Skalpel: A Type Error Slicer for Standard ML

Vincent Rahli

Cornell University, Ithaca

Joe Wells and John Pirie and Fairouz Kamareddine

Heriot-Watt University, Edinburgh

Abstract

Compilers for languages with type inference algorithms produce confusing type error messages and give a single error location which is often far away from the real location of the type error. Attempts at solving this problem 1) fail to include the multiple program points which make up the type error, 2) often 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 consists of a sophisticated new constraint generator which is linear in size and a new constraint solver which is terminating.

Keywords: Automated type inference, Automated error diagnosis, Improved error reports.

1 Introduction & Related Work

Programming languages like 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 an early stage (at compile time). Unfortunately, these compilers 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 exactly the parts (slice) of an ill-typed program contributing to an error. The produced slice contains all and only the program parts related to the error.

The original type-checking algorithm for Standard ML is algorithm W [Damas and Milner(1982)], which blames a single abstract syntax tree node when unification fails. Variations on this algorithm such as M [O. Lee(1998)] and W' [Mcadam(1998)], have been developed to solve the left-to-right bias of the W algorithm. However, all these algorithms still blame a single node in the abstract syntax tree for an error which is made up of multiple error locations. In addition,

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

the errors reported by existing compilers are confusing, as they often give incorrect type information/diagnosis and report abstract syntax tree fragments which do not correspond to the user program.

Automatically finding type errors in programming languages is a difficult task. Successful attempts need to address constraint systems (systems which use a constraint based approach in order to locate errors, unlike compilers which use a substitution-based approach) but these have only been built for toy-like languages in [Müller(1994)] and [Hage and Heeren(2009)]. A more promising approach has been taken in [Zhang and Myers(2014)], but again the supported portion of the languages used to demonstrate the key ideas is small. Moreover, existing proposals to solve poor type error reporting (e.g., [Braßel(2004)], [Lerner and Grossman(2006)], and [Schilling(2012)]) simply repeat calls to the compiler and remove/add back in portions of the untypable program to narrow the point of error. The problem of finding type errors and of reporting possible solutions is very difficult and to solve it automatically is even more difficult. Every piece of syntax in the program must be automatically labelled, constraints need to be automatically generated and solved and finding solutions can lead to new constraints and a combinatorial constraints size explosion.

We have developed a new method and tool (Skalpel) which solves the above problems. Skalpel attaches program points (*labels*) to constraints that are generated, so that when unification fails, we can report the labels attributed to the constraints which were generated, giving a full description of the error. We annotate constraints with these labels to describe what set of program points a constraint is involved with. When Skalpel is asked to check a program for type errors, it runs its sophisticated constraint generator/solver (which is linear in size and terminating). If solving the constraints fails (i.e., if there is an error in the code), Skalpel must automatically decide which parts (slice) of the program was 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:

- Unlike other algorithms which use a substitution approach to solving, such as M [O. Lee(1998)] and W' [Mcadam(1998)], Skalpel will only show program fragments which originate from the user program.
- Skalpel will show **all** the program locations that contribute to the error.
- Skalpel is general enough to deal not only with one file containing source code with a single type error, but also type error slices that we pass to the user may involve more than one file of source code and highlighting is given in all affected files. Furthermore, if the source code fed to Skalpel contains multiple separate type errors, Skalpel produces all the culprit multiple program slices.
- The constraint generator is linear in the size of the program and the constraint solver is terminating (Lemmas 3.1 and 3.3).

- Skalpel is the first attempt at handling an entire programming language using a constraint approach, the core of which is given in this paper.

In Section 2 we discuss the basic notation used. In Section 3 we give 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 constraint generation is linear and that constraint solving terminates. We conclude in Section 4.

2 Mathematical notations

Let i, j, m, n, p, q range over the set $\mathbb{N}$ of natural numbers. If v ranges over a class C, 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 $\mathsf{dj}(s_1, \ldots, s_n)$ ("disjoint") hold iff for all $i, j \in \{1, \ldots, n\}$, if $i \neq j$ then $s_i \cap s_j = \emptyset$. Let $s_1 \uplus s_2$ be $s_1 \cup s_2$ if $\mathsf{dj}(s_1, s_2)$ and undefined otherwise. Let $\llparenthesis x, y \rrparenthesis$ be the pair of x and y. If *rel* is a binary relation (a pair set), let $(x \mathrel{\mathit{rel}} y)$ iff $\llparenthesis x, y \rrparenthesis \in \mathit{rel}$, let the inverse of *rel* be rel^{-1} defined as $\{\llparenthesis x, y \rrparenthesis \mid \llparenthesis y, x \rrparenthesis \in \mathit{rel}\}$, let $\mathsf{dom}(\mathit{rel}) = \{x \mid \exists y. \llparenthesis x, y \rrparenthesis \in \mathit{rel}\}$, let $\mathsf{ran}(\mathit{rel}) = \{y \mid \exists x. \llparenthesis x, y \rrparenthesis \in \mathit{rel}\}$, let $s \lhd \mathit{rel} = \{\llparenthesis x, y \rrparenthesis \in \mathit{rel} \mid x \in s\}$, and let $s \rhd \mathit{rel} = \{\llparenthesis x, y \rrparenthesis \in \mathit{rel} \mid x \not\in s\}$. Let f range over functions (a special case of binary relations), let $s \to s' = \{f \mid \mathsf{dom}(f) \subseteq s \wedge \mathsf{ran}(f) \subseteq s'\}$, and let $x \mapsto y$ be an alternative notation for $\llparenthesis x, y \rrparenthesis$ used when writing some functions. 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 $s_1, \ldots, s_n$ be $\{\langle x_1, \ldots, x_n \rangle \mid \forall i \in \{1, \ldots, n\}. x_i \in s_i\}$. Note that $s_1, \ldots, s_n \subseteq \mathsf{tuple}(s_1 \cup \cdots \cup s_n)$. For some reduction relation R we write R^* for its reflexive and transitive closure.

3 Technical Core of Skalpel

We refer to the system which is defined in this section as the *Skalpel core*, comprising of the constraint generator and solver which are defined in this section.

We begin by introducing the external labelled syntax given in Figure 1 which describes a subset of the SML language, chosen to present the core ideas.[1] Most syntactic forms have labels (l), which are generated to track blame for errors. We surround some terms such as function application with $\lceil\ \rceil$ in order to provide a visually convenient place for labels.

We will present a running example throughout this paper. The SML program

[1] We do not enforce all the syntactic restrictions of the SML syntax e.g. in `val rec` *pat* $\overset{l}{=}$ *exp*, the expression *exp* must be an `fn`-expression (which we do not enforce in this paper).

Fig. 1 External labelled syntax: The subset of SML that Skalpel handles

l	$\in$	Label	(labels)		$\in$	(Union of below sets)		
tv	$\in$	TyVar	(type variables)	vid	$\in$	VId	$::=$	$vvar \mid dcon$
tc	$\in$	TyCon	(type constructors)	ltc	$\in$	LabTyCon	$::=$	tc^{l}
$strid$	$\in$	StrId	(structure identifiers)	$ldcon$	$\in$	LabDatCon	$::=$	$dcon^{l}$
$vvar$	$\in$	ValVar	(value variables)	dn	$\in$	DatName	$::=$	$\lceil tv\ tc\rceil^{l}$
$dcon$	$\in$	DatCon	(datatype constructors)	$atpat$	$\in$	AtPat	$::=$	$vid^{l}_{\mathtt{p}}$

cb	$\in$	ConBind	$::=$ $dcon^{l}_{\mathtt{c}} \mid dcon\ \texttt{of}\ ^{l}\ ty$
$atexp$	$\in$	AtExp	$::=$ $vid^{l}_{\mathtt{e}} \mid \texttt{let}^{l}\ dec\ \texttt{in}\ exp\ \texttt{end}$
pat	$\in$	Pat	$::=$ $atpat \mid \lceil ldcon\ \ atpat\rceil^{lab}$
ty	$\in$	Ty	$::=$ $tv^{l} \mid ty_1 \overset{l}{\to} ty_2 \mid \lceil ty\ ltc\rceil^{l}$
$strdec$	$\in$	StrDec	$::=$ $dec \mid \texttt{structure}\ strid \overset{l}{=} strexp$
$strexp$	$\in$	StrExp	$::=$ $strid^{l} \mid \texttt{struct}^{l}\ strdec_1 \cdots strdec_n\ \texttt{end}$
dec	$\in$	Dec	$::=$ $\texttt{val rec}\ pat \overset{l}{=} exp\ \mid \texttt{open}^{l}\ strid \mid \texttt{datatype}\ dn \overset{l}{=} cb$
exp	$\in$	Exp	$::=$ $atexp \mid \texttt{fn}\ pat \overset{l}{\Rightarrow} exp \mid \lceil exp\ atexp\rceil^{l}$
id	$\in$	Id	$::=$ $vid \mid strid \mid tv \mid tc$
$term$	$\in$	Term	$::=$ $ltc \mid ldcon \mid ty \mid cb \mid dn \mid exp \mid pat \mid strdec \mid strexp$

we will use as an example is shown below. We present this here in order to show how syntax is annotated with labels.

$$\texttt{fn y}^{l_2}\ \overset{l}{\Rightarrow}\ \texttt{let}^{l_3}\ \texttt{val rec f}^{\mathtt{l}_8}\ =^{\mathtt{l}_7}\ \texttt{fn x}^{\mathtt{l}_9}\ \overset{\mathtt{l}_{10}}{\Rightarrow}\ \lceil\texttt{x}^{\mathtt{l}_{12}}\ \texttt{y}^{\mathtt{l}_{13}}\rceil^{\mathtt{l}_{11}}\ \texttt{in}\ \lceil\texttt{f}^{\mathtt{l}_4}\ \texttt{y}^{\mathtt{l}_5}\rceil^{\mathtt{l}_6}\ \texttt{end}$$

In Figure 1, 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, Section 14.1 of [Rahli(2010)] handles identifier statuses. Also, to simplify the presentation of the Skalpel core for this paper, datatypes have been restricted to one constructor and one type argument.

3.1 Constraint syntax

In this section we give in Figure 2 our constraint syntax for the Skalpel core. This syntax is used to represent constraints, for example in the constraint generator where we build the constraints that will be used to establish whether a program is typable or is erroneous (Section 3.2) and in the constraint solver (Section 3.3) which locates errors.

Sections 3.1.1 ... 3.1.3 explain the various parts of this syntax. The motivation is to build environments that avoid duplication at initial constraint generation or during constraint solving. Note that earlier systems (e.g. [Di Cosmo et al.(2005)Di Cosmo, Pottier, and Rémy]) are too restrictive to represent module systems because they only support very limited cases of our binders. With our constraints, we can easily define a compositional constraint generation algorithm.

Fig. 2 Syntax of constraint terms

$$\begin{array}{llll}
 & \in & & \text{(Union of below sets and } \mathsf{Label}\text{)}
\end{array}$$

$$\begin{array}{llllllll}
ev & \in & \mathsf{EnvVar} & \text{(environment variables)} & \gamma & \in & \mathsf{TyConName} & \text{(type constructor names)} \\
\delta & \in & \mathsf{TyConVar} & \text{(type constructor variables)} & \alpha & \in & \mathsf{ITyVar} & \text{(internal type variables)}
\end{array}$$

$$\begin{array}{llllllllll}
\mu & \in & \mathsf{ITyCon} & ::= & \delta \mid \gamma \mid \mathtt{arr} \mid \langle \mu, \overline{l} \rangle & tcs & \in & \mathsf{ITyConScheme} & ::= & \forall \overline{v}.\, \mu \\
\tau & \in & \mathsf{ITy} & ::= & \alpha \mid \tau\, \mu \mid \tau_1 \rightarrow \tau_2 \mid \langle \tau, \overline{l} \rangle & es & \in & \mathsf{EnvScheme} & ::= & \forall \overline{v}.\, e
\end{array}$$

$$\begin{array}{lllll}
ts & \in & \mathsf{ITyScheme} & ::= & \forall \overline{v}.\, \tau \qquad\qquad c \in \mathsf{EqCs} ::= \mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2 \\
bind & \in & \mathsf{Bind} & ::= & \downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \\
acc & \in & \mathsf{Accessor} & ::= & \uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha \\
e & \in & \mathsf{Env} & ::= & \top \mid ev \mid bind \mid acc \mid c \mid \mathtt{poly}(e) \mid \exists a.e \mid e_2;e_1 \mid \langle e, \overline{l} \rangle
\end{array}$$

extra metavariables

$$\begin{array}{llllllllll}
ct & \in & \mathsf{CsTerm} & ::= & \tau \mid \mu \mid e & v & \in & \mathsf{Var} & ::= & \alpha \mid \delta \mid ev \\
\sigma & \in & \mathsf{Scheme} & ::= & ts \mid tcs \mid es & a & \in & \mathsf{Atom} & ::= & v \mid \gamma \mid l \\
dep & \in & \mathsf{Dependent} & ::= & \langle ct, \overline{l} \rangle & & & & &
\end{array}$$

During analysis, a dependent form $\langle , \overline{l} \rangle$ depends on the program nodes with labels in $\overline{l}$ e.g. the dependent equality constraint $\langle \tau_1{=}\tau_2, \overline{l} {\cup} \{l\} \rangle$ might be generated for the labelled 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. In order to manipulate our labels, we define two functions strip and $\mathsf{collapse}$ below, which respectively allow us to take all labels off any given term, and to union nested labels of terms. Note that $\mathsf{dom}(\mathsf{strip}) = \mathsf{dom}(\mathsf{collapse}) =$, and $\mathsf{ran}(\mathsf{strip})$ is any piece of syntax which is not a dependent form, while $\mathsf{ran}(\mathsf{collapse}) =$.

$$\mathsf{strip}() = \begin{cases} \mathsf{strip}(y) & \text{if } = \langle y, \overline{l} \rangle \\ & \text{otherwise} \end{cases} \qquad \mathsf{collapse}() = \begin{cases} \mathsf{collapse}(\langle y, \overline{l} \cup \overline{l}' \rangle) \\ \quad \text{if } = \langle (\langle y, \overline{l} \rangle), \overline{l}' \rangle \\ \quad \text{otherwise} \end{cases}$$

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

3.1.1 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.[2] We have a special kind of type constructor `arr`, which is used to create a constraint in the constraint solving process between a unary type constructor and an arrow ($\rightarrow$) type.

3.1.2 Schemes (σ)

There are three kinds of universally quantified schemes: type schemes (similar to those in [Neubauer and Thiemann(2003)]), type constructor schemes, and environment schemes. All schemes are subject to alpha-conversion (e.g. the schemes $\forall \alpha_1.\, \alpha_1$

[2] Section 14.10 in [Rahli(2010)] presents a solution whereby type constructors can have any arity.

and $\forall\alpha_2.\,\alpha_2$ are equivalent).

3.1.3 The constraint/environment form (e)

The form e should be considered as both a constraint and an environment. Such a form can be any of the following:

(i) **The empty environment/satisfied constraint**. This is represented by $\top$.

(ii) **An environment variable**. 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.

(iii) **A composition environment**. We use the operator ';' to compose environments, which is associative. Note that $e;\top$, $\top;e$, and e are equivalent.

(iv) **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 an identifier id that are being bound, and accessors represent a place where that binding is used e.g., in the environment $\downarrow vid{=}x;\uparrow vid{=}\alpha$ the internal type variable α is constrained through the binding of vid to be an instance of x. 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 $\downarrow vid{=}x;...;\uparrow vid{=}\alpha$ it is *possible* that the binder and accessor of vid are connected. There are some environment forms that can be in the omitted (...) section which will mean that the accessor and the binder will be disconnected. Section 3.1.5 describes *shadowing*, which specifies which forms would cause this.

We abbreviate $\downarrow vid{=}\forall\varnothing.\,ct$ by $\downarrow vid{=}ct$ and abbreviate a dependent form $\langle\downarrow vid{=}ct, y\rangle$ by $\downarrow vid \overset{y}{=} ct$. Similarly for accessors.

(v) **An equality constraint**. A constraint where two pieces of constraint syntax are made to be equal.

(vi) **Existential environment**. The form $\exists x.e$, binds all free occurrences of x that occur free in e. We use the notation $\exists\langle x_1.\cdots, x_n\rangle.e$ to abbreviate $\exists x_1.\cdots\exists x_n.e$.

(vii) **A polymorphic environment**. This promotes the binders in the argument to `poly` to be polymorphic.

(viii) **Dependent form**. Label-annotated environments.

3.1.4 Atomic forms and Semantics of constraints/environments

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

$$\mathsf{vars}() = \mathsf{atoms}() \cap \mathsf{Var} \qquad\qquad \mathsf{labs}() = \mathsf{atoms}() \cap \mathsf{Label}$$

Note that $\mathsf{dom}(\mathsf{atoms}) = \mathsf{dom}(\mathsf{labs}) = \mathsf{dom}(\mathsf{vars}) =$, $\mathsf{ran}(\mathsf{atoms}) = \mathsf{Var} \cup \mathsf{Label}, \mathsf{ran}(\mathsf{labs}) = \mathsf{Label}$, and $\mathsf{ran}(\mathsf{vars}) = \mathsf{Var}$.

Checking parts of the program for mismatch requires substitution, unification, renaming, and accessing shadowed hidden information. These notions are defined in this section.

We define the sets of renamings Ren and substitutions Sub. Note $\mathsf{Ren} \subset \mathsf{Sub}$.

$$ren \in \mathsf{Ren} = \{\mathsf{ITyVar} \to \mathsf{ITyVar} \mid ren \text{ is injective} \wedge \mathsf{dj}(\mathsf{dom}(ren), \mathsf{ran}(ren))\}$$

$$sub \in \mathsf{Sub} = \{f_1 \cup f_2 \mid f_1 \in \mathsf{Unifier} \wedge f_2 \in \mathsf{TyConName} \to \mathsf{TyConName}\}$$

We also define our unifier set as a directed acyclic graph $\mathcal{U} \in \mathsf{Unifier} = \{,\}$ where $= \mathsf{ITyVar} \cup \mathsf{ITy} \cup \mathsf{ITyCon}$ and $= \mathbb{P}(\times)$ which specify directional edges. Note that for each $V_x \in$, the edge $V_x \mapsto V_x'$ occurs at most once, and so we also consider $\mathcal{U}$ as a function. When using an application $\mathcal{U}(V_x)$, vertex V_x' will be returned where a path from V_x to V_x' exists (if it does not, $V_x = V_x'$) and $V_x' \mapsto V_x''$ does not exist e.g., where $\mathcal{U} = \{\{V_1, V_2, V_3, V_4, V_5, V_6\}, \{V_1 \mapsto V_3, V_3 \mapsto V_2, V_4 \mapsto V_5, V_2 \mapsto V_6\}\}, \mathcal{U}(V_1) = V_6$. During application, if $\mathcal{U}(v) =_x$ and $\mathsf{vars}() \neq \{\}$, then for each $v' \in \mathsf{vars}()$ if $\mathcal{U}(v') \neq v'$ then it is replaced by $\mathcal{U}(v')$.

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 generated by our constraint solver (see Section 3.3). Substitution is defined in Figure 3, where given a constraint term and a substitution, a resulting constraint term is produced.

Fig. 3 Substitution semantics on constraint terms (from constraint terms to constraint terms)

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

3.1.5 Shadowing, Accessing and Instance

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 an environment it may be the case that some parts are shadowed and so inaccessible. Consider the environment $bind_1; ev; bind_2$. In the event that $ev \notin \mathsf{dom}(\mathcal{U})$, we say that ev *shadows* $bind_1$ because ev could potentially be bound to an environment which rebinds $bind_1$. We define $\mathsf{shadowsAll}$ by:

- $\mathsf{shadowsAll}(\langle \mathcal{U},\ e \rangle) \iff$

$$\begin{cases} \quad (e = ev \quad\ \wedge (\mathsf{shadowsAll}(\langle \mathcal{U}, \mathcal{U}(ev)\rangle) \vee \quad ev \notin \mathsf{dom}(\mathcal{U}))) \\ \vee\ (e = (e_1;e_2) \wedge (\mathsf{shadowsAll}(\langle \mathcal{U},\ e_1\rangle)\ \vee \quad \mathsf{shadowsAll}(\langle \mathcal{U},\ e_2\rangle)))) \\ \vee\ (e = \langle e', \overline{l}\rangle \quad \wedge \mathsf{shadowsAll}(\langle \mathcal{U},\ e'\rangle)) \\ \vee\ (e = \exists a.e' \quad \wedge \mathsf{shadowsAll}(\langle \mathcal{U},\ e'\rangle) \wedge \quad a \notin \mathsf{dom}(\mathcal{U})) \end{cases}$$

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

Note that $\mathsf{dom}(\mathsf{shadowsAll}) = \mathsf{tuple}(\mathcal{U} \times e)$ and $\mathsf{ran}(\mathsf{shadowsAll})$ is either true or false. We now present how to access the semantics of an identifier in an environment below, in the context where we have access to a unifier set $\mathcal{U}$ during constraint solving.

$$\begin{array}{lll} (\downarrow id{=}\sigma)(id) & = \sigma & \\ (e^{\overline{l}})(id) & = \forall \overline{v}.\ ct^{\overline{l}},\ \text{if } (e)(id) = \forall \overline{v}.\ ct & \\ (e_1;e_2)(id) & = \begin{cases} (e_2)(id), & \text{if } (e_2)(id) \text{ is defined} \\ \text{undefined}, & \text{if } (e_2)(id) \text{ is undefined} \\ & \text{and } \mathsf{shadowsAll}(\langle \mathcal{U},\ e_2\rangle) \\ (e_1)(id), & \text{otherwise} \end{cases} & \end{array}$$

$$\begin{array}{lll} (ev)(id) & = & \begin{cases} (e)(id), \text{if } \mathcal{U}(ev) = e \\ \text{undefined, otherwise} \end{cases} \\ (\langle e\rangle)(id) & = & e(id) \\ (\langle e_1\rangle @ \langle e_2\rangle)(id) & = & (e_1;\, e_2)(id) \end{array}$$

Since an existential environment represents incomplete information, its application to an identifier is undefined. Finally, we define two instance relations here, the use of which can be seen in constraint solving.

$$\forall \overline{v}.\ ct, sub \xrightarrow{\mathsf{instance}} ct[sub] \text{ if } \mathsf{dom}(sub) = \overline{v} \qquad\qquad \sigma \xrightarrow{e} ct \text{ if } \exists sub.\sigma, sub \xrightarrow{\mathsf{instance}} e, ct$$

3.2 Constraint generation

In this section we introduce our constraint generator, which generates constraints between parts of the user program which affect each other in some way. Our *constraint generator* is defined in Figure 4. Note that there are other types of constraints during the solving process.

Let $\overline{v}$ be a function with two arguments, the first a labelled piece of user program , and the second a set of free variables occurring in . Each of the constraint generation rules is written either as $[\![]\!] = e$ (which abbreviates $\{\} = e$) or as $[\![, v]\!] = e$ (which abbreviates $\{v\} = e$). Let $= \{\}$

It can be seen that datatype declarations only have one constructor by looking at rules (G17), (G14), and (G16). We have defined the core in this manner in order to reduce the complexity of the core. In rule (G13) we define the datatype names to have exactly one type variable argument.

Structure declarations are handled in rule (G20). To reduce complexity, we do not handle signatures in the core but this theory can be seen in [Pirie(2014)].

To allow us to slice out environments correctly, we annotate environment variables with labels, such as in rule (G4). We must annotate such environment variables with a label, otherwise we would not be able to slice it out, and that environment variable would then shadow any following environment.

In order to generate constraints for our running example, we must apply rule (G4) to the program we labelled for the fn-expression, and rule (G6) to handle the pattern of the anonymous function. These two rules are used to produce the below:

$$[\exists\langle\alpha_1,\alpha_2,ev\rangle.(ev = {\downarrow}\mathtt{y} \overset{l_2}{=} \alpha_1); ev^l; [\![exp,\alpha_2]\!]; (\alpha \overset{l}{=} \alpha_1 \rightarrow \alpha_2)]$$

The *exp* here represents the body of the function, which we can see is a let statement. For this we use rule (G2) to produce:

$$[\exists\alpha_3.[\![dec]\!]; [\![exp,\alpha_3]\!]; (\alpha_2 \overset{l_3}{=} \alpha_3)]$$

where *dec* represents the declarations and *exp* represents the expression of the let statement. We deal with the declarations first, applying rules (G17) to create constraints for the `val rec` statement and (G6) to handle the name of the function (`f`) to give:

$$\exists\langle\alpha_4,\alpha_5,ev_2\rangle.(ev_2 = \mathtt{poly}({\downarrow}\mathtt{f} \overset{l_8}{=} \alpha_4; [\![exp,\alpha_5]\!]; (\alpha_4 \overset{l_7}{=} \alpha_5))); ev_2^{l_7}$$

Constraints continue to be generated in this way, until we reach the final generated constraints for this program, which are shown below.

$$[\exists\langle\alpha_1,\alpha_2,ev\rangle.(ev = {\downarrow}\mathtt{y} \overset{l_2}{=} \alpha_1); ev^l; [\exists\alpha_3.\exists\langle\alpha_4,\alpha_5,ev_2\rangle.(ev_2 = \mathtt{poly}({\downarrow}\mathtt{f} \overset{l_8}{=} \alpha_4; [\exists\langle\alpha_6,\alpha_7,ev_3\rangle.(ev_3 = {\downarrow}\mathtt{x} \overset{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8,\alpha_9\rangle.{\uparrow}\mathtt{x} \overset{l_{12}}{=} \alpha_8; {\uparrow}\mathtt{y} \overset{l_{13}}{=} \alpha_9; (\alpha_8 \overset{l_{11}}{=} \alpha_9 \rightarrow \alpha_7); \alpha_5 \overset{l_{10}}{=} \alpha_6 \rightarrow \alpha_7]; (\alpha_4 \overset{l_7}{=} \alpha_5))); ev_2^{l_7}; \exists\langle\alpha',\alpha''\rangle.{\uparrow}\mathtt{f} \overset{l_4}{=} \alpha'; {\uparrow}\mathtt{y} \overset{l_5}{=} \alpha''; (\alpha' \overset{l_6}{=} \alpha'' \rightarrow \alpha_2); (\alpha_2 \overset{l_3}{=} \alpha_3)]; (\alpha \overset{l}{=} \alpha_1 \rightarrow \alpha_2)]$$

Next, we show that constraint generation is linear in size, and that our constraint generation algorithm terminates.

Lemma 3.1 (Size of Constraint Generation)
Constraint generation is linear in the 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 poly and composition environments, and binders force solving the constraints for the body before copying its type for each use of the function. □

Lemma 3.2 (Termination of Constraint Generation Algorithm) *The constraint generator shown in Figure 4 terminates.*

Proof. Let us define an *atomic constraint generation rule* as constraint generation rule which does not create a recursive call e.g., the atomic constraint generation rules in Figure 4 are (G1), (G5), (G6), (G7) (G9), (G10), (G13), (G14), (G19), and (G21). For a constraint generation run $\overline{v}$ either will be atomic in nature or it will not. If not, we recurse with $'\overline{v}'$, on some $'$ inside , such that $'$ is strictly smaller than . Rules which recurse with strictly smaller parts of external syntax are rules (G2) (`let` syntax

Fig. 4 Constraint generator ($\rightarrow$ Env)

Expressions (*exp*)

(G1) $\llbracket vid^{l}_{\mathtt{e}}, \alpha\rrbracket = \uparrow vid \overset{l}{=} \alpha$ (G2) $\llbracket \mathtt{let}^{l}\ dec\ \mathtt{in}\ exp\ \mathtt{end}, \alpha\rrbracket = [\exists \alpha_2 . \llbracket dec\rrbracket ; \llbracket exp, \alpha_2\rrbracket ; (\alpha \overset{l}{=} \alpha_2)]$

(G3) $\llbracket \lceil exp\ \ atexp\rceil^{l}, \alpha\rrbracket = \exists\langle \alpha_1, \alpha_2\rangle . \llbracket exp, \alpha_1\rrbracket ; \llbracket atexp, \alpha_2\rrbracket ; (\alpha_1 \overset{l}{=} \alpha_2 \to \alpha)$

(G4) $\llbracket \mathtt{fn}\ pat \overset{l}{\Rightarrow} exp, \alpha\rrbracket = [\exists\langle \alpha_1, \alpha_2, ev\rangle . (ev = \llbracket pat, \alpha_1\rrbracket) ; ev^{l} ; \llbracket exp, \alpha_2\rrbracket ; (\alpha \overset{l}{=} \alpha_1 \to \alpha_2)]$

Labelled datatype constructors (*ldcon*)

(G5) $\llbracket dcon^{l}, \alpha\rrbracket = \uparrow dcon \overset{l}{=} \alpha$

Patterns (*pat*)

(G6) $\llbracket vvar^{l}_{\mathtt{p}}, \alpha\rrbracket = \downarrow vvar \overset{l}{=} \alpha$ (G7) $\llbracket dcon^{l}_{\mathtt{p}}, \alpha\rrbracket = \uparrow dcon \overset{l}{=} \alpha$

(G8) $\llbracket \lceil ldcon\ \ atpat\rceil^{l}, \alpha\rrbracket = \exists\langle \alpha_1, \alpha_2\rangle . \llbracket ldcon, \alpha_1\rrbracket ; \llbracket atpat, \alpha_2\rrbracket ; (\alpha_1 \overset{l}{=} \alpha_2 \to \alpha)$

Labelled type constructors (*ltc*)

(G9) $\llbracket tc^{l}, \delta\rrbracket = \uparrow tc \overset{l}{=} \delta$

Types (*ty*)

(G10) $\llbracket tv^{l}, \alpha\rrbracket = \uparrow tv \overset{l}{=} \alpha$ (G11) $\llbracket \lceil ty\ \ ltc\rceil^{l}, \alpha'\rrbracket = \exists\langle \alpha, \delta\rangle . \llbracket ty, \alpha\rrbracket ; \llbracket ltc, \delta\rrbracket ; (\alpha' \overset{l}{=} \alpha\, \delta)$

(G12) $\llbracket ty_1 \overset{l}{\rightarrow} ty_2, \alpha\rrbracket = \exists\langle \alpha_1, \alpha_2\rangle . \llbracket ty_1, \alpha_1\rrbracket ; \llbracket ty_2, \alpha_2\rrbracket ; (\alpha \overset{l}{=} \alpha_1 \to \alpha_2)$

Datatype names (*dn*)

(G13) $\llbracket \lceil tv\ \ tc\rceil^{l}, \alpha'\rrbracket = \exists\langle \alpha, \gamma\rangle . (\alpha' \overset{l}{=} \alpha\, \gamma) ; (\downarrow tc \overset{l}{=} \gamma) ; (\downarrow tv \overset{l}{=} \alpha)$

Constructor bindings (*cb*)

(G14) $\llbracket dcon^{l}_{\mathtt{c}}, \alpha\rrbracket = \downarrow dcon \overset{l}{=} \alpha$ (G16) $\llbracket dcon\ \mathtt{of}^{\,l}\ ty, \alpha\rrbracket = \exists\langle \alpha', \alpha_1\rangle . \llbracket ty, \alpha_1\rrbracket ; (\alpha' \overset{l}{=} \alpha_1 \to \alpha) ; (\downarrow dcon \overset{l}{=} \alpha')$

Declarations (*dec*)

(G17) $\llbracket \mathtt{val\ rec}\ pat \overset{l}{=} exp\rrbracket = \exists\langle \alpha_1, \alpha_2, ev\rangle . (ev = \mathtt{poly}(\llbracket pat, \alpha_1\rrbracket ; \llbracket exp, \alpha_2\rrbracket ; (\alpha_1 \overset{l}{=} \alpha_2))) ; ev^{l}$

(G18) $\llbracket \mathtt{datatype}\ dn \overset{l}{=} cb\rrbracket = \exists\langle \alpha_1, \alpha_2, ev\rangle . (ev = ((\alpha_1 \overset{l}{=} \alpha_2) ; \llbracket dn, \alpha_1\rrbracket ; \mathtt{poly}(\llbracket cb, \alpha_2\rrbracket))) ; ev^{l}$

(G19) $\llbracket \mathtt{open}^{l}\ strid\rrbracket = \exists ev . (\uparrow strid \overset{l}{=} ev) ; ev^{l}$

Structure declarations (*strdec*)

(G20) $\llbracket \mathtt{structure}\ strid \overset{l}{=} strexp\rrbracket = \exists\langle ev, ev'\rangle . [\llbracket strexp, ev\rrbracket] ; (ev' = (\downarrow strid \overset{l}{=} ev)) ; ev'^{l}$

Structure expressions (*strexp*)

(G21) $\llbracket strid^{l}, ev\rrbracket = \uparrow strid \overset{l}{=} ev$

(G22) $\llbracket \mathtt{struct}^{l}\ strdec_1 \cdots strdec_n\ \mathtt{end}, ev\rrbracket = \exists ev' . (ev \overset{l}{=} ev') ; (ev' = (\llbracket strdec_1\rrbracket ; \cdots ; \llbracket strdec_n\rrbracket))$

removed in recursive call), (G3) (application syntax removed), (G4) (`fn` syntax removed), (G8) (application removed), (G11) (application removed), (G12) (arrow removed), (G16) (`of` syntax removed), (G17) (`val rec` removed), (G18) (`datatype` syntax removed), (G20) (`structure` syntax removed), and (G22) (`struct` syntax removed). When we inevitably reach an atomic , we halt and return our generated e form. □

3.3 *Constraint solving*

In this section we present our new constraint solver, which solves the constraints that were generated by the constraint generator in the previous section. It is in this process where we will determine if the program the user submitted is erroneous, and will return all relevant parts of the program involved in the error if that is indeed the case. Additional syntactic forms that are used by the constraint solver (defined in Figure 6) are given in Figure 5. The symbol $\overrightarrow{st}$ is defined in Section 3.3.2, and is used to keep track of future environments that we have yet to solve.

Fig. 5 Extra syntactic forms for constraint solving

m	$\in$ Monomorphic	$::= \langle \alpha, \overline{l} \rangle$
er	$\in$ Error	$::= \langle ek, \overline{l} \rangle$
ek	$\in$ ErrKind	$::= \mathtt{clash}(\mu_1, \mu_2) \mid \mathtt{circularity}$
$state$	$\in$ State	$::= \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e') \mid \mathtt{succ} \mid \mathtt{err}(er)$

Constraint solving starts by $\mathtt{slv}(\langle \top \rangle, \varnothing, \varnothing, \langle\rangle, e)$, and ends either by $\mathtt{succ}$ (for success), or in the state $\mathtt{err}(er)$ where er is either a type constructor clash or a circularity error. The relations isErr and $\mathsf{solvable}$ are defined below, where $\to$ indicates a constraint solving step.

$$\begin{array}{lcl}
e \overset{\mathsf{isErr}}{\to} er & \Leftrightarrow & \mathtt{slv}(\top, \overline{l}, \varnothing, \varnothing, e) \to^{*} \mathsf{err}(er) \\[1ex]
\mathsf{solvable}(e) & \Leftrightarrow & \mathtt{slv}(\top, \overline{l}, \varnothing, \varnothing, e) \to^{*} \mathtt{succ} \\[1ex]
\mathsf{solvable}(strdec) & \Leftrightarrow & \exists e.strdec \to e \wedge \ \mathsf{solvable}(e)
\end{array}$$

3.3.1 Unifiers

When constraint solving starts, the set of unifiers $\mathcal{U}$ is initialised to the empty set ($\mathcal{U} = \varnothing$). During constraint solving, nothing is ever subtracted from $\mathcal{U}$, we only add to this set. The set of unifiers is used during constraint solving only (e.g. see rule (U3) of Figure 6).

Fig. 6 Constraint solver (1 of 2) : $\mathsf{State}\backslash\{\mathtt{succ}, \mathtt{err}(\mathsf{Error})\} \to state$

`equality constraint reversing`

(R) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, ct = ct') \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, ct' = ct)$, if $s = \mathsf{Var} \cup \mathsf{Dependent} \wedge ct' \in s \wedge ct \notin s$

`equality simplification`

(S1) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, ct = ct) \to \mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st})$

(S2) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, ct^{\overline{l}'} = ct') \to \mathtt{slv}(\overrightarrow{e}, \overline{l} \cup \overline{l}', m, \overrightarrow{st}, ct = ct')$

(S3) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \tau_1\ \mu_1 = \tau_2\ \mu_2) \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, (\mu_1 = \mu_2);(\tau_1 = \tau_2))$

(S4) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \tau_1 \to \tau_2 = \tau_3 \to \tau_4) \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, (\tau_1 = \tau_3);(\tau_2 = \tau_4))$

(S5) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \tau_1 = \tau_2) \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \mu = \mathtt{arr})$, if $\{\tau_1, \tau_2\} = \{\tau\ \mu, \tau_3 \to \tau_4\}$

(S6) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \mu_1 = \mu_2) \to \mathtt{err}(\langle \mathtt{clash}(\mu_1, \mu_2), \overline{l} \rangle)$, if$\{\mu_1, \mu_2\} \in \{\{\gamma, \gamma'\}, \{\gamma, \mathtt{arr}\}\} \wedge \gamma \neq \gamma'$

`unifier access`

Rules (U1) through (U4) have also the common side condition $v \neq ct \wedge y = \mathcal{U}(x^{\overline{l}}) \wedge v \notin \mathsf{dom}(\mathcal{U})$

(U1) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, v = ct) \to \mathtt{err}(\langle \mathtt{circularity}, \mathsf{deps}(y) \rangle)$, if $v \in \mathsf{vars}(y)\backslash\mathsf{Env} \ \wedge\ \mathsf{strip}(y) \neq v$

(U2) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, v = ct) \to \mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st})$, if $v \notin \mathsf{Env} \wedge\ \mathsf{strip}(y) \ = \ v$

(U3) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, v = ct) \to \mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st})$, if $v \notin \mathsf{vars}(y) \cup \mathsf{Env} \ \wedge\ \mathcal{U} = \mathcal{U} \oplus \{v \mapsto y\}$

(U4) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, v = ct) \to \mathtt{slv}(\overrightarrow{e} @ \langle \top \rangle, \overline{l}, m, \overrightarrow{st} @ \overrightarrow{st}', ct)$, if $v \in \mathsf{Env} \wedge \overrightarrow{st}' = \langle\langle \mathtt{new}, \overline{l}, m, v \rangle\rangle$

(U6) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, v = ct) \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, z = ct)$, if $\mathcal{U}(v) = z$

`composition environments`

(C1) $\mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e_1;e_2) \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st} @ \langle\langle \mathtt{new}, \overline{l}, \mathtt{new}, e_2 \rangle\rangle, e_1)$

3.3.2 The environment stack

The fourth argument to the $\mathtt{slv}$ function of the constraint solver in Figure 6, denoted as $\overrightarrow{st}$, is used as a stack of environments or other tasks which are still to be solved/completed. Below, we introduce some metavariables needed to define the stack:

Fig. 7 Constraint solver (2 of 2)

`binders/empty/dependent/variables`

(B) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, {\downarrow} vid{=}\alpha) \rightarrow \texttt{isSucc}(\overrightarrow{e}; {\downarrow} vid \overset{\overline{l}}{=} \alpha, m \cup \{\alpha^{\overline{l}}\}, \overrightarrow{st})$

(B2) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, bind) \rightarrow \texttt{isSucc}(\overrightarrow{e}; bind^{\overline{l}}, m, \overrightarrow{st})$, if $bind \neq {\downarrow} vid{=}\alpha$

(X) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \exists a.e') \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l} \cup \overline{l}', m, \overrightarrow{st}, e'[\{a \mapsto a'\}])$, if $a' \notin \mathsf{atoms}(\langle \mathcal{U}, e' \rangle)$

(E) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \top) \rightarrow \texttt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st})$

(D) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e'^{\overline{l}'}) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l} \cup \overline{l}', m, \overrightarrow{st}, e')$

(V) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, ev) \rightarrow \texttt{isSucc}(\overrightarrow{e}; ev^{\overline{d}}, m, \overrightarrow{st})$

`accessors`

(A1) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, {\uparrow} id{=}v) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l} \cup \overline{l}', m, \overrightarrow{st}, v = \tau)$,
if $\overrightarrow{e}(id), ren \xrightarrow{\mathsf{instance}} \tau, \overline{l}' \wedge \mathsf{dj}(\mathsf{vars}(\langle \overrightarrow{e}, v \rangle), \mathsf{ran}(ren))$

(A3) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, {\uparrow} id{=}v) \rightarrow \texttt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st})$, if $\overrightarrow{e}(id)$ undefined

`polymorphic environments`

(P1) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}({\downarrow} vid \overset{\overline{l}'}{=} \alpha)) \rightarrow \texttt{isSucc}(\overrightarrow{e}; \sigma, m, \overrightarrow{st})$,
if $\overline{\alpha} = \mathsf{ityvars}(\mathcal{U}(\alpha)) \setminus \bigcup \{\mathsf{ityvars}(\mathcal{U}(x)) \mid x \in m\}$
$\wedge\ \overline{l}'' = \overline{l}' \cup \mathsf{deps}(\mathsf{vars}(\mathcal{U}(\alpha)) \lhd \{\mathcal{U}(x) \mid x \in m\})$
$\wedge\ \sigma = {\downarrow} vid{=}\langle \forall \overline{\alpha}.\mathcal{U}(\alpha), \overline{l}'' \rangle$

(P2) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}(bind; e')) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st} @ \langle\langle \overrightarrow{e}, \overline{l}, m, \texttt{poly}(bind) \rangle\rangle, bind; e')$

(P3) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}(e_1^{\overline{l}})) \rightarrow \texttt{slv}(\overrightarrow{e} @ \langle \top \rangle, \overline{l}, m, \overrightarrow{st} @ \langle\langle \texttt{new}, \overline{l}, \texttt{new}, \overline{l} \rangle\rangle, \texttt{poly}(e_1))$

(P4) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}(e_1; e_2)) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st} @ \langle\langle \texttt{new}, \overline{l}, \texttt{new}, \texttt{poly}(e_2) \rangle\rangle, \texttt{poly}(e_1))$, if $\wedge e_1 \neq bind$

(P5) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}(e')) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st} @ \langle\langle \overrightarrow{e}; e', \overline{l}, m, \texttt{done} \rangle\rangle, e')$, if $e' \neq \exists a.e''$

(P6) $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}(\exists a.e')) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \texttt{poly}(e'[\{a \mapsto a'\}]))$ if $a' \notin \mathsf{atoms}(\langle \mathcal{U}, e' \rangle)$

$$\begin{array}{llll}
stackEv & \in & \mathsf{StackEv} & = & e \mid \texttt{new} \\
stackMono & \in & \mathsf{StackMono} & = & m \mid \texttt{new} \\
stackAction & \in & \mathsf{StackAction} & = & e \mid v \mid \overline{l} \mid \texttt{done}
\end{array}$$

This stack is a tuple where each element is itself a tuple which has four components: *stackEv*, $\overline{l}$, *stackMono*, and *stackAction*. *stackEv* is used to represent which environment we should use when taking action on the *stackAction* parameter. This can either be the symbol `new`, in which case we use the environment of the constraint solver when the `isSucc` function was called which deals with handling stack items, or instead it can be a specified environment e, in which case we use the environment pushed to the stack at the time when this stack item was created. $\overline{l}$ is a set of dependencies. *stackMono* is the same as *stackEv*, but with monomorphic variable sets instead of environments. *stackAction* contains operations to be performed. What we do in cases of *stackAction* can be seen in the declaration of `isSucc`′ in Figure 8, which checks for success.

When we have finished with solving the environment in the last position of the `slv` argument tuple, `isSucc` is called which solves the argument at the top of $\overrightarrow{st}$ stack, (the constraint solver terminates in the success state if it is empty). The definition of `isSucc` is given in Figure 8, where given a tuple of environments, a set of monomorphic variables and a stack of remaining environments still to process, will either recurse, return the constraint solver success state, or run the constraint solver on some environment.

Let us now continue our example. We now show the start form of the constraint generator and proceed from there. We start with the function call:

$\mathtt{slv}(\langle\top\rangle, \varnothing, \varnothing, \langle\rangle, e_1)$ where e_1 is the environment returned from the initial constraint generator, shown below.

$$[\exists\langle\alpha_1, \alpha_2, ev\rangle.(ev = {\downarrow}\mathtt{y} \overset{l_2}{=} \alpha_1); ev^{l}; [\exists\alpha_3.\exists\langle\alpha_4, \alpha_5, ev_2\rangle.(ev_2 = \mathtt{poly}({\downarrow}\mathtt{f} \overset{l_8}{=} \alpha_4; [\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = {\downarrow}\mathtt{x} \overset{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.{\uparrow}\mathtt{x} \overset{l_{12}}{=} \alpha_8; {\uparrow}\mathtt{y} \overset{l_{13}}{=} \alpha_9; (\alpha_8 \overset{l_{11}}{=} \alpha_9 \to \alpha_7); \alpha_5 \overset{l_{10}}{=} \alpha_6 \to \alpha_7]; (\alpha_4 \overset{l_7}{=} \alpha_5))); ev_2^{l_7}; \exists\langle\alpha', \alpha''\rangle.{\uparrow}\mathtt{f} \overset{l_4}{=} \alpha'; {\uparrow}\mathtt{y} \overset{l_5}{=} \alpha''; (\alpha' \overset{l_6}{=} \alpha'' \to \alpha_2); (\alpha_2 \overset{l_3}{=} \alpha_3)]; (\alpha \overset{l}{=} \alpha_1 \to \alpha_2)]$$

In this step we apply rules (U4) and (X) to remove the [] notation and existential quantification, renaming α_1, α_2, and ev to α_0, α_1, and ev' respectively. We now apply rules (C1) to break up the environment composition, then rules (U4), (D) to strip off the dependency on the binder, and (B) to handle the binder. Rules (C1), (D), and (V) are applied to handle the ev'^{l} expression, and we are then in the state shown below.

$\mathtt{slv}(\langle ev'^{\{l,l_2\}}\rangle, \{l, l_2\}, \{\alpha_0\}, \langle\rangle, e_3)$ where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto {\downarrow}\mathtt{y} \overset{l}{=} \alpha_0\}$ and e_3 is

$$[\exists\alpha_3.\exists\langle\alpha_4, \alpha_5, ev_2\rangle.(ev_2 = \mathtt{poly}({\downarrow}\mathtt{f} \overset{l_8}{=} \alpha_4; [\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = {\downarrow}\mathtt{x} \overset{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.{\uparrow}\mathtt{x} \overset{l_{12}}{=} \alpha_8; {\uparrow}\mathtt{y} \overset{l_{13}}{=} \alpha_9; (\alpha_8 \overset{l_{11}}{=} \alpha_9 \to \alpha_7); \alpha_5 \overset{l_{10}}{=} \alpha_6 \to \alpha_7]; (\alpha_4 \overset{l_7}{=} \alpha_5))); ev_2^{l_7}; \exists\langle\alpha', \alpha''\rangle.{\uparrow}\mathtt{f} \overset{l_4}{=} \alpha'; {\uparrow}\mathtt{y} \overset{l_5}{=} \alpha''; (\alpha' \overset{l_6}{=} \alpha'' \to \alpha_1); (\alpha_1 \overset{l_3}{=} \alpha_3)]; (\alpha \overset{l}{=} \alpha_0 \to \alpha_1)$$

Application of the constraint solving rules continue in this way until either the program is deemed typable, or an error is determined. A complete description of all steps used is too verbose to give here but can be seen in Section 8.2.2 of [Pirie(2014)]. The constraint solver terminates with the `circularity` error, and the gathered labels (program points) are used to highlight *all* the relevant parts of the program to the user. Such an error report is a significant benefit from what the compiler reports which is merely one program point where unification failed. With our errors, as shown in Figure 9, the user sees *all* of the information they need to solve a type error, and not just a *small portion* of that information.

Fig. 8 Success definition ($\mathtt{isSucc} : \mathsf{tuple}(\mathsf{Env}) \times \mathsf{Monomorphic} \times \mathsf{tuple}(\mathsf{Env}) \to \mathit{state}\backslash\mathtt{err}(\mathsf{Error})$)

$$\mathtt{isSucc}(\overrightarrow{e}, m, \langle\rangle) \to \mathtt{succ}$$
$$\mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st}@\langle\langle\mathtt{new}, \overline{l}, \mathtt{new}, x\rangle\rangle) \to \mathtt{isSucc}'(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, x)$$
$$\mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st}@\langle\langle\overrightarrow{e}_1, \overline{l}, \mathtt{new}, x\rangle\rangle) \to \mathtt{isSucc}'(\overrightarrow{e}_1, \overline{l}, m, \overrightarrow{st}, x)$$
$$\mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st}@\langle\langle\mathtt{new}, \overline{l}, m', x\rangle\rangle) \to \mathtt{isSucc}'(\overrightarrow{e}, \overline{l}, m', \overrightarrow{st}, x)$$
$$\mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st}@\langle\langle\overrightarrow{e}_1, \overline{l}, m', x\rangle\rangle) \to \mathtt{isSucc}'(\overrightarrow{e}_1, \overline{l}, m', \overrightarrow{st}, x)$$

$$\mathtt{isSucc}'(\overrightarrow{e}@\langle e_1, e_2\rangle, \overline{l}, m, \overrightarrow{st}, v) \to \mathtt{isSucc}(\overrightarrow{e}@\langle e_1;e_2\rangle, m, \overrightarrow{st}),\ \text{if } \mathcal{U} = \mathcal{U} \oplus \{v \mapsto e_2\}$$
$$\mathtt{isSucc}'(\overrightarrow{e}@\langle e_1, e_2\rangle, \overline{l}, m, \overrightarrow{st}, \overline{l}) \to \mathtt{isSucc}(\overrightarrow{e}@\langle e_1;e_2^{\overline{l}}\rangle, m, \overrightarrow{st})$$
$$\mathtt{isSucc}'(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, \mathtt{done}) \to \mathtt{isSucc}(\overrightarrow{e}, m, \overrightarrow{st})$$
$$\mathtt{isSucc}'(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e') \to \mathtt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e')$$

We further analyze some interesting constraint solving rules. Rule (C1) demonstrates how we handle our composition environments. We take the first environment and recurse on that first to solve the constraints inside, and only after they are handled we inspect the second environment. Polymorphism is handled in rule (P1), where we make a binder polymorphic by quantifying over the type variables which are to be made polymorphic, and creating a new binder with this information.

Lemma 3.3 (Constraint Solving Terminates)

It holds that either $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e) \rightarrow^* \texttt{succ}$ *or* $\texttt{slv}(\overrightarrow{e}, \overline{l}, m, \overrightarrow{st}, e) \rightarrow^* \texttt{err}(er)$*.*

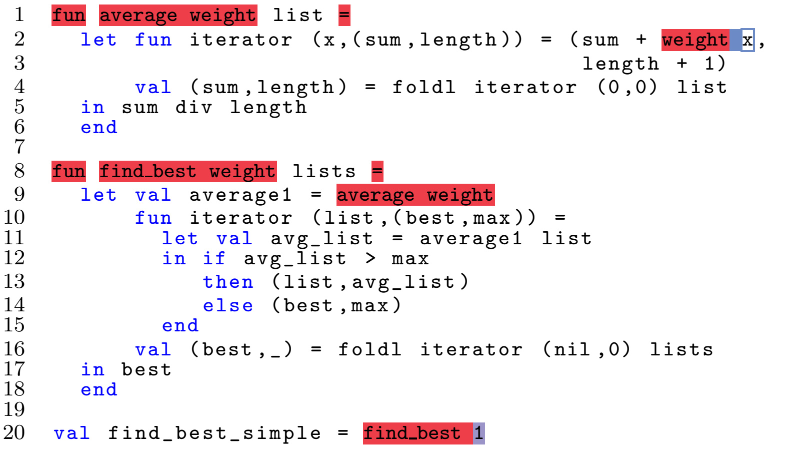
Proof. By inspection of the rules. We only summarize the proof for the important rules ([Pirie(2014)] contains a more thorough treatment). (R) flips constraints and flipped constraints can never be re-flipped. (S1) Throws away argument/adds to environment or unifier, and checks for success. (S3)/(S4) Break two applications (resp. arrow types) into two equality constraint terms. We never build new applications of constraints (resp. arrow types), so we cannot return to this point. The only rules which can be the final rules to be executed and raise error are rules `U1` and `S6` which terminate in the form $\texttt{err}(er)$, otherwise, the constraint solver will terminate in the `succ` state shown in Figure 8. □

3.4 Comprehensive errors

A crucial property of Skalpel is that it **must** present to the user **all** of the possible points where the user may fix the error. Skalpel **must not** present any program points which are irrelevant to the error. In order to ensure that this is always the case, we perform *minimisation*. When the constraint solver terminates with an error (which contains the program points, $\overline{l}$) the minimisation algorithm tests that *all* of the labels present in the reported error. It does this by removing a program point l from the program, replacing it with a dummy expression, and running the constraint solver again. If this run terminates in success, then the label was crucial to the error, and so it must be presented to the user. If the constraint solver terminates with the same error, then we know that this program point is actually irrelevant, and so we discard it from $\overline{l}$. We do this process for *all* labels in $\overline{l}$ reported as part of an error output from the constraint solver. A formal treatment of this algorithm can be seen in Section 6.5 of [Pirie(2014)]. We then present these regions to the user as shown in Figure 9. Note that a standard SML compiler, such as PolyML [POL(web)], will only report line 20 as the source of the error, which in a larger program, could cause a great deal of confusion (especially if the error is spread across multiple files - which Skalpel also handles by highlighting all areas in affected files).

Naturally, Skalpel is at its most effective in large codebases. If global changes to an entire project are needed to fix a type error, Skalpel will highlight where the problem may be fixed in all areas of the project. Furthermore, when large code bases are used, and the type error is limited to a few small functions, Skalpel will eliminate the rest of the program for the user, which is irrelevant, as opposed to existing compilers which do not rule out anything, as they only present the point where unification failed. This is achieved as a) determining which program parts to highlight (labels involved in the error) is calculated accurately by our constraint solver, as label sets are attached to each constraint, so we know which parts of the user program include conditions on other parts of the program, and b) the process of minimisation also ensures that no irrelevant part of the program is highlighted to the user.

Fig. 9 Skalpel highlighting



```
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
```

Note that Skalpel does not merely just find one error, but can find all distinct errors. We do not present the details of this mechanism here, but they can be found in Section 6.5 of [Pirie(2014)].

4 Conclusion

Automatically finding type errors in programming languages is a difficult task. Successful attempts need to address constraint systems but these have only been built for toy-like languages. Moreover, existing proposals to solve poor type error reporting simply repeat calls to the compiler and remove/add back portions of the untypable program to narrow the point of error.

In this paper we present Skalpel, which:

(i) Takes an SML program and returns exactly the erroneous parts of the program;

(ii) Does not report any portion of internally modified syntax, as can be presented by the available compilers for the language;

(iii) Will display *all* parts of an error to the user;

(iv) Is completely unbiased in its analysis as compilers are;

(v) Handles errors which occur across multiple modules and/or source files.

Skalpel automatically achieves all of the above by first labelling all parts of the program generating constraints annotated with these labels, solving these constraints and if errors are found, performing minimisation to verify the integrity of the error that we present to the user.

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. This solution is related to earlier constraint systems for ML let bindings [Di Cosmo et al.(2005)Di Cosmo, Pottier, and Rémy] however these earlier ideas are unsuitable for module systems which is why we needed a new constraint representation. 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.

To our knowledge, no work exists that attempts to handle an entire programming language using a constraint system approach such as ours, the core of which is presented in this paper.

References

[POL(web)] , web. PolyML compiler. www.polyml.org/, last accessed 20th January 2014.

[Braßel(2004)] Braßel, B., 2004. Typehope: There is hope for your type errors. In: In 16th International Workshop on Implementation and Application of Functional Languages (IFL'04). Lübeck, Germany, September 8-10 2004. University of Kiel. Report 0408.

[Damas and Milner(1982)] Damas, L., Milner, R., 1982. Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '82. ACM, New York, NY, USA, pp. 207–212.
URL http://doi.acm.org/10.1145/582153.582176

[Di Cosmo et al.(2005)Di Cosmo, Pottier, and Rémy] Di Cosmo, R., Pottier, F., Rémy, D., Apr. 2005. Subtyping recursive types modulo associative commutative products. In: Seventh International Conference on Typed Lambda Calculi and Applications (TLCA'05). Vol. 3461 of Lecture Notes in Computer Science. Springer, Nara, Japan, pp. 179–193.
URL http://gallium.inria.fr/~fpottier/publis/dicosmo-pottier-remy-tlca05.ps.gz

[Hage and Heeren(2009)] Hage, J., Heeren, B., Apr. 2009. Strategies for solving constraints in type and effect systems. Electron. Notes Theor. Comput. Sci. 236, 163–183.
URL http://dx.doi.org/10.1016/j.entcs.2009.03.021

[Lerner and Grossman(2006)] Lerner, B., Grossman, 2006. Seminal: searching for ML type-error messages. In: Proceedings of the 2006 workshop on ML. ML '06. ACM, New York, NY, USA, pp. 63–73.
URL http://doi.acm.org/10.1145/1159876.1159887

[Mcadam(1998)] Mcadam, B. J., 1998. On the unification of substitutions in type inference. In: Implementation of Functional Languages (IFL '98). Springer-Verlag, pp. 139–154.

[Müller(1994)] Müller, M., Jun.23–25 1994. A constraint-based recast of ML-polymorphism. In: Lugiez, D. (Ed.), 8^{th} International Workshop on Unification. Technical Report, Université de Nancy, to appear.

[Neubauer and Thiemann(2003)] Neubauer, M., Thiemann, P., 2003. Discriminative sum types locate the source of type errors. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. ICFP '03. ACM, New York, NY, USA, pp. 15–26.
URL http://doi.acm.org/10.1145/944705.944708

[O. Lee(1998)] O. Lee, K. Y., 1998. Proofs about a folklore let-polymorphic type inference algorithm. ACM Transactions on Programming Languages and Systems (TOPLAS) 20, 707–723.

[Pavlinovic et al.(2014)Pavlinovic, King, and Wies] Pavlinovic, Z., King, T., Wies, T., 2014. Finding minimum type error sources. In: OOPSLA 2014. ACM, pp. 525–542.
URL http://doi.acm.org/10.1145/2660193.2660230

[Pirie(2014)] Pirie, J., 2014. New Developments to Skalpel: A Type Error Slicing Method for Explaining Errors in Type and Effect Systems. Ph.D. Thesis. Available at http://www.macs.hw.ac.uk/~jp95/jpirie-thesis.pdf.

[Rahli(2010)] Rahli, V., 2010. Investigations in intersection types: Confluence and semantics of expansion in the lamba-calculus, and a type error slicing method. http://www.macs.hw.ac.uk/~rahli/articles/thesis.pdf, ph.D. Thesis. Last accessed Monday 16th July 2012.

[Schilling(2012)] Schilling, T., 2012. Constraint-free type error slicing. In: Trends in Functional Programming. Vol. 7193 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–16.
URL http://dx.doi.org/10.1007/978-3-642-32037-8_1

[Zhang and Myers(2014)] Zhang, D., Myers, A. C., 2014. Toward general diagnosis of static errors. In: POPL '14. ACM, pp. 569–582.
URL http://doi.acm.org/10.1145/2535838.2535870

[Zhang et al.(2015)Zhang, Myers, Vytiniotis, and Peyton-Jones] Zhang, D., Myers, A. C., Vytiniotis, D., Peyton-Jones, S., 2015. Diagnosing type errors with class, PLDI'2015, available at http://www.cs.cornell.edu/~zhangdf/pub/pldi15.pdf.