Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)

[Electronic Notes in Theoretical Computer Science 312 (2015) 197–213](http://dx.doi.org/10.1016/j.entcs.2015.04.012)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Skalpel: A Type Error Slicer for Standard ML

Vincent Rahli

*Cornell University, Ithaca*

Joe Wells, 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.

# 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 [[3](#_bookmark22)], which blames a single abstract syntax tree node when unification fails. Variations on this algorithm such as M [[10](#_bookmark30)] and W’ [[7](#_bookmark25)], have been developed to solve the left-to-right bias of the W algorithm. However, all these algorithms still blame a single node in

<http://dx.doi.org/10.1016/j.entcs.2015.04.012>

1571-0661/© 2015 The Authors. Published by Elsevier B.V.

This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

the abstract syntax tree for an error which is made up of multiple error locations. In addition, 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 con- straint 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 [[8](#_bookmark26)] and [[5](#_bookmark27)]. A more promising approach has been taken in [[14](#_bookmark34)], 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., [[2](#_bookmark23)], [[6](#_bookmark28)], and [[13](#_bookmark33)]) 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 so- lutions 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 gen- erated, 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 anno- tate 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 terminat- ing). 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 high- lighted 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

[[10](#_bookmark30)] and W’ [[7](#_bookmark25)], 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 (Lemma [3.1](#_bookmark11) and [3.3](#_bookmark18)).

* + 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](#_bookmark0) we discuss the basic notation used. In Section [3](#_bookmark1) 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](#_bookmark20).

# Mathematical notations

Let *i, j, m, n, p, q* range over the set N of natural numbers. If *v* ranges over a class *C* , then *vx* (where *x* can be anything) and *vj, vjj*, etc., also range over *C* . Let *s* range over sets. If *v* ranges over *s*, then let *v* range over P(*s*), the power set of *s*. Let dj(*s*1*,..., sn*) (“disjoint”) hold iff for all *i, j ∈ {*1*,..., n}*, if *i /*= *j* then *si∩sj* = *∅*. Let *s*1 *] s*2 be *s*1 *∪s*2 if dj(*s*1*, s*2) and undefined otherwise. Let Q*x, y*¢ be the pair of *x* and

*y*. If *rel* is a binary relation (a pair set), let (*x rel y*) iff Q*x, y*¢ *∈ rel* , let the inverse of *rel* be *rel —*1 defined as *{*Q*x, y*¢ *|* Q*y, x*¢ *∈ rel }*, let dom(*rel* )= *{x | ∃y.*Q*x, y*¢ *∈ rel }*, let ran(*rel* ) = *{y | ∃x.*Q*x, y*¢ *∈ rel }*, let *s a rel* = *{*Q*x, y*¢ *∈ rel | x ∈ s}*, and let *s d rel* = *{*Q*x, y*¢ *∈ rel | x /∈ s}*. Let *f* range over functions (a special case of binary relations), let *s → sj* = *{f |* dom(*f* ) *⊆ s ∧* ran(*f* ) *⊆ sj}*, and let *x '→ y* be an alternative notation for Q*x, y*¢ used when writing some functions. A tuple *t* is a function such that dom(*t* ) *⊂* N and if 1 *≤ j ∈* dom(*t* ) then *j −* 1 *∈* dom(*t* ).

Let *t* range over tuples. If *v* ranges over *s* then let *−→v* range over tuple(*s*) = *{t |*

ran(*t* ) *⊆ s}*. We write the tuple *{*0 *'→ x*0*,...,n '→ xn}* as *⟨x*0*,..., xn⟩*. Let @ append tuples: *⟨x*1*,..., xi⟩*@*⟨y*1*,..., yj⟩* = *⟨x*1*,..., xi, y*1*,..., yj⟩*. Given *n* sets *s*1*,..., sn*,

let *s*1*, ..., sn* be *{⟨x*1*,..., xn⟩ | ∀i ∈ {*1*,..., n}.xi ∈ si}*. Note that *s*1*, ..., sn ⊆* tuple(*s*1 *∪· · ·∪ sn*). For some reduction relation R we write R*∗* for its reflexive and transitive closure.

# 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](#_bookmark3) which describes a subset of the SML language, chosen to present the core ideas. [1](#_bookmark2) Most syntactic forms have labels (*l* ), which are generated to track blame for errors. We

1 We do not enforce all the syntactic restrictions of the SML syntax e.g. in val rec *pat* = *exp*, the expression

*exp* must be an fn-expression (which we do not enforce in this paper).

*l*

surround some terms such as function application with *⊆ |* in order to provide a visually convenient place for labels.

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

*l ∈* Label (labels) *PL ∈* ExtLabSynt (Union of below sets)

*tv ∈* TyVar (type variables)

*tc ∈* TyCon (type constructors) *strid ∈* StrId (structure identifiers) *vvar ∈* ValVar (value variables)

*dcon ∈* DatCon (datatype constructors)

*vid ∈* VId ::= *vvar | dcon*

*ltc ∈* LabTyCon ::= *tcl ldcon ∈* LabDatCon ::= *dconl*

*dn ∈* DatName ::= *[tv tc|l atpat ∈* AtPat ::= *vidl*

p

|  |  |  |  |
| --- | --- | --- | --- |
| *cb* | *∈* | ConBind | ::= *dconl | dcon* of *l ty*  c |
| *atexp* | *∈* | AtExp | ::= *vidl |* let*l dec* in *exp* end  e |
| *pat* | *∈* | Pat | ::= *atpat | [ldcon atpat|lab* |
| *ty* | *∈* | Ty | ::= *tvl | ty l ty | [ty ltc|l*  1 *→* 2 |
| *strdec* | *∈* | StrDec | ::= *dec |* structure *strid l strexp*  = |
| *strexp* | *∈* | StrExp | ::= *stridl |* struct*l strdec*1 *··· strdecn* end |
| *dec* | *∈* | Dec | ::= val rec *pat l exp |* open*l strid |* datatype *dn l cb*  = = |
| *exp* | *∈* | Exp | ::= *atexp |* fn *pat l exp | [exp atexp|l*  *⇒* |
| *id* | *∈* | Id | ::= *vid | strid | tv | tc* |
| *term* | *∈* | Term | ::= *ltc | ldcon | ty | cb | dn | exp | pat | strdec | strexp* |

We will present a running example throughout this paper. The SML program we will use as an example is shown below. We present this here in order to show how syntax is annotated with labels.

fn y*l*2 *l* let*l*3

*⇒*

val rec f

l8 =l7 fn x

l9 l10

l12

*⇒ [*x

yl13 *|*l11 in *[*f

l4 yl5 *|*l6 end

In Figure [1](#_bookmark3), value identifiers (*vid* ) are subscripted to disambiguate rules for expressions (*vidl* ), datatype constructor definitions (*dconl* ), and pattern (*vidl* ) oc-

e c p

currences. 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 as- signing 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 [[12](#_bookmark32)] 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.

* 1. *Constraint syntax*

In this section we give in Figure [2](#_bookmark4) 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](#_bookmark10)) and in the constraint solver (Section [3.3](#_bookmark13)) which locates errors.

Sections [3.1.1](#_bookmark5) ... [3.1.3](#_bookmark7) 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. [[4](#_bookmark24)]) 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

*CL ∈* IntLabSynt (Union of below sets and Label)

*ev ∈* EnvVar (environment variables)

*δ ∈* TyConVar (type constructor variables)

*μ ∈* ITyCon ::= *δ | γ |* arr *| ⟨μ, l⟩*

*τ ∈* ITy ::= *α | τ μ | τ*1 ⭢ *τ*2 *| ⟨τ, l⟩*

*γ ∈* TyConName (type constructor names)

*α ∈* ITyVar (internal type variables)

*tcs ∈* ITyConScheme ::= *∀v.μ*

*es ∈* EnvScheme ::= *∀v. e*

*e*1 = *e*2 *| τ*1 = *τ*2

|  |  |  |  |
| --- | --- | --- | --- |
| *ts*  *bind* | *∈*  *∈* | ITyScheme  Bind | ::= *∀v.τ c ∈* EqCs ::= *μ*1 = *μ*2 *|*  ::= ↓*tc*=*tcs |* ↓*strid* =*es |* ↓*tv* =*ts |* ↓*vid* =*ts* |
| *acc* | *∈* | Accessor | ::= †*tc*=*δ |* †*strid* =*ev |* †*tv* =*α |* †*vid* =*α* |
| *e* | *∈* | Env | ::= *T| ev | bind | acc | c |* poly(*e*) *| ∃a.e | e*2;*e*1 *| ⟨e, l⟩* |

**extra metavariables**

*ct ∈* CsTerm ::= *τ | μ | e*

*σ ∈* Scheme ::= *ts | tcs | es dep ∈* Dependent ::= *⟨ct, l⟩*

*v ∈* Var ::= *α | δ | ev a ∈* Atom ::= *v | γ | l*

During analysis, a dependent form *⟨CL, l ⟩* depends on the program nodes with labels in *l* e.g. the dependent equality constraint *⟨τ*1 =*τ*2*, l ∪{l}⟩* might be generated for the labelled function application *⊆exp atexp|l*, indicating the equality constraint *τ*1 = *τ*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 collapse below, which respectively allow us to take all labels off any given term, and to union nested labels of terms. Note that dom(strip)= dom(collapse)= IntLabSynt, and ran(strip) is any piece of syntax which is not a dependent form, while ran(collapse)= IntLabSynt.

*L* (strip(*y*) if *CL* = *⟨y, l ⟩*

⎧⎪⎨collapse(*⟨y, l ∪ l j⟩*)

strip(*C* )= *L*

collapse(*C* )= if *CL* = *⟨*(*⟨y, l ⟩*)*, l ⟩*

*C* otherwise

*L*

*j*

⎪⎩*CL* otherwise

Note that we sometimes write *⟨ct, l ⟩* for *⟨ct, {l}⟩*. Given a label or a set of labels

y, we write *cty* to abbreviate *⟨ct, y⟩*

*y*

, and *ct* 1 = *ct* 2 for

*⟨ct* 1

= *ct*

2*, y⟩*.

* + 1. *Internal types (τ) and their constructors (μ)*

The ITy and ITyCon sets contain internal types and internal type constructors re- spectively. In order to maintain some simplicity for the core, only unary type con- structors are supported. [2](#_bookmark6) 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 (*→*) type.

2 Section 14.10 in [[12](#_bookmark32)] presents a solution whereby type constructors can have any arity.

* + 1. *Schemes (σ)*

There are three kinds of universally quantified schemes: type schemes (similar to those in [[9](#_bookmark29)]), type constructor schemes, and environment schemes. All schemes are subject to alpha-conversion (e.g. the schemes *∀α*1*. α*1 and *∀α*2*. α*2 are equivalent).

* + 1. *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:

* + - 1. **The empty environment/satisfied constraint**. This is represented by *≤*.
      2. **An environment variable**. We write [*e*] to abbreviate (*∃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.
      3. **A composition environment**. We use the operator ’;’ to compose environ- ments, which is associative. Note that *e*;*≤*, *≤*;*e*, and *e* are equivalent.
      4. **A binder/accessor**. A binder is of the form ↓*id* =*σ*, and an accessor is of the form †*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 ↓*vid* =*x*;†*vid* =*α* 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 ↓*vid* =*x*;*...*;†*vid* =*α* 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](#_bookmark9) describes *shadowing*, which specifies which forms would cause this.

We abbreviate ↓*vid* =*∀*∅*. ct* by ↓*vid* =*ct* and abbreviate a dependent form

*y*

*⟨*↓*vid* =*ct, y⟩* by ↓*vid* = *ct* . Similarly for accessors.

* + - 1. **An equality constraint**. A constraint where two pieces of constraint syntax are made to be equal.
      2. **Existential environment**. The form *∃x.e*, binds all free occurrences of x that occur free in *e*. We use the notation *∃⟨x*1*. ··· , xn⟩.e* to abbreviate

*∃x*1*. ··· ∃xn.e*.

* + - 1. **A polymorphic environment**. This promotes the binders in the argument to poly to be polymorphic.
      2. **Dependent form**. Label-annotated environments.
    1. *Atomic forms and Semantics of constraints/environments*

Let atoms(*CL*) be the syntactic form set belonging to Var *∪* Label and occurring in

*CL*. In addition, we define the forms as shown below.

vars(*CL*)= atoms(*CL*) *∩* Var labs(*CL*)= atoms(*CL*) *∩* Label

Note that dom(atoms) = dom(labs) = dom(vars) = IntLabSynt, ran(atoms) =

Var *∪* Label*,* ran(labs)= Label, and ran(vars)= 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 Ren *⊂* Sub.

*ren ∈* Ren = *{*ITyVar *→* ITyVar *| ren* is injective *∧* dj(dom(*ren*)*,* ran(*ren*))*}*

*sub ∈* Sub = *{f*1 *∪ f*2 *| f*1 *∈* Unifier *∧ f*2 *∈* TyConName *→* TyConName*}*

We also define our unifier set as a directed acyclic graph *U ∈* Unifier = *{*V*,* E*}*

where V = ITyVar *∪* ITy *∪* ITyCon and E = P(V *×* V) which specify directional edges.

Note that for each *Vx ∈* V, the edge *Vx '→ V j*

*x*

occurs at most once, and so we

also consider *U* as a function. When using an application *U* (*Vx*), vertex *V j* will be

*x*

returned where a path from *Vx* to *V j* exists (if it does not, *Vx* = *V j*) and *V j '→ V jj*

*x x x* *x*

does not exist e.g., where *U* = *{{V*1*, V*2*, V*3*, V*4*, V*5*, V*6*}, {V*1 *'→ V*3*, V*3 *'→ V*2*, V*4 *'→*

*V*5*, V*2 *'→ V*6*}}, U* (*V*1) = *V*6. During application, if *U* (*v* ) = *CLx* and vars(*CL*) */*= *{}*, then for each *v j ∈* vars(*CL*) if *U* (*vj*) */*= *v j* then it is replaced by *U* (*vj*).

Environments contain information on external identifiers. We also need infor- mation on internal type variables which we get through our unifiers. Renamings are used to instantiate type schemes. The Unifier set consists of unifiers generated by our constraint solver (see Section [3.3](#_bookmark13)). Substitution is defined in Figure [3](#_bookmark8), 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 con- straint terms)

*a*[*sub*] = *x,* if *sub*(*a*)= *x*

*a,* otherwise

(*τ μ*)[*sub*] = *τ* [*sub*] *μ*[*sub*]

(*τ*1 ⭢ *τ*2)[*sub*] = *τ*1[*sub*] ⭢ *τ*2[*sub*]

*ctl* [*sub*] = *ct* [*sub*]*l*

(*ct* 1 = *ct* 2)[*sub*] = (*ct* 1[*sub*]= *ct* 2[*sub*]) (*e*1;*e*2)[*sub*] = *e*1[*sub*];*e*2[*sub*]

(*6v. ct* )[*sub*] = *6v. ct* [*sub*] s.t. dj(*v,* atoms(*sub*))

(*Ea.e*)[*sub*] = *Ea.e*[*sub*] s.t. dj(*{a},* atoms(*sub*))

,

(†*id* =*v* )[*sub*] = (†*id* =*v* [*sub*])*,* if *v* [*sub*] *∈* Var

undefined*,* otherwise

(↓*id* =*σ*)[*sub*] = (↓*id* =*σ*[*sub*])

poly(*e*)[*sub*] = poly(*e*[*sub*])

*x*[*sub*] = *x*, otherwise

* + 1. *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 ∈/* dom(*U* ), we say that *ev shadows bind* 1 because *ev* could potentially be bound to an environment which rebinds *bind* 1. We define shadowsAll by:

* shadowsAll(*⟨U, e⟩*) *⇐⇒*

⎧⎪ (*e* = *ev ∧* (shadowsAll(*⟨U, U* (*ev* )*⟩*) *∨ ev /∈* dom(*U* )))

⎪⎨

*∨* (*e* = (*e*1;*e*2) *∧* (shadowsAll(*⟨U, e*1*⟩*) *∨* shadowsAll(*⟨U, e*2*⟩*)))

*∨* (*e* = *⟨ej, l ⟩ ∧* shadowsAll(*⟨U, ej⟩*))

⎪⎪⎩

*∨* (*e* = *∃a.ej ∧* shadowsAll(*⟨U, ej⟩*) *∧ a /∈* dom(*U* ))

* shadowsAll(*e*) *⇐⇒* shadowsAll(*⟨∅, e⟩*)

Note that dom(shadowsAll)= tuple(*U × e*) and ran(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 *U* during constraint solving.

,

(↓*id* =*σ*)(*id* ) = *σ*

(*el* )(*id* ) = *6v. ctl* , if (*e*)(*id* )= *6v. ct*

⎧⎪ (*e* )(*id* )*,* if (*e* )(*id* ) is defined

2

2

(*ev* )(*id* ) =

(*e*)(*id* )*,* if *C* (*ev* )= *e* undefined*,* otherwise

(*e* ;*e* )(*id* ) = ⎪⎨ undefined*,* if (*e*2)(*id* ) is undefined

(*⟨e⟩*)(*id* ) = *e*(*id* )

1 2

⎪⎪⎩

and shadowsAll(*⟨C, e*2*⟩*) (*e*1)(*id* )*,* otherwise

(*⟨e*1*⟩*@*⟨e*2*⟩*)(*id* ) = (*e*1; *e*2)(*id* )

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

*∀v. ct, sub* ***−***in*−*st*−*a*−*nc*−→*e

*ct* [*sub*] if dom(*sub*)= *v σ* ***−****→e*

*ct* if *∃sub.σ, sub* ***−***in*−*st*−*a*−*nc*−→*e

*e, ct*

* 1. *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 *con- straint generator* is defined in Figure [4](#_bookmark12). Note that there are other types of con- straints during the solving process.

Let cstgen*j*(*PL, v* ) be a function with two arguments, the first a labelled piece of user program *PL*, and the second a set of free variables occurring in *PL*. Each

of the constraint generation rules is written either as *PL*) = *e* (which abbreviates cstgen*j*(*PL, {}*) = *e*) or as *PL, v* ) = *e* (which abbreviates cstgen*j*(*PL, {v}*) = *e*). Let cstgen(*PL*)= cstgen*j*(*PL, {}*)

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 [[11](#_bookmark31)].

To allow us to slice out environments correctly, we annotate environment vari- ables 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:

*l*2 *l l*

[*E⟨α*1*, α*2*, ev⟩.*(*ev* = ↓y = *α*1); *ev* ; *exp, α*2); (*α* = *α*1 *→ α*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:

*l*3

[*Eα*3*.* *dec*); *exp, α*3); (*α*2 = *α*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 con- straints for the val rec statement and (G6) to handle the name of the function (f) to give:

*l*8 *l*7 *l*7

*E⟨α*4*, α*5*, ev* 2*⟩.*(*ev* 2 = poly(↓f = *α*4; *exp, α*5); (*α*4 = *α*5))); *ev* 2

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

*l*2 *l*

[*E⟨α*1*, α*2*, ev⟩.*(*ev* = ↓y = *α*1); *ev* ; [*Eα*3*.E⟨α*4*, α*5*, ev* 2*⟩.*(*ev* 2 =

*l*8 *l*9

*l*10

*l*12

*l*13

*l*11

*l*10

poly(↓f = *α*4; [*E⟨α*6*, α*7*, ev* 3*⟩.*(*ev* 3 = ↓x = *α*6); *ev* 3 ; *E⟨α*8*, α*9*⟩.*†x = *α*8; †y = *α*9; (*α*8 = *α*9 *→ α*7); *α*5 =

*l*7 *l*7

*′ ′′*

*l*4 *′*

*l*5 *′′*

*′ l*6 *′′*

*l*3 *l*

*α*6 *→ α*7]; (*α*4 = *α*5))); *ev* 2 ; *E⟨α ,α ⟩.*†f = *α* ; †y = *α* ; (*α* = *α*

*→ α*2); (*α*2 = *α*3)]; (*α* = *α*1 *→ α*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. *2*

**Fig. 4** Constraint generator (ExtLabSynt *→* Env)

**Expressions (***exp***)**

(G1 *l l*

(G2 *l*

*l*

2 2 2

) *vid* e*, α*) = †*vid* = *α*

) let *dec* in *exp* end*, α*) = [*Eα .* *dec*); *exp,α* );(*α* = *α* )]

(G3

*l* 1 2 1

*l*

2 1 2

) *[exp atexp| , α*) = *E⟨α ,α ⟩.* *exp,α* ); *atexp,α* );(*α*

= *α*

⭢ *α*)

(G4 *l* 1 2

1 *l* 2 *l* 1 2

) fn *pat ⇒ exp, α*) = [*E⟨α ,α , ev⟩.*(*ev* = *pat ,α* ));*ev* ; *exp,α* );(*α* = *α*

**Labelled datatype constructors (***ldcon***)**

⭢ *α* )]

(G5 *l l*

) *dcon , α*) = †*dcon* = *α*

**Patterns (***pat* **)**

(G6) *vvarl , α*) = ↓*vvar l α*

p =

p =

(G7) *dconl , α*) = †*dcon l α*

(G8

*l* 1 2

*l*

1 2 1 2

) *[ldcon atpat| , α*) = *E⟨α ,α ⟩.* *ldcon,α* ); *atpat,α* );(*α* = *α*

**Labelled type constructors (***ltc***)**

⭢ *α*)

(G9 *l l*

) *tc , δ*) = †*tc* = *δ*

**Types (***ty* **)**

(G10 *l l* (G11 *l ′ ′ l*

) *tv , α*) = †*tv* = *α*

) *[ty ltc| ,α* ) = *E⟨α, δ⟩.* *ty, α*); *ltc, δ*);(*α* = *α δ*)

(G12 *l*

1 2 1

*l*

2 1 2

) *ty* 1 *→ ty* 2*, α*) = *E⟨α ,α ⟩.* *ty* 1*,α* ); *ty* 2*,α* );(*α* = *α*

**Datatype names (***dn***)**

⭢ *α* )

(G13 *l ′*

*′ l l l*

) *[tv tc| ,α* ) = *E⟨α, γ⟩.*(*α* = *αγ*);(↓*tc* = *γ*);(↓*tv* = *α*)

**Constructor bindings (***cb***)**

(G14 *l*

*l* (G16 *l*

*′* 1 1

*′ l* 1 *l ′*

) *dcon*c*, α*) = ↓*dcon* = *α*

**Declarations (***dec***)**

) *dcon* of

*ty, α*) = *E⟨α ,α ⟩.* *ty,α* );(*α* = *α*

⭢ *α*);(↓*dcon* = *α* )

(G17 *l* 1 2

) val rec *pat* = *exp*) = *E⟨α ,α , ev⟩.*(*ev* = poly( *pat ,α* ); *exp,α* );(*α*

*l l*

1 2 1 2

= *α* )));*ev*

(G18

*l l l*

1 2 1 2 1 2

) datatype *dn* = *cb*) = *E⟨α ,α , ev⟩.*(*ev* = ((*α*

= *α* ); *dn,α* );poly( *cb,α* ))));*ev*

*l*

(G19

) open*l*

*strid* ) = *Eev.*(†*strid* = *ev* );*evl*

**Structure declarations (***strdec***)**

(G20

*l ′ ′*

*l ′l*

) structure *strid* = *strexp*)=*E⟨ev, ev ⟩.*[ *strexp, ev* )];(*ev*

**Structure expressions (***strexp***)**

= (↓*strid* = *ev* ));*ev*

(G21 *l l*

) *strid , ev* ) = †*strid* = *ev*

(G22) struct*l strdec*1 *··· strdecn*

*′ l ′ ′*

1 *n*

end*, ev* )=*Eev .*(*ev* = *ev* );(*ev*

= ( *strdec* ); *···* ; *strdec*

)))

**Lemma 3.2 (Termination of Constraint Generation Algorithm)** *The con- straint generator shown in Figure* [*4*](#_bookmark12) *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](#_bookmark12) are (G1)*,* (G5)*,* (G6)*,* (G7) (G9), (G10), (G13), (G14), (G19), and

(G21). For a constraint generation run cstgen*j*(*PL, v* ) either *PL* will be atomic in nature or it will not. If not, we recurse with cstgen*j*(*PLj, v j*), on some *PLj* inside *PL*, such that *PLj* is strictly smaller than *PL*. Rules which recurse with strictly smaller parts of external syntax are rules (G2) (let syntax 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 *PL*, we halt and return our generated *e* form. *2*

* 1. *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](#_bookmark16)) are given in Figure [5](#_bookmark14). The symbol is defined in Section [3.3.2](#_bookmark15), and is

*st*

used to keep track of future environments that we have yet to solve.

**Fig. 5** Extra syntactic forms for constraint solving

*m ∈* Monomorphic ::= *⟨α, l⟩ er ∈* Error ::= *⟨ek, l⟩*

*ek ∈* ErrKind ::= clash(*μ*1*, μ*2) *|* circularity

::= slv( *e , l , m, st , e* ) *|* succ *|* err(*er* )

*state ∈* State

*—→ * *—→ ′*

Constraint solving starts by slv(*⟨≤⟩,* ∅*,* ∅*, ⟨⟩, e*), and ends either by succ (for success), or in the state err(*er* ) where *er* is either a type constructor clash or a circularity error. The relations isErr and solvable are defined below, where *→* indicates a constraint solving step.

*e* is*→*Err *er ⇔* slv(*≤, l ,* ∅*,* ∅*, e*) *→∗* err(*er* ) solvable(*e*) *⇔* slv(*≤, l ,* ∅*,* ∅*, e*) *→∗* succ solvable(*strdec*) *⇔ ∃e.strdec → e ∧* solvable(*e*)

* + 1. *Uniﬁers*

When constraint solving starts, the set of unifiers *U* is initialised to the empty set (*U* = ∅). During constraint solving, nothing is ever subtracted from *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](#_bookmark16)).

* + 1. *The environment stack*

The fourth argument to the slv function of the constraint solver in Figure [6](#_bookmark16), de-

*−→*noted as is used as a stack of environments or other tasks which are still to be

*st* ,

solved/completed. Below, we introduce some metavariables needed to define the stack:

**Fig. 6** Constraint solver (1 of 2) : State*\{*succ*,* err(Error)*}→ state*

equality constraint reversing

(R *—→ * *—→ ′*

*—→ * *—→ ′*

Var *′*

) slv( *e , l , m, st , ct* = *ct* ) *→* slv( *e , l , m, st , ct*

equality simplification

) slv( *e , l , m, st , ct* = *ct* ) *→* isSucc( *e , m, st* )

= *ct* ), if *s* =

*∪* Dependent *∧ ct*

*∈ s ∧ ct /∈ s*

(S1

) slv( *e , l , m, st , ct*

*—→ * *—→*

*—→ * *—→*

(S2

*—→ * *—→ l' ′*

*—→ ′ * *—→ ′*

(S3

= *ct* ) *→* slv( *e , l ∪ l , m, st , ct* = *ct* )

*—→ * *—→*

1 1 2 2

) slv( *e , l , m, st ,τ*

*μ*

= *τ*

*μ* ) *→* slv( *e , l , m, st ,* (*μ*

*—→ * *—→*

1

2 1 2

(S4

= *μ* );(*τ*

= *τ* ))

*—→ * *—→*

1 2 3 4

) slv( *e , l , m, st ,τ* ⭢ *τ*

= *τ*

⭢ *τ* ) *→* slv( *e , l , m, st ,* (*τ*

*—→ * *—→*

1

3 2 4

(S5

= *τ* );(*τ*

= *τ* ))

*—→ * *—→*

*—→ * *—→*

) slv( *e , l , m, st , τ*1 = *τ*2) *→* slv( *e , l , m, st ,μ* = arr), if *{τ*1*, τ*2*}* = *{τ μ, τ*3 ⭢ *τ*4*}*

(S6

*—→ * *—→*

1 2

1 2 1 2 *′ ′*

) slv( *e , l , m, st ,μ* = *μ* ) *→* err(*⟨*clash(*μ ,μ* )*, l⟩*), if*{μ ,μ }∈ {{γ, γ }, {γ,* arr*}} ∧γ /*= *γ*

unifier access

Rules (U1) through (U4) have also the common side condition *v /*= *ct ∧ y* = *C* (*xl* ) *∧ v ∈/* dom(*C* )

) slv( *e , l , m, st , v* = *ct* ) *→* err(*⟨*circularity*,*

(U1

) slv( *e , l , m, st , v* = *ct* ) *→* isSucc( *e , m, st* ), if *v ∈/*

*—→ * *—→*

deps(*y*)*⟩*), if *v ∈* vars(*y*)*\*Env *∧* strip(*y*) */*= *v*

(U2

) slv( *e , l , m, st , v* = *ct* ) *→* isSucc( *e , m, st* ), if *v ∈/*

*—→ * *—→*

*—→ * *—→*

Env*∧* strip(*y*) = *v*

(U3

) slv( *e , l , m, st , v* = *ct* ) *→* slv( *e* @*⟨T⟩, l , m, st* @ *st , ct* ), if *v ∈*

*—→ * *—→*

*—→ * *—→*

vars(*y*) *∪* Env *∧ C* = *C ⊕ {v '→ y}*

(U4

*—→ * *—→*

*—→ * *—→*

*—→′*

Env

*—→′*

(U6

*∧ st*

= *⟨⟨*new*, l , m, v⟩⟩*

*—→ * *—→*

*—→ * *—→*

) slv( *e , l , m, st , v* = *ct* ) *→* slv( *e , l , m, st ,z* = *ct* ), if *C* (*v*)= *z*

composition environments

(C1

*—→ * *—→*

1 2

) slv( *e , l , m, st , e* ;*e* ) *→* slv( *e , l , m, st* @*⟨⟨*new*, l ,* new*, e ⟩⟩, e* )

*—→ * *—→*

2 1

**Fig. 7** Constraint solver (2 of 2)

binders/empty/dependent/variables

(B *—→ * *—→*

*—→ l*

*l —→*

) slv( *e , l , m, st ,* ↓*vid* =*α*) *→* isSucc( *e* ; ↓*vid* = *α, m ∪ {α }, st* )

(B2

*—→ * *—→*

*—→ l * *—→*

) slv( *e , l , m, st , bind*) *→* isSucc( *e* ; *bind , m, st* ), if *bind /*= ↓*vid* =*α*

*∈/*

(X *—→*

) slv( *e , l , m, st , Ea.e* ) *→* slv( *e , l ∪ l , m, st , e* [*{a '→ a }*]), if *a*

*—→ ′*

*—→ ′*

*—→ ′ ′*

*′* atoms(*⟨C, e′⟩*)

(E *—→ * *—→*

) slv( *e , l , m, st , T*) *→* isSucc( *e , m, st* )

*—→ * *—→*

(D *—→*

) slv( *e , l , m, st , e*

*—→ ′l'*

*—→ ′*

*—→ ′*

(V *—→ * *—→*

) *→* slv( *e , l ∪ l , m, st , e* )

*—→ d*¯  *—→*

) slv( *e , l , m, st , ev* ) *→* isSucc( *e* ; *ev*

accessors

) slv( *e , l , m, st ,* †*id* =*v* ) *→* slv( *e , l ∪ l , m, st , v* = *τ* ),

*, m, st* )

(A1

*—→ * *—→*

*—→ ′ * *—→*

*—→* instance *′ —→*

if *e* (*id* )*, ren —————→ τ, l ∧* dj(vars(*⟨ e , v⟩*)*,* ran(*ren*))

(A3

*—→ * *—→*

*—→ * *—→ —→*

) slv( *e , l , m, st ,* †*id* =*v* ) *→* isSucc( *e , m, st* ), if

polymorphic environments

*e* (*id* ) undefined

*—→ * *—→ l' —→ * *—→*

(P1) slv( *e , l , m, st ,* poly(↓*vid* = *α*)) *→* isSucc( *e* ; *σ, m, st* ),

if *α* = ityvars(*C* (*α*))*\* *{*ityvars(*C* (*x*)) *| x ∈ m}*

*′′ ′*

*∧ l* = *l*

*∪* deps(vars(*C* (*α*)) *a {C* (*x* ) *| x ∈ m}*)

*′′*

*∧ σ* = ↓*vid*=*⟨6α. C* (*α*)*, l ⟩*

(P2

*—→ * *—→*

*′ —→*

*—→ —→ * *′*

) slv( *e , l , m, st ,* poly(*bind*;*e* )) *→* slv( *e , l , m, st* @*⟨⟨ e , l , m,* poly(*bind*)*⟩⟩, bind*; *e* )

(P3

*—→ * *—→ l —→ * *—→*

) slv( *e , l , m, st ,* poly(*e*1)) *→* slv( *e* @*⟨T⟩, l , m, st* @*⟨⟨*new*, l ,* new*, l⟩⟩,* poly(*e*1))

(P4

*—→ * *—→*

*—→ * *—→*

) slv( *e , l , m, st ,* poly(*e*1;*e*2)) *→* slv( *e , l , m, st* @*⟨⟨*new*, l ,* new*,* poly(*e*2)*⟩⟩,* poly(*e*1)), if *∧e*1 */*= *bind*

(P5

*—→ * *—→ ′*

*—→ * *—→*

*—→ ′*

*′ ′ ′′*

) slv( *e , l , m, st ,* poly(*e* )) *→* slv( *e , l , m, st* @*⟨⟨ e* ; *e , l , m,* done*⟩⟩, e* ), if *e /*= *Ea.e*

) slv( *e , l , m, st ,* poly(*Ea.e* )) *→* slv( *e , l , m, st ,* poly(*e* [*{a '→ a }*])) if *a*

*∈/*

(P6

*—→ * *—→*

*′ —→ * *—→ ′ ′*

*′* atoms(*⟨C, e′⟩*)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *stackEv* | *∈* | StackEv | = | *e |* new |
| *stackMono* | *∈* | StackMono | = | *m |* new |
| *stackAction* | *∈* | StackAction | = | *e | v | l |* done |

This stack is a tuple where each element is itself a tuple which has four compo- nents: *stackEv* , *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. *l* is a set of de- pendencies. *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*j* in Figure [8](#_bookmark17), 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 *st*

stack, (the constraint solver terminates in the success state if it is empty). The definition of isSucc is given in Figure [8](#_bookmark17), 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:

slv(*⟨T⟩,* ∅*,* ∅*, ⟨⟩, e*1) where *e*1 is the environment returned from the initial constraint generator, shown below.

*l*2 *l*

[*E⟨α*1*, α*2*, ev⟩.*(*ev* = ↓y = *α*1); *ev* ; [*Eα*3*.E⟨α*4*, α*5*, ev* 2*⟩.*(*ev* 2 =

*l*8 *l*9

*l*10

*l*12

*l*13

*l*11

*l*10

poly(↓f = *α*4; [*E⟨α*6*, α*7*, ev* 3*⟩.*(*ev* 3 = ↓x = *α*6); *ev* 3 ; *E⟨α*8*, α*9*⟩.*†x = *α*8; †y = *α*9; (*α*8 = *α*9 *→ α*7); *α*5 =

*l*7 *l*7

*′ ′′*

*l*4 *′*

*l*5 *′′*

*′ l*6 *′′*

*l*3 *l*

*α*6 *→ α*7]; (*α*4 = *α*5))); *ev* 2 ; *E⟨α ,α ⟩.*†f = *α* ; †y = *α* ; (*α* = *α*

*→ α*2); (*α*2 = *α*3)]; (*α* = *α*1 *→ α*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 *evj* 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 *evjl* expression, and we are then in the state shown below.

slv(*⟨ev′{l,l*2 *}*

*⟩, {l, l*2*}, {α*0*}, ⟨⟩, e*3) where the set of unifiers *C* is *{ev′*

*l*

*'→* ↓y = *α*0*}* and *e*3 is

[*Eα*3*.E⟨α*4*, α*5*, ev* 2*⟩.*(*ev* 2 =

*l*8

*l*9 *l*10

*l*12

*l*13

*l*11

*l*10

poly(↓f = *α*4; [*E⟨α*6*, α*7*, ev* 3*⟩.*(*ev* 3 = ↓x = *α*6); *ev* 3 ; *E⟨α*8*, α*9*⟩.*†x = *α*8; †y = *α*9; (*α*8 = *α*9 *→ α*7); *α*5 =

*l*7 *l*7

*′ ′′*

*l*4 *′*

*l*5 *′′*

*′ l*6 *′′*

*l*3 *l*

*α*6 *→ α*7]; (*α*4 = *α*5))); *ev* 2 ; *E⟨α ,α ⟩.*†f = *α* ; †y = *α* ; (*α* = *α*

*→ α*1); (*α*1 = *α*3)]; (*α* = *α*0 *→ α*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 [[11](#_bookmark31)]. 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](#_bookmark19), 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 (isSucc : tuple(Env) *×* Monomorphic *×* tuple(Env) *→*

*state\*err(Error))

isSucc(*—→e , m, ⟨⟩*) *→* succ

*—→ * *—→*

*′ —→ * *—→*

isSucc( *e , m, st* @*⟨⟨*new*, l ,* new*, x⟩⟩*) *→* isSucc ( *e , l , m, st , x*)

*—→ * *—→ —→ ′ —→ * *—→*

isSucc( *e , m, st* @*⟨⟨ e* 1*, l ,* new*, x⟩⟩*) *→* isSucc ( *e* 1*, l , m, st , x*)

*—→ * *—→ ′*

*′ —→*

*′ —→*

isSucc( *e , m, st* @*⟨⟨*new*, l , m , x⟩⟩*) *→* isSucc ( *e , l , m , st , x*)

*—→ * *—→ —→ ′*

*′ —→*

*′ —→*

isSucc( *e , m, st* @*⟨⟨ e* 1*, l , m , x⟩⟩*) *→* isSucc ( *e* 1*, l , m , st , x*)

*′ —→ * *—→*

*—→ * *—→*

isSucc ( *e* @*⟨e*1*, e*2*⟩, l , m, st , v* ) *→* isSucc( *e* @*⟨e*1;*e*2*⟩, m, st* )*,* if *C* = *C⊕{v '→ e*2*}*

*′ —→*

*—→ —→*

*l * *—→*

isSucc ( *e* @*⟨e*1*, e*2*⟩, l , m, st , l* ) *→* isSucc( *e* @*⟨e*1;*e*2*⟩, m, st* )

*′ —→ * *—→*

*—→ * *—→*

isSucc ( *e , l , m, st ,* done) *→* isSucc( *e , m, st* )

*′ —→*

*—→ ′*

*—→ * *—→ ′*

isSucc ( *e , l , m, st , e* ) *→* slv( *e , l , m, st , e* )

We further analyze some interesting constraint solving rules. Rule (C1) demon- strates 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 han- dled 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* slv( *e , l , m, st , e*) *→* succ *or* slv( *e , l , m, st , e*) *→*

err(*er* )*.*

**Proof.** By inspection of the rules. We only summarize the proof for the important rules ([[11](#_bookmark31)] contains a more thorough treatment). (R) flips constraints and flipped constraints can never be re-flipped. (S1) Throws away argument/adds to environ- ment 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 err(*er* ), otherwise, the constraint solver will terminate in the succ state shown in Figure [8](#_bookmark17). *2*

* 1. *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, *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 *l* . We do this process for *all* labels in *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 [[11](#_bookmark31)]. We then present these regions to the user as shown in Figure [9](#_bookmark19). Note that an standard SML compiler, such as PolyML [[1](#_bookmark21)], 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

* + 1. fun average weight list =
    2. let fun iterator (x ,( sum , length )) = ( sum + weight x ,
    3. length + 1)
    4. val ( sum , length ) = foldl iterator (0 ,0) list
    5. in sum div length
    6. end

7

1. fun find best weight lists =
2. let val average1 = average weight
3. fun iterator ( list ,( best , max )) =
4. let val avg\_list = average 1 list
5. in if avg\_list > max
6. then ( list , avg\_list )
7. else ( best , max )
8. end
9. val ( best , \_) = foldl iterator ( nil ,0) lists
10. in best
11. end

19

20 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 [[11](#_bookmark31)].

# Conclusion

Automatically finding type errors in programming languages is a difficult task. Suc- cessful attempts need to address constraint systems but these have only been built

for toy-like languages. Moreover, existing proposals to solve poor type error re- porting 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:

1. Takes an SML program and returns exactly the erroneous parts of the program;
2. Does not report any portion of internally modified syntax, as can be presented by the available compilers for the language;
3. Will display *all* parts of an error to the user;
4. Is completely unbiased in its analysis as compilers are;
5. 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 con- straints 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 ter- minating 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 [[4](#_bookmark24)] 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 rep- resentation 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

1. *PolyML compiler*, [www.polyml.org/](http://www.polyml.org/) (web), last accessed 20th January 2014.
2. Braßel, B., *Typehope: There is hope for your type errors*, in: *In 16th International Workshop on Implementation and Application of Functional Languages (IFL’04)*, 2004.
3. Damas, L. and R. Milner, *Principal type-schemes for functional programs*, in: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’82 (1982), pp. 207–212.

URL <http://doi.acm.org/10.1145/582153.582176>

1. Di Cosmo, R., F. Pottier and D. R´emy, *Subtyping recursive types modulo associative commutative products*, in: *Seventh International Conference on Typed Lambda Calculi and Applications (TLCA’05)*, Lecture Notes in Computer Science **3461** (2005), pp. 179–193.

URL <http://gallium.inria.fr/~fpottier/publis/dicosmo-pottier-remy-tlca05.ps.gz>

1. Hage, J. and B. Heeren, *Strategies for solving constraints in type and effect systems*, Electron. Notes Theor. Comput. Sci. **236** (2009), pp. 163–183.

URL <http://dx.doi.org/10.1016/j.entcs.2009.03.021>

1. Lerner, B. and Grossman, *Seminal: searching for ML type-error messages*, in: *Proceedings of the 2006 workshop on ML*, ML ’06 (2006), pp. 63–73.

URL <http://doi.acm.org/10.1145/1159876.1159887>

1. Mcadam, B. J., *On the unification of substitutions in type inference*, in: *Implementation of Functional Languages (IFL ’98)* (1998), pp. 139–154.
2. Mu¨ller, M., *A constraint-based recast of ML-polymorphism (extended abstract)*, Technical report, 8th international workshop on unification (1994).
3. Neubauer, M. and P. Thiemann, *Discriminative sum types locate the source of type errors*, in: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP ’03 (2003), pp. 15–26.

URL <http://doi.acm.org/10.1145/944705.944708>

1. O. Lee, K. Y., *Proofs about a folklore let-polymorphic type inference algorithm*, ACM Transactions on Programming Languages and Systems (TOPLAS) **20** (1998), pp. 707–723.
2. Pirie, J., *New developments to skalpel: A type error slicing method for explaining errors in type and* *effect systems* (2014), ph. D thesis. Available at [http://www.macs.hw.ac.uk/~jp95/jpirie-thesis.](http://www.macs.hw.ac.uk/~jp95/jpirie-thesis.pdf) [pdf](http://www.macs.hw.ac.uk/~jp95/jpirie-thesis.pdf).
3. Rahli, V., *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> (2010), ph.D. Thesis. Last accessed Monday 16th July 2012.
4. Schilling, T., *Constraint-free type error slicing*, in: *Trends in Functional Programming*, Lecture Notes in Computer Science **7193**, Springer Berlin Heidelberg, 2012 pp. 1–16.

URL <http://dx.doi.org/10.1007/978-3-642-32037-8_1>

1. Zhang, D. and A. C. Myers, *Toward general diagnosis of static errors*, SIGPLAN Not. **49** (2014),

pp. 569–581.

URL <http://doi.acm.org/10.1145/2578855.2535870>