Skalpel: A Constraint-Based Type Error Slicer for Standard ML

Vincent Rahli[a], Joe Wells[b], John Pirie[b], Fairouz Kamareddine[b]

[a] *SnT, University of Luxembourg, Luxembourg*

[b] *Heriot-Watt University, Edinburgh, UK*

Abstract

Compilers for languages with type inference algorithms often produce confusing type error messages and give a single error location which is sometimes far away from the real location of the error. Attempts at solving this problem often (1) fail to include the multiple program points which make up the type error; (2) report tree fragments which do not correspond to any place in the user program; and (3) give incorrect type information/diagnosis which can be highly confusing. We present Skalpel, a type error slicing tool which solves these problems by giving the programmer **all and only** the information involved with a type error to significantly aid in diagnosis and repair of type errors. Skalpel relies on a simple and general constraint system, a sophisticated constraint generator which is linear in program size, and a constraint solver which is terminating. Skalpel's constraint system can elegantly and efficiently handle intricate features such as SML's *open*. We also show that the Skalpel tool is general enough to deal not only with one source code file and one single error, but highlights all and only the possible locations of the error(s) in all affected files and produces all the culprit multiple program slices.

Keywords: Constraint-based type inference, Automated type inference, Automated error diagnosis, Type error slicing, Improved error reports.

1. Introduction

Higher-order functional programming languages such as Standard ML (SML), Haskell, and OCaml rely on type systems which allow automatic type inference, freeing programmers from explicitly writing types. These type inference algorithms allow one to detect programming errors at compile time. Unfortunately, these compilers often give confusing type error reports which waste users' valuable time during error correction. We present *Skalpel*, a type error slicing tool

which helps programmers by isolating a program *slice* containing exactly the parts of the ill-typed program contributing to an error. The produced slice contains all and only the program parts related to the error.

Milner's W algorithm [18] is the original type-checking algorithm of the functional core of ML. Implementations of W generally locate errors at or near the syntax tree node being visited when unification fails, and this is unsatisfactory. Also, it blames a single abstract syntax tree node when unification fails. Following W, other algorithms try to do better by reorganizing the type-checking algorithm so that it fails at a node that is meant to be closer to the "real" error location. For example, Lee and Yi proved that the folklore algorithm M [36] finds errors "earlier" (this measure is based on the number of recursive calls of the algorithm) than W and claimed that the combination of these two algorithms "can generate strictly more informative type-error messages than either of the two algorithms alone can". Similar claims are made for W' [39] and UAE [64]. McAdam observes that W suffers a left-to-right bias and tries to eliminate it by replacing the unification algorithm used in the application case of W by another operation called *unification of substitutions*. McAdam explains that the left-to-right bias in W arises because in the case of applications, "the substitution from a left-hand subexpression is applied to the type-environment before traversing the right-hand side expression" [39]. His unification of solutions allows one "to infer types and substitutions for each subexpression independently" [39]. Unification of substitutions is then used to combine the inferred substitutions. Yang claims that UAE's primary advantage is that it also eliminates this bias. However, all the algorithms mentioned above retain a left-to-right bias in handling of let-bindings and they all blame only one syntax tree node for each type error when in fact a node set is at fault. When only one node is reported as the error site, it is often far away from the actual programming error. The situation is made worse because the blamed node depends on internal implementation details such as the order in which the abstract syntax tree is traversed. The confusion is worsened because these algorithms usually exhibit in error messages (1) an internal representation of the program subtree at the blamed location which often has been transformed substantially from what the programmer wrote, and (2) inferred type details which were not written by the programmer and which are anyway erroneous and confusing.

Since these early algorithms, many approaches have been proposed to infer types and report type errors. For example, constraint-based type inference algorithms [43, 47, 48] separate *constraint generation* and *constraint solving*. Many approaches use this idea to improve error reporting. A probably incomplete list includes [41, 25, 26, 23, 33, 32, 27, 34, 29, 52, 53, 54, 50, 44, 67, 68]. Independently from this separation, there exist other approaches toward improving errors [66]: error explanation systems [8, 20, 19, 65] which focus on explaining the reasoning steps leading to a type error, and error reporting systems [60, 57, 13] which focus on trying to precisely locate errors in pieces of code. There are also approaches that report type errors together with sugges-

tions for changes that would solve the errors [28, 38]. Some of these approaches are discussed in Sec. 8.

The problem of automatically finding type errors and of reporting possible solutions is very difficult and to solve it automatically is even more difficult. Every bit of the syntax, every part of the program must be automatically labeled for precise blaming, and in constraint based approaches, constraints need to be automatically generated and solved, and finding solutions can lead to new constraints and a combinatorial constraints size explosion.

Skalpel is based on Haack and Wells' type error slicer [25, 26], which reports all and only the program parts related to errors. Haack and Wells' system makes use of intersection types to deal with polymorphic let-bindings. Given a let-expression of the form `let val x = e in b end`, their system makes copies of the constraints generated for `e`, and this for every occurrence of `x` in `b`. Constraints associated to identifiers are accumulated using intersection types. This can therefore lead to the exponential growth of the size of the constraint term generated for an expression. Instead, to solve this issue, Skalpel adopts variants of the polymorphic let-constraints and type scheme instantiations of Pottier and Rémy [48]. These constraints "allow building a constraint of linear size" [47]. In a concern for efficiency, other error reporting system have also adopted similar constraints, such as [29].

Skalpel (and its precursor developed by Haack and Wells) differs from the above systems by reporting all and only the program parts related to errors without any bias. Skalpel attaches program points (*labels*) to generated constraints, so that when unification of two constraints fails, we can report the labels responsible for this failure, giving a full description of the error in term of program points. We annotate constraint syntax forms with these labels, written $\langle ct, \bar{l} \rangle$ to describe that the constraint term ct is involved with the set of program points $\bar{l}$. When Skalpel is asked to check a program for type errors, it runs its sophisticated constraint generator/solver. If solving the constraints fails, i.e., if there is an error in the code, Skalpel automatically decides which parts of the program are responsible for the error. Then, Skalpel generates a type error slice highlighting the minimum amount of information responsible for the type error in the code. By looking at the highlighted regions, the user can be confident that the type error can be fixed in one of the highlighted locations and that non-highlighted locations do not contribute to any error. Our contributions include the following:

- Skalpel avoids Haack and Wells' combinatorial constraint size explosion using Pottier and Rémy's-like let-constraints [48].
- Skalpel handles almost all of SML, including structures and signatures, and intricate features such as `open` (see our webpage for example for more details [5]). We achieve this with our novel hybrid constraint/environments.

- The constraint generator is linear in the size of the program and the constraint solver is terminating (Lem. 6.2 and 6.4).
- Skalpel only shows program fragments which originate from the user program.
- Skalpel shows **all** the program locations that contribute to an error.
- If the source code fed to Skalpel contains multiple separate type errors, Skalpel produces all the culprit multiple program slices (see Fig. 6).
- In order to deal with "real" size programs, the type error slices that Skalpel reports to the user may involve more than one file of source code and highlighting is given in all affected files (see Fig. 5).

Sec. 2 discusses the basic notation used. Sec. 3 provides some motivating examples. Sec. 4 informally describes the overall design of Skalpel. Sec. 6 gives the technical core of Skalpel. In particular, we discuss our new constraint representation which was vital for us overcoming the constraint size explosion challenge when dealing with an entire programming language such as SML. We show that given a program, our constraint generator produces a constraint of size linear in the size of the program. We also show that constraint solving terminates. In addition to this, we present our minimization algorithm, responsible for gathering precise errors, and our enumeration algorithm, which allows us to locate entirely distinct error slices, sometimes providing different explanations for the same programming error. We then give our representation for *type error slices*. We show that if the user's program had an error, then at the end of constraint solving, minimization and enumeration, the user is given a type error slice which contains all and only the pieces of code relevant to the error. Sec. 7 presents our design principles for Skalpel. Sec. 5 illustrates with a worked example how Skalpel automatically carries out error finding and reporting. Related work is discussed in Sec. 8. Finally, we conclude in Sec. 9.

2. Notation

Let i, j, m, n, p, q range over the set $\mathbb{N}$ of natural numbers. If v ranges over a class C (proper or small), then v_x (where x can be anything) and v', v'', etc., also range over C. Let s range over sets. If v ranges over s, then let $\overline{v}$ range over $\mathbb{P}(s)$, the power set of s. Let $\mathsf{dj}(s_1, \dots, s_n)$ ("disjoint") hold iff for all $i, j \in \{1, \dots, n\}$, if $i \neq j$ then $s_i \cap s_j = \varnothing$. Let $s_1 \uplus s_2$ be $s_1 \cup s_2$ if $\mathsf{dj}(s_1, s_2)$ and undefined otherwise. Let $(\!|x, y|\!)$ be the pair of x and y. If rel is a binary relation (a pair set), let $(x\ rel\ y)$ iff $(\!|x, y|\!) \in rel$, let the inverse of rel be rel^{-1} defined as $\{(\!|x, y|\!) \mid (\!|y, x|\!) \in rel\}$, let $\mathsf{dom}(rel) = \{x \mid (\!|x, y|\!) \in rel\}$, let $\mathsf{ran}(rel) = \{y \mid (\!|x, y|\!) \in rel\}$, let $s \lhd rel = \{(\!|x, y|\!) \in rel \mid x \in s\}$, and let $s \rhd rel = \{(\!|x, y|\!) \in rel \mid x \notin s\}$. Let f range over functions (a special case of

Figure 1 Conditionals, pattern matching, tuples (testcase 121)

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

binary relations), let $s \rightarrow s' = \{f \mid \mathsf{dom}(f) \subseteq s \wedge \mathsf{ran}(f) \subseteq s'\}$, and let $x \mapsto y$ be an alternative notation for $(\!|x, y|\!)$ used when writing some functions. Let $f_1 + f_2 = f_2 \cup (\mathsf{dom}(f_2) \triangleright f_1)$. If $f_1, f_2 \in s_1 \rightarrow \mathbb{P}(s_2)$ then let $f_1 \Cup f_2 = \{x \mapsto f_1 \cup f_2 \mid x \in \mathsf{dom}(f_1) \cap \mathsf{dom}(f_2)\} \cup \mathsf{dom}(f_2) \triangleright f_1 \cup \mathsf{dom}(f_1) \triangleright f_2$. A tuple t is a function such that $\mathsf{dom}(t) \subset \mathbb{N}$ and if $1 \leq j \in \mathsf{dom}(t)$ then $j - 1 \in \mathsf{dom}(t)$. Let t range over tuples. If v ranges over s then let $\overrightarrow{v}$ range over $\mathsf{tuple}(s) = \{t \mid \mathsf{ran}(t) \subseteq s\}$. We write the tuple $\{0 \mapsto x_0, \ldots, n \mapsto x_n\}$ as $\langle x_0, \ldots, x_n\rangle$. Let @ append tuples: $\langle x_1, \ldots, x_i\rangle @ \langle y_1, \ldots, y_j\rangle = \langle x_1, \ldots, x_i, y_1, \ldots, y_j\rangle$. Given n sets $s_1, \ldots, s_n$, let $s_1, ..., s_n$ be $\{\langle x_1, \ldots, x_n\rangle \mid \forall i \in \{1, \ldots, n\}.x_i \in s_i\}$. Note that $s_1, ..., s_n \subseteq \mathsf{tuple}(s_1 \cup \cdots \cup s_n)$. We write R^* for the reflexive transitive closure of reduction relation R.

3. Motivating Examples

This section gives examples extracted from our testcase database motivating Skalpel. Our testcase database is distributed with the packages and archives we provide [5]. Type error slices are highlighted with red. Purple and blue highlight error *end points* (sources of conflict).

3.1. Conditionals, pattern matching, records

Fig. 1 shows an untypable piece of code involving, among other things, the following derived forms: a conditional and a record selector (`# u`). Derived forms are syntactic sugar for core or module forms. For example, the conditional `if exp1 then exp2 else exp3`, is a derived form equivalent to the core expression `case exp1 of true => exp2 | false => exp3`. Suppose the *programming error* in the code presented in Fig. 1 is that we wrote `y` (the circled one at the top of Fig. 1) instead of `x`. We also call the programming error location, the *real error location*. The function `g` can be used to perform computations on integers. For example `(g true (fn x => x + 1) 2)` evaluates to `2` and `(g false (fn x => x + 1) 2)` evaluates to `3`. This piece of code is untypable because of the following reasons highlighted in Fig. 1: `y`, being a parameter of a function, has a monomorphic type; `y` is constrained to be a Boolean via the conditional; and finally, `u`'s first component is applied to `y`, where `u`'s first

component is the function f which is constrained by the two branches of the conditional to take a function as argument. SML's compiler SML/NJ (v.110.72) [3] reports a type constructor clash in line 6 (more precisely, the circled portion of code (#1 u) y in Fig. 1 is blamed) as follows:

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

Because of the small size of the piece of code, the programmer's error is not too far away from the location reported by SML/NJ. It is not always the case. The real error location might even be in another file. Nonetheless, note that SML/NJ reports only one location which is far from the real error location w.r.t. the size of the piece of code. Also, the type 'Z -> 'Z reported by SML/NJ is an internal type made up during type inference. Finally, the reported expression does not match the source code[1].

Fig. 1 highlights a slice for the type error described above. This highlighting contains the minimal amount of information necessary to understand and fix the type error. Also, it highlights the real error location. Note that the fact that most of the piece of code is highlighted is due to the small size of the piece of code. We present below larger examples where a smaller percentage of the pieces of code is highlighted[2].

3.2. Datatypes, pattern matching, type functions

Fig. 2 shows how Skalpel helps for intricate errors. The code declares the datatype t and the function trans to deal with user defined colors. This function is then applied to an instance of a colour (the first element of the pair x). Suppose the programming error is that we wrote 'b instead of 'c in Green's definition at location ①. SML/NJ (v.110.72) reports a type constructor clash

[1] SML/NJ has transformed the code because the derived form #1 is equivalent to the function (fn {1=y,...} => y) in SML. Note also that (fn {1=<pat>,...} => 1) is SML/NJ's pretty printing of #1, but the two functions are different because (fn {1=<pat>,...} => 1) returns always 1 while #1 takes a record and returns the field of field name 1 in the record, which is confusing. SML's compilers MLton [1] and Poly/ML [6] do not transform the code.

[2] A slice for a type error will always contain exactly the portion of the program required to explain the error. We do not influence how much or how little of a piece of code is included in a type error slice. The type error itself decides which parts are included. In our experience in using Skalpel, the size of slices does not vary much depending on the size of the program but it varies mainly depending on the kind of error.

Figure 2 Datatypes, pattern matching, type functions (testcase 114)

```
datatype ('a,'b,'c) t = Red    of 'a * 'b * 'c
                      | Blue   of 'a * 'b * 'c
                      | Pink   of 'a * 'b * 'c
                      | Green  of 'a * 'b * 'b①
                      | Yellow of 'a * 'b * 'c
                      | Orange of 'a * 'b * 'c
fun trans (Red   (x, y, z)) = Blue  (y, x, z)
  | trans (Blue  (x, y, z)) = Pink  (y, x, z)
  | trans (Pink  (x, y, z)) = Green (y, x, z)
  | trans (Green (x,②y, z)) = Yellow(y, x, z)③
  | trans (Yellow(x, y, z)) = Orange(y, x, z)
  | trans (Orange(x, y, z)) = Red   (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t *⑤'b
val x = (Red (2, 2, false), true)
val y : (int, bool) u = (trans (#1 x)④, #2 x)
```

at ④ as follows:

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

The reported code is far from the actual error and does not match the source code. SML/NJ gives the same error message if, instead of the error described above, one writes `x` instead of `z` in the right-hand-side of any branch of `trans`. Thus, one might need to inspect the entire program to find the error.

Fig. 2 highlights a slice for this error. The programming error location being in the slice, we track it down by considering only the highlighted code, starting from the clashing types on the last line. The type annotation `(int, bool) u` constrains the result type of `trans`'s application. The part of `trans` in the slice is about `Green`. At ①, `Green`'s second and third arguments are constrained to be of the same type. At ②, `y` is therefore constrained to be of the same type as `z`. At ③, because `y` and `z` are respectively `Yellow`'s first and third arguments and using `Yellow`'s definition, we infer that the type of `Yellow`'s application to its three arguments (returned by `trans`) is `t` where its first and third parameters have to be equal. At ④ and ⑤ we can see that `trans` is constrained to return a `t` where its first (`int`) and third (`bool`) parameters differ.

3.3. Chained opens *and nested structures*

The most challenging feature for full SML was the `open` declaration, which splices another structure into the current environment (see example below), and has been criticized in the literature [7, 9, 10, 30]. Harper writes [30]: "*it is*

Figure 3 Chained *opens* and nested structures (testcase 450)

```
structure S = struct
  structure Y = struct
    structure A = struct val x = false end
    structure X = struct val x = false end
    structure M = struct val x = true end
  end
  open Y
  val m = M.x
  val x = if m then true else false
end
structure T = struct
  structure X = struct val x = 1 end
  open S
  open X
  val y = if m then 1 else x
end
```

hard to control its behaviour, since it incorporates the entire body of a structure, and hence may inadvertently shadow identifiers that happen to be also used in the structure". Blume [9] shows that certain automatic dependency analyses become NP-complete in the presence of `open`, and writes: "*Programs are not only read by analysis tools; human read them as well. A language construct like open that serves to confuse the analysis tool is also likely to confuse the human reader*". We believe `open` is one of the most difficult programming language features to analyze, but our constraint/environments make it easy and simple, and we believe this highlights the generality of our machinery. Skalpel clarifies otherwise obscure type errors involving `open` and enhances its usability.

Fig. 3 has an intricate type error involving chained `open`s. Let us describe what the code was meant to do. Structure `T` declares structure `X` declaring integer `x`. Structure `S` is opened to access the Boolean `m`. Then, `X` is opened to access the integer `x`. Finally, if `m` is true then we return `1` otherwise we return `x`. This is untypable and SML/NJ blames `y`'s body as follows:

```
Error:  types of if branches do not agree [literal]
  then branch:  int
  else branch:  bool
  in expression:
    if m then 1 else x
```

The programming error, as our type error slice shows, is that opening `S` causes `S`'s declarations to shadow the current typing environment. Because `Y` is opened in `S`, the structures `A`, `X` and `M` are part of `S`'s declarations. Hence, when opening `S` in `T`, the structure `X`, which was in our current typing environment, is shadowed by the one defined in `Y`. If the programmer's intent is as described above, one can solve this error by replacing "`open S open X`" by "`open S X`",

Figure 4 Type error slice highlighted using our Emacs interface (testcase 1)

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

fun find_best weight lists =
    let val average1 = average weight
        fun iterator (list,(best,max)) =
            let val avg_list = average1 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;
-:---  code1.sml       All (1,0)     Git-master  (SML) 21:34
Error (click to toggle associated info):
Constant 1 overloaded to the overloading class Int not including arrow

Slice:

〈..fun 〈..〉 〈..〈..average weight..=
                                          〈..weight 〈..〉..〉..〉..〉
 ..fun 〈..〉 〈..〈..find_best weight..=
                               〈..average weight..〉..〉..〉
                         ..find_best 1..〉
```

which opens `S` and `X` simultaneously. Opening `X` results in the opening of `X` declared in `T` because it is not shadowed by the one declared in `Y` declared in `S`.

Our type error slice rules out `x`'s declarations in `X` and `S`, and clearly shows why `x` does not have the expected type. Traditional reports leave us to track down `x`'s binding by hand.

3.4. Emacs interface

Fig. 4 shows an example of Skalpel running with our Emacs user interface. The type error slice is at the bottom of the figure (bottom Emacs buffer), and the error locations found by Skalpel are highlighted in the original user code at the top of the figure (top Emacs buffer).

Figure 5 Example of Skalpel handling multiple source files

```
datatype t = C                          val (C x) = 2
                                        fun C () = ();

-:@---  file1.sml      All (2,0) -:@---  file2.sml      All (3,0)
[Skalpel: unification...]
Non applied constructor that is defined to take an argument

Slice in context:

/home/jpirie/mulitple-files-test/file1.sml:
1:                datatype t = C
/home/jpirie/mulitple-files-test/file2.sml:
1:                val (C x) = 2

U:@**-  *zsh-1*        84% (24,25)    (Term: line run) 18:10
```

3.5. Multiple files

Fig. 5, shows an example of how Skalpel deals with errors involving multiple files. We have a special control file, which we call a *tes* file. This file contains all of the source files the user wishes to run Skalpel on, in the order they should be run. Fig. 5 shows Skalpel's output when run on `files.tes` which contains two lines: `file1.sml` and `file2.sml`. Users can use such files to run Skalpel on their entire codebase.

Fig. 6 shows that Skalpel can show multiple program slices to the user which exist in the same source code file. We do this by highlighting only one at a time, leaving any additional slices in a different colour to indicate they are not highlighted. To the right hand side of the image, we can see that we also present each program slice to the user as a formal type error slice as described in Sec. 6.7 below.

4. Skalpel's Design

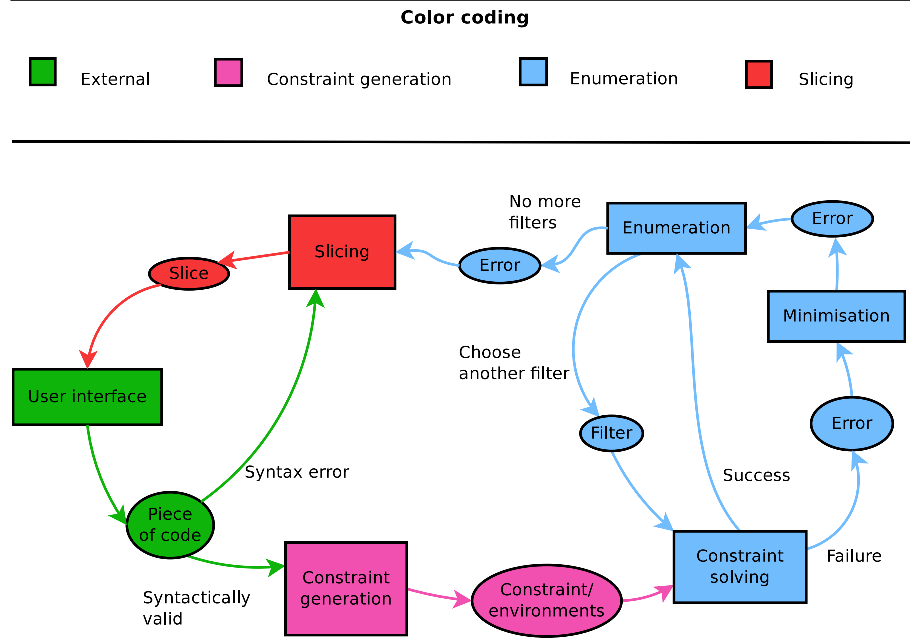
Fig. 7 informally presents how the different modules of Skalpel interact with each other. We use different colors to differentiate the different components of Skalpel. The green parts are user interface related. The red parts are related to slicing. The purple parts are related to constraint generation. These parts are external language related. The blue parts are related to the enumeration of type errors. These parts are external language unrelated. Boxes represent algorithms and ovals represent data.

Given an SML structure declaration *strdec* (see Sec. 6.1), the initial con-

Figure 6 Example of Skalpel showing multiple slices

```
let
    fun f (x ::_) (y ::_) z = (z x, z y)
in f [1, 2] [(), 4]
end;

-:**-  code6.sml                    (5,0)    Git-equality-
Error (click to toggle associated info):
Constant 1 overloaded to the overloading class Int not including record

Slice:

⟨..fun ⟨..⟩ ⟨..⟨..f (x :: ⟨..⟩) (y :: ⟨..⟩)..⟨..z..⟩..= ⟨..z x..z y..⟩..⟩..⟩
    ..f [1,⟨..⟩] [(),⟨..⟩]..⟩

Error (click to toggle associated info):
Constant 4 overloaded to the overloading class Int not including record

Slice:

⟨..[(),4]..⟩
```

Figure 7 Overall design of Skalpel



straint generator[3] defined in Sec. 6.4 generates a constraint/environment e (see Sec. 6.2). Then, the type error enumerator defined in Sec. 6.6.5 enumerates the type errors of e. Each error found by the enumerator is minimized by the minimizer defined in Sec. 6.6.3. Enumerating all errors in a piece of code is an iterative process: see the blue loops in Fig. 7. From each minimized error and *strdec*, the slicing algorithm defined in Sec. 6.7 computes a type error slice. Both enumeration and minimization rely on the constraint solver defined in Sec. 6.5. The computed type error slices are finally reported to the user. A type error report includes a type error slice, a highlighting of the slice directly in the SML user code, and a message explaining the kind of the error. Formally, our overall algorithm skalpel is defined as follows (the undefined relations, functions, and other syntactic forms used in this definition of Skalpel' overall algorithm are all defined in the remaining sections):

$$\begin{array}{lll} \mathsf{skalpel}(\mathit{strdec}) = \{\langle \mathit{strdec}', \mathit{ek}\rangle & | & [\![\mathit{strdec}]\!] = e \\ & \wedge & \mathtt{enum}(e) \rightarrow^{*}_{\mathsf{e}} \mathtt{errors}(\overline{\mathit{er}} \cup \{\langle \mathit{ek}, \overline{l}\rangle\}) \\ & \wedge & \mathsf{sl}(\mathit{strdec}, \overline{l}) = \mathit{strdec}'\} \end{array}$$

We define our constraint generator and our slicing algorithm as deterministic total recursive functions, while our constraint solver, minimizer and enumerator, which are deterministic partial recursive functions in our implementation, are defined here as non-deterministic relations presented as rewriting systems. We do so in order to avoid providing implementation details that would complicate the presentation of these algorithms. For example, our constraint solver non-deterministically generates "fresh" renamings in rule (A1) in Fig. 22, and we thus do not have to provide a deterministic algorithm that generates such "fresh" renamings.

5. Worked Example

We now illustrate how Skalpel works using the piece of code (EX1) defined below. This section uses the labeled syntax introduced in Sec. 6.1, and refers to our constraint generation (see Sec. 6.4 below), constraint solving (see Sec. 6.5 below), enumeration (see Sec. 6.6 below), and slicing (see Sec. 6.7 below) algorithms all defined below in the technical Sec. 6. This piece of code is untypable because f is defined as taking a `'a t` and is applied to a `'a u`. The labelled version of this piece of code, which we call $\mathit{strdec}_{\mathrm{EX}}$, is as follows:

[3]We sometimes refer to our constraint generation algorithm as the "initial constraint generator" in order to distinguish with our constraint solver, which also generates constraints. We will sometimes refer to our initial constraint generator simply as our *constraint generator*.

$$
\text{(EX1)}\qquad
\begin{array}{ll}
\mathtt{structure}\ \mathtt{X} \stackrel{\underline{l_1}}{=} & \mathtt{struct}^{l_2} \\
 & \quad \mathtt{structure}\ \mathtt{S} \stackrel{\underline{l_3}}{=} \mathtt{struct}^{l_4}\ \mathtt{datatype}\ \lceil \mathtt{'a\ u} \rceil^{l_6} \stackrel{\underline{l_5}}{=} \mathtt{U}_{\mathtt{c}}^{l_7}\ \mathtt{end} \\
 & \quad \mathtt{datatype}\ \lceil \mathtt{'a\ t} \rceil^{l_9} \stackrel{\underline{l_8}}{=} \mathtt{T}_{\mathtt{c}}^{l_{10}} \\
 & \quad \mathtt{val\ rec}\ \mathtt{f}_{\mathtt{p}}^{l_{12}} \stackrel{\underline{l_{11}}}{=} \mathtt{fn}\ \mathtt{T}_{\mathtt{p}}^{l_{14}} \stackrel{\underline{l_{13}}}{\Rightarrow} \mathtt{T}_{\mathtt{e}}^{l_{15}} \\
 & \quad \mathtt{val\ rec}\ \mathtt{g}_{\mathtt{p}}^{l_{17}} \stackrel{\underline{l_{16}}}{=} \mathtt{let}^{l_{18}}\ \mathtt{open}^{l_{19}}\ \mathtt{S}\ \mathtt{in}\ \lceil \mathtt{f}_{\mathtt{e}}^{l_{21}}\ \mathtt{U}_{\mathtt{e}}^{l_{22}} \rceil^{l_{20}}\ \mathtt{end} \\
 & \mathtt{end}
\end{array}
$$

5.1. Constraints generated for (EX1)

The environment generated for `datatype 'a u = U`, which we call e_0, is as follows:

$$
\begin{array}{rcl}
e_0 & = & \exists \langle \alpha_1, \alpha_2, ev_4 \rangle.((ev_4 = ((\alpha_1 \stackrel{\underline{l_5}}{=} \alpha_2);e_0'';e_0'));ev_4^{l_5}) \\
 & & \text{such that} \left\{ \begin{array}{lll} e_0' & \text{is} & \mathtt{poly}(\downarrow\mathtt{U} \stackrel{\underline{l_7}}{=} \alpha_2) \\ e_0'' & \text{is} & \exists \langle \alpha_1', \gamma_1 \rangle.(\alpha_1 \stackrel{\underline{l_6}}{=} \alpha_1'\ \gamma_1);(\downarrow\mathtt{u} \stackrel{\underline{l_6}}{=} \gamma_1);(\downarrow\mathtt{'a} \stackrel{\underline{l_6}}{=} \alpha_1') \end{array} \right.
\end{array}
$$

The environment generated for `structure S = struct datatype 'a u = U end`, which we call e_1, is as follows:

$$
e_1 \;=\; \exists \langle ev_1, ev_2 \rangle.[\exists \langle ev_3 \rangle.((ev_2 \stackrel{\underline{l_4}}{=} ev_3);(ev_3 = e_0))];(ev_1 = (\downarrow\mathtt{S} \stackrel{\underline{l_3}}{=} ev_2));ev_1^{l_3}
$$

The environment generated for `datatype 'a t = T`, which we call e_2, is as follows:

$$
\begin{array}{rcl}
e_2 & = & \exists \langle \alpha_3, \alpha_4, ev_5 \rangle.((ev_5 = ((\alpha_3 \stackrel{\underline{l_8}}{=} \alpha_4);e_2'';e_2'));ev_5^{l_8}) \\
 & & \text{such that} \left\{ \begin{array}{lll} e_2' & \text{is} & \mathtt{poly}(\downarrow\mathtt{T} \stackrel{\underline{l_{10}}}{=} \alpha_4) \\ e_0'' & \text{is} & \exists \langle \alpha_3', \gamma_2 \rangle.(\alpha_3 \stackrel{\underline{l_9}}{=} \alpha_3'\ \gamma_2);(\downarrow\mathtt{t} \stackrel{\underline{l_9}}{=} \gamma_2);(\downarrow\mathtt{'a} \stackrel{\underline{l_9}}{=} \alpha_3') \end{array} \right.
\end{array}
$$

The environment generated for `val rec f = fn T => T`, which we call e_3, is as follows:

$$
\begin{array}{rcl}
e_3 & = & \exists \langle \alpha_5, \alpha_6, ev_6 \rangle.(ev_6 = \mathtt{poly}((\downarrow\mathtt{f} \stackrel{\underline{l_{12}}}{=} \alpha_5);e_3';(\alpha_5 \stackrel{\underline{l_{11}}}{=} \alpha_6)));ev_6^{l_{11}} \\
 & & \text{such that } e_3' = [\exists \langle \alpha_7, \alpha_8, ev_7 \rangle.(ev_7 = (\uparrow\mathtt{T} \stackrel{\underline{l_{14}}}{=} \alpha_7));ev_7^{l_{13}};(\uparrow\mathtt{T} \stackrel{\underline{l_{15}}}{=} \alpha_8);(\alpha_6 \stackrel{\underline{l_{13}}}{=} \alpha_7 \to \alpha_8)]
\end{array}
$$

The environment generated for `val rec g = let open S in f U end`, which we call e_4, is as follows:

$$
\begin{array}{rcl}
e_4 & = & \exists \langle \alpha_9, \alpha_{10}, ev_8 \rangle.(ev_8 = \mathtt{poly}((\downarrow\mathtt{g} \stackrel{\underline{l_{17}}}{=} \alpha_9);[\exists \alpha_{11}.e_4';e_4'';(\alpha_{10} \stackrel{\underline{l_{18}}}{=} \alpha_{11})];(\alpha_9 \stackrel{\underline{l_{16}}}{=} \alpha_{10})));ev_8^{l_{16}} \\
 & & \text{such that} \left\{ \begin{array}{lll} e_4' & \text{is} & \exists ev_9.(\uparrow\mathtt{S} \stackrel{\underline{l_{19}}}{=} ev_9);ev_9^{l_{19}} \\ e_4'' & \text{is} & \exists \langle \alpha_{12}, \alpha_{13} \rangle.(\uparrow\mathtt{f} \stackrel{\underline{l_{21}}}{=} \alpha_{12});(\uparrow\mathtt{U} \stackrel{\underline{l_{22}}}{=} \alpha_{13});(\alpha_{12} \stackrel{\underline{l_{20}}}{=} \alpha_{13} \to \alpha_{11}) \end{array} \right.
\end{array}
$$

Finally, the environment generated for the entire piece of code is the following environment, which we call e_{EX}:

$$e_{\text{EX}} = \exists\langle ev_{11}, ev_{10}\rangle. [\exists\langle ev_{12}\rangle.(ev_{11} \stackrel{l_2}{=} ev_{12});(ev_{12} = (e_1;e_2;e_3;e_4))];(ev_{10} = (\downarrow\mathtt{X} \stackrel{l_1}{=} ev_{11}));ev_{10}^{l_1}$$

5.2. Solving of the constraint generated for (EX1)

We present below how our constraint solver solves the environment $(e_1;e_2;e_3;e_4)$, which is part of e_{EX}. First, let us solve e_1, which was generated for the declaration `structure S = struct datatype 'a u = U end`. The solved version (see Sec. 6.5.1) of e_1 is as follows:

$$ev_1^{l_3} \text{ such that } \begin{cases} ev_1 \mapsto \downarrow\mathtt{S} \stackrel{l_3}{=} ev_2 \\ ev_2 \mapsto ev_3^{l_4} \\ ev_3 \mapsto ((\downarrow\mathtt{u} \stackrel{l_6}{=} \gamma_1);(\downarrow\mathtt{'a} \stackrel{l_6}{=} \alpha_1');(\downarrow\mathtt{U} \stackrel{l_7}{=} \forall\{\alpha_1'\}.\,(\alpha_1'\,\gamma_1)^{\{l_5,l_6\}}))^{l_5} \end{cases}$$

As mentioned in the "Recursive datatypes" paragraph in Sec. 6.4.1, ev_3 is mapped to an environment containing a binder for `'a` because we have not yet introduced any mechanism to partially export environments (we want a mechanism other than $e_1;e_2$ that exports, e.g., the binders of e_1 but not those of e_2). This issue is resolved in [49] using local environments.

Let us now solve e_2, which was generated for `datatype 'a t = T`. The solved version of e_2 is as follows:

$$ev_5^{l_8} \text{ such that } ev_5 \mapsto (\downarrow\mathtt{t} \stackrel{l_9}{=} \gamma_2);(\downarrow\mathtt{'a} \stackrel{l_9}{=} \alpha_3');(\downarrow\mathtt{T} \stackrel{l_{10}}{=} \forall\{\alpha_3'\}.\,(\alpha_3'\,\gamma_2)^{\{l_8,l_9\}})$$

We now solve e_3, which was generated for `val rec f = fn T => T`. The solved version of e_3 is as follows:

$$ev_6^{l_{11}} \text{ such that } \begin{cases} ev_6 \mapsto (\downarrow\mathtt{f} \stackrel{l_{12}}{=} \forall\{\alpha_3'', \alpha_3'''\}.\,\tau_1) \\ \tau_2 = ((\alpha_3''\,\gamma_2)^{\{l_8,l_9,l_{10},l_{14}\}}) \\ \tau_3 = ((\alpha_3'''\,\gamma_2)^{\{l_8,l_9,l_{10},l_{15}\}}) \\ \tau_1 = ((\tau_2 \to \tau_3)^{\{l_{11},l_{13}\}}) \end{cases}$$

Note that in the binder generated at constraint solving for `f`, l_{15} only labels $(\alpha_3'''\,\gamma_2)$ and does not label the whole binder. Having dependencies on types as well as on environments allows a precise blaming (dependency tracking).

We now solve e_4, which was generated for `val rec g = let open S in f U end`. We start by solving e_4'. Its solved version is as follows:

$$ev_9^{l_{19}} \text{ such that } ev_9 \mapsto ev_2^{\{l_3,l_{19}\}}$$

Then, we solve e_4''. The dependent accessor $(\uparrow\mathtt{f} \stackrel{l_{21}}{=} \alpha_{12})$ accesses f's binder through ev_6. It leads to the generation of the following mapping:

$$\alpha_{12} \mapsto ((\alpha_4''\,\gamma_2)^{\{l_8,l_9,l_{10},l_{14}\}} \to (\alpha_4'''\,\gamma_2)^{\{l_8,l_9,l_{10},l_{15}\}})^{\{l_{11},l_{12},l_{13},l_{21}\}}$$

The dependent accessor $(\uparrow\mathtt{U} \stackrel{l_{22}}{=} \alpha_{13})$ accesses U's binder through ev_9, ev_2, and ev_3. It leads to the generation of the following mapping:

$$\alpha_{13} \mapsto (\alpha_1''\,\gamma_1)^{\{l_3,l_4,l_5,l_6,l_7,l_{19},l_{22}\}}$$

Finally, our constraint solver returns a type error (terminates in an error state) when dealing with the equality constraint $(\alpha_{12} \stackrel{l_{20}}{=} \alpha_{13} \to \alpha_{11})$, because $\gamma_1 \neq \gamma_2$. The error found is $er_{\mathrm{EX}} = \langle ek_{\mathrm{EX}}, \bar{l}_{\mathrm{EX}}\rangle$, where $ek_{\mathrm{EX}} = \mathtt{clash}(\gamma_1,\gamma_2)$ and $\bar{l}_{\mathrm{EX}} = \{l_3,l_4,l_5,l_6,l_7,l_8,l_9,l_{10},l_{11},l_{12},l_{13},l_{14},l_{19},l_{20},l_{21},l_{22}\}$.

5.3. Enumerating all the errors in example (EX1)

It turns out that (EX1) has only one minimal type error which is er_{EX}. This error is already minimal when found by the enumeration algorithm and therefore the minimiser does not do anything in this case, but is still called by the enumerator. Therefore we obtain the following enumeration steps (we superscript $\to_{\mathsf{e}}$ and $\to_{\mathsf{e}}^*$ with the names of the rules used to obtain the provided enumeration steps):

$$\begin{array}{ll}
 & \mathtt{enum}(e_{\mathrm{EX}}) \\
\to_{\mathsf{e}}^{(\mathsf{ENUM1})} & \mathtt{enum}'(e_{\mathrm{EX}},\varnothing,\{\varnothing\}) \\
\to_{\mathsf{e}}^{(\mathsf{ENUM4})} & \mathtt{enum}'(e_{\mathrm{EX}},\{er_{\mathrm{EX}}\},\{\{l\} \mid l \in \bar{l}_{\mathrm{EX}}\}) \\
\to_{\mathsf{e}}^{*(\mathsf{ENUM3})} & \mathtt{enum}'(e_{\mathrm{EX}},\{er_{\mathrm{EX}}\},\varnothing) \\
\to_{\mathsf{e}}^{(\mathsf{ENUM2})} & \mathtt{errors}(\{er_{\mathrm{EX}}\})
\end{array}$$

5.4. Generating type error slices for (EX1)

We saw in Sec. 5.1 that given (EX1), our constraint generator generates the environment e_{EX}. We saw in Sec. 5.3 that given e_{EX}, our enumeration algorithm enumerates only one error, namely er_{EX}. Sec. 5.2 defines $er_{\mathrm{EX}} = \langle ek_{\mathrm{EX}}, \bar{l}_{\mathrm{EX}}\rangle$ where $\bar{l}_{\mathrm{EX}} = \{l_3,l_4,l_5,l_6,l_7,l_8,l_9,l_{10},l_{11},l_{12},l_{13},l_{14},l_{19},l_{20},l_{21},l_{22}\}$. Let us present the slice that our slicing algorithm computes when given er_{EX}, i.e., we compute $\mathsf{sl}(strdec_{\mathrm{EX}}, \bar{l}_{\mathrm{EX}})$.

Figure 8 Result of applying toTree to $strdec_{\mathrm{EX}}$

$$\langle\langle \mathtt{strdec}, \mathtt{strdecStr}\rangle, l_1, \langle \mathtt{X}, \langle\langle \mathtt{strexp}, \mathtt{strexpSt}\rangle, l_2, \langle tree_1, tree_2, tree_3, tree_4\rangle\rangle\rangle\rangle$$

where

$$\begin{array}{ll}
tree_1 = & \langle\langle \mathtt{strdec}, \mathtt{strdecStr}\rangle, l_3, \\
& \quad \langle \mathtt{S}, \\
& \quad\quad \langle\langle \mathtt{strexp}, \mathtt{strexpSt}\rangle, l_4, \\
& \quad\quad\quad \langle\ \langle\ \langle \mathtt{dec}, \mathtt{decDat}\rangle, l_5, \\
& \quad\quad\quad\quad\quad \langle\langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l_6, \langle \mathtt{'a}, \mathtt{u}\rangle\rangle, \langle\langle \mathtt{conbind}, \mathtt{id}\rangle, l_7, \langle \mathtt{U}\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle \\
tree_2 = & \langle\langle \mathtt{dec}, \mathtt{decDat}\rangle, l_8, \langle\langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l_9, \langle \mathtt{'a}, \mathtt{t}\rangle\rangle, \langle\langle \mathtt{conbind}, \mathtt{id}\rangle, l_{10}, \langle \mathtt{T}\rangle\rangle\rangle\rangle \\
tree_3 = & \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l_{11}, \\
& \quad \langle\langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l_{12}, \langle \mathtt{f}\rangle\rangle, \\
& \quad \langle\langle \mathtt{exp}, \mathtt{expFn}\rangle, l_{13}, \langle\langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l_{14}, \langle \mathtt{T}\rangle\rangle, \langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l_{15}, \langle \mathtt{T}\rangle\rangle\rangle\rangle\rangle\rangle \\
tree_4 = & \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l_{16}, \\
& \quad \langle\langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l_{17}, \langle \mathtt{g}\rangle\rangle, \\
& \quad\quad \langle\langle \mathtt{atexp}, \mathtt{atexpLet}\rangle, l_{18}, \\
& \quad\quad\quad \langle\langle\langle \mathtt{dec}, \mathtt{decOpn}\rangle, l_{19}, \langle \mathtt{S}\rangle\rangle, \\
& \quad\quad\quad \langle\langle \mathtt{exp}, \mathtt{app}\rangle, l_{20}, \langle\langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l_{21}, \langle \mathtt{f}\rangle\rangle, \langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l_{22}, \langle \mathtt{U}\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle
\end{array}$$

Fig. 8 shows the tree (which we call $tree_{\mathrm{EX}}$) obtained when applying toTree to $strdec_{\mathrm{EX}}$. Finally, $\mathsf{sl}(\mathsf{toTree}(strdec_{\mathrm{EX}}), \bar{l}_{\mathrm{EX}})$ returns the following tree where $tree_1$ and $tree_2$ are the ones defined above, and $tree'_3$ and $tree'_4$, are obtained from $tree_3$ and $tree_4$ respectively:

$$\langle \mathtt{dotD}, \langle tree_1, tree_2, tree'_3, tree'_4\rangle\rangle$$

$$\begin{array}{lll}
\text{where} & tree'_3 = & \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l_{11}, \\
& & \quad \langle\langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l_{12}, \langle \mathtt{f}\rangle\rangle, \\
& & \quad \langle\langle \mathtt{exp}, \mathtt{expFn}\rangle, l_{13}, \langle\langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l_{14}, \langle \mathtt{T}\rangle\rangle, \langle \mathtt{dotE}, \langle\rangle\rangle\rangle\rangle\rangle\rangle \\
& tree'_4 = & \langle \mathtt{dotE},\ \ \langle\langle\langle \mathtt{dec}, \mathtt{decOpn}\rangle, l_{19}, \langle \mathtt{S}\rangle\rangle, \\
& & \quad\quad \langle\langle \mathtt{exp}, \mathtt{app}\rangle, l_{20}, \langle\langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l_{21}, \langle \mathtt{f}\rangle\rangle, \langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l_{22}, \langle \mathtt{U}\rangle\rangle\rangle\rangle\rangle\rangle
\end{array}$$

This slice is displayed as follows:

```
⟨..structure S = struct datatype 'a u = U end
 ..datatype 'a t = T
 ..val rec f = fn T => ⟨..⟩
 ..⟨..open S..f U..⟩..⟩
```

6. Technical Core of Skalpel

We refer to the system which is defined in this section as the *Skalpel core*, comprising, as mentioned in Sec. 4, its constraint generator, constraint solver, minimizer, enumerator, and slicer which are all defined in this section. Sec. 5

Figure 9 External labeled syntax: SML's subset handled by Skalpel core

l	$\in$	Label		(labels)
$\mathcal{P}$	$\in$	ExtLabSynt	$=$	The union of all sets below.
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
vid	$\in$	VId	$::=$	$vvar \mid dcon$
ltc	$\in$	LabTyCon	$::=$	tc^l
$ldcon$	$\in$	LabDatCon	$::=$	$dcon^l$
ty	$\in$	Ty	$::=$	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	$::=$	$dcon^l_{\mathtt{c}} \mid dcon\ \mathtt{of}^l\ ty$
dn	$\in$	DatName	$::=$	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	$::=$	$\mathtt{val\ rec}\ pat \overset{l}{=} exp \mid \mathtt{open}^l\ strid \mid \mathtt{datatype}\ dn \overset{l}{=} cb$
$atexp$	$\in$	AtExp	$::=$	$vid^l_{\mathtt{e}} \mid \mathtt{let}^l\ dec\ \mathtt{in}\ exp\ \mathtt{end}$
exp	$\in$	Exp	$::=$	$atexp \mid \mathtt{fn}\ pat \overset{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l$
$atpat$	$\in$	AtPat	$::=$	$vid^l_{\mathtt{p}}$
pat	$\in$	Pat	$::=$	$atpat \mid \lceil ldcon\ atpat \rceil^l$
$strdec$	$\in$	StrDec	$::=$	$dec \mid \mathtt{structure}\ strid \overset{l}{=} strexp$
$strexp$	$\in$	StrExp	$::=$	$strid^l \mid \mathtt{struct}^l\ strdec_1 \cdots strdec_n\ \mathtt{end}$

extra metavariables

id	$\in$	Id	$::=$	$vid \mid strid \mid tv \mid tc$
$term$	$\in$	Term	$::=$	$ltc \mid ldcon \mid ty \mid cb \mid dn \mid exp \mid pat \mid strdec \mid strexp$

illustrates how these algorithms compute on a simple program. The reader might want to look at that section whenever we introduce a new algorithm. Also, we sometimes forward reference our design principles presented in Sec. 7 below. Whenever we mention these principles, the reader might want to jump to that section for more information regarding choices we have made while designing Skalpel.

6.1. External labeled syntax

Fig. 9 describes a subset of SML, chosen to present Skalpel's core ideas. We refer to this language as the external labeled syntax.[4] Most syntactic forms have labels (l), which are generated to track precise blame for errors. We surround some terms such as function application with $\lceil\ \rceil$ in order to provide a visually convenient place for labels.

[4] We do not enforce all the syntactic restrictions of the SML syntax. For example, in a recursive declaration such as $\mathtt{val\ rec}\ pat \overset{l}{=} exp$, the expression exp must be an `fn`-expression, which we do not enforce in this paper. More details can be found in [49].

Figure 10 Syntax of constraint terms (internal labeled syntax)

$\mathcal{C}$	$\in$	IntLabSynt	$=$ The union of all sets below and Label.
ev	$\in$	EnvVar	(environment variables)
δ	$\in$	TyConVar	(type constructor variables)
γ	$\in$	TyConName	(type constructor names)
α	$\in$	ITyVar	(internal type variables)
μ	$\in$	ITyCon	$::= \delta \mid \gamma \mid \texttt{arr} \mid \langle \mu, \bar{l} \rangle$
τ	$\in$	ITy	$::= \alpha \mid \tau\, \mu \mid \tau_1 \to \tau_2 \mid \langle \tau, \bar{l} \rangle$
ts	$\in$	ITyScheme	$::= \forall \bar{v}.\, \tau$
tcs	$\in$	ITyConScheme	$::= \forall \bar{v}.\, \mu$
es	$\in$	EnvScheme	$::= \forall \bar{v}.\, e$
c	$\in$	EqCs	$::= \mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::= \downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts$
acc	$\in$	Accessor	$::= \uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha$
e	$\in$	Env	$::= \top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1 \mid \langle e, \bar{l} \rangle$

extra metavariables

ct	$\in$	CsTerm	$::= \tau \mid \mu \mid e$	v	$\in$	Var	$::= \alpha \mid \delta \mid ev$
σ	$\in$	Scheme	$::= ts \mid tcs \mid es$	a	$\in$	Atom	$::= v \mid \gamma \mid l$
dep	$\in$	Dependent	$::= \langle ct, \bar{l} \rangle$				

In Fig. 9, value identifiers (vid) are subscripted to disambiguate rules for expressions ($vid^l_{\mathtt{e}}$), datatype constructor definitions ($dcon^l_{\mathtt{c}}$), and pattern ($vid^l_{\mathtt{p}}$) occurrences. The non-ambiguous (hence non-subscripted) value identifiers occur at unary positions in patterns and datatype declarations.

Although SML distinguishes value variables and datatype constructors by assigning statuses in the type system, we distinguish them by defining two disjoint sets ValVar and DatCon. As opposed to the Skalpel core, for fully correct minimal error slices, [49, Sec.14.1] handles identifier statuses. Also, to simplify the presentation of the Skalpel core, datatypes have one constructor and one type argument.

6.2. Constraint syntax

Fig. 10 defines our constraint syntax for the Skalpel core. In our constraint based approach, we use constraints to compute the static semantics of programs using our constraint generator defined in Sec. 6.4, and we use our constraint solver defined in Sec. 6.5 to check whether constraints are solvable, i.e., whether programs are typable.

Sections 6.2.1 to 6.2.3 explain the various parts of this syntax. Note our novel hybrid constraint/environment forms e where binders, accessors and composition environments interact. The motivation is to build environments that avoid duplication at initial constraint generation or during constraint solving (see also design principle **DP4** in Sec. 7). Our binders and accessors are also

novel. Earlier systems (e.g. [48]) are too restrictive to easily represent module systems because they only support very limited cases of our binders. With our constraints, we can easily define a compositional constraint generator (see Sec. 6.4.2). We use the symbol $\mathcal{C}$ to represent either a label or any syntactic form presented in Fig. 10.

During analysis, a dependent form $\langle ct, \bar{l} \rangle$ depends on the program nodes with labels in $\bar{l}$. For that reason, labels are sometimes called dependencies. For example, the dependent equality constraint $\langle \tau_1 = \tau_2, \bar{l} \cup \{l\} \rangle$ might be generated for the labeled function application $\lceil exp\ atexp \rceil^l$, indicating the equality constraint $\tau_1 = \tau_2$ need only be true if node l has not been sliced out (see our design principle **DP2** in Sec. 7). In order to manipulate our labels, we define the two functions (of type $\mathsf{IntLabSynt} \rightarrow \mathsf{IntLabSynt}$) strip and $\mathsf{collapse}$ below, which respectively allow us to take all outer labels off any given term, and to union outer nested labels of terms.

$$\mathsf{strip}(\mathcal{C}) = \begin{cases} \mathsf{strip}(ct) & \text{if } \mathcal{C} = \langle ct, \bar{l} \rangle \\ \mathcal{C} & \text{otherwise} \end{cases}$$

$$\mathsf{collapse}(\mathcal{C}) = \begin{cases} \mathsf{collapse}(\langle y, \bar{l} \cup \bar{l}' \rangle) & \text{if } \mathcal{C} = \langle (\langle y, \bar{l} \rangle), \bar{l}' \rangle \\ \mathcal{C} & \text{otherwise} \end{cases}$$

Note that we sometimes write $\langle ct, l \rangle$ for $\langle ct, \{l\} \rangle$. Given a label or a set of labels x, we write ct^x to abbreviate $\langle ct, x \rangle$, and $ct_1 \stackrel{x}{=} ct_2$ for $\langle ct_1 = ct_2, x \rangle$.

6.2.1. Internal types (τ) and their constructors (μ)

The ITy and ITyCon sets contain internal types and internal type constructors respectively. In order to maintain some simplicity for the core, only unary type constructors are supported[5]. We have a special kind of type constructor `arr`, which is used to create a constraint in the constraint solving process (rule ($\mathsf{S5}$)) between a unary type constructor and a function type.

6.2.2. Schemes (σ)

There are three kinds of universally quantified schemes: type schemes, type constructor schemes, and environment schemes. All schemes are subject to α-conversion (e.g., the schemes $\forall\{\alpha_1\}.\,\alpha_1$ and $\forall\{\alpha_2\}.\,\alpha_2$ are equivalent).

[5] [49, Sec.14.10] presents a solution whereby type constructors can have any arity.

6.2.3. The constraint/environment form (e)

A constraint/environment e is an hybrid form that should be considered both as a constraint and as an environment. Constraint/environments can both specify type semantics and binding semantics of programs and provide an expressive way to structure constraints. Such a form can be any of the following:

1. **The satisfied (true) constraint**. This is represented by the symbol $\top$. Logically, $\top$ could be defined as $\exists\alpha.\alpha = \alpha$, but we chose to have a separate constraint for simplicity (e.g., $\top$ is simpler to solve).
2. **An environment variable**. An environment variable stands for any environment. This is especially useful to deal with structures and connect constraints on environments. (See also our design principle **DP7** in Sec. 7)
3. **A composition environment**. We use the operator ';' to compose environments, which is associative with unit $\top$: we consider $e;\top$, $\top;e$, and e to be equivalent. In a composition of the form $e_1;e_2$, the accessors of e_2 are in the scope of the binders of e_1. It acts as a logical conjunction requiring e_1 to be satisfied, and e_2 to be satisfied when the bindings of e_1 are in scope.
4. **A binder/accessor**. A binder is of the form $\downarrow id{=}\sigma$, and an accessor is of the form $\uparrow id{=}v$. Binders represent program occurrences of identifiers at binding positions, and accessors represent occurrences of identifiers at bound positions. For example, in the environment

$$\downarrow vid{=}\sigma;\uparrow vid{=}\alpha$$

   the internal type variable α is constrained through the binding of *vid* to be an instance of σ. In this case, we say that the binder and the accessor of *vid* are *connected*. Moreover, binders and accessors can often be connected without being next to each other. E.g., in the environment

$$\downarrow vid{=}\sigma;\ldots;\uparrow vid{=}\alpha$$

   it is *possible* that the binder and accessor of *vid* are connected. Some environments in the omitted $(\ldots)$ part might disconnect the accessor and the binder. Sec. 6.3.2 specifies which forms would cause this.
   We abbreviate $\downarrow vid{=}\forall\varnothing.\,ct$ by $\downarrow vid{=}ct$, $\downarrow vid{=}\forall\{v\}.\,ct$ by $\downarrow vid{=}\forall v.\,ct$, and $\langle\downarrow vid{=}ct, y\rangle$ by $\downarrow vid \overset{y}{=} ct$ (and similarly for accessors).
5. **An equality constraint**. A constraint where two pieces of constraint syntax are made to be equal.
6. **Existential environment**. The form $\exists a.e$, binds all occurrences of a that occur free in e. We abbreviate $\exists a_1.\cdots\exists a_n.e$ by $\exists\langle a_1,\ldots,a_n\rangle.e$.
   We write $[e]$ to abbreviate $(\exists ev.ev = e)$, where ev does not occur in e. This is a constraint which enforces the logical constraint nature of e while limiting the scope of its bindings. Note that the bindings can still have an effect if e constrains an environment variable.

7. **A polymorphic environment**. This promotes the binders in the argument to `poly` to be polymorphic.
8. **Dependent form**. Label-annotated environments act as environments only when the labels are satisfied.

6.2.4. Atomic forms

Let $\mathsf{atoms}(\mathcal{C})$ be the syntactic form set belonging to $\mathsf{Var} \cup \mathsf{Label}$ and occurring in $\mathcal{C}$. In addition, we define:

$$\begin{array}{rcl} \mathsf{vars}(\mathcal{C}) & = & \mathsf{atoms}(\mathcal{C}) \cap \mathsf{Var} \\ \mathsf{labs}(\mathcal{C}) & = & \mathsf{atoms}(\mathcal{C}) \cap \mathsf{Label} \end{array}$$

We also define $\mathsf{freevars}(\mathcal{C})$ as the set of free variables occurring in $\mathsf{vars}(\mathcal{C})$.

6.3. Semantics of constraints/environments

Checking parts of the program for mismatch requires substitution, unification and renaming which are used as part of our constraint solver outlined in section 6.5. These notions are defined in this section.

6.3.1. Renamings, unifers, and substitutions

We define renamings, unifiers, and substitutions as follows (note that $\mathsf{Ren} \subset \mathsf{Unifier} \subset \mathsf{Sub}$):

$$\begin{array}{llll}
ren & \in & \mathsf{Ren} & = \left\{ f \mid \begin{array}{ll} & f \in \mathsf{ITyVar} \rightarrow \mathsf{ITyVar} \\ \wedge & f \text{ is injective} \\ \wedge & \mathsf{dj}(\mathsf{dom}(f), \mathsf{ran}(f)) \end{array} \right\} \\
u & \in & \mathsf{Unifier} & = \left\{ f_1 \cup f_2 \cup f_3 \mid \begin{array}{ll} & f_1 \in \mathsf{ITyVar} \rightarrow \mathsf{ITy} \\ \wedge & f_2 \in \mathsf{TyConVar} \rightarrow \mathsf{ITyCon} \\ \wedge & f_3 \in \mathsf{EnvVar} \rightarrow \mathsf{Env} \end{array} \right\} \\
sub & \in & \mathsf{Sub} & = \left\{ f_1 \cup f_2 \mid \begin{array}{ll} & f_1 \in \mathsf{Unifier} \\ \wedge & f_2 \in \mathsf{TyConName} \rightarrow \mathsf{TyConName} \end{array} \right\} \\
\Delta & \in & \mathsf{Context} & ::= \langle u,\, e \rangle
\end{array}$$

Environments contain information on external identifiers. We also need information on internal type variables which we get through our unifiers. Renamings are used to instantiate type schemes. The $\mathsf{Unifier}$ set consists of unifiers

Figure 11 Substitution on constraint terms

$$
\begin{array}{lcl}
a[sub] & = & \left\{\begin{array}{ll} x, & \text{if } sub(a) = x \\ a, & \text{otherwise} \end{array}\right. \\
(\tau\ \mu)[sub] & = & \tau[sub]\ \mu[sub] \\
(\tau_1 \rightarrow \tau_2)[sub] & = & \tau_1[sub] \rightarrow \tau_2[sub] \\
ct^{\overline{l}}[sub] & = & \mathsf{collapse}(ct[sub]^{\overline{l}}) \\
(ct_1 = ct_2)[sub] & = & (ct_1[sub] = ct_2[sub]) \\
(e_1;e_2)[sub] & = & e_1[sub];e_2[sub] \\
(\forall\overline{v}.\ ct)[sub] & = & \forall\overline{v}.\ ct[sub] \text{ s.t. } \mathsf{dj}(\overline{v}, \mathsf{atoms}(sub)) \\
(\exists a.e)[sub] & = & \exists a.e[sub] \text{ s.t. } \mathsf{dj}(\{a\}, \mathsf{atoms}(sub)) \\
(\uparrow id{=}v)[sub] & = & \left\{\begin{array}{ll} (\uparrow id{=}v[sub]), & \text{if } v[sub] \in \mathsf{Var} \\ \text{undefined}, & \text{otherwise} \end{array}\right. \\
(\downarrow id{=}\sigma)[sub] & = & (\downarrow id{=}\sigma[sub]) \\
\mathtt{poly}(e)[sub] & = & \mathtt{poly}(e[sub]) \\
\mathcal{C}[sub] & = & \mathcal{C}, \text{ otherwise}
\end{array}
$$

Figure 12 Context application

$$
\begin{array}{lcl}
\langle u,\ \downarrow id{=}\sigma\rangle(id) & = & \sigma \\
\langle u,\ e^{\overline{l}}\rangle(id) & = & \forall\overline{v}.\ ct^{\overline{l}}, \text{ if } \langle u,\ e\rangle(id) = \forall\overline{v}.\ ct \\
\langle u,\ e_1;e_2\rangle(id) & = & \left\{\begin{array}{ll} \langle u,\ e_2\rangle(id), & \text{if } \langle u,\ e_2\rangle(id) \text{ is defined} \\ \text{undefined}, & \text{if } \langle u,\ e_2\rangle(id) \text{ is undefined} \\ & \text{and } \mathsf{shadowsAll}(\langle u,\ e_2\rangle) \\ \langle u,\ e_1\rangle(id), & \text{otherwise} \end{array}\right. \\
\langle u,\ ev\rangle(id) & = & \left\{\begin{array}{l} \langle u,\ e\rangle(id), \text{if } u(ev) = e \\ \text{undefined, otherwise} \end{array}\right.
\end{array}
$$

generated by our constraint solver (see Sec. 6.5). Fig. 11 defines the substitution function: it takes a constraint term and a substitution, and produces a constraint term. In addition, we define another substitution-like function called build, which, as compared to the substitution function defined in Fig. 11, calls itself recursively in the atom case (the other cases are the same as in Fig. 11):

$$
\mathsf{build}(a, sub) = \left\{\begin{array}{ll} \mathsf{build}(ct, sub), & \text{if } sub(a) = ct \\ a, & \text{otherwise} \end{array}\right.
$$

A constraint solving context (or just context for short) Δ of the form $\langle u,\ e\rangle$ is the context in which the meaning of constraint/environments is defined w.r.t. the semantic rules provided below in Sec. 6.3.3. Such forms are also used in our constraint solver defined in Sec. 6.5 as contexts in which the solvability of constraint/environments is checked. As mentioned above, in our system, unifiers and environments are complementary: unifiers contain information on internal variables and environments on external identifiers.

Fig. 12 presents how to access the semantics of an identifier in a context. Note that the application of an existential environment to an identifier is undefined because in general it represents incomplete information.

In general, we allow functions and relations that take an environment e to take instead a context $\langle u,\ e\rangle$, and functions and relations that take a unifier u to take instead a context $\langle u,\ e\rangle$. For example, let $\langle u,\ e\rangle(v) = u(v)$, let $\langle u,\ e\rangle;e' = \langle u,\ e;e'\rangle$, let $\mathsf{dom}(\langle u,\ e\rangle) = \mathsf{dom}(u)$, let $\langle u,\ e\rangle \cup u' = \langle u \cup u',\ e\rangle$, and let $\mathsf{build}(ct, \langle u,\ e\rangle) = \mathsf{build}(ct, u)$.

6.3.2. Shadowing

Finding the source of errors in a program is all about accessing and getting to know every bit of the program, so that any mismatches are identified. Error finding is elusive because in a context it may be the case that some parts are shadowed and so inaccessible. Consider the context $\langle u,\ bind_1;ev;bind_2\rangle$. When $ev \notin \mathsf{dom}(u)$, we say that *ev shadows* $bind_1$ because ev could potentially be constrained to be an environment that rebinds $bind_1$. We define $\mathsf{shadowsAll}$ by:

$$\mathsf{shadowsAll}(\langle u,\ e\rangle) \iff \left\{\begin{array}{lll} & (e = ev & \wedge(\mathsf{shadowsAll}(\langle u,\ u(ev)\rangle) \vee ev \notin \mathsf{dom}(u))) \\ \vee & (e = (e_1;e_2) & \wedge(\mathsf{shadowsAll}(\langle u,\ e_1\rangle) \vee \mathsf{shadowsAll}(\langle u,\ e_2\rangle))) \\ \vee & (e = \langle e', \bar{l}\rangle & \wedge\mathsf{shadowsAll}(\langle u,\ e'\rangle)) \\ \vee & (e = \exists a.e' & \wedge\mathsf{shadowsAll}(\langle u,\ e'\rangle) \wedge a \notin \mathsf{dom}(u)) \end{array}\right.$$

$$\mathsf{shadowsAll}(e) \iff \mathsf{shadowsAll}(\langle \varnothing,\ e\rangle)$$

If $\mathsf{shadowsAll}(e)$ then it means that some of the binders in e might be shadowed, and especially it means that in $e;ev$, the environment ev shadows the entire environment e (no binder from e is accessible in $e;ev$). In the rest of this paper we will use $\mathsf{shadowsAll}$ as a function that returns a Boolean—it is decidable.

6.3.3. Semantic rules

We now present the semantics of our constraint/environments. First, we define two $\mathsf{instance}$ relations, which we use to instantiate type schemes as follows:

$$\begin{array}{lll} \forall\overline{v}.\ ct, sub \xrightarrow{\mathsf{instance}} ct[sub] & \text{if} & \mathsf{dom}(sub) = \overline{v} \\ \sigma \xrightarrow{\mathsf{instance}} ct & \text{if} & \exists sub.\sigma, sub \xrightarrow{\mathsf{instance}} ct \end{array}$$

Semantic judgements are of the form:

$$\Phi \in \mathsf{SemanticJudgement} ::= u, e \triangleright e_1 \hookrightarrow e_2$$

Fig. 13 defines the semantics of our constraint/environments, ignoring dependencies at first. This figure uses the function toPoly which is formally defined

Figure 13 Semantics of constraint/environments, ignoring dependencies

$$\frac{}{u, e \triangleright \top \hookrightarrow \top}\ (\top) \qquad \frac{}{u, e \triangleright ev \hookrightarrow ev}\ (\mathsf{evar})$$

$$\frac{\forall i \in \{1,2\}. u, e \triangleright ct_i \hookrightarrow ct'_i \quad ct'_1[u] = ct'_2[u]}{u, e \triangleright (ct_1 = ct_2) \hookrightarrow \top}\ (\mathsf{eq})$$

$$\frac{e(id) \xrightarrow{\mathsf{instance}} ct \quad u, e \triangleright ct = v \hookrightarrow \top}{u, e \triangleright (\uparrow id{=}v) \hookrightarrow \top}\ (\mathsf{acc}) \qquad \frac{e(id) \text{ undefined}}{u, e \triangleright (\uparrow id{=}v) \hookrightarrow \top}\ (\mathsf{acc}')$$

$$\frac{}{u, e \triangleright (\downarrow id{=}ct) \hookrightarrow (\downarrow id{=}ct[u])}\ (\mathsf{bind})$$

$$\frac{u, e \triangleright e' \hookrightarrow e''}{u, e \triangleright \mathtt{poly}(e') \hookrightarrow \mathsf{toPoly}(\langle u, e\rangle, e'')}\ (\mathsf{poly})$$

$$\frac{u, e \triangleright e_1 \hookrightarrow e'_1 \quad u, (e;e'_1) \triangleright e_2 \hookrightarrow e'_2}{u, e \triangleright (e_1;e_2) \hookrightarrow (e'_1;e'_2)}\ (\mathsf{comp})$$

$$\frac{u + u', e \triangleright e_1 \hookrightarrow e_2 \quad \mathsf{dom}(u') = \{a\}}{u, e \triangleright \exists a. e_1 \hookrightarrow e_2}\ (\mathsf{exists})$$

below in Fig. 17 in Sec. 6.3.4, and which transforms monomorphic environments into polymorphic ones. We say that an environment e is satisfiable iff there exist u and e' such that $u, \top \triangleright e \hookrightarrow e'$. For example, let us consider the following environments:

$$\begin{array}{rcl} e_0 & = & (\uparrow vid{=}\alpha_2);(\alpha_2 = \alpha\,\gamma);(\alpha_1 = \alpha_3 \to \alpha_4) \\ e_1 & = & \exists\langle\alpha_1, \alpha_2\rangle.\mathtt{poly}(\downarrow vid{=}\alpha_0);e_0 \end{array}$$

We show that e_1 is satisfiable by deriving $\Phi = \varnothing, \top \triangleright e_1 \hookrightarrow e'$, where

$$\begin{array}{rcl} e' & = & (\downarrow vid{=}\forall\alpha_0.\,\alpha_0) \\ u_1 & = & \{\alpha_1 \mapsto (\alpha_3 \to \alpha_4), \alpha_2 \mapsto \alpha\,\gamma\} \\ \Phi_1 & = & u_1, e' \triangleright e_0 \hookrightarrow \top \end{array}$$

Fisst, we derive Φ_1 as follows:

$$\dfrac{\dfrac{\dfrac{\alpha_2[u_1] = \alpha_2[u_1]}{u_1, e' \triangleright (\alpha_2 = \alpha_2) \hookrightarrow \top}}{u_1, e' \triangleright (\uparrow vid{=}\alpha_2) \hookrightarrow \top} \quad \dfrac{\dfrac{\alpha_2[u_1] = (\alpha\,\gamma)[u_1]}{u_1, e' \triangleright (\alpha_2 = \alpha\,\gamma) \hookrightarrow \top} \quad \dfrac{\alpha_1[u_1] = (\alpha_3 \to \alpha_4)[u_1]}{u_1, e' \triangleright (\alpha_1 = \alpha_3 \to \alpha_4) \hookrightarrow \top}}{u_1, e' \triangleright (\alpha_2 = \alpha\,\gamma);(\alpha_1 = \alpha_3 \to \alpha_4) \hookrightarrow \top}}{\Phi_1}$$

Then, we derive Φ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\alpha_1 \mapsto \alpha_3 \to \alpha_4, \alpha_2 \mapsto \alpha\,\gamma\}, \top \rhd {\downarrow}vid{=}\alpha_0 \hookrightarrow {\downarrow}vid{=}\alpha_0}}{\{\alpha_1 \mapsto \alpha_3 \to \alpha_4, \alpha_2 \mapsto \alpha\,\gamma\}, \top \rhd \mathtt{poly}({\downarrow}vid{=}\alpha_0) \hookrightarrow e'} \quad \Phi_1}{\{\alpha_1 \mapsto \alpha_3 \to \alpha_4, \alpha_2 \mapsto \alpha\,\gamma\}, \top \rhd \mathtt{poly}({\downarrow}vid{=}\alpha_0);e_0 \hookrightarrow e'}}{\{\alpha_1 \mapsto \alpha_3 \to \alpha_4\}, \top \rhd \exists\alpha_2.\mathtt{poly}({\downarrow}vid{=}\alpha_0);e_0 \hookrightarrow e'}}{\Phi}$$

Let us mention an issue regarding the semantics of our constraint/environments and our constraint solver defined below in Sec. 6.5. Let us consider the following environment, similar to e_1, which we call e_2:

$$\mathtt{poly}({\downarrow}vid{=}\alpha_1);({\uparrow}vid{=}\alpha_2);(\alpha_2 = \alpha\,\mu);(\alpha_1 = \alpha_3 \to \alpha_4)$$

The environment e_2 only differs from e_1 by the replacement of α_0 by α_1. There are now two occurrences of α_1 in e_2 that are used at two separate unrelated places. Because of these two occurrences of α_1, the environment e_2 fails to be satisfiable w.r.t. the rules defined in Fig. 13. However, e_2 is satisfiable w.r.t. our constraint solver defined below in Sec. 6.5. The issue is that our constraint solver considers the two occurrences of α_1 to be different when with the semantics defined in this section, these two occurrences are considered to be the same. Note that e_2 cannot be generated by our initial constraint generation algorithm defined below in Sec. 6.4, so this bug is not triggered. (To overcome this issue, our semantic rules and constraint solver could be modified to fail on e_2 by (1) keeping track of the generalized type variables, and by (2) failing when dealing with a constraint in which occurs a generalized type variable.)

Let us now consider dependencies. We define semantic judgements considering dependencies as follows:

$$\begin{array}{lcllcl} ds & \in & \mathsf{DepStatus} & ::= & \mathtt{keep} \mid \mathtt{drop} \mid \mathtt{keep\text{-}only\text{-}binders} \\ de & \in & \mathsf{DepEnv} & = & \mathsf{Label} \to \mathsf{DepStatus} \\ \Psi & \in & \mathsf{SemanticJudgementDep} & ::= & u, e, de \rhd e_1 \hookrightarrow e_2 \end{array}$$

We define the application of dependency environments to dependency sets as follows:

$$de(\bar{l}) = \{de(l) \mid l \in \bar{l}\}$$

Let us add dependencies to the rules defined in Fig. 13. Semantic judgements are now of the form $u, e, de \rhd e_1 \hookrightarrow e_2$. Except for these additions, the rules defined in Fig. 13 do not differ and are therefore not repeated. In addition, Fig. 14 defines three new rules: (keep), (drop), and ($\mathsf{keep\text{-}only\text{-}binders}$) to deal with dependencies. Fig. 14 uses the function dum which is formally defined below in Fig. 23 in Sec. 6.6.2. This function transforms an environment e into

Figure 14 Semantics of the constraint/environments, considering dependencies

$$\frac{u, e, de \triangleright e' \hookrightarrow e'' \quad de(\bar{l}) = \{\texttt{keep}\}}{u, e, de \triangleright \langle e', \bar{l}\rangle \hookrightarrow \langle e'', \bar{l}\rangle}\ (\mathsf{keep}) \qquad \frac{\texttt{drop} \in de(\bar{l})}{u, e, de \triangleright \langle e', \bar{l}\rangle \hookrightarrow \mathsf{dum}(e')}\ (\mathsf{drop})$$

$$\frac{\{\texttt{keep-only-binders}\} = de(\bar{l}) \setminus \{\texttt{keep}\}}{u, e, de \triangleright \langle e', \bar{l}\rangle \hookrightarrow \top}\ (\mathsf{keep\text{-}only\text{-}binders})$$

a similar dummy environment e' which contains dummy versions of the binders from e that cannot participate in any error.

We say that an environment e is satisfiable w.r.t. the dependency environment de iff there exist u and e' such that $u, \top, de \triangleright e \hookrightarrow e'$. Given a dependency environment de, a dependency l is said to be satisfied if $de(l) = \texttt{keep}$, and it is said to be unsatisfied if $de(l) = \texttt{drop}$. The dependency status `keep-only-binders` is more complicated. This status is needed for scoping issues which are further discussed below in Sec. 6.6.2. If an environment e is annotated by a dependency which has status `keep-only-binders` then e's binders and environment variables (which could potentially bind any identifier) are turned into dummy binders and dummy environment variables respectively. Other environments, such as equality constraints, are discarded. The environment e' is the semantics of e in the context $\langle u, \top, de\rangle$.

6.3.4. Polymorphic environments

This section shows how to generate polymorphic environments for polymorphic functions. It defines the function toPoly which closes environments w.r.t. their contexts [40, Sec.4.8]. In order to do that and report precise type error slices, we need to assign precise blames to variables that cannot be generalized. Because of that, this section is fairly technical and can be skipped all together by the reader. The main takeaway is that the toPoly function closes environments w.r.t. their context and is similar to the `Clos` function defined in [40, Sec.4.8].

The function toPoly is used in rule (poly) in Fig. 13 above and in rule (P1) of our constraint solver defined below in Sec. 6.5 to generate polymorphic environments from monomorphic ones by quantifying the type variables not occurring in the types of the monomorphic binders of the current constraint solving context. Therefore, we need to keep track of the reasons why some variables cannot be generalized, i.e., quantified over:

$$m \in \mathsf{Monos} = \mathsf{ITyVar} \to \mathbb{P}(\mathsf{Label})$$

These functions are used to associate blames with type variables as to why they are monomorphic. In that context, a type variable is said to be monomorphic

Figure 15 Blame computation regarding monomorphic variables

$$
\begin{array}{lcl}
\mathsf{blames}(u,\alpha) & = & \begin{cases} \mathsf{blames}(u,\tau), & \text{if } u(\alpha)=\tau \\ \{\alpha \mapsto \varnothing\}, & \text{otherwise} \end{cases} \\
\mathsf{blames}(u,\tau\,\mu) & = & \mathsf{blames}(u,\tau) \\
\mathsf{blames}(u,\tau_1 \to \tau_2) & = & \mathsf{blames}(u,\tau_1) \Cup \mathsf{blames}(u,\tau_2) \\
\mathsf{blames}(u,\tau^{\overline{l}}) & = & \{\alpha \mapsto \overline{l} \cup \overline{l}' \mid \mathsf{blames}(u,\tau)(\alpha) = \overline{l}'\} \\
\mathsf{blames}(\langle u,\ e\rangle,\tau) & = & \mathsf{blames}(u,\tau)
\end{array}
$$

Figure 16 Decoration with blames

$$
\begin{array}{lcl}
\mathsf{decorate}(\alpha,m) & = & \begin{cases} \alpha^{\overline{l}}, & \text{if } m(\alpha)=\overline{l} \\ \alpha, & \text{otherwise} \end{cases} \\
\mathsf{decorate}(\tau\,\mu,m) & = & \mathsf{decorate}(\tau,m)\,\mu \\
\mathsf{decorate}(\tau_1 \to \tau_2,m) & = & \mathsf{decorate}(\tau_1,m) \to \mathsf{decorate}(\tau_2,m) \\
\mathsf{decorate}(\downarrow id{=}\sigma,m) & = & (\downarrow id{=}\mathsf{decorate}(\sigma,m)) \\
\mathsf{decorate}(\forall\overline{v}.\ ct,m) & = & \forall\overline{v}.\,\mathsf{decorate}(ct,\overline{v} \triangleright m) \\
\mathsf{decorate}(\uparrow id{=}\alpha,m) & = & (\uparrow id{=}v)^{\overline{l}},\ \text{if } m(v)=\overline{l} \\
\mathsf{decorate}(ct_1 = ct_2,m) & = & (\mathsf{decorate}(ct_1,m) = \mathsf{decorate}(ct_2,m)) \\
\mathsf{decorate}(\mathtt{poly}(e),m) & = & \mathtt{poly}(\mathsf{decorate}(e,m)) \\
\mathsf{decorate}(\exists a.e,m) & = & \exists a.\mathsf{decorate}(e,\{a\} \triangleright m) \\
\mathsf{decorate}(e_1;e_2,m) & = & \mathsf{decorate}(e_1,m);\mathsf{decorate}(e_2,m) \\
\mathsf{decorate}(ct^{\overline{l}},m) & = & \mathsf{decorate}(ct,m)^{\overline{l}} \\
\mathsf{decorate}(ct,m) & = & ct,\ \text{otherwise}
\end{array}
$$

if it occurs in the type of a monomorphic binder such as a fn-binder.

Fig. 15 and fig. 16 define the two functions blames and decorate. The function blames computes a function m in Monos: $\mathsf{blames}(u, ct)$ associates with α the dependency set occurring in ct on the paths from its root node to any free occurrence of α, and this for each α occurring free in ct. The function decorate adds dependencies to constraint terms: $\mathsf{decorate}(ct, m)$ results in the precise constraining of the free occurrences of α in ct with the dependency set $m(\alpha)$. The function $\mathsf{monos} : \mathsf{Context} \to \mathsf{Monos}$ defined below computes the set of dependent monomorphic type variables occurring in a given context, i.e., the type variables occurring in the types of the monomorphic binders:

$$\mathsf{monos}(\Delta) = \Cup_{\tau \text{ s.t. } \exists vid.\Delta(vid)=\forall\varnothing.\,\tau}\ \mathsf{blames}(\Delta,\tau)$$

For example, $\mathsf{monos}(\langle\{\alpha \mapsto (\alpha_1^{\overline{l}_1} \to \alpha_2^{\overline{l}_2})^{\overline{l}_0}\},\ \downarrow vid{=}\alpha\rangle) = \{\alpha_1 \mapsto \overline{l}_0 \cup \overline{l}_1, \alpha_2 \mapsto \overline{l}_0 \cup \overline{l}_2\}$. The type variable α_1 occurring in the monomorphic type associated with vid depends only on $\overline{l}_0 \cup \overline{l}_1$ and not on $\overline{l}_2$ (and similarly for α_2).

Finally, using all these functions, we define toPoly in Fig. 17.

Figure 17 Monomorphic to polymorphic environment

$$
\begin{array}{lll}
\mathsf{toPoly}(\Delta, \downarrow id{=}\forall\varnothing.\ ct) & = \forall\overline{v}.\ ct', & \text{where} \left\{ \begin{array}{ll} \overline{v} & = \mathsf{freevars}(\langle e, ct\rangle) \setminus \mathsf{dom}(\mathsf{monos}(\Delta)) \\ ct' & = \mathsf{decorate}(\mathsf{build}(ct, \Delta), \mathsf{monos}(\Delta)) \end{array} \right. \\
\mathsf{toPoly}(\Delta, e^{\overline{l}}) & = e'^{\overline{l}}, & \text{if } e' = \mathsf{toPoly}(\Delta, e) \\
\mathsf{toPoly}(\Delta, e_1;e_2) & = e_1';e_2' & \text{if } e_1' = \mathsf{toPoly}(\Delta, e_1) \text{ and } e_2' = \mathsf{toPoly}(\Delta;e_1', e_2) \\
\mathsf{toPoly}(\Delta, e) & = e, & \text{if none of the above applies}
\end{array}
$$

Figure 18 Constraint generator (1 of 2) ($\mathsf{ExtLabSynt} \rightarrow \mathsf{Env}$)

Expressions (exp)

(G1) $[\![vid^l_{\mathtt{e}}, \alpha]\!] = \uparrow vid \stackrel{l}{=} \alpha$

(G2) $[\![\mathtt{let}^l\ dec\ \mathtt{in}\ exp\ \mathtt{end}, \alpha]\!] = [\exists\alpha_2.[\![dec]\!];[\![exp, \alpha_2]\!];(\alpha \stackrel{l}{=} \alpha_2)]$

(G3) $[\![\lceil exp\ \ atexp\rceil^l, \alpha]\!] = \exists\langle\alpha_1, \alpha_2\rangle.[\![exp, \alpha_1]\!];[\![atexp, \alpha_2]\!];(\alpha_1 \stackrel{l}{=} \alpha_2 \rightarrow \alpha)$

(G4) $[\![\mathtt{fn}\ pat \stackrel{l}{\Rightarrow} exp, \alpha]\!] = [\exists\langle\alpha_1, \alpha_2, ev\rangle.(ev = [\![pat, \alpha_1]\!]);ev^l;[\![exp, \alpha_2]\!];(\alpha \stackrel{l}{=} \alpha_1 \rightarrow \alpha_2)]$

Labeled datatype constructors ($ldcon$)

(G5) $[\![dcon^l, \alpha]\!] = \uparrow dcon \stackrel{l}{=} \alpha$

Patterns (pat)

(G6) $[\![vvar^l_{\mathtt{p}}, \alpha]\!] = \downarrow vvar \stackrel{l}{=} \alpha$

(G7) $[\![dcon^l_{\mathtt{p}}, \alpha]\!] = \uparrow dcon \stackrel{l}{=} \alpha$

(G8) $[\![\lceil ldcon\ \ atpat\rceil^l, \alpha]\!] = \exists\langle\alpha_1, \alpha_2\rangle.[\![ldcon, \alpha_1]\!];[\![atpat, \alpha_2]\!];(\alpha_1 \stackrel{l}{=} \alpha_2 \rightarrow \alpha)$

Labeled type constructors (ltc)

(G9) $[\![tc^l, \delta]\!] = \uparrow tc \stackrel{l}{=} \delta$

Types (ty)

(G10) $[\![tv^l, \alpha]\!] = \uparrow tv \stackrel{l}{=} \alpha$

(G11) $[\![\lceil ty\ \ ltc\rceil^l, \alpha']\!] = \exists\langle\alpha, \delta\rangle.[\![ty, \alpha]\!];[\![ltc, \delta]\!];(\alpha' \stackrel{l}{=} \alpha\,\delta)$

(G12) $[\![ty_1 \stackrel{l}{\rightarrow} ty_2, \alpha]\!] = \exists\langle\alpha_1, \alpha_2\rangle.[\![ty_1, \alpha_1]\!];[\![ty_2, \alpha_2]\!];(\alpha \stackrel{l}{=} \alpha_1 \rightarrow \alpha_2)$

Datatype names (dn)

(G13) $[\![\lceil tv\ \ tc\rceil^l, \alpha']\!] = \exists\langle\alpha, \gamma\rangle.(\alpha' \stackrel{l}{=} \alpha\,\gamma);(\downarrow tc \stackrel{l}{=} \gamma);(\downarrow tv \stackrel{l}{=} \alpha)$

Constructor bindings (cb)

(G14) $[\![dcon^l_{\mathtt{c}}, \alpha]\!] = \downarrow dcon \stackrel{l}{=} \alpha$

(G16) $[\![dcon\ \mathtt{of}^l\ ty, \alpha]\!] = \exists\langle\alpha', \alpha_1\rangle.[\![ty, \alpha_1]\!];(\alpha' \stackrel{l}{=} \alpha_1 \rightarrow \alpha);(\downarrow dcon \stackrel{l}{=} \alpha')$

6.4. Constraint generation

Fig. 18 and 19 define our *constraint generator*. As mentioned in Sec. 4, constraints are not only generated here, but are also generated as part of the constraint solving process.

Figure 19 Constraint generator (2 of 2) (ExtLabSynt → Env)

Declarations (*dec*)

(G17) $[\![\texttt{val rec}\ pat \stackrel{l}{=} exp]\!] =$
$\exists\langle\alpha_1, \alpha_2, ev\rangle.(ev = \texttt{poly}([\![pat, \alpha_1]\!];[\![exp, \alpha_2]\!];(\alpha_1 \stackrel{l}{=} \alpha_2)));ev^l$

(G18) $[\![\texttt{datatype}\ dn \stackrel{l}{=} cb]\!] =$
$\exists\langle\alpha_1, \alpha_2, ev\rangle.(ev = ((\alpha_1 \stackrel{l}{=} \alpha_2);[\![dn, \alpha_1]\!];\texttt{poly}([\![cb, \alpha_2]\!])));ev^l$

(G19) $[\![\texttt{open}^l\ strid]\!] = \exists ev.(\uparrow strid \stackrel{l}{=} ev);ev^l$

Structure declarations (*strdec*)

(G20) $[\![\texttt{structure}\ strid \stackrel{l}{=} strexp]\!] =$
$\exists\langle ev, ev'\rangle.[[\![strexp, ev]\!];(ev' = (\downarrow strid \stackrel{l}{=} ev));ev'^l$

Structure expressions (*strexp*)

(G21) $[\![strid^l, ev]\!] = \uparrow strid \stackrel{l}{=} ev$

(G22) $[\![\texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}, ev]\!] =$
$\exists ev'.(ev \stackrel{l}{=} ev');(ev' = ([\![strdec_1]\!]; \cdots ;[\![strdec_n]\!]))$

Let $\mathsf{cstgen}(\mathcal{P}, \overline{v})$ be a function with two arguments: (1) a labeled piece of user program $\mathcal{P}$, and (2) a set of free variables occurring in $\mathcal{P}$. Let $\mathsf{cstgen}(\mathcal{P}) = \mathsf{cstgen}(\mathcal{P}, \varnothing)$. Each constraint generation rule is written either as $[\![\mathcal{P}]\!] = e$, which abbreviates $\mathsf{cstgen}(\mathcal{P})$, or as $[\![\mathcal{P}, v]\!] = e$, which abbreviates $\mathsf{cstgen}(\mathcal{P}, \{v\})$.

As mentioned above, in order to simplify the presentation of Skalpel core, datatype declarations only have one constructor: see rules (G11) and (G13). Structure declarations are handled in rule (G20). Again to simplify the presentation of Skalpel core, we do not handle signatures in this paper but we worked out the theory in [45].

6.4.1. Discussion of some constraint generation rules

Recursive value declarations. In rule (G17), to handle the recursivity of such declarations, the environment e_2 generated for *exp* must be in the scope of environment e_1 generated for *pat*. The binders in e_1 are monomorphic. Polymorphic type schemes are generated at constraint solving when dealing with the `poly` constraint. Within the `poly` environment, binders need to be monomorphic because SML does not allow polymorphic recursion. Allowing `poly` constraints on environments other than just a single binder (such as in $\texttt{poly}(bind;e;acc)$, where *acc* can potentially refer to *bind*) allows one to delay the generation of polymorphic types. Therefore, given a recursive function declaration, one can generate only one binder for the function (in a naive approach two might be needed: one monomorphic for the function's body and one polymorphic for the function's scope).

Recursive datatypes. In rule (G18), to handle the recursivity of such declarations, the environment `poly`(e_2) generated for the datatype constructor of the declared type constructor must be in the scope of e_1 generated for the declared type constructor. For example, in the declaration `datatype nat = z | s of nat`, `nat`'s second occurrence refers to its first occurrence. Note that e_1 also binds explicit type variables in Skalpel. This extends the scope of the bound external type variable further than needed, but causes no harm here because all type variables only occur inside datatype constructor bindings. This issue is resolved in [49] using local environments.

Unlabeled equality constraints. Rules (G4), (G17), (G18), (G20) and (G22) generate unlabeled environment equality constraints. Given a structure expression *strexp* of the form $\mathtt{struct}^{l}$ $strdec_1 \cdots strdec_n$ `end`, rule (G22) generates a constraint c of the form $ev' = (e_1; \cdots; e_n)$. Such a constraint needs to be unlabeled because each e_i does not depend on *strexp* itself, but only on the corresponding declaration $strdec_i$ which happens to be packed together with other declarations in *strexp*. When slicing out the packaging created by *strexp* (by slicing out l above), we must not discard the e_is, which is what would happen if we were to label c with l and discard it when slicing out l. The information related only to *strexp*, carried by c, is the fact that a sequence of declarations, corresponding to the composition environment $e_1; \cdots; e_n$, is packed into a structure. This information depends on *strexp* via the extra labeled equality constraint $ev \stackrel{l}{=} ev'$ (see also design principle **DP3** in Sec. 7). In rules (G4), (G17), (G18) and (G20), we use labeled environment variables of the form ev^{l} for this purpose.

Environment variables. Rules (G4), (G17), (G18), (G19) and (G20) label environment variables. In rule (G19), we do so to prevent sliced out declarations from shadowing their context. For example, if ev is unconstrained, it shadows e in $e;ev$. In the other rules we do so to be able to disconnect accessors from their binders. Let us focus on rule (G19). In that rule, ev represents the entire opening declaration, and is labeled with l, the label associated with the declaration. Without l, ev would be a constraint that always has to be satisfied, even when the corresponding opening declaration has been sliced out. For example, slicing out `open S` in `structure S = struct end; val x = 1; open S; val y = x 1;` would result in the environment variable generated for `open S` shadowing its context, which contains the declaration `val x = 1`. Failing from labeling ev using l in rule (G19) would prevent from finding the error that `x` is declared as an integer in the piece of code presented above, and is also applied to an argument in `y`'s body. With the label, the environment variable is a constraint that has to be satisfied only when the declaration is not sliced out. (See also our design principle **DP7** in Sec. 7 for more information.)

6.4.2. Properties

This section discusses some properties of our constraint generator. (1) It is compositional, i.e., given a piece of code, we can first generate separately constraints for its various parts and then compose the analysis of these parts to form a constraint/environment for the larger piece of code. Note that even though our constraint generator is compositional, this is not true about the entire Skalpel procedure (see Sec. 4) mainly because of the way schemes are generated from binders at constraint solving (Haack and Wells' slicer was inefficient but allowed compositional analysis [25, 26]). (2) It generates constraints that are linear in the size of the input program. We will show that our program does not suffer from a constraint explosion the larger the program grows. (3) It terminates.

Remark 6.1 (Compositionality of constraint generator). *The constraint generator shown in Fig. 18 and 19 is compositional.*

Lemma 6.2 (Size of initially generated constraints). *Constraints generated by our constraint generator shown in Fig. 18 and 19 have a size linear in the input program's size.*

Proof. By inspection of the rules. For a polymorphic (let-bound) function (rules (G2), (G6) and (G17)) we do not eagerly copy constraints for the function body. Instead, we generate polymorphic and composition environments, and binders force solving the constraints for the body before copying its type for each use of the function. □

Lemma 6.3 (Termination of constraint generator). *The constraint generator shown in Fig. 18 and 19 terminates, i.e., for all strdec,* $\mathsf{cstgen}(\mathit{strdec})$ *returns a constraint/environment of the form* e*.*

Proof. This can easily be proved using the size of the input program as measure. □

6.5. Constraint solving

6.5.1. Syntax

Fig. 20 defines additional syntactic forms used by our constraint solver (itself defined in Fig. 21 and 22), where a constraint solving step is defined by the relation $\rightarrow$, and where $\rightarrow^*$ is its reflexive (w.r.t. State) and transitive closure. Given an environment e to solve, our constraint solver starts in the state

Figure 20 Syntactic forms used by the constraint solver

ek	$\in$	ErrKind	$::=$	$\texttt{clash}(\mu_1, \mu_2) \mid \texttt{circ}$
er	$\in$	Error	$::=$	$\langle ek, \bar{l} \rangle$
$istate$	$\in$	InternalState	$::=$	$\texttt{slv}(\Delta, \bar{l}, e, e')$
$fstate$	$\in$	FinalState	$::=$	$\texttt{succ}(\Delta, e) \mid \texttt{err}(er)$
$state$	$\in$	State	$::=$	$istate \mid fstate$

$\texttt{slv}(\langle \varnothing, \top \rangle, \varnothing, e, \top)$, and either succeeds with final state $\texttt{succ}(\Delta, e')$ returning its current constraint solving context Δ, and e', the "solved" version of e, or fails with final state $\texttt{err}(er)$ returning an error which can either be a type constructor clash or a circularity error[6] (see Lem. 6.4). Given a state $\texttt{slv}(\Delta, \bar{l}, e, e')$, if the dependencies in $\bar{l}$ are satisfied and e is solvable in the context Δ then the constraint solver succeeds with final state $\texttt{succ}(\Delta', e';e'')$ for some Δ' and e''.

6.5.2. Algorithm

Fig. 21 and 22 define our constraint solver which can be regarded as a rewriting system. A finite computation is then a finite sequence of states $\langle state_1, \ldots, state_n \rangle$ such that for each $i \in \{1, \ldots, n-1\}$, the state $state_{i+1}$ is obtained by applying one of the constraint solving rules as defined in Fig. 21 and 22 to the state $state_i$, i.e., the pair $\langle state_i, state_{i+1} \rangle$ is obtained by instantiating one of the constraint solving rules, where $state_i$ is the instantiation of the left-hand-side of the rule and $state_{i+1}$ is the instantiation of the right-hand-side.

Note that rule (A2) in Fig. 22 could be used to report free identifiers. If $\texttt{slv}(\Delta, \bar{l}, \uparrow id{=}v, e) \rightarrow \texttt{succ}(\Delta, e)$ and $\neg\mathsf{shadowsAll}(\Delta)$, then it means that there is no binder for id and so that it is a free identifier. Free identifiers are in any case important to report, but it is especially vital for structure identifiers in `open` declarations. In our approach, a free opened structure is considered as potentially redefining its entire context. For example, in Skalpel the program `val x = 1; open S; val y = x 1` does not have an error involving `x` because `x`'s first occurrence is hidden by the declaration `open S`. This could be confusing if `S` was not reported as being free. Let us explain how a free opened structure shadows its context. Given a declaration `open S` labeled by l, our initial constraint generator generates an environment of the form $(\uparrow\texttt{S} \overset{l}{=} ev);ev^l$. Because `S` is free, we have to use rule (A2) when solving $\uparrow\texttt{S}{=}ev$. The environment variable ev is then unconstrained. Hence, when solving ev, we use rule (V) and $e;ev$ (from the right-hand-side of rule (V)) results in the shadowing of all the binders in e by ev.

Let the relations isErr and solvable be defined as follows:

[6]More error kinds are handled in [49, Ch.14] for example.

Figure 21 Constraint solver (1 of 2): InternalState $\rightarrow$ FinalState

equality constraint reversing

(R) $\mathtt{slv}(\Delta, \bar{l}, ct = ct', e) \rightarrow \mathtt{slv}(\Delta, \bar{l}, ct' = ct, e)$,
if $(ct' \in s \wedge ct \notin s)$ where $s = \mathsf{Var} \cup \mathsf{Dependent}$

equality simplification

(S1) $\mathtt{slv}(\Delta, \bar{l}, ct = ct, e) \rightarrow \mathtt{succ}(\Delta, e)$

(S2) $\mathtt{slv}(\Delta, \bar{l}, ct^{\bar{l}'} = ct', e) \rightarrow \mathtt{slv}(\Delta, \bar{l} \cup \bar{l}', ct = ct', e)$

(S3) $\mathtt{slv}(\Delta, \bar{l}, \tau_1\, \mu_1 = \tau_2\, \mu_2, e) \rightarrow \mathtt{slv}(\Delta, \bar{l}, (\mu_1 = \mu_2);(\tau_1 = \tau_2), e)$

(S4) $\mathtt{slv}(\Delta, \bar{l}, \tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4, e) \rightarrow \mathtt{slv}(\Delta, \bar{l}, (\tau_1 = \tau_3);(\tau_2 = \tau_4), e)$,

(S5) $\mathtt{slv}(\Delta, \bar{l}, \tau = \tau', e) \rightarrow \mathtt{slv}(\Delta, \bar{l}, \mu = \mathtt{arr}, e)$,
if $\{\tau, \tau'\} = \{\tau_1\, \mu, \tau_2 \rightarrow \tau_3\}$

(S6) $\mathtt{slv}(\Delta, \bar{l}, \mu_1 = \mu_2, e) \rightarrow \mathtt{err}(\langle \mathtt{clash}(\mu_1, \mu_2), \bar{l}\rangle)$,
if $\{\mu_1, \mu_2\} \in \{\{\gamma, \gamma'\}, \{\gamma, \mathtt{arr}\}\} \wedge \gamma \neq \gamma'$

unifier access

Rules (U1) $\cdots$ (U4) assume: $v \notin \{\mathsf{strip}(ct)\} \cup \mathsf{dom}(\Delta)$ and $ct' = \mathsf{build}(\Delta, ct^{\bar{l}})$.

(U1) $\mathtt{slv}(\Delta, \bar{l}, v = ct, e) \rightarrow \mathtt{err}(\langle \mathtt{circ}, \mathsf{labs}(ct')\rangle)$, if $v \in \mathsf{freevars}(ct') \setminus \mathsf{Env}$

(U2) $\mathtt{slv}(\Delta, \bar{l}, v = ct, e) \rightarrow \mathtt{succ}(\Delta \cup \{v \mapsto ct^{\bar{l}}\}, e)$, if $v \notin \mathsf{freevars}(ct') \cup \mathsf{Env}$

(U3) $\mathtt{slv}(\Delta, \bar{l}, v = ct, e) \rightarrow \mathtt{succ}(\Delta' \cup \{v \mapsto e'\}, e)$,
if $v \in \mathsf{Env} \wedge \mathtt{slv}(\Delta, \bar{l}, ct, \top) \rightarrow^* \mathtt{succ}(\Delta', e')$

(U4) $\mathtt{slv}(\Delta, \bar{l}, v = ct, e) \rightarrow \mathtt{err}(er)$,
if $v \in \mathsf{Env} \wedge \mathtt{slv}(\Delta, \bar{l}, ct, \top) \rightarrow^* \mathtt{err}(er)$

(U5) $\mathtt{slv}(\Delta, \bar{l}, v' = ct, e) \rightarrow \mathtt{slv}(\Delta, \bar{l}, ct' = ct, e)$, if $\Delta(v') = ct'$

binders/existentials/true/dependent/variables

(B) $\mathtt{slv}(\Delta, \bar{l}, bind, e) \rightarrow \mathtt{succ}(\Delta, e;bind^{\bar{l}})$

(X) $\mathtt{slv}(\Delta, \bar{l}, \exists a.e', e) \rightarrow \mathtt{slv}(\Delta, \bar{l}, e'[\{a \mapsto a'\}], e)$, if $a' \notin \mathsf{atoms}(\langle \Delta, e'\rangle)$

(E) $\mathtt{slv}(\Delta, \bar{l}, \top, e) \rightarrow \mathtt{succ}(\Delta, e)$

(D) $\mathtt{slv}(\Delta, \bar{l}, e'^{\bar{l}'}, e) \rightarrow \mathtt{slv}(\Delta, \bar{l} \cup \bar{l}', e', e)$

(V) $\mathtt{slv}(\Delta, \bar{l}, ev, e) \rightarrow \mathtt{succ}(\Delta, e;ev^{\bar{l}})$

$$e \xrightarrow{\mathsf{isErr}} er \iff \mathtt{slv}(\langle \varnothing, \top\rangle, \varnothing, e, \top) \rightarrow^* \mathtt{err}(er)$$
$$\mathsf{solvable}(e) \iff \exists \Delta. \exists e'. \mathtt{slv}(\langle \varnothing, \top\rangle, \varnothing, e, \top) \rightarrow^* \mathtt{succ}(\Delta, e')$$
$$\mathsf{solvable}(strdec) \iff \exists e. [\![strdec]\!] = e \wedge \mathsf{solvable}(e)$$

These relations are used, among other things, to define our minimization and enumeration algorithms in Sec. 6.6.

6.5.3. Properties

As mentioned above in Sec. 6.5.1, given an environment e, to solve our constraint solver either succeeds and returns a final success state of the form $\mathtt{succ}(\Delta, e')$, or fails and returns a final failure state of the form $\mathtt{err}(er)$. Let us make this more formal here by proving that our constraint solver algorithm terminates in either of these states. Note also that success and failure states are irreducible, and our constraint solver is confluent, which means that there

Figure 22 Constraint solver (2 of 2): InternalState $\rightarrow$ FinalState

composition environments

(C1) $\mathtt{slv}(\Delta, \bar{l}, e_1;e_2, e) \quad \rightarrow \quad \mathtt{succ}(\Delta_2, e;e_1';e_2')$,
if $\mathtt{slv}(\Delta, \bar{l}, e_1, \top) \rightarrow^* \mathtt{succ}(\Delta_1, e_1') \wedge \mathtt{slv}(\Delta_1;e_1', \bar{l}, e_2, \top) \rightarrow^* \mathtt{succ}(\Delta_2;e_1', e_2')$

(C2) $\mathtt{slv}(\Delta, \bar{l}, e_1;e_2, e) \quad \rightarrow \quad \mathtt{err}(er)$,
if $\mathtt{slv}(\Delta, \bar{l}, e_1, \top) \rightarrow^* \mathtt{succ}(\Delta_1, e_1') \wedge \mathtt{slv}(\Delta_1;e_1', \bar{l}, e_2, \top) \rightarrow^* \mathtt{err}(er)$

(C3) $\mathtt{slv}(\Delta, \bar{l}, e_1;e_2, e) \quad \rightarrow \quad \mathtt{err}(er)$,
if $\mathtt{slv}(\Delta, \bar{l}, e_1, \top) \rightarrow^* \mathtt{err}(er)$

accessors

(A1) $\mathtt{slv}(\Delta, \bar{l}, \uparrow id{=}v, e') \quad \rightarrow \quad \mathtt{slv}(\Delta, \bar{l}, ct = v, e')$,
if $\Delta(id), ren \xrightarrow{\mathsf{instance}} ct \wedge \mathsf{dj}(\mathsf{vars}(\langle \Delta, v \rangle), \mathsf{ran}(ren))$

(A2) $\mathtt{slv}(\Delta, \bar{l}, \uparrow id{=}v, e') \quad \rightarrow \quad \mathtt{succ}(\Delta, e')$,
if $\Delta(id)$ undefined

polymorphic environments

(P1) $\mathtt{slv}(\Delta, \bar{l}, \mathtt{poly}(e), e_1) \quad \rightarrow \quad \mathtt{succ}(\Delta', e_1;\mathsf{toPoly}(\Delta', e_2))$,
if $\mathtt{slv}(\Delta, \bar{l}, e, \top) \rightarrow^* \mathtt{succ}(\Delta', e_2)$

(P2) $\mathtt{slv}(\Delta, \bar{l}, \mathtt{poly}(e), e_1) \quad \rightarrow \quad \mathtt{err}(er)$,
if $\mathtt{slv}(\Delta, \bar{l}, e, \top) \rightarrow^* \mathtt{err}(er)$

is only one (success or failure) final state that a state can reduce to.

Lemma 6.4 (Termination of constraint solver). *Either* $\mathtt{slv}(\Delta, \bar{l}, e, e') \rightarrow^* \mathtt{succ}(\Delta, e)$ *or* $\mathtt{slv}(\Delta, \bar{l}, e, e') \rightarrow^* \mathtt{err}(er)$.

Proof. We use the following measure: given a state of the form $\mathtt{err}(er)$ or $\mathtt{succ}(\Delta, e)$, we return 0, and given a state of the form $\mathtt{slv}(\Delta, \bar{l}, e, e')$, we return $\mathsf{size}(\Delta, e)$ (defined below). We then have to prove that whenever $\mathtt{slv}(\Delta_1, \bar{l}_1, e_1, e_1') \rightarrow \mathtt{slv}(\Delta_2, \bar{l}_2, e_2, e_2')$, then $\mathsf{size}(\Delta_2, e_2) < \mathsf{size}(\Delta_1, e_1)$, and similarity for recursive calls to slv. I.e., we will have to inspect the rules (R), (S2), (S3), (S4), (S5), (U5), (X), (D), (C1), (C2), (C3), (A1), (P1), and (P2). First, let us define the size function:

$$
\begin{array}{lcl}
\mathsf{size}(\Delta, v) & = & \left\{\begin{array}{ll} 1 + \mathsf{size}(\Delta, ct), & \text{if } \Delta(\alpha) = ct \\ 1, & \text{otherwise} \end{array}\right. \\
\mathsf{size}(\Delta, \tau\, \mu) & = & 3 + \mathsf{size}(\Delta, \mu) + \mathsf{size}(\Delta, \tau) \\
\mathsf{size}(\Delta, \tau_1 \to \tau_2) & = & 3 + \mathsf{size}(\Delta, \tau_1) + \mathsf{size}(\Delta, \tau_2) \\
\mathsf{size}(\Delta, \downarrow id{=}\sigma) & = & 1 + \mathsf{size}(\Delta, \sigma) \\
\mathsf{size}(\Delta, \forall \overline{v}.\, ct) & = & 1 + \mathsf{size}(\Delta, ct) \\
\mathsf{size}(\Delta, \uparrow id{=}v) & = & 3 + \mathsf{size}(\Delta, v) \\
\mathsf{size}(\Delta, ct_1 = ct_2) & = & \left\{\begin{array}{l} 2 + \mathsf{size}(\Delta, ct_1) + \mathsf{size}(\Delta, ct_2), \\ \quad \text{if } ct_2 \in \mathsf{Var} \cup \mathsf{Dependent} \\ \quad \text{and } ct_1 \notin \mathsf{Var} \cup \mathsf{Dependent} \\ 1 + \mathsf{size}(\Delta, ct_1) + \mathsf{size}(\Delta, ct_2), \\ \quad \text{otherwise} \end{array}\right. \\
\mathsf{size}(\Delta, \mathtt{poly}(e)) & = & 1 + \mathsf{size}(\Delta, e) \\
\mathsf{size}(\Delta, \exists a.e) & = & 1 + \mathsf{size}(\Delta, e) \\
\mathsf{size}(\Delta, e_1; e_2) & = & 1 + \mathsf{size}(\Delta, e_2) + \mathsf{size}(\Delta, e_1) \\
\mathsf{size}(\Delta, ct^{\overline{l}}) & = & 1 + \mathsf{size}(\Delta, ct) \\
\mathsf{size}(\Delta, ct) & = & 1, \text{otherwise}
\end{array}
$$

Let us now inspect the above mentioned rules. The only non-trivial ones are (R), (S3), (S4), (U5), and (A1). We have a $2 + _$ in the definition of $\mathsf{size}(\Delta, ct_1 = ct_2)$ to handle rule (R). We have a $3 + _$ in the definitions of $\mathsf{size}(\Delta, \tau\, \mu)$ and $\mathsf{size}(\Delta, \tau_1 \to \tau_2)$ to handle rules (S3) and (S4). We use 3 instead of 2 because these rules generate new equality constraints, and as mentioned above we sometimes add 2 instead of 1 when computing the size of equality constraints. Similarity, we use $3 + _$ in the definition of $\mathsf{size}(\Delta, \uparrow id{=}v)$ to handle rule (A1). Finally, we look up inside Δ in the definition of $\mathsf{size}(\Delta, v)$ in order to deal with rule (U5).
□

6.6. Minimization and enumeration

So far, Skalpel has taken a program, labeled it, and has generated and solved constraints. If the output of the constraint solver is successful, then the program is typable and we do not generate an error. Alternatively, if the constraint solver detects an error, then the error produced from that algorithm contains a set of labels. However, not all these labels might be responsible for the error. To find the minimal set responsible for the error, we run our minimization algorithm which we explain in Sec. 6.6.3. For minimization to take place successfully, we need to determine precisely which program parts represented in the error are responsible for the error produced. We do this by experimenting with these parts, i.e., by automatically adding and removing them, e.g., via dummy binders or via constraint filtering as given in Sec. 6.6.2. The minimization algorithm is called by our enumeration algorithm, which is given in 6.6.5 and which is in charge of enumerating all minimal errors.

6.6.1. Why is minimization necessary?

Given an environment generated for a piece of code, i.e., given e such that $[\![strdec]\!] = e$ for a given *strdec*, our enumeration algorithm (see Sec. 6.6.5) works as follows: it selects a filter from its search space, it filters out the constraints labeled by the filter in the environment and runs the constraint solver on the filtered environment. If the constraint solver succeeds (terminates in a success state) then the enumerator keeps searching for type errors using the rest of the search space. If the constraint solver fails (terminates in an error state) then the enumerator has found a new error. This new error might not be minimal. In that case, the enumerator runs the minimizer on the found error and once a minimal error has been found, keeps searching for other type errors. The minimizer is necessary because when the constraint solver returns an error at enumeration, this error might not be minimal. A simple example is as follows:

```
val rec f = fn x => (x (fn z => z), x (fn () => ()))
val rec g = fn y => y true
val u = f g
```

This piece of code is untypable and the highlighting of one of the type errors of this piece of code is as follows:

```
val rec f = fn x => (x (fn z => z), x (fn () => ()))
val rec g = fn y => y true
val u = f g
```

The corresponding type error slice (see Sec. 6.7) is (we have adapted the slice returned by our implementation to the restricted language presented in this document):

```
⟨..val rec f = fn x => ⟨..x (fn () => ⟨..⟩)..⟩
 ..val rec g = fn y => ⟨..y true..⟩
 ..f g..⟩
```

The issue is that because of the first component returned by the function `f` (i.e., the application `x (fn z => z)`) and because of `x`'s monomorphism, when the error presented above is first found at enumeration, it is not minimal. The error first found by the enumerator, before minimization, is as follows:

```
⟨..val rec f = fn x => ⟨..x (fn z => ⟨..⟩)..x (fn () => ⟨..⟩)..⟩
 ..val rec g = fn y => ⟨..y true..⟩
 ..f g..⟩
```

Because `x` is monomorphic, it is constrained by both `z` and `()`. This is a typical example that shows the necessity of the minimization algorithm. We

Figure 23 Constraint filtering

$$\mathsf{filt}(e,\bar{l}_1,\bar{l}_2) = \mathsf{filt}'\left(\begin{array}{ll} & \{l \mapsto \mathtt{keep} \mid l \in \bar{l}_1 \setminus \bar{l}_2\} \\ e, \cup & \{l \mapsto \mathtt{keep\text{-}only\text{-}binders} \mid l \in \bar{l}_2\} \\ \cup & \{l \mapsto \mathtt{drop} \mid l \in \mathsf{labs}(e) \setminus (\bar{l}_1 \cup \bar{l}_2)\}\end{array}\right)$$

$$\mathsf{filt}'(e^{\bar{l}}, de) = \begin{cases} \mathsf{filt}'(e,de)^{\bar{l}}, & \text{if } de(\bar{l}) = \{\mathtt{keep}\} \\ \top, & \text{if } \mathtt{drop} \in de(\bar{l}) \\ \mathsf{dum}(e), & \text{if } de(\bar{l}) \setminus \{\mathtt{keep}\} = \{\mathtt{keep\text{-}only\text{-}binders}\} \end{cases}$$

$$\begin{array}{lll} \mathsf{filt}'(e_1 = e_2, de) & = & (\mathsf{filt}'(e_1,de) = \mathsf{filt}'(e_2,de)) \\ \mathsf{filt}'(e_1;e_2, de) & = & \mathsf{filt}'(e_1,de);\mathsf{filt}'(e_2,de) \\ \mathsf{filt}'(\mathtt{poly}(e), de) & = & \mathtt{poly}(\mathsf{filt}'(e,de)) \\ \mathsf{filt}'(\exists a.e, de) & = & \exists a.\mathsf{filt}'(e,de) \\ \mathsf{filt}'(e, de) & = & e, \text{ otherwise} \end{array}$$

conversion of environments into dummy environments

$$\begin{array}{lllllll} \mathsf{dum}(\downarrow vid{=}ts) & = & \downarrow vid{=}\forall\alpha.\,\alpha & & \mathsf{dum}(ev) & = & \exists ev.ev \\ \mathsf{dum}(\downarrow tv{=}ts) & = & \downarrow tv{=}\forall\alpha.\,\alpha & & \mathsf{dum}(e_1;e_2) & = & \mathsf{dum}(e_1);\mathsf{dum}(e_2) \\ \mathsf{dum}(\downarrow tc{=}tcs) & = & \downarrow tc{=}\forall\delta.\,\delta & & \mathsf{dum}(\mathtt{poly}(e)) & = & \mathsf{dum}(e) \\ \mathsf{dum}(\downarrow strid{=}es) & = & \downarrow strid{=}\forall ev.\,ev & & \mathsf{dum}(\exists a.e) & = & \mathsf{dum}(e) \end{array}$$

$$\begin{array}{lllllll} \mathsf{dum}(e^{\bar{l}}) & = & \mathsf{dum}(e) & & \mathsf{dum}(acc) & = & \top \\ \mathsf{dum}(c) & = & \top & & \mathsf{dum}(\top) & = & \top \end{array}$$

have not yet found a way to directly obtain the first slice presented above without the help of the minimizer. The investigation of such a system is left for future work. (See also design principle **DP5** in Sec. 7 for more information.)

6.6.2. Dummy binders and constraint filtering

To determine precisely which program parts are responsible for the error, we experiment by removing potentially large sections of code, such as structures, datatype definitions, etc., because we want to deal with the skeletal structure of the program. We do this by replacing binders with dummy binders. For that we define the function lBinds, which gathers the labels of all binders in a given environment e:

$$\mathsf{lBinds}(e) = \{l \mid bind^l \text{ occurs in } e\}$$

The definition of the constraint filtering function $\mathsf{filt} : \mathsf{Env} \times \mathbb{P}(\mathsf{Label}) \times \mathbb{P}(\mathsf{Label}) \rightarrow \mathsf{Env}$ is given in Fig. 23 (where $\mathsf{filt}' : \mathsf{Env} \times \mathsf{DepEnv} \rightarrow \mathsf{Env}$). This function is used to check the solvability of constraints in the case that some have been discarded, and we use label sets to accomplish this. In $\mathsf{filt}(e,\bar{l}_1,\bar{l}_2)$, e is the environment we want to filter, and $\bar{l}_1$ contains the labels that we want to keep. Any accessors or equality constraints annotated with a label which is present in the $\bar{l}_2$ set, we do not want to keep. If a binder label is in that set $\bar{l}_2$, then we turn it into a dummy binder. Any environment labeled with a label which is not in $\bar{l}_1 \cup \bar{l}_2$ is discarded. Note that we distinguish binders that we discard, i.e., any binder with a label not in $\bar{l}_1 \cup \bar{l}_2$, and the binders that we turn into

Figure 24 Minimization algorithm

(MIN1)	$\langle e, \bar{l}_1, \{l\} \uplus \bar{l}_2 \rangle$	$\rightarrow_{\mathsf{test}}$	$\langle e, \bar{l}_1 \cap \bar{l}, \bar{l}_2 \cap \bar{l} \rangle$,	if $\mathsf{filt}(e, \bar{l}_1 \cup \bar{l}_2, \{l\}) \xrightarrow{\mathsf{isErr}} \langle ek, \bar{l} \rangle$
(MIN2)	$\langle e, \bar{l}_1, \{l\} \uplus \bar{l}_2 \rangle$	$\rightarrow_{\mathsf{test}}$	$\langle e, \bar{l}_1 \cup \{l\}, \bar{l}_2 \rangle$,	if $\mathsf{solvable}(\mathsf{filt}(e, \bar{l}_1 \cup \bar{l}_2, \{l\}))$
(MIN3)	$\langle e, er \rangle$	$\xrightarrow{\mathsf{min}}$	er',	

if $\mathsf{lBinds}(e) = \bar{l}$
$\wedge \langle e, \mathsf{labs}(er) \setminus \bar{l}, \mathsf{labs}(er) \cap \bar{l} \rangle \rightarrow^*_{\mathsf{test}} \langle e, \bar{l}_1, \varnothing \rangle$(Phase 1)
$\wedge \langle e, \varnothing, \bar{l}_1 \rangle \rightarrow^*_{\mathsf{test}} \langle e, \bar{l}_2, \varnothing \rangle$(Phase 2)
$\wedge \mathsf{filt}(e, \bar{l}_2, \varnothing) \xrightarrow{\mathsf{isErr}} er'$

dummy binders, i.e., those in $\bar{l}_2$, so that when throwing away an environment, we make sure that accessors in the resulting environment do not get captured by a different binder with the same name (see Sec. 6.6.4). We use elements of the sets DepStatus resp. DepEnv defined in Sec. 6.3.3 to keep or drop binders resp. to map a dependency to a dependency status.

Anticipating Sec. 6.6.4, given a user program $\mathcal{P}$, and constraints e for that program of the form $\downarrow vid \stackrel{l}{=} \sigma; e_1; \uparrow vid \stackrel{l'}{=} \alpha$, where e_1 contains a binder of the form $\downarrow vid \stackrel{l''}{=} \sigma'$, we will not filter out the parts of $\mathcal{P}$ responsible for e_1 and create e' such that the accessor to *vid* will be connected to a previously existing binding. That is, α will not be connected to σ by rule (A1) of Fig. 22, and will instead be connected to a dummy binder created by $\mathsf{dum}(e_1)$.

6.6.3. Minimization

As mentioned above, the labels attributed to an error discovered by our constraint solver may be extraneous. The minimization algorithm given in Fig. 24 removes unnecessary labels. A *minimal error* is an error which has no extraneous labels, i.e., it is untypable and removing any one label from it makes it typable. The $\rightarrow_{\mathsf{test}}$ relation determines whether a given label can be removed from the set of labels associated with an error without losing a label crucial to the error. We separate the process of minimization into two phases:

- (Phase 1) turns binders into dummy binders which can potentially remove large sections of code.
- (Phase 2) tries to remove labels one at a time until we find the minimal amount of labels attributed to the error.

In a minimization step $\langle e, \bar{l}_1, \{l\} \uplus \bar{l}_2 \rangle \xrightarrow{\mathsf{test}} \langle e, \bar{l}_3, \bar{l}_4 \rangle$, it holds that $\bar{l}_3$ and $\bar{l}_4$ depend upon the solvability of $\mathsf{filt}(e, \bar{l}_1 \cup \bar{l}_2, \{l\})$, which we will call e'. The full set of labels for the error the minimizer is working on is the set $\bar{l}_1 \cup \bar{l}_2 \cup \{l\}$, and $\{l\} \uplus \bar{l}_2$ is the label set we still have to attempt to discard. The new environment e' is obtained from e by:

- filtering out the constraints not labeled by $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$.
- Filtering out accessors and equality constraints annotated with l.
- Creating dummy binders for binders annotated with l, and similarly, dummy environment of the form $\exists ev.ev$ for environment variables annotated with l.

If e' is solvable, then label l must be in the error's label set for the error to occur, and so $\overline{l}_3 = \overline{l}_1 \cup \{l\}$ and $\overline{l}_4 = \overline{l}_2$, i.e., we keep l (rule (MIN2)). If e' is unsolvable then the solver fails and we obtain a new smaller error, which contains strictly less labels. This means that l is extraneous, and any environment labeled by l can be completely filtered out. The label sets $\overline{l}_3$ and $\overline{l}_4$ are then restricted to the newly found error (rule (MIN1)).

The next conjecture says that our minimization algorithm is meant to compute minimal errors, where a minimal error is untypable and such that no remaining labels after minimization are extraneous. (This is simply a conjecture and not a lemma because, as mentioned in the proof sketch below, a full proof would necessitate several extra lemmas regarding our filtering and minimization algorithms. Proving these lemmas is left for future work.)

Conjecture 6.5 (Minimal errors). *After minimizing* $\langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \overline{l}\rangle$, *the two following properties hold:*

1. $\langle e, \langle ek, \overline{l}\rangle\rangle$ *is an error:* $\exists er.\mathsf{filt}(e, \overline{l}, \varnothing) \xrightarrow{\mathsf{isErr}} er$.
2. $\langle e, \langle ek, \overline{l}\rangle\rangle$ *is minimal:* $\forall l \in \overline{l}.\mathsf{solvable}(\mathsf{filt}(e, \overline{l}, \{l\}))$.

Proof Sketch. By definition of the minimization algorithm, we get that there exists $\overline{l}_b$, $\overline{l}_1$, and $\overline{l}_2$ such that $\overline{l}_b = \mathsf{lBinds}(e)$, $\langle e, \mathsf{labs}(er) \setminus \overline{l}_b, \mathsf{labs}(er) \cap \overline{l}_b\rangle \rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_1, \varnothing\rangle$, $\langle e, \varnothing, \overline{l}_1\rangle \rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_2, \varnothing\rangle$, and $\mathsf{filt}(e, \overline{l}_2, \varnothing) \xrightarrow{\mathsf{isErr}} \langle ek, \overline{l}\rangle$. Note that we should be able to derive that:

$$\overline{l} \subseteq \overline{l}_2 \subseteq \overline{l}_1 \subseteq (\mathsf{labs}(er) \setminus \overline{l}_b) \cup (\mathsf{labs}(er) \cap \overline{l}_b)$$

It should be a property of our constraint solver that:

$$\forall e, ek, \overline{l}.e \xrightarrow{\mathsf{isErr}} \langle ek, \overline{l}\rangle \Rightarrow \exists er.\mathsf{filt}(e, \overline{l}, \varnothing) \xrightarrow{\mathsf{isErr}} er$$

Therefore, from $\mathsf{filt}(e, \overline{l}_2, \varnothing) \xrightarrow{\mathsf{isErr}} \langle ek, \overline{l}\rangle$ we should be able to deduce that there exists er such that $\mathsf{filt}(\mathsf{filt}(e, \overline{l}_2, \varnothing), \overline{l}, \varnothing) \xrightarrow{\mathsf{isErr}} er$. Because $\overline{l} \subseteq \overline{l}_2$, we would get $\mathsf{filt}(e, \overline{l}, \varnothing) \xrightarrow{\mathsf{isErr}} er$. Let us now consider the minimality property. Let $l \in \overline{l}$. We have to show that $\mathsf{solvable}(\mathsf{filt}(e, \overline{l}, \{l\}))$. By definition of our minimization algorithm (Phase 2 and rule (MIN2)), there was a minimization state $\langle e, \overline{l}_3, \{l\} \uplus \overline{l}_4\rangle$ such that $\mathsf{solvable}(\mathsf{filt}(e, \overline{l}_3 \cup \overline{l}_4, \{l\}))$ and $\langle e, \overline{l}_3, \{l\} \uplus \overline{l}_4\rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_3 \cup \{l\}, \overline{l}_4\rangle$

$\rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_2, \varnothing\rangle$. Therefore, we have $\overline{l} \subseteq \overline{l}_3 \cup \overline{l}_4$. As mentioned below in Sec. 6.6.4, it should be a property of our constraint solver and minimization algorithm that whenever we apply rule (MIN1) and discard labels, bindings do not get mixed up. Therefore, it should be the case that if $\mathsf{solvable}(\mathsf{filt}(e, \overline{l}_3 \cup \overline{l}_4, \{l\}))$ then $\mathsf{solvable}(\mathsf{filt}(e, \overline{l}, \{l\}))$. In addition we should also be able to derive $\mathsf{filt}(e, \overline{l}, \varnothing) \xrightarrow{\mathsf{isErr}} \langle ek, \overline{l}\rangle$. □

6.6.4. Regarding binding discarding during minimization

Let us describe a step of the first phase of our minimization algorithm. Let $\mathcal{P}$ be our original untypable piece of code, let e be the environment generated for $\mathcal{P}$, and let $\overline{l} \cup \{l\}$ be the label set labeling the slice that is being minimized, where l is associated with a binder *bind* from the slice. We test whether we can remove l (and still obtain a type error slice) by first filtering out the constraints of $\mathcal{P}$ as follows: $\mathsf{filt}(e, \overline{l}, \{l\})$, to obtain e'. In order not to mix up the bindings, at constraint filtering, the binder *bind* associated with l is not discarded but is replaced by a non labelled dummy binder $bind'$ (such that $bind' = \mathsf{dum}(bind)$) that cannot participate in any error but that still acts as a binder. If we then solve e' and obtain an error then no label labeling an accessor to $bind'$ in e' will occur in the found error (we give below an informal argument as why none of these accessors will be part of the new error). The bindings in this new error are then not mixed up[7]. The found error is then the new slice to try to minimize further and next time the constraints will be filtered w.r.t. this new slice, the binder *bind* and its accessors will be completely thrown away (as well as the other constraints not participating in the new error).

Let us consider the following unsolvable environment, which we call e:

$$\alpha_1 \stackrel{l_1}{=} \mathtt{int}; \alpha_2 \stackrel{l_2}{=} \mathtt{unit}; \downarrow vid \stackrel{l_3}{=} \alpha_1; \downarrow vid \stackrel{l_4}{=} \alpha_2; \alpha_3 \stackrel{l_5}{=} \mathtt{unit}; \uparrow vid \stackrel{l_6}{=} \alpha_3; \alpha_1 \stackrel{l_7}{=} \alpha_3$$

The only labels necessary for an error to occur are l_1, l_5 and l_7. Note that *vid*'s accessor refers to *vid*'s binder labelled by l_4 (second binder) and not to the one labelled by l_3 (first binder). Let us run our minimization algorithm on e and let the first step be to try to discard l_4. First, the filtering function is called on e as follows: $\mathsf{filt}(e, \{l_1, l_2, l_3, l_5, l_6, l_7\}, \{l_4\})$, which results in the following environment, called e':

$$\alpha_1 \stackrel{l_1}{=} \mathtt{int}; \alpha_2 \stackrel{l_2}{=} \mathtt{unit}; \downarrow vid \stackrel{l_3}{=} \alpha_1; \downarrow vid {=} \forall \alpha.\, \alpha; \alpha_3 \stackrel{l_5}{=} \mathtt{unit}; \uparrow vid \stackrel{l_6}{=} \alpha_3; \alpha_1 \stackrel{l_7}{=} \alpha_3$$

Let us run our constraint solver on e'. When dealing with the accessor $\uparrow vid \stackrel{l_6}{=}$

[7] Note that bindings can be mixed up in a filtered environment if and only if an accessor refers to a binder to which it does not refer to in the non filtered environment.

α_3, the dummy binder $\downarrow vid{=}\forall\alpha.\,\alpha$ is accessed and the equality constraint $\alpha{=}\alpha_3$ is generated by rule (A1), where α is "fresh enough". Rule (U2) is then used and a binding of the form $\alpha \mapsto \alpha_3^{\bar{l}}$ is added to the current context. Because of the freshness of α, this binding will never be accessed. In effect the accessor and its label are discarded at constraint solving and cannot occur in any error. In our example, the constraint solver terminates in an error state on e', which means that l_4 is unnecessary for an error to occur. The error returned by the constraint solver does not involve l_4 or l_6 and especially, in the next step of the minimization process, *vid*'s accessor cannot refer to *vid*'s first binder.

Note that filtering itself does not prevent bindings to get mixed up because, e.g., filtering allows one to throw away the binder generated for the second occurrence of `x` in `fn x => fn x => x` while not throwing away the binder generated for the first occurrence of `x` and not throwing away its accessor. However, we give below an informal argument as why we never filter a binder without filtering its accessor.

Let us now present an informal argument as why when our constraint solver returns an error, the error does not involve accessors to dummy binders or accessors without their corresponding binders.

According to rules (A1) and (A2), during constraint solving the label labeling an accessor only gets recorded in a constraint solving context Δ of the form $\langle u,\, e\rangle$ if the accessed identifier is in the type environment e stored in Δ in the current state (the state in which the constraint solving process is when the rule applies). There are two possible scenarios. In the environment e (1) either (as mentioned above) the accessed identifier has a dummy static semantics (resulting from filtering) of the form $\forall v.\, v$, and according to rule (U2) the label of the accessor gets recorded into the constraint solving context Δ, but will never be used because of the freshness condition used in rule (A1): given an accessor $\uparrow id{=}v$, according to rule (A1), a constraint of the form $v'{=}v$ is generated, where v' is "fresh enough". (2) Or the accessed identifier has a labelled non-dummy static semantics, and the labels associated with the binder and the label associated with the bound occurrence will always occur together in the constraint solving context. The main point being that in our system if a binder is not a dummy binder then it is labelled. This is why we strongly believe that an identifier occurring at a non-binding position in a sliced piece of code (represented by an accessor in our constraint language) only occurs in a slice if it is bound and its binder occurs in the slice as well.

This informal argument relies on the fact that our labelled external syntax together with our constraint generator enforce that each bound occurrence of an identifier is labelled by a unique label that does not label a larger piece of code and therefore the label labeling an accessor does not label any other constraint term (see our design principle **DP6** in Sec. 7). Therefore in case (1) described above, once the accessor and the generated equality constraint have

Figure 25 Enumeration algorithm

(ENUM1)	$\mathtt{enum}(e)$	$\rightarrow_{\mathsf{e}}$	$\mathtt{enum}'(e, \varnothing, \{\varnothing\})$
(ENUM2)	$\mathtt{enum}'(e, \overline{er}, \varnothing)$	$\rightarrow_{\mathsf{e}}$	$\mathtt{errors}(\overline{er})$
(ENUM3)	$\mathtt{enum}'(e, \overline{er}, \bar{\bar{l}} \uplus \{\bar{l}\})$	$\rightarrow_{\mathsf{e}}$	$\mathtt{enum}'(e, \overline{er}, \bar{\bar{l}})$, if $\mathsf{solvable}(\mathsf{filt}(e, \mathsf{labs}(e), \bar{l}))$
(ENUM4)	$\mathtt{enum}'(e, \overline{er}, \bar{\bar{l}} \uplus \{\bar{l}\})$	$\rightarrow_{\mathsf{e}}$	$\mathtt{enum}'(e, \overline{er} \cup \{\langle ek, \bar{l}\rangle\}, \bar{\bar{l}}' \cup \bar{\bar{l}})$
	if $\mathsf{filt}(e, \mathsf{labs}(e), \bar{l}) \xrightarrow{\mathsf{isErr}} er$		
	$\wedge\ \langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \bar{l}\rangle$		
	$\wedge\ \bar{\bar{l}}' = \{\bar{l} \cup \{l\} \mid l \in \bar{l} \wedge \forall \bar{l}_0 \in \bar{\bar{l}}. \bar{l}_0 \not\subseteq \bar{l} \cup \{l\}\}$		

been dealt with, the label labeling the accessor occurs in the state in which the constraint solver is but cannot be reached, and therefore cannot be part of any error. This would not necessarily be the case with a constraint generator that would generate $\langle \alpha, ((\alpha \overset{l}{=} \alpha_1 \rightarrow \alpha_2);(\uparrow id \overset{l}{=} \alpha_1))\rangle$ for some term. As a matter of fact, we could imagine a scenario where α is further constrained to, e.g., be equal to `int`. We would therefore obtain a type constructor clash between `int` and the arrow type constructor that involves l but does not require the accessor to be resolved. The accessor being kept alive in this error, at the next step of the minimization algorithm, we would have no guarantee that it does not refer to a different binder than the one it refers to (if referring to any) in the non filtered environment.

6.6.5. Enumeration

Our enumeration algorithm filters out those parts of the program which are irrelevant to the error and attempts to find all distinct errors that exist in the user program. We have three enumeration states as follows:

$$\mathsf{EnumState} ::= \mathtt{enum}(e) \mid \mathtt{enum}'(e, \overline{er}, \bar{\bar{l}}) \mid \mathtt{errors}(\overline{er})$$

The enumeration process describes in Fig. 25 starts in the state $\mathtt{enum}(e)$ and ends in the state $\mathtt{errors}(\overline{er})$. The enumeration rule (ENUM1) is only used once to start the enumeration process by creating *filters*, which form the search space built when searching for errors. Initially, there is only one filter, which is the empty filter, causing all constraints to be considered. An intermediate state is of the form $\mathtt{enum}'(e, \overline{er}, \bar{\bar{l}})$ where $\overline{er}$ is the collection of minimal errors found so far, and $\bar{\bar{l}}$ is the collection of filters still to try. Finding new minimal errors leads to new filters being created: after an error has been found and the error has been minimized, we see in rule (ENUM4) that the labels of the minimized error are used to build new filters, namely the set $\bar{\bar{l}}'$. When a filter leads to a solvable filtered environment, the filter is discarded using rule (ENUM3). Once there are no more filters to try, the enumeration process finishes with rule (ENUM2).

After the enumeration algorithm has stopped, the errors that have been

found are all the minimal type errors in the analyzed piece of code. The next conjecture says that errors found during enumeration and presented to the user are meant to be minimal. (This is simply a conjecture and not a lemma because, as mentioned in the proof sketch below, a full proof would necessitate several extra lemmas regarding our filtering, minimization, and enumeration algorithms. Proving these lemmas is left for future work.)

Conjecture 6.6 (Enumeration of minimal errors). *Given an enumeration output* $\texttt{errors}(\overline{er})$, $\forall er \in \overline{er}$, *it holds that* er *is a minimal error.*

PROOF SKETCH. When we detect an error er during enumeration (rule (ENUM4)), we will always perform the step $\langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \bar{l}\rangle$. Conjecture 6.5 says that this error is meant to be minimal. As $\langle ek, \bar{l}\rangle \in \overline{er}$, and we do not locate errors in other enumeration rules nor do we add or remove labels from errors during enumeration, we retain the property that these errors are minimal. □

6.7. Slicing

After all minimal errors have been enumerated, Skalpel knows exactly those parts of the program responsible for the error. Next, the user must be informed of those parts, in the most helpful manner. To do so, we produce a *type error slice* (as discussed e.g. in [25] and [50]), which is the program the user submitted to Skalpel with all irrelevant pieces of code removed and replaced by *dot terms*.

6.7.1. Dot terms

When Skalpel locates an error $\langle ek, \bar{l}\rangle$ in the user program, it makes a type error slice from the labels $\bar{l}$ and the error kind ek. This is done by the slicing function sl defined below in Sec 6.7.3. Program nodes annotated with labels not occurring in the set of labels as part of the error are replaced by "dot" terms, which are used to show that some program nodes have been thrown away as they do not contribute to the error. Fig. 26 extend our external labeled syntax and our constraint generator to dot terms. A *slice* is any syntactic form that can be produced using the grammar rules defined in Fig. 26 and Fig. 9. A *type error slice* is a slice for which the constraint generator (which has been extended to dot terms) only generates unsolvable constraints. Let us consider the labeled program $\lceil 1^{l_1} ()^{l_2} \rceil^{l_3}$. This is an error, as 1 is being used as a function and applied to (). In this case, we do not want to report in our error what 1 is being applied to, as it is irrelevant. What matters is that the application itself exists. For this reason we remove the node annotated with label l_2, giving the labeled program $\lceil 1^{l_1} \texttt{dot-e}(\varnothing) \rceil^{l_3}$. This is displayed to the user as 1 $\langle .. \rangle$.

Fig. 27 provides an alternative generic definition of our external labeled syntax. This definition helps defining functions such as our slicing algorithm in

Figure 26 Constraint syntax/generator for "dot" terms

LabTyCon	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
LabDatCon	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
Ty	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
ConBind	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
DatName	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
Dec	::=	$\cdots$ | $\mathtt{dot\text{-}d}(\overrightarrow{term})$
AtExp	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
Exp	::=	$\cdots$ | $\mathtt{dot\text{-}e}(\overrightarrow{term})$
AtPat	::=	$\cdots$ | $\mathtt{dot\text{-}p}(\overrightarrow{pat})$
Pat	::=	$\cdots$ | $\mathtt{dot\text{-}p}(\overrightarrow{pat})$
StrDec	::=	$\cdots$ | $\mathtt{dot\text{-}d}(\overrightarrow{term})$
StrExp	::=	$\cdots$ | $\mathtt{dot\text{-}s}(\overrightarrow{term})$

(G24) $[\![\mathtt{dot\text{-}d}(\langle term_1, \ldots, term_n \rangle)]\!] = [[\![term_1]\!]; \cdots ;[\![term_n]\!]]$
(G25) $[\![\mathtt{dot\text{-}p}(\langle pat_1, \ldots, pat_n \rangle), \alpha]\!] = [\![pat_1]\!]; \cdots ;[\![pat_n]\!]$
(G26) $[\![\mathtt{dot\text{-}s}(\langle term_1, \ldots, term_n \rangle), ev]\!] = [[\![term_1]\!]; \cdots ;[\![term_n]\!]]$
(G27) $[\![\mathtt{dot\text{-}e}(\langle term_1, \ldots, term_n \rangle), \alpha]\!] = [[\![term_1]\!]; \cdots ;[\![term_n]\!]]$

Figure 27 Generic definition of our external labeled syntax

class	$\in$	Class	::= `lTc` | `lDcon` | `ty` | `conbind` | `datname` | `dec` | `atexp` | `exp` | `atpat` | `pat` | `strdec` | `strexp`
prod	$\in$	Prod	::= `tyArr` | `tyCon` | `conbindOf` | `datnameCon` | `decRec` | `decDat` | `decOpn` | `atexpLet` | `expFn` | `strdecDec` | `strdecStr` | `strexpSt` | `id` | `app` | `seq`
dot	$\in$	Dot	::= `dotE` | `dotP` | `dotD` | `dotS`
node	$\in$	Node	::= $\langle class, prod \rangle$
tree	$\in$	Tree	::= $\langle node, l, \overrightarrow{tree} \rangle$ | $\langle dot, \overrightarrow{tree} \rangle$ | id

a compact way. A node in a tree *tree* can either be a labeled node of the form $\langle node, l, \overrightarrow{tree} \rangle$, an unlabeled "dot" node of the form $\langle dot, \overrightarrow{tree} \rangle$, or a *leaf* of the form id. Fig. 28, defines the function toTree : ExtLabSynt $\rightarrow$ Tree, which takes a piece of external labeled syntax and produces a generic tree form.

We now define getDot : Class $\times$ Prod $\rightarrow$ Dot, which generates terms in Dot from nodes. This function is used in our slicing algorithm to generate dot nodes from labeled nodes.

getDot($\langle x, prod \rangle$)=`dotE`, if x $\in$ {`lTc`, `lDcon`, `ty`, `conbind`, `datname`, `atexp`, `exp`}
getDot($\langle x, prod \rangle$)=`dotD`, if x $\in$ {`dec`, `strdec`}
getDot($\langle x, prod \rangle$)=`dotP`, if x $\in$ {`atpat`, `pat`}
getDot($\langle x, prod \rangle$)=`dotS`, if x $\in$ {`strexp`}

Figure 28 From terms to trees

$$
\begin{array}{lcl}
\mathsf{toTree}(tv^{l}) & = & \langle\langle \mathtt{ty}, \mathtt{id}\rangle, l, \langle tv\rangle\rangle \\
\mathsf{toTree}(dcon^{l}) & = & \langle\langle \mathtt{lDcon}, \mathtt{id}\rangle, l, \langle dcon\rangle\rangle \\
\mathsf{toTree}(tc^{l}) & = & \langle\langle \mathtt{lTc}, \mathtt{id}\rangle, l, \langle tc\rangle\rangle \\
\mathsf{toTree}(vid^{l}_{\mathtt{p}}) & = & \langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l, \langle vid\rangle\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}e}(\overrightarrow{term})) & = & \langle \mathtt{dotE}, \mathsf{toTree}(\overrightarrow{term})\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}d}(\overrightarrow{term})) & = & \langle \mathtt{dotD}, \mathsf{toTree}(\overrightarrow{term})\rangle \\
\mathsf{toTree}(\mathtt{open}^{l}\ strid) & = & \langle\langle \mathtt{dec}, \mathtt{decOpn}\rangle, l, \langle strid\rangle\rangle \\
\mathsf{toTree}(vid^{l}_{\mathtt{e}}) & = & \langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l, \langle vid\rangle\rangle \\
\mathsf{toTree}(dcon^{l}_{\mathtt{c}}) & = & \langle\langle \mathtt{conbind}, \mathtt{id}\rangle, l, \langle dcon\rangle\rangle \\
\mathsf{toTree}(strid^{l}) & = & \langle\langle \mathtt{strexp}, \mathtt{id}\rangle, l, \langle strid\rangle\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}p}(\overrightarrow{pat})) & = & \langle \mathtt{dotP}, \mathsf{toTree}(\overrightarrow{pat})\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}s}(\overrightarrow{term})) & = & \langle \mathtt{dotS}, \mathsf{toTree}(\overrightarrow{term})\rangle \\
\mathsf{toTree}(ty_1 \stackrel{l}{\rightarrow} ty_2) & = & \langle\langle \mathtt{ty}, \mathtt{tyArr}\rangle, l, \langle \mathsf{toTree}(ty_1), \mathsf{toTree}(ty_2)\rangle\rangle \\
\mathsf{toTree}(\lceil ty\ ltc\rceil^{l}) & = & \langle\langle \mathtt{ty}, \mathtt{tyCon}\rangle, l, \langle \mathsf{toTree}(ty), \mathsf{toTree}(ltc)\rangle\rangle \\
\mathsf{toTree}(dcon\ \mathtt{of}^{\,l}\ ty) & = & \langle\langle \mathtt{conbind}, \mathtt{conbindOf}\rangle, l, \langle dcon, \mathsf{toTree}(ty)\rangle\rangle \\
\mathsf{toTree}(\lceil tv\ tc\rceil^{l}) & = & \langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l, \langle tv, tc\rangle\rangle \\
\mathsf{toTree}(\mathtt{val\ rec}\ pat \stackrel{l}{=} exp) & = & \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l, \langle \mathsf{toTree}(pat), \mathsf{toTree}(exp)\rangle\rangle \\
\mathsf{toTree}(\mathtt{datatype}\ dn \stackrel{l}{=} cb) & = & \langle\langle \mathtt{dec}, \mathtt{decDat}\rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(cb)\rangle\rangle \\
\mathsf{toTree}(\mathtt{let}^{l}\ dec\ \mathtt{in}\ exp\ \mathtt{end}) & = & \langle\langle \mathtt{atexp}, \mathtt{atexpLet}\rangle, l, \langle \mathsf{toTree}(dec), \mathsf{toTree}(exp)\rangle\rangle \\
\mathsf{toTree}(\mathtt{fn}\ pat \stackrel{l}{\Rightarrow} exp) & = & \langle\langle \mathtt{exp}, \mathtt{expFn}\rangle, l, \langle \mathsf{toTree}(pat), \mathsf{toTree}(exp)\rangle\rangle \\
\mathsf{toTree}(\lceil exp\ atexp\rceil^{l}) & = & \langle\langle \mathtt{exp}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(exp), \mathsf{toTree}(atexp)\rangle\rangle \\
\mathsf{toTree}(\lceil ldcon\ atpat\rceil^{l}) & = & \langle\langle \mathtt{pat}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(ldcon), \mathsf{toTree}(atpat)\rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{toTree}(\mathtt{structure}\ strid \stackrel{l}{=} strexp) \\
\qquad = \langle\langle \mathtt{strdec}, \mathtt{strdecStr}\rangle, l, \langle strid, \mathsf{toTree}(strexp)\rangle\rangle \\
\mathsf{toTree}(\mathtt{struct}^{l}\ strdec_1 \cdots strdec_n\ \mathtt{end}) \\
\qquad = \langle\langle \mathtt{strexp}, \mathtt{strexpSt}\rangle, l, \mathsf{toTree}(\langle strdec_1, \ldots, strdec_n\rangle)\rangle \\
\mathsf{toTree}(\langle term_1, \ldots, term_n\rangle) \\
\qquad = \langle \mathsf{toTree}(term_1), \ldots, \mathsf{toTree}(term_n)\rangle
\end{array}
$$

6.7.2. Tidying

In order for the output slice computed by Skalpel not to be cluttered with extraneous symbols indicating where parts of programs are omitted, we define two functions flat and tidy that make this output elegant. The function flat flattens nested dot terms. For example, flattening $\langle..1..\langle..()..\rangle..\rangle$ gives $\langle..1..()..\rangle$. Not all nested dot terms are flattened. In order not to mix up bindings in a slice, we do not let declarations escape dot terms. For example we do not flatten

$$\langle..\mathtt{val\ x\ =\ false}..\langle..\mathtt{val\ x\ =\ 1}..\rangle..\mathtt{x\ +\ 1}..\rangle$$

to become

$$\langle..\mathtt{val\ x\ =\ false}..\mathtt{val\ x\ =\ 1}..\mathtt{x\ +\ 1}..\rangle$$

because the semantics has changed: the first is a typable slice and the second is not. In the first slice, the third occurrence of `x` is bound by the first, while in

the second slice it is bound by the second. In order to define our flat and tidy functions, we define the predicates $\mathsf{isClass}$, $\mathsf{declares}$ and $\mathsf{pattern}$, which are used to check whether a given tree has any binders or is a pattern (in the rest these predicates are considered as functions returning Booleans—they are decidable):

$$
\begin{array}{ll}
\mathsf{isClass}(tree, \{class\} \cup \overline{class}) & \iff tree = \langle\langle class, prod\rangle, l, \overrightarrow{tree}\rangle \\
\mathsf{declares}(tree) & \iff \mathsf{isClass}(tree, \{\mathtt{dec}, \mathtt{strdec}, \mathtt{datname}, \mathtt{conbind}\}) \\
\mathsf{pattern}(tree) & \iff \mathsf{isClass}(tree, \{\mathtt{atpat}, \mathtt{pat}\})
\end{array}
$$

With these in place, we define $\mathsf{flat} : \mathsf{tuple}(\mathsf{Tree}) \to \mathsf{tuple}(\mathsf{Tree})$ as follows:

$$
\begin{array}{lll}
\mathsf{flat}(\langle\rangle) & = & \langle\rangle \\
\mathsf{flat}(\langle tree\rangle @ \overrightarrow{tree}) & = & \begin{cases} \langle tree_1, \ldots, tree_n\rangle @ \mathsf{flat}(\overrightarrow{tree}), \\ \quad \text{if } tree = \langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle \\ \quad \wedge(\forall i \in \{1, \ldots, n\}.\neg\mathsf{declares}(tree_i) \text{ or } \overrightarrow{tree} = \langle\rangle) \\ \langle tree\rangle @ \mathsf{flat}(\overrightarrow{tree}), \text{otherwise} \end{cases}
\end{array}
$$

The condition "$\forall i \in \{1, \ldots, n\}.\neg\mathsf{declares}(tree_i)$" ensures that bindings are not mixed up as explained above. However, flattening the last dot term (if it actually is a dot term) cannot mix up the bindings because there is no identifier left to bind. Therefore, flattening ⟨`..val x = 1..`⟨`..val x = true..`⟩`..`⟩ would lead to ⟨`..val x = 1..val x = true..`⟩. We however have not yet found a concrete example where this situation occurs.

The function tidy merges dot terms containing declarations in structures:

$$
\begin{array}{l}
\mathsf{tidy}(\langle\rangle) = \langle\rangle \\
\mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{tree}_1\rangle, \langle \mathtt{dotD}, \overrightarrow{tree}_2\rangle\rangle @ \overrightarrow{tree}) \\
\quad = \mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{tree}_1 @ \overrightarrow{tree}_2\rangle\rangle @ \overrightarrow{tree}), \text{ if } \forall tree \in \mathsf{ran}(\overrightarrow{tree}_1).\neg\mathsf{declares}(tree) \\
\mathsf{tidy}(\langle\langle \mathtt{dotD}, \varnothing\rangle\rangle @ \overrightarrow{tree}) \\
\quad = \mathsf{tidy}(\overrightarrow{tree}), \text{ if none of the above applies} \\
\mathsf{tidy}(\langle tree\rangle @ \overrightarrow{tree}) \\
\quad = \langle tree\rangle @ \mathsf{tidy}(\overrightarrow{tree}), \text{ if none of the above applies}
\end{array}
$$

6.7.3. Algorithm

Fig. 29 formally defines the slicing algorithm. In this figure, $\mathsf{sl}(strdec, \bar{l})$ abbreviates $\mathsf{sl}(\mathsf{toTree}(strdec), \bar{l})$. Note that $\mathsf{sl}, \mathsf{sl}_1, \mathsf{sl}_2 : \mathsf{Tree} \times \bar{l} \to \mathsf{Tree}$.

6.7.4. Properties

After sending an erroneous program as input to Skalpel, errors are generated that show all and only the parts of the program that are responsible for the

Figure 29 Slicing algorithm

$$(\mathsf{SL1})\ \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \bar{l}) = \begin{cases} \langle node, l, \mathsf{sl}_1(\overrightarrow{tree}, \bar{l})\rangle, \\ \qquad \text{if } l \in \bar{l} \wedge \mathsf{getDot}(node) \neq \mathtt{dotS} \\ \langle node, l, \mathsf{tidy}(\mathsf{sl}_1(\overrightarrow{tree}, \bar{l}))\rangle, \\ \qquad \text{if } l \in \bar{l} \wedge \mathsf{getDot}(node) = \mathtt{dotS} \\ \langle \mathsf{getDot}(node), \mathsf{flat}(\mathsf{sl}_2(\overrightarrow{tree}, \bar{l}))\rangle, \\ \qquad \text{otherwise} \end{cases}$$

(SL2)	$\mathsf{sl}_1(\langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle, \bar{l})$	$=$	$\langle dot, \mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \bar{l}), \ldots, \mathsf{sl}_2(tree_n, \bar{l})\rangle)\rangle$
(SL3)	$\mathsf{sl}_2(\langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle, \bar{l})$	$=$	$\langle dot, \mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \bar{l}), \ldots, \mathsf{sl}_2(tree_n, \bar{l})\rangle)\rangle$
(SL4)	$\mathsf{sl}_1(\langle node, l, \overrightarrow{tree}\rangle, \bar{l})$	$=$	$\mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \bar{l})$
(SL5)	$\mathsf{sl}_2(\langle node, l, \overrightarrow{tree}\rangle, \bar{l})$	$=$	$\mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \bar{l})$
(SL6)	$\mathsf{sl}_1(\langle tree_1, \ldots, tree_n\rangle, \bar{l})$	$=$	$\langle \mathsf{sl}_1(tree_1, \bar{l}), \ldots, \mathsf{sl}_1(tree_n, \bar{l})\rangle$
(SL7)	$\mathsf{sl}_2(\langle tree_1, \ldots, tree_n\rangle, \bar{l})$	$=$	$\langle \mathsf{sl}_2(tree_1, \bar{l}), \ldots, \mathsf{sl}_2(tree_n, \bar{l})\rangle$
(SL8)	$\mathsf{sl}_1(id, \bar{l})$	$=$	id
(SL9)	$\mathsf{sl}_2(id, \bar{l})$	$=$	$\langle \mathtt{dotE}, \langle\rangle\rangle$

error to the user. Given a user program, Skalpel labels it producing $\mathcal{P}$ and runs $\mathsf{cstgen}(\mathcal{P})$ to produce the constraint/environment e. Skalpel then runs $\mathsf{enum}(e)$ to find all the distinct errors by generating filters of the form $\bar{l}$. Given a filter $\bar{l}$, if $\mathsf{filt}(e, \mathsf{labs}(e), \bar{l}) \xrightarrow{\mathsf{isErr}} er$, then Skalpel computes a minimal error by running: $\langle e, er\rangle \overset{\mathsf{min}}{\to} \langle ek, \bar{l}'\rangle$. Eventually, Skalpel computes the minimal error set $\overline{er}$, and performs slicing on each error in $\overline{er}$ to present them clearly to the user.

Note that using the slicing algorithm presented in Fig. 29, only those parts of the program which they wrote and contribute to an error are shown to the user, that is, syntax (apart from dots) is not shown which was not written by the user and which does not contribute to the error.

7. Design Principles

While developing Skalpel we discovered, developed, and followed the following principles.

DP1: **Variables.** Each syntactic sort of constraint terms should have a case ranging over an infinite variable set. This allows incomplete information everywhere, which allows one to consider every possible way of slicing out parts of the program. This is essential to get precise slices that include all relevant details and exclude the irrelevant. Thus, the sorts μ, τ, and e have the variable cases δ, α, and ev.

DP2: **Dependencies.** Each syntactic sort of constraint terms should support dependencies. This allows precise blame, enabling precise slicing. Thus, sorts μ, τ, σ, and e have dependency cases $\langle \mu, \overline{d}\rangle$, $\langle \tau, \overline{d}\rangle$, $\langle \sigma, \overline{d}\rangle$, and $\langle e, \overline{d}\rangle$.

DP3: **Connections.** Our constraint generator returns an environment used for constraints and bindings. The generated constraints may connect information from the results for a program node's subtrees to the other subtrees or to the node's results.

The principle is that these connections should generally be via constraints that carry the syntax tree node's label and that are "shallow", i.e., contain only connection details and not constraints from program subtrees. Fresh variables should be used as needed. This allows a program syntax node to be "disconnected" for type errors that depend on the node's details, while still keeping type errors that arise solely due to connections between environment accessors and bindings that pass through the node.

For example, rule (G22) of our constraint generator defined in Fig. 19 in Sec. 6.4 builds the unlabeled constraint $ev' = (e_1; \cdots ; e_n)$. This "deep" unlabeled constraint packs together a sequence of environments generated from the declarations in the corresponding structure. The resulting environment is connected to the main result by the labeled shallow constraint $ev \stackrel{l}{=} ev'$.

DP4: **Irredundant.** Duplicating constraints should be unnecessary. This seems obvious, but some previous formalisms seem too weak for the needed sharing. For example, rule (G22) of our constraint generator defined in Fig. 19 in Sec. 6.4 builds a structure's environment as the sequential composition of its component declarations' environments: $e_1; \cdots ; e_n$. Here, the first declaration's environment e_1 is available for subsequent declarations and also in the result (if its bindings are not shadowed) which avoids duplicating it. A previous version of our system had a weaker constraint system with let-constraints similar to those of Pottier and Rémy [48], and the best solution we could find duplicated the constraints for each declaration's bindings, causing severe performance problems.

DP5: **Propagation.** Dependencies must be propagated during solving exactly where needed. If dependencies are not propagated where they "should", minimization could over-minimize yielding non-errors. This can be detected. More insidiously, propagating dependencies where they are unneeded can keep alive unneeded parts of error slices much longer during minimization, resulting in severe slowdowns. Because correct results happen eventually, detecting such bugs is harder so this requires great care. For example, an earlier version of our solver copied dependencies from declarations in a structure to the structure's main result. The minimizer had to remove declarations one at a time. Debugging this was hard because only speed suffered. Furthermore, the system should yield error slices (before minimization) that are as close to minimal as can be reasonably achieved. If constraint solving yields a non-minimal error slice, then solving steps must have annotated a constraint with a location on which it does not uniquely depend.

DP6: **Bindings.** Sec. 6.6.4 already mentioned this principle. In the labeled external syntax, identifiers which can occur at bound positions must be labeled by a unique label that does not label a piece of code larger than the identifier itself. Moreover, for those labeled identifiers, the constraint generator should in general generate no more than a labeled accessor. (Note that to simplify the presentation of Skalpel we do otherwise for structure openings (see constraint generation rule (G19) in Fig. 19) but this is in general unsafe.) The risk of not following this principle is that during minimization, a bound occurrence of an identifier can be kept in a slice while its binding occurrence is discarded. This can then result in the identifier at a bound position being bound to a different binding occurrence than the one to which it is originally bound in the original piece of code. This can then lead to generating wrong identifier bindings and finding false errors.

DP7: **Environment variables.** Environment variables, when not generated as part of a shallow environment in an equality constraint (e.g., as the direct left or right-hand-side of an equality constraint), should always be labeled. An unlabeled environment variable is a constraint that can never be filtered out and has to always be satisfied, independently from any program location. Because an environment variable shadows its context (e.g., in (ev;e), the environment variable ev shadows e), if such an environment variable is unlabeled and is not constrained to be equal to anything, it always shadows its context whatever filtering is applied to it. This behaviour is undesirable because the shadowing of an environment should in general be dependent on a program location (see, e.g., constraint generation rule (G19) in Fig. 19 for `open` declarations).

In our constraint generator, most of the environment variables not generated as part of a shallow environment in an equality constraint cannot shadow their environments. It is the case for rules (G4), (G17) and (G18). Note that in these rules, each generated environment variable has to be labeled to carry the dependency on the program point responsible for its generation. Each of these rules generates an environment variable that is constrained by an unlabeled equality constraint on the environment variable itself (these unlabeled equality constraints cannot be filtered out). If these equality constraints were labeled, but the environment variables were not, the equality constraints could be filtered out and the environment variables could then be unconstrained and therefore shadow their contexts. Given a piece of code containing a recursive value declaration dec, for rule (G17) for example this would mean that filtering out the constraints associated with dec would allow this declaration to shadow its entire context in the analyzed piece of code, which is undesirable. For example, when slicing out the recursive value declaration in `val x = 1 val rec f = fn x => x val y = x x`, we do not want it do shadow `val x = 1`, i.e., we do not want the environment generated for the declaration `val rec f = fn x => x` to shadow the environment generated for `val x = 1` when the label associated with `val rec f = fn x => x` is sliced out in the environment generated for the entire piece of code. Rule (G19) stands out by

generating environment variables that are constrained by labeled accessors. If this rule was generating $((\uparrow strid \stackrel{l}{=} ev);ev)$ instead $((\uparrow strid \stackrel{l}{=} ev);ev^l)$, where the environment variable is unlabeled, ev would then be totally unconstrained when filtering out the accessor. This would prevent one to slice out `open` declarations. Worse, this could lead to finding typable type error slices. Let us illustrate this last point with the following example:

```
structure S = struct end
val x = 1
open S
val y = x 1
```

Because the structure `S` is empty, `open S` does not do anything, and especially `x` is not rebound. Let e be the environment generated by our constraint generator for this sequence of declarations. Our enumeration algorithm finds the following slice:

```
⟨..val x = 1
 ..x ⟨..⟩..⟩
```

Assuming that unlabeled environment variables are generated for `open` declarations instead of labeled environment variables as we currently do, filtering out the constraints in e w.r.t. this slice would lead to an environment e' where the unlabeled environment variable generated for `open S` shadows the environment generated for `x`'s declarations. This environment e' would then be solvable.

8. Related Work

8.1. Related constraint systems

Our constraint system has evolved over the years. An earlier version had constraints similar to Pottier and Rémy's let-constraints [48, 47], which we discuss here first.[8]

Pottier and Rémy. Pottier and Rémy define a constraint system [48] that reduces "type inference problems for HM(X) to constraint solving problems". Their HM(X) is similar to Odersky, Sulzmann, and Wehr's HM(X) constraint system [43]. This system is a "general framework for Hindley/Milner style type systems with constraints". Sulzmann, Odersky and Wehr say about their system

[8]Technically, the let-constraints of Pottier and Rémy are based on their more primitive def-constraints.

that "particular type systems can be obtained by instantiating the parameter X to a specific constraint system" and that "the Hindley/Milner system itself is obtained by instantiating X to the trivial constraint system". Variants of this system have been proposed over the years, for example in [55, 56]. Similarly, in order to deal with Haskell's rich type system, Vytiniotis et al. developed the parameterized OutsideIn(X) constraint-based type inference algorithm [59]. Using these let-constraints Pottier and Rémy "achieve the desired separation between constraint generation, on the one hand, and constraint solving and simplification, on the other hand, without compromising efficiency" [48]. Pottier defines a very similar system in [47]. In our discussion, we will collectively refer to these two systems as PR (Pottier/Rémy) and ignore their technical differences, although our presentation will follow more closely the presentation of Pottier and Rémy [48].

PR has let-constraints, subtyping constraints, type scheme instantiation constraints, conjunction constraints, and the satisfied `true` constraint. A PR let-constraint looks like: **let** $id{:}\dot{\sigma}$ **in** C, where $\dot{\sigma}$ ranges over type schemes, and C ranges over constraints. In PR, type schemes are of the form $\forall\overline{X}[C].T$ where $\overline{X}$ is a type variable set, C is a constraint, and T is a type. We borrow for our discussion two of Pottier and Rémy's abbreviations: (1) $\forall\overline{X}.T$ stands for the type scheme $\forall\overline{X}[\mathtt{true}].T$, and (2) **let** $id{:}T$ **in** C stands for **let** $id{:}\forall\varnothing.T$ **in** C.

The idea of let-constraints is that a constraint of the form

$$\textbf{let } id{:}\forall\overline{X}[C].T \textbf{ in } (id = T_1 \wedge id = T_2)$$

is (roughly) equivalent to a constraint of this form:

$$(\exists\overline{X}.(C \wedge T = T_1)) \wedge (\exists\overline{X}.(C \wedge T = T_2)) \wedge (\exists\overline{X}.C)$$

The key point is that one can get the effect of making the appropriate number of copies of C and T while keeping the size of the constraint proportional to the program size by eagerly simplifying C and calculating T as much as possible before making any copies.

Informally, a let-constraint of the form **let** $id{:}\forall\overline{X}[C_1].T$ **in** C_2 generated for an SML recursive `let`-binding would be represented in Skalpel by (using a combination of rules (G2) and (G17) in Fig. 18 and 19):

$$[\mathtt{poly}((\downarrow id{=}\tau);e_1);e_2]$$

where C_i is represented by e_i and T is represented by τ. (Let-constraints generated for other SML forms would not necessarily be modeled this way.) There is no explicit representation of $\overline{X}$ in Skalpel; instead the correct set of type variables that can be quantified over is calculated by toPoly which generates type schemes when it handles environments of the form $\mathtt{poly}(e)$ (see Fig. 17).

We now analyze the different parts of a let-constraint: **let** $id{:}\forall\overline{X}[C_1].T$ **in** C_2 (1) assigns static semantics to the identifier id (thanks to $id{:}\dot{\sigma}$), (2) quantifies

the static semantics associated with *id* over a set of variables (generates a polymorphic type), (3) makes the access to *id*'s semantics local to C_2, and (4) defines an order in which the constraints have to be solved (C_1 before C_2). Such a constraint can then be seen as the combination of (at least) four primitive constraints. The first one is a binder in Skalpel, the second one is a `poly` environment, the third one is an environment of the form $[e]$ (defined in Sec. 6.2.3), and the fourth one is an environment of the form $e_1;e_2$.

Let us now compare PR and Skalpel through the following example (a recursive value SML declaration):

$$\texttt{let val rec f = fn z => } \mathit{exp}_1 \texttt{ in } \mathit{exp}_2$$

where exp_1 and exp_2 are two sub-expressions. PR generates:

$$\textbf{let f}{:}\forall XY[\textbf{let f}{:}X \to Y \textbf{ in let z}{:}X \textbf{ in } C_1].X \to Y \textbf{ in } C_2$$

where X and Y are internal type variables, XY is PR's notation for the set $\{X, Y\}$, C_i (for $i \in \{1, 2\}$) is the constraint generated for exp_i, and Y is the result type of exp_1. Due to the way let-constraints declare a local environment, PR needs two binders for `f`: the outer one polymorphically binds `f`'s occurrences in exp_2, and the inner one monomorphically binds `f`'s occurrences in exp_1.

Skalpel generates[9] the following constraint (technically, an environment):

$$[\texttt{poly}(\downarrow\texttt{f}{=}\alpha_1 \to \alpha_2;[(\downarrow\texttt{z}{=}\alpha_1);e_1]);e_2]$$

In contrast to how PR handles this example, Skalpel only needs one binder for `f`. Two Skalpel features interact to allow this. First, in a composition environment ($e_1;e_2$), the bindings from e_1 are available in e_2, but also form part of the result (except when shadowed by bindings in e_2). Second, we use environments of the form $\texttt{poly}(e)$ to change the status of binders. In our example, `f`'s binder is monomorphic within the scope of `poly` (in e_1) and polymorphic outside (in e_2).

Hage and Heeren [34, 32, 27, 29]. Their approach is as follows: given a piece of code, first a constraint tree is generated, then this constraint tree is converted into a list (many conversions are possible resulting in different lists), and finally constraints are solved. Because different tree-to-list conversions are allowed, their system can emulate algorithms such as W [18], M [36] or UAE [64].

In their system, a constraint tree can among other things (we only present some of their constructs) be a strict node $T_1 \ll T_2$ where T_1 and T_2 are

[9] We have omitted labels and simplified a bit. Still omitting labels Skalpel actually generates: $[(ev_2{=}\texttt{poly}(\downarrow\texttt{f}{=}\alpha_1;[(ev_1{=}(\updownarrow\texttt{z}{=}\alpha_2));ev_1;e_1;c_1];c_2));ev_2;e_2;c_3]$ where $c_1 = (\alpha_3{=}\alpha_2{\to}\alpha_4)$, $c_2 = (\alpha_1 = \alpha_3)$, $c_3 = (\alpha_5 = \alpha_6)$, $\langle\alpha_4, e_1\rangle$ is generated for exp_1, $\langle\alpha_6, e_2\rangle$ is generated for exp_2, and α_5 is the type of the entire let-expression. See [49, 14.1] for the meaning of $\updownarrow\texttt{z}{=}\alpha_2$.

constraint trees. A constraint can be attached to a tree using, for example, $c \diamond T$, which makes the constraint c "part of the constraint associated with the root of T" [29]. A tree can also pack together trees as follows: $\bullet[T_1, \ldots, T_n]\bullet$ A constraint itself can among other things be: an equality constraint $\tau_1 \equiv \tau_2$, a generalisation constraint $\sigma{:=}\mathtt{GEN}(M, \tau)$, where M is a (monomorphic) type variable set and σ is a scheme variable, or an instantiation constraint $\tau \preceq \sigma$. Hage and Heeren [29] say about their generalisation and instantiation constraints: "The reason we have constraints to explicitly represent generalization and instantiation is the same as why, e.g., Pottier and Rémy do [48]: otherwise we would be forced to (make a fresh) duplicate of the set of constraints every single time we use a polymorphically defined identifier". The same goes for our system.

Trees are sophisticated constraints that provide extra structure on constraint sets. In Skalpel a single equality constraint can be an environment. Similarly, in their system a single constraint can be a tree. A strict node of the form $T_1 \ll T_2$ can be seen as a restricted version of a composition environments of the form $e_1;e_2$. Environments of the form $e_1;e_2$ also enforce that e_1 has to be solved before e_2. A major difference is that in Skalpel, in an environment $e_1;e_2$, not only the environment e_1 has to be solved before e_2 but also e_2 has access to e_1's binders. Also a major difference between trees and constraint/environments is that in their system trees do not act as environments, they do not associate static semantics with identifiers. We do not allow non-strict nodes (such as their nodes of the form $\bullet\ T_1, \ldots, T_n\ \bullet$) because Skalpel does not reorder constraints. Their constraint rearrangement mechanism can be seen as a restriction of our enumeration algorithm.

Enforcing to solve some constraints before others introduces a bias. One of our early design principle was to build an unbiased type error slicer[10]. Skalpel is unbiased thanks to our enumeration algorithm, which, given an environment e, runs our constraint solver on the different environments that can be obtain from e using our filtering function. Hage and Heeren opted for ordering strategies instead.

The main difference between their transformation of a type inference problem into a constraint solving problem and ours (and so the main difference between our constraint systems) is that bindings are solved at constraint solving in Skalpel (using binders and accessors) while they are solved at constraint generation in Hage and Heeren's system. We moved from a binding resolution at initial constraint generation to a binding resolution at constraint solving in order to handle SML features such as `open`. Thanks to our binders and accessors, we can generate a "faithful" (used loosely here) representation of an SML program that uses intricate features such as `open`.

[10][49, 16.2] explains how Skalpel meets Yang et al.'s list of criteria for good type error reports [66]

Digression on the generation of "faithful" program representations using constraints. In addition to the motivation of generating "faithful" program representations in our constraint language, we believe that binders and accessors are necessary to distinctly separate the constraint generation and constraint solving phases of a constraint based type inference algorithm for SML. To illustrate this point let us consider the following typable SML program:

```
structure S = struct val c = fn () => () end
structure T = S
structure U = T
open U
val d = c ()
```

Without binders and accessors, one needs to use type environments at constraint generation to access identifiers' static semantics when analyzing identifiers at bound positions. To generate a proper environment for `open U` so that it can be used when dealing with `val d = c ()` at constraint generation, one needs to resolve the chain of structure equalities. This means that solving structures' static semantics at constraint generation becomes necessary which goes against a clear separation between constraint generation and constraint solving.

Müller [41]. Müller designed the ρ_{deep} calculus to "implement the classical Damas-Milner polymorphic type inference algorithm". This calculus allows one to generate constraints of linear size. It does so by generating identifier binders with which are associated static semantics. The semantics attached to an identifier binder can then be simplified before being "used", i.e., before instantiating the polymorphic type. Müller considers the λ-calculus extended with polymorphic let-expressions. His constraint language is as follows, where ϕ and ψ are called constraints, and E and F are called (constraint) expressions:

$$\begin{array}{lcl} \phi, \psi & ::= & \top \mid \bot \mid \exists\alpha\ \phi \mid \phi \wedge \psi \mid \alpha = \beta \mid \alpha = \beta \to \gamma \\ E, F & ::= & \phi \mid E \wedge F \mid \exists\alpha\ E \mid x{:}\alpha/E \mid [\![M]\!]\alpha \end{array}$$

where M is a λ-expression and α, β and γ are type variables. $\top$ is the satisfied (or true) constraint, and $\bot$ is the unsatisfied (or false) constraint. Constraints and expressions of the forms $\exists\alpha\ \phi$ and $\exists\alpha\ E$ introduce fresh variables. Constraints and expressions of the form $\phi \wedge \psi$ and $E \wedge F$ are conjunctions. The two last forms of constraints are shallow equality constraints. The most interesting forms of ρ_{deep} are: $x{:}\alpha/E$ and $[\![M]\!]\alpha$.

An expression $x{:}\alpha/E$ is called an *abstraction*. They are similar to Pottier and Rémy's let-constraints. Such an abstraction associates the constrained static semantics α, constrained by E, with the identifier x. This is called an abstraction because $x{:}\alpha/E$ abstracts the type variable α. The polymorphism of such forms comes from the existential expressions. For example, one can generate the following abstraction (binder) for the polymorphic identity function `id`

(where some expressions are omitted for clarity): $\mathtt{id}{:}\gamma/\exists\beta\ \gamma = \beta \to \beta$. A bound occurrence of id with which is associated the static semantics α will result in the application of the above abstraction α: $\exists\beta\ \alpha = \beta \to \beta$. A particularity of ρ_{deep} is that computation can occur inside abstractions allowing one "to simplify the abstractions encoding polymorphic types before application" [41]. We believe that an abstraction of the form $x{:}\alpha/E$ would be modeled in Skalpel by an environment of the form $\mathtt{poly}(e;\downarrow x{=}\alpha)$, where E is represented by e.

An expression of the form $[\![M]\!]\alpha$ is called a *proof obligation* and it "represent the constraint $\alpha = \tau$ for the principal type τ of M", where τ is an internal type in ρ_{deep}. A constraint expression of the form $[\![M]\!]\alpha$ is used to analyse (infer a type for) the λ-expression M.

Müller's constraint based type inference algorithm does not distinguish between constraint generation and constraint solving and no specific constraint solving strategy is presented (constraint generation and solving interleave).

Gustavsson and Svenningsson [24]. In their system, solutions can be found in cubic time. Their constraint syntax is based on: the satisfied constraint $\top$, inequality constraints on variables of the form $a \leq b$, conjunctions of constraints of the form $M \wedge N$, and existential constraints of the form $\exists a.M$. In addition, they also have abstractions, applications, and let-constraints.

Abstractions are inspired by let-expressions and are of the form: $f\ \vec{a} = M$, where f is a constraint abstraction variable (the name of an abstraction), $\vec{a}$ is a set of variables, and M is a constraint. Abstractions are used in let-constraints of the form: $\mathtt{let}\ \{\vec{F}\}\ \mathtt{in}\ M$, where $\vec{F}$ is a set of abstractions and M is a constraint. Abstractions in a let-constraint are mutually recursive. Therefore, in a let-constraint $\mathtt{let}\ \{\vec{F}\}\ \mathtt{in}\ M'$, if $f\ \vec{a} = M$ is a constraint abstraction in $\vec{F}$, then the uses of f in $\vec{F}$ and M' all refer to this occurrence of f.

We believe that a let-constraint of the form $\mathtt{let}\ \{f_1\ \vec{a}_1 = M_1, \ldots, f_n\ \vec{a}_n = M_n\}\ \mathtt{in}\ M$ would be represented in Skalpel by an environment of the form $[\mathtt{poly}(\downarrow f_1{=}\alpha_1;\cdots;\downarrow f_n{=}\alpha_n;e_1;\cdots;e_n);e]$, where M_i would be represented by e_i for each $i \in \{1, \ldots, n\}$, where M would be represented by e, and where $\vec{a}_i$, for each $i \in \{1, \ldots, n\}$, would be computed when dealing at constraint solving with the poly constraint. Abstractions are applied thanks to application constraints of the form $f\ \vec{a}$. An application of the form $f\ \vec{a}$ would be represented in Skalpel by an accessor of the form $\uparrow f{=}\alpha$.

Gustavsson and Svenningsson define a constraint solving algorithm and prove it to be of cubic complexity. Such a result is obtained by enforcing that abstractions are simplified before being applied.

Tip and Dinesh [57]. They report type error slices for a Pascal-like language called CLaX, which is an explicitly typed language (where explicit types are enforced, e.g., on function parameters). CLaX's type checker is a rewriting system, which rewrites a piece of code either into a type if the piece of code is typable, or into a list of error messages if the piece of code is untypable. To compute slices they use "dependence tracking" [21, 22]. Tip and Dinesh explain that "Dependence tracking is a method for computing term slices that relies on an analysis of rewriting rules to determine how the application of rewriting rules causes *creation* of new function symbols, and the *residuation* (i.e., copying, moving around, or erasing) of previously existing subterms" [57]. Developments (w.r.t. a sequence of rewriting steps on a piece of code) are trimmed to retain only the necessary symbols of a piece of code, i.e., the ones responsible for an error to occur. Tip and Dinesh also applied their techniques to Mini-ML, "a simple typed λ-calculus with constants, products, conditionals, and recursive function definitions" [17]. Because they do not use any minimization algorithm, Tip and Dinesh faced some minimality issues when applying their method to Mini-ML: "in some cases slices are computed that seem larger than necessary" [57].

Neubauer and Thiemann [42]. They use flow analysis to compute type dependencies for a small ML-like language and report type errors. Their system uses discriminative sum types and can analyze any term. In a first "collecting" phase, they label terms and infer type information. This analysis generates a set of program point sets. These program points are directly stored in the discriminative sum types. A conflicting type ("multivocal") is then paired with the locations responsible for its generation. A second "reporting" phase consists in generating error reports from the conflicts generated during the first phase. Slices are built from which highlighting are produced. An interesting detail is that a type derivation can be viewed as the description of all type errors in an untypable piece of code, from which another step computes error reports.

Chameleon. Similar to ours is work by Stuckey, Sulzmann and Wazny [54, 61] (based on earlier work without slices [52, 53]). They do type inference, type checking and report type errors for the Chameleon language (a modified Haskell subset). Chameleon includes algebraic data types, type-class overloading, and functional dependencies. They code the typing problem into a constraint problem and attach labels to constraints to track program locations and highlight parts of untypable pieces of code. First they compute a minimal unsatisfiable set of generated constraints from which they select one of the type error locations, and finally they provide a highlighting and an error message depending on the selected location. Their slice highlighting strategy is different from ours. They focus on explaining conflicts in the inferred types at one program point inside the error location set. To the best of our knowledge, they do not intend to exactly (no more and no less) highlight code fragments as to provide complete

explanations of type errors. For example, they do not highlight applications because "they have no explicit tokens in the source code". The authors further explain: "We leave it to the user to understand when we highlight a function position we may also refer to its application". Skalpel highlights all the program locations responsible for an error even if these are white spaces. Also, they do not appear to highlight the parts of datatype declarations relevant to type errors.

When running on a translation of the code presented in Sec. 3.2 into Haskell, ChameleonGecko outputs the error report partially displayed below (the rest of the output seems to be internal information from their solver).

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

This highlighting identifies the same location as SML/NJ and would not help solve the error.

Gast [23]. Gast generates "detailed explanations of ML type errors in terms of data flows" in three steps: (1) generation of subtyping constraints annotated by reasons for their generation; (2) gathering of reasons during constraint solving; (3) transformation of the gathered reasons into explanations by data flows. He provides a visually convenient display of the data flows with arrows in XEmacs. Gast's method (which seems to be designed only for a small portion of OCaml) can be considered as a slicing method with data flow explanations.

Braßel [13]. Braßel presents a generic approach (implemented for the Curry language) for type error reporting that consists of two procedures. The first one tries to replace portions of code by dummy terms that can be assigned any type. If an untypable piece of code becomes typable when one of its subtrees has been replaced by a dummy term then the process goes on to apply the same strategy inside the subtree. The second procedure consists in using of a heuristic to guide the search of type errors. The heuristic is based on two principles: it will always "prefer an inner correction point to an outer one" and will always "prefer the point which is located in a function farther away in the call graph from the function which was reported by the type checker as the error location". Braßel's method does not seem to compute proper slices but instead singles out different locations that might be the cause of a type error inside a piece of code.

Weijers, Hage, and Holdermans [62, 63]. In these papers, the authors present an error diagnosis tool for security label errors that combines a type error slicer as in by Haack and Wells' approach, with heuristics as used in Helium (discussed below in Sec. 8.3). The language they consider, called sFun++, is a let-polymorphic applied λ-calculus extended with *declassify* and *protect* security constructs. They first generate minimal error slices and then prune even

further these slices using various kinds of heuristics that are meant to pinpoint likely sources of errors. Their current system purposely report a single error message and they mention that reporting a collection of error diagnoses "is only a matter of engineering". Also, their system only reports security errors but could presumably be generalized to support other kinds of errors.

8.3. Significant non-slicing type explanation methods

Let us now present some non-slicing type explanation methods. We believe that all of these could be combined with Skalpel to provide better error reports. This is left for future work.

Helium [33, 31, 34, 28]. Heeren et al. designed a method to provide error messages for the Helium language (a subset of Haskell), relying on a constraint-based type inference algorithm. First, a constraint graph is generated from a piece of code. Given an ill-typed piece of code, a conflicting path called an inconsistency is extracted from the constraint graph. A conflicting path is a structured unsolvable set of type constraints. A trust value is associated with each type constraint and depending on these values and the other heuristics, some constraints are discarded until the inconsistency is removed. They also provide "program correcting heuristics" used to search for a typable piece of code from an untypable one. Such a heuristic is, e.g., the permutation of parameters, which is a common mistake in programming. Their approach is, among other things, meant to help students learn functional programming. Using pieces of code written by students and their expertise of the language they refined their heuristics. They also designed a system of "directives", which are commands specified by the programmer to constrain the set of types derivable from a type class. This approach differs from ours by privileging locations over others by the use of some heuristics. They do not compute minimal slices and highlightings.

We present below the most interesting part of the error report obtained using Helium on a translation of the code presented in Sec. 3.2 into Haskell. It comes with some warnings (not displayed here) about identifiers' bindings such as the binding of `y` in `trans` (some of these warnings explain, e.g., that `y`'s declaration at the end of the code does not bind any of the `y`'s in `trans`'s definition).

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

Compilation failed with 1 error
```

It is reported that `x1` and `trans` do not have the expected types. The application at the end of the code is blamed when our programming error is at the

very beginning of the code.

They have also tackled the task to report type errors for Java [11, 12]. Error reports provided by standard compilers can be of little help, especially in the presence of generics. El Boustani and Hage try to do better by keeping track of more information during type checking, allowing a more "global view" of type errors, and therefore leading to more informative error reports. The main difference between type error reporting for SML and Java is that in Java "types are instantiated based on local information only and not through a long and complicated sequence of unifications" [11].

More recently, Chen, Erwig, and Smeltzer studied the combination of Helium with another of their tool called Lazy Typing (that aims at pinpointing type errors and that suggests fixes) [14], making use of the fact that different error reporting tools have their own strengths and weaknesses.

SEMINAL [37, 38]. Lerner, Flower, Grossman and Chambers designed algorithms that search for type errors by replacing code fragments by an expression that type-checks in any context, such as raising a dummy exception: `raise Foo`. The same method is used by Schilling to compute type error slices in the context of Haskell [50]. They then present type error messages by constructing well-typed programs from ill-typed ones using techniques such as switching parameters (like Heeren et al. [28]). Automatically generated modifications of ill-typed pieces of code are checked for typability. They target Caml, and also developed a prototype for C++. The new typable generated code is presented as possible code that the programmer might have intended.

Chitil. Chitil developed several tools to understand and debug type errors. For example, Typeview [16] is an interactive tool that can show the type of any identifier in a Haskell program. When a program is not typable, "it is possible to query the types of all expressions that were inferred so far". In [15] Chitil describes a type explanation and debugging tool for Haskell. This tool allows programmers to explore how unexpected types are inferred in order for them to fix their errors. This is done by interactive navigation through a type explanation graph. (More recently, Tsushima and Asai developed a similar tool for OCaml [58], and Plociniczak, Miller, and Odersky for Scala [46].) In [51], Silva and Chitil explore the combination of two debugging techniques: algorithmic debugging and program slicing. The way algorithmic debugging works is that, given a program, it automatically produces series of questions regarding the execution behavior of the program and processes the programmer's answer to try and locate errors. However, as mentioned by Silva and Chitil, questions can be numerous and arbitrarily long and complex. Their solution is to combine algorithmic debugging with program slicing in order to reduce the number of questions needed to locate errors.

Zhang and Myers [67]. As in [62, 63], Zhang and Myers' also aim at diagnosing security errors. Their error diagnosis tool is motivated by two languages: OCaml and Jif [35] (a "security-typed programming language that extends Java with support for information flow control and access control, enforced at both compile time and run time"). They are therefore aiming at a system that is general enough to deal with, not only type errors, but various other kinds of static errors such as failures of security label checking. Their method, also constraint based, relies on solving constraint graphs. They use a Bayesian posterior distribution to try and pinpoint programming errors, and suggests missing hypotheses. Their constraint language does not feature let-constraints. Instead they use Haack and Wells' approach [25, 26] of duplicating constraint sets when dealing with let-expressions, which, as mentioned above, could lead to an exponential growth of the number of generated constraints. The authors acknowledge this issue and report that they "find performance is still reasonable with this approach" [67]. They then showed that their method scales up to more expressive type systems such as Haskell's type system [68]. There, let-expressions are annotated by user defined constrained type schemes, which in practice results in a smaller number of generated constraints than when using Haack and Wells' approach.

Pavlinovic, King, and Wies [44]. Their approach is also based on constraint solving. Their tool try to pinpoint type errors (for a subset of OCaml) by computing "minimum" sets of untypable portions of programs, where minimum is a compiler dependent property. As the authors mention, a "compiler may only be interested in those error causes that require the fewest changes to fix the program". Their approach consists in generating weighted constraints, and reducing the search for minimum error sources to an optimization problem that they formulate in terms of weighted maximum satisfiabiity modulo theories. Finally, they use SMT solvers to solve these problems. Their constraint generator is similar to Haack and Wells'. It can lead to an exponential growth of the number of generated constraints.

9. Conclusion

In this paper we presented Skalpel, which takes a program written in core SML (see Sec. 9), and returns exactly the erroneous parts of the program. Skalpel automatically achieves this by first labeling all parts of the program generating constraints annotated with these labels, solving these constraints and if errors are found, minimizing the errors to produce a small and elegant program slice that is passed to the user showing them exactly those parts of the program that contribute to the error.

Skalpel is based on a novel constraint syntax, generator and solver which is terminating and avoids a combinatorial explosion in the number of constraints. We retain a compositional generation of constraints but solve constraints in a

strict left-to-right order. Furthermore, in order to scale constraints while also handling module system features, we introduced a novel representation of hybrid constraint/environments. This allows for environments that avoid duplication at constraint generation and during constraint solving.

In the future, we wish to perform some analysis as a result of this work to allow us to determine interesting information such as how much faster software developers can develop their SML programs using the Skalpel tool, or on which kind of errors the Skalpel tool is particularly effective.

To our knowledge, no work exists that returns minimal type error slices for the entirety of a powerful programming language such as SML (we only present the core of our approach in this paper, more details about Skalpel can be found in [49, 45] and on the Skalpel webpage [2]). We believe that with further research, techniques such as the ones presented in this paper can be applied to larger languages with more sophisticated type systems and wider user bases such as Haskell, and combined with other techniques such as the ones presented in the related work section.

References

[1] MLton compiler. `www.mlton.org/`. Last accessed 21st May 2015.

[2] Skalpel project homepage. `http://www.macs.hw.ac.uk/ultra/skalpel`. Last accessed 13th July 2012.

[3] SML/NJ compiler. `www.smlnj.org/`. Last accessed 17th July 2012.

[4] *16th Int'l Workshop, IFL 2004*, volume 3474 of *LNCS*. Springer, 2005.

[5] Skalpel project, 2010. `http://www.macs.hw.ac.uk/ultra/skalpel`.

[6] PolyML compiler. `www.polyml.org/`, web. Last accessed 20th January 2014.

[7] Andrew W. Appel. A critique of Standard ML. *J. Funct. Program.*, 3(4):391–429, 1993.

[8] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4):17–30, 1993.

[9] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, November 1997.

[10] Matthias Blume. Dependency analysis for Standard ML. *ACM Trans. Program. Lang. Syst.*, 21(4):790–812, 1999.

[11] Nabil El Boustani and Jurriaan Hage. Improving type error messages for Generic Java. In *PEPM*, pages 131–140. ACM, 2009.

[12] Nabil El Boustani and Jurriaan Hage. Corrective hints for type incorrect Generic Java programs. In *PEPM*, pages 5–14. ACM, 2010.

[13] Bernd Braßel. TypeHope - there is hope for your type errors. In *16th Int'l Workshop, IFL 2004* [4], pages 185–198.

[14] Sheng Chen, Martin Erwig, and Karl Smeltzer. Let's hear both sides: On combining type-error reporting tools. In *VL/HCC 2014*, pages 145–152. IEEE, 2014.

[15] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 193–204, New York, NY, USA, 2001. ACM.

[16] Olaf Chitil, Frank Huch, and Axel Simon. Typeview: A tool for understanding type errors. In *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 63–69. Aachner Informatik-Berichte, 2000.

[17] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: mini-ML. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 13–27, New York, NY, USA, 1986. ACM.

[18] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL82*, pages 207–212, New York, NY, USA, 1982. ACM.

[19] Dominic Duggan. Correct type explanation. In *In ACM SIGPLAN Workshop on ML*, pages 49–58, 1998.

[20] Dominic Duggan and Frederick Bent. Explaining type inference. *Sci. Comput. Program.*, 27(1):37–83, 1996.

[21] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 415–431, London, UK, 1994. Springer-Verlag.

[22] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11-12):609–636, 1998.

[23] Holger Gast. Explaining ML type errors by data flows. In *16th Int'l Workshop, IFL 2004* [4], pages 72–89.

[24] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *PADO*, volume 2053 of *LNCS*, pages 63–83. Springer, 2001.

[25] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *ESOP*, volume 2618 of *LNCS*, pages 284–301. Springer, 2003.

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

[27] Jurriaan Hage and Bastiaan Heeren. Ordering type constraints: A structured approach. Technical report, Institute of information and computing sciences, utrecht university, 2005.

[28] Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In *18th Int'l Symp., IFL 2006*, volume 4449 of *LNCS*, pages 199–216. Springer, 2007.

[29] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electron. Notes Theor. Comput. Sci.*, 236:163–183, 2009.

[30] Robert Harper. Programming in Standard ML, 2009. Working draft of August 20, 2009.

[31] Bastiaan Heeren and Jurriaan Hage. Type class directives. In *7th Int'l Symp., PADL 2005*, volume 3350 of *LNCS*, pages 253–267. Springer, 2005.

[32] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59–80, Cork, September 2003.

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

[34] Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[35] Jif. `http://www.cs.cornell.edu/jif/`.

[36] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, jul 1998.

[37] Benjamin Lerner, Dan Grossman, and Craig Chambers. SEMINAL: Searching for ML type-error messages. In *ACM Workshop on ML*, pages 63–73. ACM, 2006.

[38] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *ACM SIGPLAN 2007 Conf. PLDI*. ACM, 2007.

[39] Bruce J. McAdam. On the unification of substitutions in type inference. In *10th Int'l Workshop, IFL'98*, volume 1595 of *LNCS*, pages 137–152. Springer, 1999.

[40] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (Revised).* The MIT Press, 55 Hayward Street, Cambridge, MA 02142-1493, USA, 1998.

[41] Martin Müller. A constraint-based recast of ML-polymorphism (extended abstract). Technical report, 8th International Workshop on unification, 1994.

[42] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *8th ACM SIGPLAN Int'l Conf., ICFP 2003*, pages 15–26. ACM, 2003.

[43] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[44] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *OOPSLA 2014*, pages 525–542. ACM, 2014.

[45] John Pirie. *New Developments to Skalpel: A Type Error Slicing Method for Explaining Errors in Type and Effect Systems.* PhD thesis, Heriot-Watt University, 2014. Available at `http://www.macs.hw.ac.uk/~jp95/jpirie-thesis.pdf`.

[46] H. Plociniczak, H. Miller, and M. Odersky. Improving human-compiler interaction through customizable type feedback. Under submission (`http://infoscience.epfl.ch/record/197948`), 2015.

[47] François Pottier. A modern eye on ML type inference: old techniques and recent developments. Lecture notes for the APPSEM Summer School, September 2005.

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

[49] Vincent Rahli. *Investigations in intersection types: Confluence and semantics of expansion in the lamba-calculus, and a type error slicing method.* PhD thesis, Heriot-Watt University, 2010. Last accessed Monday 16th July 2012: `http://www.cs.cornell.edu/~rahli/articles/thesis.pdf`.

[50] Thomas Schilling. Constraint-free type error slicing. In *TFP 2011*, volume 7193 of *LNCS*, pages 1–16. Springer, 2011.

[51] Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *PPDP 2006*, pages 157–166. ACM, 2006.

[52] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in Haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, New York, NY, USA, 2003. ACM.

[53] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 80–91, New York, NY, USA, 2004. ACM.

[54] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Type processing by constraint reasoning. In *4th Asian Symp., APLAS 2006*, volume 4279 of *LNCS*, pages 1–25. Springer, 2006.

[55] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical report, 1999.

[56] Martin Franz Sulzmann. *A general framework for Hindley/Milner type systems with constraints.* PhD thesis, Yale University, New Haven, CT, USA, 2000. Director - Paul Hudak.

[57] Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55, 2001.

[58] Kanae Tsushima and Kenichi Asai. An embedded type debugger. In *IFL 2012*, volume 8241 of *LNCS*, pages 190–206. Springer, 2012.

[59] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

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

[61] Jeremy Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions.* PhD thesis, University of Melbourne, Australia, 2006.

[62] Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. Security type error diagnosis for higher-order, polymorphic languages. In *PEPM 2013*, pages 3–12. ACM, 2013.

[63] Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. Security type error diagnosis for higher-order, polymorphic languages. *Sci. Comput. Program.*, 95:200–218, 2014.

[64] Jun Yang. Explaining type errors by finding the source of a type conflict. In *SFP'99: Selected papers from the 1st Scottish Functional Programming Workshop*, pages 59–67, Exeter, UK, UK, 2000. Intellect Books.

[65] Jun Yang, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

[66] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. Improved type error reporting. In *12th Int'l Workshop, IFL 2000*, volume 2011 of *LNCS*, pages 71–86. Springer, 2001.

[67] Danfeng Zhang and Andrew C. Myers. Toward general diagnosis of static errors. In *POPL '14*, pages 569–582. ACM, 2014.

[68] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing type errors with class. PLDI'2015, available at `http://www.cs.cornell.edu/~zhangdf/pub/pldi15.pdf`, 2015.