer variable binder. We cannot expect a minimality theorem without a precondition on distinctness of bound variables, if our definition of $\sqsubset$ allows independent masking of bound variables. We have to live with this slight cosmetic shortcoming, and we do not propose to α-convert type error slices, because this would create great confusion to programmers in most cases.

8 Conclusion and Future Work

We have introduced the notion of type error slices as sets of program points. We have defined the criteria of completeness and minimality of type error slices, and explained why these criteria are useful. We have illustrated using the output of our prototype type error slicing implementation how type error slices can be presented either by highlighting the points in the context of the full program or by presenting an incomplete program which omits program points not in the set. 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 T, and then find minimal unsolvable subsets of the set of generated constraints. We have shown that the computed type error slices are both complete and minimal.

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 is explicit type annotations. These will put natural boundaries on type error slices. For instance, if library modules are always annotated with explicit signatures (module types), then type error slices for programs that use the library will never contain parts of the library implementation.

A Completeness and Minimality

A.1 An Auxiliary Relation

In this appendix, we prove completeness and minimality of slicing, as stated in theorems 16 and 17. Both completeness and minimality would be obvious if the following were true for all label sets L:

> If $(\mathit{lexp} \Downarrow \langle \Gamma, \mathit{ty}, C \rangle)$ and $(\mathsf{slice}(L, \mathit{lexp}) \Downarrow \langle \Gamma', \mathit{ty}', C' \rangle)$, then C' contains a variant of $\Pi_L(C)$.

Unfortunately, this statement does not hold, because constraints associated with variable binders may get lost. Take, for instance, $\mathit{lexp} = (\texttt{fn}\ x^{l_1}\ \texttt{=>}\ x^{l_2})^{l_3}$ and $L = \{l_1, l_3\}$. Suppose $(\mathit{lexp} \Downarrow \langle \Gamma, \mathit{ty}, C \rangle)$ and $(\mathsf{slice}(L, \mathit{lexp}) \Downarrow \langle \Gamma', \mathit{ty}', C' \rangle)$. Then C contains a constraint labeled by l_1, saying that the type of binder x^{l_1} must equal the type of expression x^{l_2}. On the other hand, C' does not contain a constraint labeled by l_1. The key to completeness and minimality is that, if L is a minimal error, then C' will still contain all constraints that are relevant for the error: *No relevant constraints get lost when slicing by a minimal error.*

As a technical device, we introduce an auxiliary relation $\Downarrow^\bullet$, which is closely related to $\Downarrow$. It is defined in figure A.1 by stating the modifications to $\Downarrow$'s rules. If $(\mathit{lexp} \Downarrow \langle \Gamma, \mathit{ty}, C \rangle)$, then the relation $\Downarrow^\bullet$ applied to *lexp* non-deterministically generates subsets of C. Note, however, that not all subsets of C can be generated.

Lemma 18 (Key lemma for completeness and minimality) *If* $(\mathit{lexp} \Downarrow \langle \Gamma, \mathit{ty}, C \rangle)$ *and* C_{min} *is a minimally unsolvable subset of* C*, then there are* $\Gamma^\bullet, \mathit{ty}^\bullet, C^\bullet$ *such that* $(\mathit{lexp} \Downarrow^\bullet \langle \Gamma^\bullet, \mathit{ty}^\bullet, C^\bullet \rangle)$*,* $C_{\mathsf{min}} \subseteq C^\bullet$ *and* $\mathsf{labels}(C^\bullet) = \mathsf{labels}(C_{\mathsf{min}})$.

The variable rule is replaced by the following two rules:

$$\frac{}{x^l \;\Downarrow^\bullet\; \langle \mathsf{empty}, a, \{\}\rangle} \quad \text{where } a \text{ fresh}$$

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

Modified side condition in function abstraction rule:

$$\{a_x \overset{l}{=} ty \mid ty \in S\} \;\subseteq\; C_0 \;\subseteq\; \{a_x \overset{l}{=} ty \mid ty \in S\} \cup \{\; a_x \texttt{ -> } ty \;\overset{l'}{=}\; a \;\}$$

instead of

$$C_0 \;=\; \{a_x \overset{l}{=} ty \mid ty \in S\} \cup \{\; a_x \texttt{ -> } ty \;\overset{l'}{=}\; a \;\}$$

Modified side condition in let-expression rule:

$$C \;\subseteq\; C_0 \;\subseteq\; C \cup \{a \overset{l'}{=} ty_2\}$$

instead of

$$C_0 \;=\; C \cup \{a \overset{l'}{=} ty_2\}$$

Modified side condition in all other rules:

$$(C_0 \;\subseteq\; \cdots) \quad \text{instead of} \quad (C_0 \;=\; \cdots)$$

Fig. A.1. System $\Downarrow^\bullet$, the modifications to $\Downarrow$'s rules

We postpone the proof of this lemma. Let $C \lesssim C'$ iff C' has a subset that is equal to C up to renaming of type variables.

Lemma 19 (Key property of $\Downarrow^\bullet$) *Suppose* $(lexp \Downarrow^\bullet \langle \Gamma^\bullet, ty^\bullet, C^\bullet\rangle)$, $(lexp \Downarrow \langle \Gamma, ty, C\rangle)$ *and* $L = \mathsf{labels}(C^\bullet)$. *Then there exists* C' *such that* $(\mathsf{slice}(L, lexp) \Downarrow \langle \Gamma^\bullet, ty^\bullet, C'\rangle)$, $C^\bullet \subseteq C' \lesssim C$ *and* $L = \mathsf{labels}(C')$.

Proof. By induction on the structure of *lexp*. It is important that the generated environment $\Gamma^\bullet$ is the the same for *lexp* and $\mathsf{slice}(L, lexp)$. This is the reason why we can get the induction working for the variable binding constructors, i.e., function abstractions and let-expressions. □

A.2 Completeness

Theorem 20 (Completeness) *If* $(lexp \Downarrow \langle \Gamma, ty, C\rangle)$, $L \in \mathsf{minErrors}(C)$ *and* $(\mathsf{slice}(L, lexp) \Downarrow \langle \Gamma', ty', C'\rangle)$, *then* $L \in \mathsf{errors}(C')$.

Proof. Suppose $(lexp \Downarrow \langle \Gamma, ty, C\rangle)$, $L \in \mathsf{minErrors}(C)$ and $(\mathsf{slice}(L, lexp) \Downarrow \langle \Gamma', ty', C'\rangle)$. Then C has a minimal unsolvable subset C_{min} such that $L = \mathsf{labels}(C_{\mathsf{min}})$. By lemma 18, there are $\Gamma^\bullet, ty^\bullet, C^\bullet$ such that $(lexp \Downarrow^\bullet \langle \Gamma^\bullet, ty^\bullet, C^\bullet\rangle)$, $C_{\mathsf{min}} \subseteq C^\bullet$ and $L = \mathsf{labels}(C^\bullet)$. Then, by lemma 19, there is C'' such that $C^\bullet \subseteq C''$, $L = \mathsf{labels}(C'')$ and $(\mathsf{slice}(L, lexp) \Downarrow \langle \Gamma^\bullet, ty^\bullet, C''\rangle)$. Then C'' is unsolvable, because C_{min} is. But then C' is unsolvable and $L = \mathsf{labels}(C')$, because $\langle \Gamma^\bullet, ty^\bullet, C''\rangle$ and $\langle \Gamma', ty', C'\rangle$ are equal up to renaming of type variables. □

A.3 Minimality

We modify the slice order to be indexed by a finite set of variables xs, the *binder environment*. The defining rules for $\sqsubset^{xs}$ are mostly the same as the rules for $\sqsubset$, with the exception of four of the congruence rules. The congruence rules for function- and let-bodies decrement the binder environment:

$$\frac{sl \sqsubset^{x,xs} sl'}{(\texttt{fn}\ x^l\ \texttt{=>}\ sl)^k \sqsubset^{xs} (\texttt{fn}\ x^l\ \texttt{=>}\ sl')^k}$$

$$\frac{sl_2 \sqsubset^{x,xs} sl_2'}{(\texttt{let val}\ x^l\ \texttt{=}\ sl_1\ \texttt{in}\ sl_2\ \texttt{end})^k \sqsubset^{xs} (\texttt{let val}\ x^l\ \texttt{=}\ sl_1\ \texttt{in}\ sl_2'\ \texttt{end})^k}$$

The congruence rules for variable binders get an additional side condition:

$$\frac{vsl \sqsubset^{xs} x^l \quad x \notin xs}{(\texttt{fn}\ vsl\ \texttt{=>}\ sl)^k \sqsubset^{xs} (\texttt{fn}\ x^l\ \texttt{=>}\ sl)^k}$$

$$\frac{vsl \sqsubset^{xs} x^l \quad x \notin xs}{(\texttt{let val}\ vsl\ \texttt{=}\ sl_1\ \texttt{in}\ sl_2\ \texttt{end})^k \sqsubset^{xs} (\texttt{let val}\ x^l\ \texttt{=}\ sl_1\ \texttt{in}\ sl_2\ \texttt{end})^k}$$

Let $\mathsf{bv}(sl)$ denote the set of bound variables of sl.

Lemma 21 *If bound variables of* sl' *are distinct and* $sl \sqsubset sl'$, *then* $sl \sqsubset^\emptyset sl'$.

Proof. One proves the following more general statement by induction on the

derivation of $sl \sqsubset sl'$: *If bound variables of* sl' *are distinct,* $\mathsf{bv}(sl') \cap xs = \emptyset$ *and* $sl \sqsubset sl'$*, then* $sl \sqsubset^{xs} sl'$. □

Let $(\wedge S \geq \wedge S')$ iff $(S \subseteq S')$. Let $(\Gamma \geq_{xs} \Gamma')$ iff $(\Gamma(x) \geq \Gamma'(x))$ for all x in xs.

Lemma 22 (Monotonicity of $\Downarrow$) *If* $sl' \sqsubset^{xs} sl$ *and* $(sl \Downarrow \langle \Gamma, ty, C \rangle)$*, then there are* Γ', C' *such that* $(sl' \Downarrow \langle \Gamma', ty, C' \rangle)$*,* $\Gamma \geq_{xs} \Gamma'$ *and* $C' \subseteq C$*. Moreover, if* sl *does not contain immediately nested* `dots`*, then* $\mathsf{labels}(C') \neq \mathsf{labels}(C)$.

Proof. By induction on the structure of sl. □

Theorem 23 (Minimality) *If* $(lexp \Downarrow \langle \Gamma, ty, C \rangle)$*,* $L \in \mathsf{minErrors}(C)$*, bound variables in* $\mathsf{slice}(L, lexp)$ *are distinct and* $sl \sqsubset \mathsf{slice}(L, lexp)$*, then* sl *is typable.*

Proof. Suppose $(lexp \Downarrow \langle \Gamma, ty, C \rangle)$, $L \in \mathsf{minErrors}(C)$, bound variables in $\mathsf{slice}(L, lexp)$ are distinct and $sl \sqsubset \mathsf{slice}(L, lexp)$. Then there exists a minimal unsolvable subset C_{min} of C such that $L = \mathsf{labels}(C_{\mathsf{min}})$. By lemma 18, there are $\Gamma^\bullet, ty^\bullet, C^\bullet$ such that $(lexp \Downarrow^\bullet \langle \Gamma^\bullet, ty^\bullet, C^\bullet \rangle)$ and $L = \mathsf{labels}(C^\bullet)$. Then, by lemma 19, there exists C' such that $(\mathsf{slice}(L, lexp) \Downarrow \langle \Gamma^\bullet, ty^\bullet, C' \rangle)$, $C' \lesssim C$ and $L = \mathsf{labels}(C')$. Then, by lemmas 21 and 22, there are Γ', C'' such that $(sl \Downarrow \langle \Gamma', ty^\bullet, C'' \rangle)$ and $C'' \subseteq C'$. Because $\mathsf{slice}(L, lexp)$ does not contain immediately nested `dots`, $\mathsf{labels}(C'')$ is a proper subset of L. Because C'' is a variant of a subset of C, it must be solvable, by minimality of L. Then, sl is typable by (an extension to slices of) fact 2(1). □

A.4 Key Lemma for Completeness and Minimality.

In this section, we prove the key lemma 18. To this end, we define an auxiliary system that attaches stamps to constraints. Stamps are not unique. There are two instances where non-uniquely stamped constraints get introduced. Firstly, the two constraints that are associated with a variable occurrence and the corresponding variable binder have the same stamp. Secondly, in the rule for let-expressions stamps do not get refreshed, and, thus, all fresh variants of a constraint keep an identical stamp.

Let Stamp be an infinite set of *stamps* that is disjoint from all other sets in this paper. Let s range over Stamp. A *stamped environment entry* is a pair $\langle ty, s \rangle$ of a type ty and a stamp s. A *stamped intersection type* is a an object of the form $\wedge S$, where S is a finite set of stamped environment entries. A *stamped type environment* is a function from Var to the set of stamped intersection types. A *stamped constraint* is a pair $\langle c, s \rangle$ of a labeled constraint c and a stamp s. We use the meta variables Γ and C to range over stamped type environments and sets of stamped constraints. (It will always be clear from the context whether

a meta variable refers to a stamped or an unstamped object.) A *fresh variant* of a stamped triple $\langle \Gamma, ty, C\rangle$ is obtained from this triple by replacing all type variables by fresh type variables, *but keeping the stamps fixed*.

The stamped system $\Downarrow^+$ is defined in figure A.2. Note that constraints for variable binders in `fn`-abstractions and `let`-expressions inherit their stamps from the environment. Note also that in the variable axiom the constraint and the environment entry carry the same stamp.

$$
\begin{aligned}
|\wedge S| &\stackrel{\text{def}}{=\!=} \wedge\{ty \mid (\exists s)(\langle ty, s\rangle \in S)\} \\
|\Gamma|(x) &\stackrel{\text{def}}{=\!=} |\Gamma(x)| \\
|C| &\stackrel{\text{def}}{=\!=} \{c \mid (\exists s)(\langle c, s\rangle \in C)\}
\end{aligned}
$$

Lemma 24 *If* $(lexp \Downarrow \langle \Gamma, ty, C\rangle)$*, then there are* Γ'*,* C' *such that* $|\Gamma'| = \Gamma$*,* $|C'| = C$ *and* $(lexp \Downarrow^+ \langle \Gamma', ty, C'\rangle)$*.*

Proof. By induction on the derivation of $(lexp \Downarrow \langle \Gamma, ty, C\rangle)$. □

$$
\begin{aligned}
\mathsf{stamps}(\wedge S) &\stackrel{\text{def}}{=\!=} \{s \mid (\exists ty)(\langle ty, s\rangle \in S)\} \\
\mathsf{stamps}(\Gamma) &\stackrel{\text{def}}{=\!=} \textstyle\bigcup_{x\in\mathsf{Var}} \mathsf{stamps}(\Gamma(x)) \\
\mathsf{stamps}(C) &\stackrel{\text{def}}{=\!=} \{s \mid (\exists c)(\langle c, s\rangle \in C)\}
\end{aligned}
$$

Let ss range over sets of stamps.

$$
\begin{aligned}
\Pi_{ss}(\wedge S) &\stackrel{\text{def}}{=\!=} \{ty \mid (\exists s \in ss)(\langle ty, s\rangle \in S)\} \\
(\Pi_{ss}(\Gamma))(x) &\stackrel{\text{def}}{=\!=} \Pi_{ss}(\Gamma(x)) \\
\Pi_{ss}(C) &\stackrel{\text{def}}{=\!=} \{c \mid (\exists s \in ss)(\langle c, s\rangle \in S)\}
\end{aligned}
$$

Lemma 25 *If* $(lexp \Downarrow^+ \langle \Gamma, ty, C\rangle)$*, then* $(lexp \Downarrow^\bullet \langle \Pi_{ss}(\Gamma), ty, \Pi_{ss}(C)\rangle)$*.*

Proof. By induction on the derivation of $(lexp \Downarrow^+ \langle \Gamma, ty, C\rangle)$. □

Definition 26 A stamped environment entry $\langle ty, s\rangle$ or a stamped constraint $\langle c, s\rangle$ is called a *stamped item*. For a stamped item $\langle ty, s\rangle$ or $\langle c, s\rangle$, we say that the item has stamp s. A set of stamped items is called a *clique* iff all its elements have the same stamp. A clique *clq* is called *atomic* iff it has one of the following two forms:

$$
cl \;=\; \{\langle a, s\rangle,\ \langle a \stackrel{l}{=\!=} a', s\rangle\} \quad \text{or} \quad \begin{cases} cl \;=\; \{\langle a' \stackrel{l}{=\!=} a, s\rangle,\ \langle a \stackrel{l'}{=\!=} a'', s\rangle\}, \\ \text{where } a' \neq a'' \text{ and } l \neq l' \end{cases}
$$

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

$$\frac{}{n^l \;\Downarrow^+\; \langle \mathsf{empty},\; a,\; \{\langle \mathtt{int} \overset{l}{=} a, s\rangle\}\rangle} \quad \text{where } a, s \text{ fresh}$$

$$\frac{\mathit{lexp}_1 \Downarrow^+ \langle \Gamma_1, ty_1, C_1\rangle; \quad \mathit{lexp}_2 \Downarrow^+ \langle \Gamma_2, ty_2, C_2\rangle}{(\mathit{lexp}_1 \texttt{ + } \mathit{lexp}_2)^l \;\Downarrow^+\; \langle \Gamma_1 \wedge \Gamma_2,\; a,\; C_0 \cup C_1 \cup C_2\rangle}$$

where a, s_1, s_2, s_3 fresh, $C_0 = \{\langle ty_1 \overset{l}{=} \mathtt{int}, s_1\rangle,\; \langle ty_2 \overset{l}{=} \mathtt{int}, s_2\rangle,$
$\langle \mathtt{int} \overset{l}{=} a, s_3\rangle\}$

$$\frac{\mathit{lexp} \Downarrow^+ \langle \Gamma[x \mapsto \wedge S], ty,\; C\rangle}{(\texttt{fn } x^l \texttt{ => } \mathit{lexp})^{l'} \;\Downarrow^+\; \langle \Gamma[x \mapsto \wedge\{\}],\; a,\; C_0 \cup C\rangle}$$

where a_x, a, s fresh, $C_0 = \{\langle a_x \overset{l}{=} ty', s'\rangle \mid \langle ty', s'\rangle \in S\} \cup \{\; \langle a_x \texttt{ -> } ty \overset{l'}{=} a, s\rangle \;\}$

$$\frac{\mathit{lexp}_1 \Downarrow^+ \langle \Gamma_1, ty_1, C_1\rangle; \quad \mathit{lexp}_2 \Downarrow^+ \langle \Gamma_2, ty_2, C_2\rangle}{(\mathit{lexp}_1\ \mathit{lexp}_2)^l \;\Downarrow^+\; \langle \Gamma_1 \wedge \Gamma_2,\; a,\; C_0 \cup C_1 \cup C_2\;\rangle}$$

where $a, a_1, a_2, s_1, s_2, s_3$ fresh, $C_0 = \{\; \langle ty_1 \overset{l}{=} a_1 \texttt{ -> } a_2, s_1\rangle,\; \langle ty_2 \overset{l}{=} a_1, s_2\rangle,$
$\langle a \overset{l}{=} a_2, s_3\rangle \;\}$

$$\frac{\mathit{lexp}_1 \Downarrow^+ \langle \Gamma_1, ty_1, C_1\rangle; \quad \mathit{lexp}_2 \Downarrow^+ \langle \Gamma_2[x \mapsto \wedge\{\langle ty'_1, s_1\rangle, \ldots, \langle ty'_n, s_n\rangle\}],\; ty_2,\; C_2\rangle}{(\texttt{let val } x^l \texttt{ = } \mathit{lexp}_1 \texttt{ in } \mathit{lexp}_2 \texttt{ end})^{l'} \;\Downarrow^+\; \langle \Gamma'_1 \wedge \Gamma_2[x \mapsto \wedge\{\}],\; a,\; C_0 \cup C'_1 \cup C_2\rangle}$$

where $\langle \Gamma_{1,1}, ty_{1,1}, C_{1,1}\rangle, \ldots, \langle \Gamma_{1,k}, ty_{1,k}, C_{1,k}\rangle$ are fresh variants of $\langle \Gamma_1, ty_1, C_1\rangle$,
$\Gamma'_1 = \Gamma_{1,1} \wedge \ldots \wedge \Gamma_{1,k},\; C'_1 = C_{1,1} \cup \ldots \cup C_{1,k},\; k = \mathsf{max}(n, 1)$,
$C = \{\; \langle ty_{1,1} \overset{l}{=} ty'_1, s_1\rangle, \ldots, \langle ty_{1,n} \overset{l}{=} ty'_n, s_n\rangle \;\}$,
a, s fresh, $C_0 = \{\langle a \overset{l'}{=} ty_2, s\rangle\} \cup C$

Fig. A.2. Stamped system $\Downarrow^+$

The type variable a in these forms is called the *subject* of the atomic clique. A subset *maxclq* of a set of items *its* is called a *maximal clique* of *its* iff it is a clique and there is no clique that is both a subset of *its* and a proper superset of *maxclq*.

$$\begin{aligned}
\mathsf{items}(\wedge S) &\stackrel{\text{def}}{=\!=} S \\
\mathsf{items}(\Gamma) &\stackrel{\text{def}}{=\!=} \textstyle\bigcup_{x\in\mathsf{Var}} \mathsf{items}(\Gamma(x)) \\
\mathsf{items}(\Gamma, C) &\stackrel{\text{def}}{=\!=} \mathsf{items}(\Gamma) \cup C
\end{aligned}$$

Lemma 27 *Suppose that* $(\mathit{lexp} \Downarrow^+ \langle \Gamma, \mathit{ty}, C\rangle)$ *and* maxclq *is a maximal clique of* $\mathsf{items}(\Gamma, C)$ *that has at least two elements. Then there is a set* $\mathcal{A}$ *of atomic cliques such that such that* $\mathit{maxclq} = \bigcup \mathcal{A}$ *and the following statements hold for all* $\mathit{atclq}, \mathit{atclq}'$ *in* $\mathcal{A}$*:*

(1) atclq' *is a variant of* atclq*.*
(2) If a *is the subject of* atclq*, then* a *does not occur in* ty *or* $(\mathsf{items}(\Gamma, C) \setminus \mathit{atclq})$*.*

Proof. By induction on the derivation of $(\mathit{lexp} \Downarrow^+ \langle \Gamma, \mathit{ty}, C\rangle)$. □

Lemma 28 *If* $(\mathit{lexp} \Downarrow^+ \langle \Gamma, \mathit{ty}, C\rangle)$ *and* $C_{\mathsf{min}} \subseteq C$ *such that* $|C_{\mathsf{min}}|$ *is a minimally unsolvable constraint set, then* $\mathsf{labels}(\Pi_{\mathsf{stamps}(C_{\mathsf{min}})}(C)) = \mathsf{labels}(|C_{\mathsf{min}}|)$.

Proof. Let $ss = \mathsf{stamps}(C_{\mathsf{min}})$, $L = \mathsf{labels}(\Pi_{ss}(C))$ and $L' = \mathsf{labels}(|C_{\mathsf{min}}|)$. Obviously, $|C_{\mathsf{min}}| \subseteq \Pi_{ss}(C)$ and, thus, $L' \subseteq L$. We need to show that $L \subseteq L'$. To this end, let $l \in L$. Then there is a stamped constraint $sc = \langle c, s\rangle$ in C_{min}, and a stamped constraint $sc_l = \langle c_l, s\rangle$ in C such that c_l is labeled by l. Let l' denote c's label. If $l' = l$, then, obviously, $l \in L'$. So, assume that $l' \neq l$. Let maxclq be a maximal clique of C that contains both sc and sc_l. Then maxclq is of the form described in lemma 27. In particular, there is a variant $sc_{l'}$ of sc such that such that $\{sc_{l'}, sc_l\}$ is an atomic clique, whose subject, call it a, does not occur in $(C \setminus \{sc_{l'}, sc_l\})$. We claim that $sc_{l'} \in C_{\mathsf{min}}$: Assume, toward a contradiction, that $sc_{l'} \notin C_{\mathsf{min}}$. Then $|C_{\mathsf{min}} \setminus \{sc_l\}|$ is unsolvable, because $|C_{\mathsf{min}}|$ is and a does not occur in $|C_{\mathsf{min}} \setminus \{sc_l\}|$. But that contradicts minimality of $|C_{\mathsf{min}}|$. □

Proof of lemma 18. Let $(\mathit{lexp} \Downarrow \langle \Gamma, \mathit{ty}, C\rangle)$ and C_{min} be a minimally unsolvable subset of C. By lemma 24, there are stamped objects Γ', C' such that $|\Gamma'| = \Gamma$, $|C'| = C$ and $(\mathit{lexp} \Downarrow^+ \langle \Gamma', \mathit{ty}, C'\rangle)$. Let C'_{min} be a subset of C' such that $|C'_{\mathsf{min}}| = C_{\mathsf{min}}$, and let $ss = \mathsf{stamps}(C'_{\mathsf{min}})$. By lemma 25, $(\mathit{lexp} \Downarrow^\bullet \langle \Pi_{ss}(\Gamma'), \mathit{ty}, \Pi_{ss}(C')\rangle)$. Then, $C_{\mathsf{min}} = |C'_{\mathsf{min}}| = \Pi_{ss}(C'_{\mathsf{min}}) \subseteq \Pi_{ss}(C')$. Moreover, by lemma 28, $\mathsf{labels}(\Pi_{ss}(C')) = \mathsf{labels}(|C'_{\mathsf{min}}|) = \mathsf{labels}(C_{\mathsf{min}})$. □

References

[1] F. Baader, T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] M. Beaven, R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2, 1993.

[3] K. L. Bernstein, E. W. Stark. Debugging type errors (full version). Technical report, State University of New York, Stony Brook, 1995.

[4] S. Carlier, C. H. Haack, J. B. Wells. Web demo: Type error slicing, 2002. `http://www.macs.hw.ac.uk/ultra/compositional-analysis/type-error-slicin%g`.

[5] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proc. 6th Int'l Conf. Functional Programming.* ACM Press, 2001.

[6] V. Choppella. *Unification source-tracking with application to diagnosis of type inference.* PhD thesis, Indiana University, 2002.

[7] V. Choppella, C. T. Haynes. Diagnosis of ill-typed programs. Technical Report 426, Indiana University, 1995.

[8] L. Damas, R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, 1982.

[9] L. M. M. Damas. *Type assignment in Programming Languages.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1985.

[10] 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.

[11] D. Duggan, F. Bent. Explaining type inference. *Sci. Comput. Programming*, 27, 1996.

[12] 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.

[13] B. Heeren, J. Hage. Parametric type inferencing for Helium. Technical Report UU-CS-2002-035, University Utrecht, 2002.

[14] B. Heeren, J. Hage, D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, University Utrecht, 2002.

[15] 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.

[16] T. Jim. What are principal typings and what are they good for? In POPL '96 [25].

[17] G. F. Johnson, J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In POPL '96 [25].

[18] 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.

[19] A. J. Kfoury, J. Tiuryn, P. Urzyczyn. ML typability is DEXPTIME complete. In *15th Colloq. Trees in Algebra and Programming*, vol. 431 of *LNCS*. Springer-Verlag, 1990. Superseded by [20].

[20] A. J. Kfoury, J. Tiuryn, P. Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2), 1994. Supersedes [19].

[21] O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. on Prog. Langs. & Systs.*, 20(4), 1998.

[22] 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.

[23] B. J. McAdam. Generalising techniques for type debugging. In Trinder et al. [28].

[24] R. Milner, M. Tofte, R. Harper, D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[25] *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[26] G. S. Port. A simple approach to finding the cause of non-unifiability. In *Proc. Fifth International Conference on Logic Programming*. MIT Press, 1988.

[27] Z. Shao, A. Appel. Smartest recompilation. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, 1993.

[28] P. Trinder, G. Michaelson, H.-W. Loidl, eds. *Trends in Functional Programming*. Intellect, 2000. A collection of papers earlier presented at the 1999 Scottish Functional Programming Workshop.

[29] M. Wand. Finding the source of type errors. In *Conf. Rec. 13th Ann. ACM Symp. Princ. of Prog. Langs.*, 1986.

[30] D. A. Wolfram. Intractable unifiability problems and backtracking. In *Proc. Third International Conference on Logic Programming*, vol. 225 of *LNCS*, 1986.

[31] J. YANG. Explaining type errors by finding the source of a type conflict. In Trinder et al. [28].

[32] 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. This is the unreviewed draft proceedings distributed at the workshop containing all submitted papers.

[33] YANG Jun, G. Michaelson, P. Trinder. Explaining polymorphic types. *Computer Journal*, 45(4), 2002.