Technical report HW-MACS-TR-0079

A constraint system for a SML type error slicer

Vincent Rahli J. B. Wells Fairouz Kamareddine

ULTRA group, Heriot-Watt University

Abstract

Existing compilers for many languages have confusing type error messages. *Type error slicing* (TES) helps the programmer by isolating the part of a program contributing to a type error, but unfortunately TES was initially done for a tiny toy language. Extending TES to a full programming language is extremely challenging, and for SML we needed a number of innovations and generalisations. Some issues would be faced for any language, and some are SML-specific but representative of the complexity of language-specific issues likely to be faced for other languages. We solve both kinds of issues and present a simple, general constraint system for providing type error slices for ill-typed programs. Our constraint system elegantly and efficiently handles features like the intricate *open* SML feature. We show how the simple clarity of type error slices can demystify language features known to confuse users.

We also provide in an appendix a case study on how to use our TES to help modifying user data types, and extend the core language presented in the main body of this report to handle more of the implementation of our system. These extensions allow handling local declarations, type declarations and some uses of signatures.

1. Introduction

Higher-order type inference. SML is a higher-order function-oriented imperative programming language. SML (and similar languages like OCaml, Haskell, etc.) has polymorphic types allowing considerable flexibility, and almost fully automatic type inference, which frees the programmer from writing explicit types. We say "almost fully" because some explicit types are required in SML, e.g., as part of datatype definitions, module types, and type annotations sometimes needed in special circumstances. Milner's W algorithm [8] is the original type-checking algorithm of the functional core of ML (variables, abstractions, applications and polymorphic let-expressions). W implementations generally give error messages relative to the syntax tree node the algorithm was visiting when unification failed, and this is often unsatisfactory.

Moving the error spot. Following W, other algorithms try to get better locations by arranging that untypability will be discovered when visiting a different syntax tree node. For example, Lee and Yi proved that the folklore algorithm M [18] finds errors "earlier" than W and claimed that their combination "can generate strictly more informative type-error messages than either of the two algorithms alone can". Similar claims are made for W′ [20] and UAE [30]. McAdam observes that W suffers a left-to-right bias and tries to eliminate it using "unification of substitutions". Yang claims that UAE's primary advantage is that it also eliminates this bias. However, all the algorithms mentioned above retain a left-to-right bias in handling of let-bindings and they all blame only one syntax tree node for each type error when in fact a node set is at fault.

When only one node is reported as the error site, this node is often far away from the actual programming error. The situation is made worse because which node is blamed depends on internal implementation details, i.e., the tree node traversal order and which constraints are accumulated and solved at different times in the traversal. The confusion is worsened because these algorithms usually exhibit in error messages (1) an internal representation of the program subtree at the blamed location which often has been transformed substantially from what the programmer wrote, and (2) details of inferred types which were not written by the programmer and which are anyway erroneous and therefore confusing.

Other improved error reporting systems. Attempting to solve this problem, constraint-based type inference algorithms [22, 23, 24] separate the two following processes: the generation of type constraints for a given term and their unification. Many works are based on this idea to improve error reporting (a probably incomplete list includes [15, 10, 11, 9, 25, 26, 27]). Independently from this separation, there exist many different approaches toward improving error reporting [32]: error explanation systems [2, 31] and error reporting systems [28]. Another approach to type error reporting is the one of Lerner et al. [19] or Hage and Heeren [12] suggesting changes to perform in the untypable code to solve type errors.

Type error slicing. Haack and Wells [11] noted that "*Identifying only one node or subtree of the program as the error location makes it difficult for programmers to understand type errors. To choose the correct place to fix a type error, the programmer must find all of the other program points that participate in the error.*" They locate type errors at *program slices* which include all parts of an untypable piece of code where changes can be made to fix the error and exclude the parts where changes cannot fix the error.

Haack and Wells gave their method of *type error slicing* (TES) for a tiny subset of SML barely larger than the λ-calculus. The TES of Haack and Wells generates constraints for SML code, enumerates minimal unsatisfiable subsets of the constraint set, and then computes type error slices. Generation and solving of constraints are not interleaved. To identify program slices responsible for type errors, each constraint is labeled by the location responsible for its generation. Error slices are portions of a program where all subterms with no responsibility for the error are elided (e.g., replaced by dots). Slices can also be shown by highlighting the source code. These slices are intended to contain all and only the information needed to solve the type errors.

The method of Haack and Wells meets the following criteria [32] for good type error reports: it reports only errors for ill-typed code (*correct*), it reports no more than the conflicting portions of code (*precise*), it reports short messages (*succinct*), it does not report internal information such as internal types generated during type inference (*a-mechanical*), it reports only code written by the programmer which has not been transformed as happens with existing SML implementations (*source-based*), it does not privilege any location over the others (*unbiased*), and it reports all the conflicting portions of code (*comprehensive*).

Slicing for a full language. We aim toward a TES method that (1) covers the full SML language, (2) is practical on real programs, and (3) has a simple and general design. As would happen for any programming language, we encountered challenges.

One challenge was avoiding a combinatorial explosion in the number of constraints. Naive constraint generation could duplicate the environment of a polymorphic declaration such as in SML's let-expressions. Our solution is related to a constraint system by Pottier and Rémy [24, 23] although it has evolved significantly beyond that, especially to handle the challenge of SML's *open* declaration. The most interesting constraints in their constraint system are "let-constraints" (generated for let-bindings). They are to some extent inspired by constraint-based type systems such as the one by Odersky, Sulzmann and Wehr [22] (and mainly by the type schemes used in that system). As explained by Pottier such constraints "allow building a constraint of linear size" [23]. We have generalised the structure of these constraints to deal with sequences of diverse (both polymorphic and monomorphic) identifier declarations (for values, types, structures, and signatures), and we developed a compatible slicing machinery. Our new constraint system replaced the type duplicating approach of Haack and Wells and gained scalability at the cost of losing compositional analysis.

Another major challenge was SML's *open* feature which splices the declarations of a structure into the current environment. This feature has been criticized in the literature [13, 3, 4, 1]. Harper writes [13]: "*it is hard to control its behaviour, since it incorporates the entire body of a structure, and hence may inadvertently shadow identifiers that happen to be also used in the structure*". Blume writes [3]: "*Programs are not only read by analysis tools; human read them as well. A language construct like open that serves to confuse the analysis tool is also likely to confuse the human reader*". Our TES provides useful type error reports when *open* is involved, clarifying otherwise obscure type errors, and enhancing the usability of *open*. To handle errors involving *open*, we designed a simple and general machinery of "constrained environments" (definition in Sec. 4.2, example in Sec. 2.2) which goes well beyond what is supported by Pottier and Rémy's let-constraints.

Another challenge is SML's value identifier statuses. In SML, a value identifier can be a value variable (the only status considered by Haack and Wells), a datatype constructor, or an exception constructor (omitted in this paper's formalism). For example, if identifier `c` has value variable status in the context, `fn c => (c 1, c())` has a unique minimal error which is that `c` has a monomorphic type but is applied to two expressions with different types: `int` and `unit`. However, this error would not exist if the code was preceded by, e.g., `datatype t = c` because the fn-binding would not bind `c`, but instead there would be a minimal error that `c` is declared as a nullary datatype constructor and is applied to an argument in `c 1`. To compute correct type error slices, we annotate constraints by context dependencies on identifier statuses. For the fn-binding presented above we generate during unification constraints relating the occurrences of `c` annotated by the dependency that `c` is a value variable and not a datatype constructor. These constraints are not generated if a context confirms that `c` must be a datatype constructor. The constraints but not the context dependency are generated if a context confirms that `c` cannot be a datatype constructor. When handling incomplete programs, we report conditional errors (warnings) that assume a sensible default truth status for the dependencies.

Later sections detail solving these and other challenges.

2. Key motivating examples

This section gives motivating examples of TES. Type error slices are highlighted with very light grey. Dark grey highlights error *end points* (e.g., the sources of conflicting types constrained to be equal). A color version has been made available.

2.1 Datatypes, pattern matching and type functions. Fig. 1 shows how TES is important for intricate errors. The code declares the datatype `t` and the function `trans` to deal with user defined

Figure 1 Datatypes, pattern matching and type functions

```
datatype ('a, 'b, 'c) t = Red    of 'a * 'b * 'c
                        | Blue   of 'a * 'b * 'c
                        | Pink   of 'a * 'b * 'c
                        | Green  of 'a * 'b * 'b   ①
                        | Yellow of 'a * 'b * 'c
                        | Orange of 'a * 'b * 'c
fun trans (Red    (x, y, z)) = Blue   (y, x, z)
  | trans (Blue   (x, y, z)) = Pink   (y, x, z)
  | trans (Pink   (x, y, z)) = Green  (y, x, z)   ②③
  | trans (Green  (x, y, z)) = Yellow (y, x, z)
  | trans (Yellow (x, y, z)) = Orange (y, x, z)
  | trans (Orange (x, y, z)) = Red    (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t * 'b
val x = (Red (2, 2, false), true)   ⑤
val y : (int, bool) u = (trans (#1 x), #2 x)
                                    ④
```

Figure 2 Chained *open*s and nested structures

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

colours. This function is then applied to an instance of a colour (the first element in the pair `x`). Assume that our programming error is that we wrote `'b` instead of `'c` in `Green`'s definition at location ①. SML/NJ (version 110.72) reports a type constructor clash at ④:

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

The reported code does not resemble our code and is far away from the programming error location. SML/NJ gives the same error message if, instead of the error described above, we write `x` instead of `z` in the right-hand-side of any branch of `trans`. This means with SML/NJ one must check the entire program to find the error.

Fig. 1 shows one of the type error slices reported by our type error slice, highlighted in the code. This error is context-dependent: it assumes that `y` and `z` are value variables and not datatype constructors. The programming error location being in the slice, we track it down by considering only the highlighted portions of code, starting from the clashing types on the last line. The type `(int, bool) u` constrains the type of `trans`'s application and the highlighted portion of `trans` is when applied to a `Green` object. At ①, `Green`'s second and third arguments are constrained to be of the same type. At ②, `y` is incidentally constrained to be of the same type as `z`. At ③, because `y` and `z` are respectively the first and third arguments of `Yellow` and using `Yellow`'s definition, we infer that the type of the application of `Yellow` to its three arguments (returned by `trans`) is `t` where its first and third parameters have to be equal. At ④ and ⑤ we can see that `trans` is constrained to return a `t` where its first (`int`) and third (`bool`) parameters differ.

2.2 Chained *open*s and nested structures. Fig. 2 presents a type error involving nested *open*s leading to intricate type errors. Let us describe what the code was meant to do. In the structure `T`, we declare a structure `X` declaring an integer `x`. We then open the structure `S` to access the Boolean `m`. We then open `X` to access the

integer `x`. Finally, if `m` is true then we return `1` otherwise we return `x`. Unfortunately, this piece of code is untypable and SML/NJ reports the following error message which blames `y`'s body:

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

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

Our type error slice rules out `x`'s declarations in `X` and `S` and clearly shows why `x` does not have the expected type. SML/NJ's report leaves us to track down `x`'s binding by hand.

2.3 Merged minimal error slices. We have found cases needing the display of many minimal errors at once. One important case is in record field name clashes where, e.g., the highlighting `val {foo,bar} = {fool=0,bar=1}` reports two minimal errors at once: that `fool` is not in $\{\texttt{foo},\texttt{bar}\}$ and `foo` is not in $\{\texttt{fool},\texttt{bar}\}$. This merged error is preferable over the minimal errors because of the explosion in the number of minimal slices. Light grey highlights the fields that are common to different minimal slices. For merged slices minimality is understood as follows: retain a single dark grey field name in one of the two clashing records and all field names in the other.

3. Mathematical definitions and notations

Let i, j, n, m be metavariables ranging over $\mathbb{N}$, the set of natural numbers. If a metavariable v ranges over a class C, then the metavariables v_x (where x can be anything) and the metavariables 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 $s_i \cap s_j = \varnothing$. Let $s_1 \uplus s_2$ be $s_1 \cup s_2$ if $\mathsf{dj}(s_1, s_2)$ and undefined otherwise. Let R range over binary relations (we write $(\!|x, y|\!)$ for a pair). Given a relation R let $\mathsf{dom}(R) = \{x \mid (\!|x, y|\!) \in R\}$ and $\mathsf{ran}(R) = \{y \mid (\!|x, y|\!) \in R\}$. Let $s \lhd R = \{(\!|x, y|\!) \in R \mid x \notin s\}$. Let f range over functions, let $s \rightarrow s' = \{f \mid \mathsf{dom}(f) \subseteq s \wedge \mathsf{ran}(f) \subseteq s'\}$, and let $x \mapsto y$ be an alternative notation for $(\!|x, y|\!)$ used when writing some functions. A tuple t is a function such that $\mathsf{dom}(t) \subset \mathbb{N}$ and if $1 \leq k \in \mathsf{dom}(t)$ then $k - 1 \in \mathsf{dom}(t)$. Let t range over tuples. We write the tuple $\{0 \mapsto x_0, \ldots, n \mapsto x_n\}$ as $\langle x_0, \ldots, x_n \rangle$. We define the appending $\langle x_1, \ldots, x_i \rangle @ \langle y_1, \ldots, y_j \rangle$ of two tuples as the tuple $\langle x_1, \ldots, x_i, y_1, \ldots, y_j \rangle$. If v ranges over s, $\overrightarrow{v}$ is defined to range over $\mathsf{tuple}(s) = \{t \mid \mathsf{ran}(t) \subseteq s\}$.

4. Technical design of our TES

The different modules of our TES are: constraint generation (Sec. 4.3), constraint solving (Sec. 4.4), minimisation (Sec. 4.5), enumeration (Sec. 4.5), and slicing (Sec. 4.6). Sec. 4.8 discusses minimality, Sec. 4.7 defines the overall algorithm of our type slicer, and Sec. 4.9 discusses the principles of our approach.

4.1 External syntax. Fig. 3 (upper half) defines our external syntax which is a subset of the SML syntax. Many syntactic forms are annotated with labels (l). These labels are generated by our TES to track locations responsible for inferences made during analysis. To provide a visually convenient place for labels, expression applications are surrounded by $\lceil\ \rceil$ which are not seen by programmers but are part of an internal representation used to avoid confusion with ( ) as part of SML syntax. Value identifiers (vid) are subscripted to distinguish between occurrences in expressions (vid^l_{e}), datatype constructor definitions (vid^l_{c}), and patterns (vid^l_{p}).

4.2 Constraint syntax. Constraint terms. Fig. 3 (lower half) defines our constraint terms.

In addition to distinguishing identifier classes (VId for value identifiers, TyCon for type constructor names, etc.), SML assigns statuses within the value identifier class to distinguish value variables, datatype constructors, and exception constructors. Because SML has no lexical distinction between, e.g., a datatype constructor and a value variable, a value identifier's status cannot always be inferred from any context smaller than the entire program.

In our constraint system, an identifier status can either be a raw status (ris) or a raw status annotated with dependencies (is). The `v` status is for value variables (e.g., the recursive function `f` in `val rec f = fn x => x` is a value variable and not a datatype constructor). Statuses `c` and `d` are for unary and nullary datatype constructors respectively (e.g., `c` in `datatype 'a t = C of 'a` and `d` in `datatype 'a t = D`). Status `u` is for unconfirmed context-dependent statuses (e.g., in `fn x => x`, the identifier `x` could be a value variable or a nullary datatype constructor). Status `p` is for unresolvable statuses such as in `let open S in fn x => x end`, where `x` could be declared as a value variable as well as a datatype constructor in the free structure `S`. Finally, status `a` is similar to a variable as it can be any status (used by our constraint filtering function filt defined in Fig. 8 in Sec. 4.4 to generate dummy environments that cannot participate in type errors).

Some syntactic forms, called *dependent forms*, are annotated by dependencies: $\langle x, \overline{d} \rangle$. A dependency d can be a label l or a value identifier vid. During analysis, if a dependency d is a label l, the annotated syntactic form depends on the program node labelled by l. For example, if the dependent equality constraint $\tau_1 \overset{\overline{d} \cup \{l\}}{=\!=\!=} \tau_2$ is generated for the annotated code $\lceil exp\ atexp \rceil^l$, then the equality constraint $\tau_1 = \tau_2$ depends on the application root node of $\lceil exp\ atexp \rceil^l$. If d is a value identifier vid, then the syntactic form depends on vid's status in the code being a `v` or `u`. Because identifiers' statuses are resolved during constraint solving, such dependencies (value identifiers) are only generated during constraint solving and not during initial constraint generation. For example, if constraint solving generates the dependent equality constraint $\tau_1 \overset{\overline{d} \cup \{vid\}}{=\!=\!=\!=\!=\!=} \tau_2$, then the equality constraint $\tau_1 = \tau_2$ needs to be satisfied only if vid cannot be a datatype constructor. Let strip be the function that strips off the outer dependencies (not nested under another constructor than the dependency constructor) of any syntactic form: $\mathsf{strip}(x) = \mathsf{strip}(y)$ if $x = \langle y, \overline{d} \rangle$ and x otherwise.

An internal type of the form $\tau\ \mu$ is called a *type construction* and is built from an internal type constructor μ and its argument τ (such as the polymorphic list type `'a list`, where `'a` is an explicit type variable in SML). We only allow type constructors to take one parameter in this paper and so we only allow internal type constructors to take one parameter in our constraint system. The internal type constructor `ar` is used during constraint solving to represent the arrow type constructor so that we can generate type constraints between the arrow type constructor and any other unary type constructor. This is necessary to compute the necessary portions of code when generating type errors. A type scheme can either be a universal quantification or an internal type.

We use $\downarrow$ to represent bindings (as in $\downarrow id{=}x$ that associates the semantics x to the binding occurrence id) and $\uparrow$ to constrain the semantics of non-binding occurrences (also called accessors) of identifiers (as in $\uparrow id{=}x$ that constrains id's semantics to be x). A binder of the form $\updownarrow vid{=}\alpha$, is an unconfirmed binder that can either be confirmed to be a binder of a value variable at constraint solving

Figure 3 External syntax and constraint system

external syntax (what the programmer sees, plus labels)

$l \in \mathsf{Label}$	(labels)	$dec \in \mathsf{Dec}$	$::= \mathtt{val\ rec}\ pat \stackrel{l}{=} exp \mid \mathtt{datatype}\ dn \stackrel{l}{=} cb \mid \mathtt{open}^l\ sid$
$vid \in \mathsf{VId}$	(value identifiers)	$atexp \in \mathsf{AtExp}$	$::= vid^l_{\mathtt{e}} \mid \mathtt{let}^l\ dec\ \mathtt{in}\ exp\ \mathtt{end}$
$sid \in \mathsf{StrId}$	(structure identifiers)	$exp \in \mathsf{Exp}$	$::= atexp \mid \mathtt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l$
$tv \in \mathsf{TyVar}$	(type variables)	$atpat \in \mathsf{AtPat}$	$::= vid^l_{\mathtt{p}}$
$tc \in \mathsf{TyCon}$	(type constructors)	$pat \in \mathsf{Pat}$	$::= atpat \mid vid^l\ atpat$
$ty \in \mathsf{Ty}$	$::= tv^l \mid ty_1 \stackrel{l}{\rightarrow} ty_2 \mid ty\ tc^l$	$sdec \in \mathsf{StrDec}$	$::= dec \mid \mathtt{structure}\ sid \stackrel{l}{=} sexp$
$cb \in \mathsf{ConBind}$	$::= vid^l_{\mathtt{c}} \mid vid\ \mathtt{of}^l\ ty$	$sexp \in \mathsf{StrExp}$	$::= sid^l \mid \mathtt{struct}^l\ sdec_1 \cdots sdec_n\ \mathtt{end}$
$dn \in \mathsf{DatName}$	$::= tv\ tc^l$		

constraint terms (syntax of entities used internally by the type error slicer and which the programmer never sees)

$ev \in \mathsf{EnvVar}$	(environment variables)	$\mu \in \mathsf{ITyCon}$	$::= \delta \mid \gamma \mid \mathtt{ar} \mid \langle \mu, \overline{d} \rangle$
$\delta \in \mathsf{TyConVar}$	(type constructor variables)	$\tau \in \mathsf{ITy}$	$::= \alpha \mid \tau\,\mu \mid \tau_1 \rightarrow \tau_2 \mid \langle \tau, \overline{d} \rangle$
$\gamma \in \mathsf{TyConName}$	(type constructor names)	$\sigma \in \mathsf{Scheme}$	$::= \tau \mid \forall \overline{\alpha}.\, \tau$
$\alpha \in \mathsf{ITyVar}$	(internal type variables)	$bind \in \mathsf{Bind}$	$::= \downarrow tc{=}\mu \mid \downarrow sid{=}e \mid \downarrow tv{=}\alpha \mid \downarrow vid{=}\sigma \mid \downarrow vid{=}is \mid \updownarrow vid{=}\alpha$
$d \in \mathsf{Dependency}$	$::= l \mid vid$	$acc \in \mathsf{Accessor}$	$::= \uparrow tc{=}\delta \mid \uparrow sid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha \mid \uparrow vid{=}ris$
$ris \in \mathsf{RawIdStatus}$	$::= \mathtt{v} \mid \mathtt{c} \mid \mathtt{d} \mid \mathtt{u} \mid \mathtt{p} \mid \mathtt{a}$	$c \in \mathsf{Constraint}$	$::= \mu_1{=}\mu_2 \mid e_1{=}e_2 \mid \tau_1{=}\tau_2 \mid is_1{=}is_2$
$is \in \mathsf{IdStatus}$	$::= ris \mid \langle is, \overline{d} \rangle$	$e \in \mathsf{Env}$	$::= \boxdot \mid ev \mid bind \mid acc \mid c \mid \mathtt{poly}(e) \mid e_2;e_1 \mid \langle e, \overline{d} \rangle$

extra metavariables mostly used in side conditions

$id \in \mathsf{Id} ::= vid \mid sid \mid tv \mid tc$ $\quad var \in \mathsf{Var} ::= \alpha \mid \delta \mid ev$ $\quad dep \in \mathsf{Dependent} ::= \langle \tau, \overline{d} \rangle \mid \langle \mu, \overline{d} \rangle \mid \langle e, \overline{d} \rangle \mid \langle is, \overline{d} \rangle$

time, and so be turned into a binder of the form $\downarrow vid{=}\alpha$ or be turned into an accessor $\uparrow vid{=}\alpha$ if it turns out that vid is a datatype constructor. This mechanism is further illustrated in Sec. 4.4.

The keystone of our constraint system is the constrained environment $e_1;e_2$ where e_1 constrains e_2. The environment $e_1;e_2$ builds a new environment from its two components where references of the form $\uparrow id$ (accessors) in e_2 can depend on occurrences of $\downarrow id$ (binders) in e_1. For example, in $\downarrow vid{=}\sigma;\uparrow vid{=}\alpha$, α is constrained to be σ through the binding of vid. The motivation for these environments is to have a general mechanism to build environments for sequential declarations.

In addition to a constrained environment of the form $e_1;e_2$, an environment can also be an empty environment $\boxdot$, an environment variable ev, a binder $bind$ associating static semantics to identifiers, an accessor to look identifiers' static semantics up in environments, an equality constraint c, a special form $\mathtt{poly}(e)$ (explained below) which grants e the possibility to be polymorphic, or a conditional environment $\langle e, \overline{d} \rangle$ depending on $\overline{d}$. Environments are special kinds of constraint (on internal types, internal type constructors, environments and statuses).

Let $e_1;\cdots;e_n$ be $\boxdot$ if $n = 0$ and $(e_1;\cdots;e_{n-1});e_n$ if $n > 0$.

"Atomic" syntactic forms. Let $\mathsf{atoms}(x)$ be the set of syntactic forms belonging to $\mathsf{Var} \cup \mathsf{TyConName} \cup \mathsf{Dependency}$ and occurring in x whatever x is. We define the following functions:

$$\begin{array}{lll} \mathsf{vars}(x) & = \mathsf{atoms}(x) \cap \mathsf{Var} & \text{(set of variables)} \\ \mathsf{labs}(x) & = \mathsf{atoms}(x) \cap \mathsf{Label} & \text{(set of labels)} \\ \mathsf{deps}(x) & = \mathsf{atoms}(x) \cap \mathsf{Dependency} & \text{(set of dependencies)} \end{array}$$

Freshness of variables. We use distinguished dummy variables: $\mathsf{DumVar} = \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}\}$. Each use of a dummy variable acts like a fresh variable. These variables are used to generate dummy environments and constraints. For example, the equality constraint $ev_{\mathtt{dum}}{=}e$ means that the environment e must be solved and does not constrain any other environment. The relation dja ensures the freshness of the generated variables and type constructor names: $\mathsf{dja}(x_1, \ldots, x_n) \Leftrightarrow \mathsf{dj}(f(x_1), \ldots, f(x_n), \mathsf{DumVar})$, where $f(x) = \mathsf{atoms}(x) \setminus \mathsf{VId}$. This also ensures that each label occurs at most once in a labelled program.

Syntactic sugar. We write $\langle x, d \rangle$ for $\langle x, \{d\} \rangle$. Let y be a d or a $\overline{d}$. We write x^y for $\langle x, y \rangle$. We write $x_1 \stackrel{y}{=} x_2$ for $\langle x_1{=}x_2, y \rangle$, and similarly for $bind$'s and acc's. We write $\downarrow vid \stackrel{y}{=} \langle \sigma, is \rangle$ for $\downarrow vid \stackrel{y}{=} is;\downarrow vid \stackrel{y}{=} \sigma$, and $\uparrow vid \stackrel{y}{=} \langle \alpha, ris \rangle$ for $\uparrow vid \stackrel{y}{=} ris;\uparrow vid \stackrel{y}{=} \alpha$ We write $[e]$ for $(ev_{\mathtt{dum}}{=}e)$. Such a constraint defines a local environment e which is not visible from outside the constraint. This is used for local bindings by rules (G2) and (G4) of our constraint generation algorithm defined in Fig. 4.

4.3 Constraint generation. Value bindings.

At constraint generation (Fig. 4), in the pattern rule (G5), we generate monomorphic, unconfirmed binders of the form $\updownarrow vid{=}\alpha$ where no type variable is yet quantified over. These binders are monomorphic because in SML, e.g., the type of a recursive function such as `f` in the let-expression `let val rec f = fn x => f x in f end`, is monomorphic within its definition (`f`'s first and second occurrences' types are equal) and generalised into a polymorphic *for all* type scheme when typing the declaration (`f`'s third occurrence's type is an instance of the generalisation of `f`'s first occurrence's type). An environment e is then turned into a polymorphic one during constraint solving (using toPoly defined in Fig. 6 in Sec. 4.4) if marked as follows: $\mathtt{poly}(e)$. Such forms are generated by the recursive value declaration rule (G12) and the datatype declaration rule (G13). In (G5) again, the binder is unconfirmed and no status constraint is generated (as opposed to, e.g., rule (G10) which forces the analysed identifier to be a nullary datatype constructor) because in SML, e.g., in `fn x => x`, without any more context, the identifier `x` could be a value variable or a datatype constructor. The status of `x` is then unknown. Because recursive functions are forced to be value variables (`v`) even when in the scope of a datatype constructor binding, toV (used by (G12)) generates a status constraint:

$$\begin{array}{ll} \mathsf{toV}(e_1;e_2) & = \mathsf{toV}(e_1);\mathsf{toV}(e_2) \\ \mathsf{toV}(\updownarrow vid \stackrel{l}{=} \alpha) & = (\downarrow vid \stackrel{l}{=} \langle \alpha, \mathtt{v} \rangle) \\ \mathsf{toV}(e) & = e, \text{ if none of the above applies} \end{array}$$

As explained in Sec. 4.4, at constraint solving, an unconfirmed binder of the form $\updownarrow vid{=}\alpha$ eventually turns into a binder of the form $\downarrow vid{=}\alpha$ or an accessor of the form $\uparrow vid{=}\alpha$. (In some cases, a status constraint is also generated from an unconfirmed binder.)

Algorithm. Fig. 4 defines our constraint generator. At initial constraint generation, the only labelled environments are equality constraints (c), binders ($bind$), accessors (acc), and environment variables (ev).

In rule (G12) for recursive function declarations, the environment $\mathsf{toV}(e_1)$ generated for the pattern part of the declaration constrains the environment e_2 generated for the expression part. This order is necessary to handle the recursivity of such declarations.

Figure 4 Constraint generation rules

Expressions

(G1) $vid^l_{\mathsf{e}} \triangleright \langle \alpha, \uparrow vid \stackrel{l}{=} \alpha \rangle$

(G2) $\texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end} \triangleright \langle \alpha, [e_1;e_2;(\alpha \stackrel{l}{=} \alpha_2)] \rangle \Leftarrow dec \triangleright e_1 \wedge exp \triangleright \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

(G3) $\lceil exp\ atexp \rceil^l \triangleright \langle \alpha, e_1;e_2;(\alpha_1 \stackrel{l}{=} \alpha_2 \to \alpha) \rangle \Leftarrow exp \triangleright \langle \alpha_1, e_1 \rangle \wedge atexp \triangleright \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

(G4) $\texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \triangleright \langle \alpha, [(ev{=}e_1);ev^l;e_2;(\alpha \stackrel{l}{=} \alpha_1 \to \alpha_2)] \rangle \Leftarrow pat \triangleright \langle \alpha_1, e_1 \rangle \wedge exp \triangleright \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \alpha, ev)$

Patterns

(G5) $vid^l_{\mathsf{p}} \triangleright \langle \alpha, \updownarrow vid \stackrel{l}{=} \alpha \rangle$ (G6) $vid^l\ atpat \triangleright \langle \alpha_2, \alpha_1 \stackrel{l}{=} \alpha \to \alpha_2; \uparrow vid \stackrel{l}{=} \langle \alpha_1, \mathsf{c} \rangle; e \rangle \Leftarrow atpat \triangleright \langle \alpha, e \rangle \wedge \mathsf{dja}(e, \alpha_1, \alpha_2)$

Types

(G7) $tv^l \triangleright \langle \alpha, \uparrow tv \stackrel{l}{=} \alpha \rangle$ (G8) $ty\ tc^l \triangleright \langle \alpha', (\uparrow tc \stackrel{l}{=} \delta); (\alpha' \stackrel{l}{=} \alpha\ \delta); e \rangle \Leftarrow ty \triangleright \langle \alpha, e \rangle \wedge \mathsf{dja}(e, \alpha', \delta)$

(G9) $ty_1 \stackrel{l}{\to} ty_2 \triangleright \langle \alpha, e_2;e_1;(\alpha \stackrel{l}{=} \alpha_1 \to \alpha_2) \rangle \Leftarrow ty_1 \triangleright \langle \alpha_1, e_1 \rangle \wedge ty_2 \triangleright \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

Constructor bindings

(G10) $vid^l_{\mathsf{c}} \triangleright \langle \alpha, \downarrow vid \stackrel{l}{=} \langle \alpha, \mathsf{d} \rangle \rangle$

(G11) $vid\ \texttt{of}^l\ ty \triangleright \langle \alpha_1, e; \alpha_2 \stackrel{l}{=} \alpha \to \alpha_1; \downarrow vid \stackrel{l}{=} \langle \alpha_2, \mathsf{c} \rangle \rangle \Leftarrow ty \triangleright \langle \alpha, e \rangle \wedge \mathsf{dja}(e, \alpha_1, \alpha_2)$

Declarations

(G12) $\texttt{val rec}\ pat \stackrel{l}{=} exp \triangleright (ev{=}\texttt{poly}(\texttt{toV}(e_1);e_2;(\alpha_1 \stackrel{l}{=} \alpha_2)));ev^l \Leftarrow pat \triangleright \langle \alpha_1, e_1 \rangle \wedge exp \triangleright \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

(G13) $\texttt{datatype}\ dn \stackrel{l}{=} cb \triangleright (ev{=}((\alpha_1 \stackrel{l}{=} \alpha_2);e_1;\texttt{poly}(e_2)));ev^l \Leftarrow dn \triangleright \langle \alpha_1, e_1 \rangle \wedge cb \triangleright \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

(G14) $\texttt{open}^l\ sid \triangleright (\uparrow sid \stackrel{l}{=} ev);ev^l$

Datatype names

(G15) $tv\ tc^l \triangleright \langle \alpha', (\alpha' \stackrel{l}{=} \alpha\ \gamma); (\downarrow tc \stackrel{l}{=} \gamma); (\downarrow tv \stackrel{l}{=} \alpha) \rangle \Leftarrow \alpha \neq \alpha'$

Structure declarations

(G16) $\texttt{structure}\ sid \stackrel{l}{=} sexp \triangleright (ev'{=}(e;(\downarrow sid \stackrel{l}{=} ev)));ev'^l \Leftarrow sexp \triangleright \langle ev, e \rangle \wedge \mathsf{dja}(e, ev')$

Structure expressions

(G17) $sid^l \triangleright \langle ev, \uparrow sid \stackrel{l}{=} ev \rangle$

(G18) $\texttt{struct}^l\ sdec_1 \cdots sdec_n\ \texttt{end} \triangleright \langle ev, (ev \stackrel{l}{=} ev'); (ev'{=}(e_1;\cdots;e_n)) \rangle \Leftarrow sdec_1 \triangleright e_1 \wedge \cdots \wedge sdec_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev, ev')$

In rule (G13) for datatype declarations, the environment e_1 generated for the declared type constructor constrains the environment $\texttt{poly}(e_2)$ generated for the datatype constructor of the declared type constructor. This order is necessary to handle the recursivity of such datatype declarations (in `datatype nat = z | s of nat`, the second occurrence of `nat` refers to its first occurrence).

Rule (G14) for structure opening (as for rules (G4), (G12), (G13) and (G16)) labels an environment variable, so that a sliced out declaration does not shadow its environment. Without this label, the environment variable would be a constraint that always have to be satisfied. With the label, the environment variable is a constraint that has to be satisfied only when the declaration is not sliced out. The link between the environment variable and the structure to open is made via the labelled accessor $\uparrow sid \stackrel{l}{=} ev$.

Rule (G18) for structure expressions (as for rules (G4), (G12), (G13) and (G16)) generates unlabelled equality constraints. An unlabelled equality constraint such as a constraint of the form $ev'{=}(e_1;\cdots;e_n)$ generated by (G18) needs to be unlabelled because each of the e_i is not dependent on the analysed structure expression itself but is dependent on the corresponding declaration packed together with other declarations in the structure expression. The information related to the analysed structure expression, carried by the unlabelled constraint is the fact that a sequence of declarations (corresponding to the constrained environment $e_1;\cdots;e_n$) is packed into a structure. This information depends on the analysed structure expression via the extra labelled equality constraint $ev \stackrel{l}{=} ev'$. In (G4), (G12), (G13) and (G16), we use labelled environment variables of the form ev^l.

4.4 Constraint solving. Syntax. Fig. 5 defines the syntactic forms used by our constraint solver (Fig. 7) where one unification step is defined by the relation $\rightarrow$, where $\rightarrow^*$ is its reflexive and transitive closure. To each state of a unification computation (except the errors), a unification context $\Delta = \langle u, e \rangle$ is associated. In a state $\texttt{solve}(\langle u, e \rangle, \overline{d}, e')$, $\langle u, e \rangle$ is the context in which e' must be solvable for the algorithm to succeed. Let $\langle u, e \rangle(var)$ be $u(var)$, let $\langle u, e \rangle;e'$ be $\langle u, e;e' \rangle$, and let $u_1 \boxplus u_2$ be $u_1 \cup (\mathsf{DumVar} \lhd u_2)$ if $\mathsf{dj}(\mathsf{dom}(u_1), \mathsf{dom}(u_2))$, and undefined otherwise.

Given constraints (of the form e), our constraint solver either succeeds with $\texttt{succ}(\Delta)$ returning its current unification context Δ, or fails with $\texttt{err}(er)$ returning an error which can be (see ek in Fig. 5) a type constructor clash, a circularity error or a status clash (discussed below with the compatible relation). The application of a renaming ren to an internal type τ is defined as usual and, is denoted $\tau[ren]$. Renamings are used to instantiate type schemes.

Environment application. Constraint solving maintains a type environment (e in Δ) where some parts might be shadowed and so inaccessible. For example, in $bind_2;ev;bind_1^d$, the usable part is $bind_1$ and ev shadows $bind_2$ because an environment variable stands for any environment and could potentially bind any identifier. During unification, no c or acc occurs in the e stored in a Δ because they are transformed into unifiers u (rules (U3) and (U4) in Fig. 7). Similarly, the $\texttt{poly}(e)$ and $\updownarrow vid{=}\alpha$ forms are eliminated. Concerning $\boxdot$, we never add it to a unification context Δ while unifying constraints, but we always start a unification process with the initial unification context $\langle \varnothing, \boxdot \rangle$. Thus, during unification if $\Delta = \langle u, e \rangle$ then e is of the form $\boxdot;e_1 \cdots;e_n$, where each e_i is either an ev or a $\downarrow id \stackrel{\overline{d}}{=} x$. Let the predicate hiding be defined as follows: $\mathsf{hiding}(e)$ be true iff $e \in \mathsf{Var}$ or e is of the form $e_1;e_2$ and $\mathsf{hiding}(e_i)$ for $i = 1$ or $i = 2$. Let $\mathsf{hiding}(\langle u, e \rangle)$ be true iff $\mathsf{hiding}(e)$. We define the application $e(id)$ as follows:

(EA1) $(e';\downarrow id \stackrel{\overline{d}}{=} \forall \overline{\alpha}.\, \tau)(id) = \forall \overline{\alpha}.\, \tau^{\overline{d}}$

(EA2) $(e';\downarrow id \stackrel{\overline{d}}{=} x)(id) = x^{\overline{d}}$, if x of the form τ, μ or e

(EA3) $(e';\downarrow id' \stackrel{\overline{d}}{=} x)(id) = e'(id)$, if $id \neq id'$ or $x \in \mathsf{IdStatus}$

We define $e[\![id]\!]$ to access value identifiers' statuses:

(EAIS1) $(e';\downarrow id \stackrel{\overline{d}}{=} is)[\![id]\!] = is^{\overline{d}}$

(EAIS2) $(e';\downarrow id' \stackrel{\overline{d}}{=} x)[\![id]\!] = e'[\![id]\!]$, if $id \neq id'$ or $x \notin \mathsf{IdStatus}$

For example, $(\downarrow vid \stackrel{\overline{d}_3}{=} \sigma;\downarrow vid \stackrel{\overline{d}_2}{=} \mathsf{v};\downarrow sid \stackrel{\overline{d}_1}{=} e)(vid) = \sigma$ but $(\downarrow vid \stackrel{\overline{d}_3}{=} \sigma;\downarrow vid \stackrel{\overline{d}_2}{=} \mathsf{v};ev^{\overline{d}};\downarrow sid \stackrel{\overline{d}_1}{=} e)(vid)$ and $(\downarrow vid \stackrel{\overline{d}_3}{=} \sigma;\downarrow vid \stackrel{\overline{d}_2}{=} \mathsf{v};\downarrow sid \stackrel{\overline{d}_1}{=} e)(tc)$ are undefined.

Let $\Delta(id) = e(id)$ and $\Delta[\![id]\!] = e[\![id]\!]$, where $\Delta = \langle u, e \rangle$.

Context dependencies solving. Context dependencies are solved during unification. An unconfirmed binder of the form $\updownarrow vid{=}\alpha$ is then either turned into a binder of the form $\downarrow vid{=}\alpha$ or

Figure 5 Syntactic forms used by the constraint solver

$$
\begin{array}{rrl}
 & u \in \mathsf{Unifier} & = \{f_1 \cup f_2 \cup f_3 \mid f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \wedge f_3 \in \mathsf{EnvVar} \to \mathsf{Env}\} \\
er \in \mathsf{Error} ::= \langle ek, \overline{d}\rangle & ek \in \mathsf{ErrKind} & ::= \mathtt{tyConsClash}(\mu_1, \mu_2) \mid \mathtt{statusClash}(is_1, is_2) \mid \mathtt{circularity} \\
\Delta \in \mathsf{UnifEnv} ::= \langle u, e\rangle & state \in \mathsf{State} & ::= \mathtt{solve}(\Delta, \overline{d}, e) \mid \mathtt{succ}(\Delta) \mid \mathtt{err}(er) \\
 & ren \in \mathsf{Ren} & = \{ren \in \mathsf{ITyVar} \to \mathsf{ITyVar} \mid ren \text{ is injective} \wedge \mathsf{dj}(\mathsf{dom}(ren), \mathsf{ran}(ren))\}
\end{array}
$$

an accessor of the form $\uparrow vid{=}\alpha$ by one of these rules: (B2)-(B5). These rules make use of the function ifNotDum which is defined as follows: $\mathsf{ifNotDum}(\alpha_{\mathtt{dum}}, is) = \mathtt{a}$ and $\mathsf{ifNotDum}(\alpha, is) = is$ if $\alpha \notin \mathsf{DumVar}$. This function is required to ensure that a dummy binder cannot bind something else than a dummy status. Rule (B2) discards binders generated under unsatisfied context dependencies, e.g., in `let datatype t = x in fn x => x end`, `x`'s second occurrence does not bind `x`'s third occurrence because of `x`'s declaration as a datatype constructor. The unconfirmed binder is then turned into an accessor. In all three other rules, the unconfirmed binder is turned into a confirmed one. Rule (B3) validates context dependencies, e.g., in `val rec x = fn x => x`, `x` is confirmed to be a value variable, `x`'s second occurrence being in the scope of `x`'s first occurrence which is a recursive function and so in SML is forced to be a value variable and not a datatype constructor. Rule (B4) generates context dependencies, e.g., in `fn x => x`, because `x` can be a value variable as well as a datatype constructor then `x`'s second occurrence is bound to `x`'s first occurrence under the context dependency that `x` is not a datatype constructor. Rule (B5) generates dummy environments when there is not enough information to check whether a context dependency is satisfied or not, e.g., in `let open S in fn x => x end`, if `S` is free, it might declare `x` as a datatype constructor or as a recursive function. Thus, we do not allow `x` to be a monomorphic binding but we still generate a dummy binding to catch status clashes (e.g., if instead of the second `x` we had `fn (x y) => y` where `x` is a unary datatype constructor, we would then have `x` occurring in patterns both at a nullary position and a unary position).

Status compatibility. Two identifier statuses are incompatible iff a unary datatype constructor, occurring in a pattern, is bound to a (context-dependent or independent) value variable as in `let val rec f = fn x => x in fn (f x) => x end` where `f`'s first occurrence is a value variable and `f`'s second occurrence is a unary datatype constructor (taking an argument in a pattern); or if a value variable in a pattern (not applied) is bound to a unary datatype constructor as in `let datatype t = x of int in fn x => x end`.

$$\mathsf{compatible}(is_1, is_2) \Leftrightarrow \{is_1, is_2\} \notin \{\{\mathtt{c}, \mathtt{v}\}, \{\mathtt{c}, \mathtt{u}\}, \{\mathtt{c}, \mathtt{p}\}\}$$

Status compatibility is checked by constraint solving rules (S6) and (S7) (in Fig. 7). Compatibility is only defined on raw statuses because constraint solving rule (S8) removes dependencies on (among other things) statuses.

Building of constraint terms. The constraint solver uses build to build, w.r.t. a given unifier, polymorphic types (Fig. 6), check circularity errors (in order not to generate a unifier where, e.g., $\alpha = \tau{\to}\alpha$), and build environment:

$$
\begin{array}{ll}
\mathsf{build}(u, var) & = \begin{cases} \mathsf{build}(u, x), \text{ if } u(var) = x \\ var, \quad \text{otherwise} \end{cases} \\
\mathsf{build}(u, \tau\,\mu) & = \mathsf{build}(u, \tau)\ \mathsf{build}(u, \mu) \\
\mathsf{build}(u, \tau_1{\to}\tau_2) & = \mathsf{build}(u, \tau_1){\to}\mathsf{build}(u, \tau_2) \\
\mathsf{build}(u, x^{\overline{d}}) & = \mathsf{build}(u, x)^{\overline{d}} \\
\mathsf{build}(u, x) & = x, \text{ if none of the above applies}
\end{array}
$$

As explained at the end of this Section, types have to be built up when generating polymorphic environments for efficiency issues. Because SML does not allow infinite types, we also use build to detect circularity issues. During unification, before augmenting any unification context, we check if it would allow allow generating infinite types (see rule (U1) of our unification algorithm defined in Fig. 7). For example, given the unifier $\{\alpha_1 \mapsto \alpha_2^{\overline{d}_1}, \alpha_2 \mapsto \langle \alpha_3^{\overline{d}_3}{\to}\alpha_4^{\overline{d}_4}, \overline{d}_2\rangle\}$, we do not allow its augmentation with, e.g., $\{\alpha_3 \mapsto \langle \alpha_5^{\overline{d}_6}{\to}\alpha_1^{\overline{d}_7}, \overline{d}_5\rangle\}$ because it would allow generating infinite types.

Environment extraction. The function diff is used by rules (U4), (P1) and (P2) of our constraint solver to extract environments generated during unification. It allows, when solving an environment, getting back its "solved version" once all of its constraints have been dealt with. By "solved version" of an environment e, we mean the sequence of environments that has been added to the unification context of the state in which the unification process was when it started to solve e. For example, if $\mathtt{solve}(\langle u, e\rangle, \overline{d}, e_0) \to^* \mathtt{succ}(\langle u', e'\rangle)$ then $e' = e; e_1 \cdots; e_n$ and $\mathsf{diff}(e, e') = \boxdot; e_1 \cdots; e_n$ which is the "solved version" of e_0.

$$
\begin{array}{ll}
\mathsf{diff}(e, e) & = \boxdot \\
\mathsf{diff}(e_1, e_2; e_3) & = \mathsf{diff}(e_1, e_2); e_3, \text{ if } e_1 \neq (e_2; e_3)
\end{array}
$$

Polymorphic environments. Fig. 6 defines toPoly which is used by rule (P1) of our constraint solver to generate a polymorphic environment by quantifying the type variables not occurring in the types of the monomorphic bindings of the unification environment of the current state.

In Fig. 6, τ is the type from which we want to generate a type scheme. First, we build up the type, using the unifier of the unification context of the current state (u), to obtain the type τ'. The set $\overline{\tau}$ is the set of types of the monomorphic bindings for which, the binding currently being generalised, is in the scope. The set $\overline{\alpha}$ is the set of type variables that are allowed to be quantified over because they do not depend on the types of the monomorphic bindings. Finally, $\overline{d}'$ is the set of dependencies "explaining" why the type variables not in $\overline{\alpha}$ but occurring in τ' (the type variables occurring in τ' and also depending on the monomorphic bindings) are not allowed to be quantified over.

Let us illustrate how this mechanism works with the fn-expression exp: `fn x => let val rec f = fn z => x z in f end`. The constraint generation algorithm generates an environment of the form $\mathtt{poly}(e_1)$ for the recursive declaration `val rec f = fn z => x z`. When solving the constraints generated for exp, the constraint solver eventually applies toPoly to a unification context $\langle u, e\rangle$ and a binding of the form $\downarrow\mathtt{f} \overset{\overline{d}}{=} \alpha_1$ (which is the "solved version" of e_1). Building up α_1 results in a type τ' of the form $\langle \alpha_2^{\overline{d}_2}{\to}\alpha_3^{\overline{d}_3}, \overline{d}_1\rangle$. Because `x`'s type is monomorphic, a monomorphic binding of the form $\downarrow\mathtt{x} \overset{\overline{d}_1}{=} \alpha_0$ occurs in e (the only monomorphic binding occurring in e) and so we build a $\overline{\tau}$ (see Fig. 6) of the form $\{\tau_0\}$ where τ_0 is obtain building up α_0 and is of the form $\langle \alpha_2^{\overline{d}_5}{\to}\alpha_3^{\overline{d}_6}, \overline{d}_4\rangle$ (equivalent to τ' up to dependencies because `f` η-reduces to `x`). We therefore build a $\overline{\alpha}$ (see Fig. 6) of the form $\varnothing$ because α_2 and α_3 both occur in τ'. We also build a $\overline{d}'$ of the form $\overline{d}_4 \cup \overline{d}_5 \cup \overline{d}_6$ which is the set of "reasons" for not allowing α_2 and α_3 to be in $\overline{\alpha}$ (set of type variables that are allowed to be generalised over when building the type scheme returned by toPoly). Finally, e is augmented with $\downarrow\mathtt{f} \overset{\overline{d}}{=} \forall\varnothing.\, \langle \alpha_2^{\overline{d}_2}{\to}\alpha_3^{\overline{d}_3}, \overline{d}_1 \cup \overline{d}'\rangle$.

Figure 6 Monomorphic to polymorphic environment

$$\mathsf{toPoly}(\langle u, e\rangle, \downarrow vid \stackrel{\overline{d}}{=} \langle \tau, is\rangle) = \langle u, e; (\downarrow vid \stackrel{\overline{d}}{=} \langle \forall \overline{\alpha}.\, \tau'^{\overline{d}'}, is\rangle)\rangle \quad \text{where} \begin{cases} \tau' = \mathsf{build}(u, \tau) \\ \overline{\tau} = \{\tau_0 \mid \exists vid.\ \tau_1 = e(vid) \wedge \tau_0 = \mathsf{build}(u, \tau_1)\}\} \\ \overline{\alpha} = (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus (\mathsf{vars}(\overline{\tau}) \cup \{\alpha_{\mathtt{dum}}\}) \\ \overline{d}' = \{d \mid \tau_0 \in \overline{\tau} \wedge d \in \mathsf{deps}(\tau_0) \wedge \neg\mathsf{dj}(\mathsf{vars}(\tau_0) \cap \mathsf{ITyVar}, \mathsf{vars}(\tau') \setminus (\overline{\alpha} \cup \{\alpha_{\mathtt{dum}}\}))\} \end{cases}$$

When solving constraints generated by our constraint generator, toPoly is only applied to $bind^{\overline{d}}$'s resulting from the solving of an environment wrapped by `poly` which in turn is only used to wrap environments built from: dependencies, a unique monomorphic binding and equality constraints.

Extracting the monomorphic type variables of a binding's type is an expansive computation. We only perform it once per polymorphic binding by, provided a unification context, first building the type of a given binding and by then looking up in the environment (in the unification context) which type variables cannot be quantified over because they are monomorphic. When accessing the type of a polymorphic binding we then only have to generate an instance of its type scheme (see rule (A1) of our unification algorithm).

Algorithm. Fig. 7 defines our constraint solver.

The accessor rule (A4) can also be used to report free identifiers. If $\mathtt{solve}(\Delta, \overline{d}, \uparrow id{=}x) \rightarrow \mathtt{succ}(\Delta)$ and $\neg\mathsf{hiding}(\Delta)$ then it means that there is no binder for id and so that it is a free identifier.

Free identifiers are in any case important to report, but it is especially vital for structure identifiers in *open* declarations. In our approach, a free opened structure is considered as potentially redefining its entire context. Hence, `val x = 1 open S val y = x 1` is typable because `x`'s first occurrence is hidden by the declaration `open S`. This might be confusing if `S` was not reported as being free. Let us explain how a free opened structure hides its context. Given a declaration `open S`, our constraint generation algorithm generates an environment of the form $(\uparrow\mathtt{S} \stackrel{l}{=} ev); ev^l$ where l is the label labelling the declaration. Because `S` is free, rule (A4) applies when solving $\uparrow\mathtt{S}{=}ev$. The environment variable ev is then not constrained to anything. Hence when solving ev, rule (V2) applies and $\Delta; ev$ (from the right-hand-side of rule (V2)) results in the hiding of Δ by ev: all the binders in Δ are hidden by ev.

Let the relations isErr and solvable be defined as follows:

$$\begin{array}{ll} e \xrightarrow{\mathsf{isErr}} er & \Leftrightarrow \mathtt{solve}(\langle\varnothing, \boxdot\rangle, \varnothing, e) \rightarrow^* \mathtt{err}(er) \\ \mathsf{solvable}(e) & \Leftrightarrow \exists\Delta.\ \mathtt{solve}(\langle\varnothing, \boxdot\rangle, \varnothing, e) \rightarrow^* \mathtt{succ}(\Delta) \\ \mathsf{solvable}(sdec) & \Leftrightarrow \exists e.\ sdec \triangleright e \wedge \mathsf{solvable}(e) \end{array}$$

4.5 Minimisation and enumeration. Extraction of environment labels. Given an environment e, lBinds extracts the labels labelling binders ($bind$) occurring in e. It is used during the first phase of our minimisation algorithm which consists in trying to remove entire sections of code (datatype declarations, functions, structures, ...) by "disconnecting" accessors from their binders:

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

Constraint filtering. Fig. 8 defines our constraint filtering function filt, used to check the solvability of constraints in which some constraints are discarded. It is only applied to constraints generated by our constraint generator. This is why we only filter environment equality constraints of the form $ev{=}e$ and not of the general from $e_1{=}e_2$. In $\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)$, $\overline{l}_1$ is the label set for which we want to keep the annotated environments (first case of the filtering rule for e^l), and $\overline{l}_2$ is the label set for which we do not want to keep the equality constraints and accessors but for which we want to turn the binders into dummy ones and keep the environment variables (second case of the filtering rule for e^l). The environments annotated by labels that are not in $\overline{l}_1 \cup \overline{l}_2$ are then discarded (third case of the filtering rule for e^l). In the context of constraint filtering, label sets are sometimes called filters. Being able to distinguish between binders to discard (not labelled by a label in $\overline{l}_1 \cup \overline{l}_2$) and binders to turn into dummy ones (labelled by a label in $\overline{l}_2$) is necessary because during minimisation, throwing away any environment might result in different bindings in the filtered constraints (corresponding to a different SML code). For example, removing the binder labelled by l_2 in $(\downarrow\mathtt{x} \stackrel{l_1}{=} \langle\tau_1, is_1\rangle); (\downarrow\mathtt{x} \stackrel{l_2}{=} \langle\tau_2, is_2\rangle); (\uparrow\mathtt{x} \stackrel{l}{=} \tau)$ would result in `x`'s accessor being bound to `x`'s first binder instead of its second. Similarly, removing the binding labelled by the label associated to `f`'s second occurrence in the environment generated for

```
let val rec f = fn x => x 1
in let val rec f = fn x => x + 1 in f true end
end
```

would result in `f`'s third occurrence to be bound to its first occurrence and so to the enumeration algorithm to find a type error that does not exist in the original piece of code. When a binding is labelled by a label from $\overline{l}_2$, it becomes a dummy unlabelled one that cannot be involved in any error and it results that the same holds for its accessors.

Minimisation algorithm. Fig. 9 defines our minimisation algorithm: the relation min that uses the relation $\rightarrow_{\mathsf{test}}$ to test if a label can be removed from a slice and where $\rightarrow^*_{\mathsf{test}}$ is its reflexive and transitive closure. It consists of two main phases. The first one ($\langle e, \mathsf{labs}(er) \setminus \overline{l}, \mathsf{labs}(er) \cap \overline{l}\rangle \rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_1, \varnothing\rangle$) tries to remove entire sections of code at once by turning bindings into dummy ones using lBinds. In a fine-grained second phase ($\langle e, \varnothing, \overline{l}_1\rangle \rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_2, \varnothing\rangle$) the algorithm tries to remove the remaining labels ($\overline{l}_1$) one at a time.

A step of our unification algorithm is as follow: $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2\rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_3, \overline{l}_4\rangle$ where $\overline{l}_3$ and $\overline{l}_4$ depend on the solvability of $\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\})$. The set $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$ is the label set of the error that the minimisation algorithm is minimising and $\{l\} \uplus \overline{l}_2$ is the label set yet to try to discard. The environment $\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\})$ is obtained from e by filtering out the constraints that are not labelled by $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$ and by turning the binders labelled by l into dummy ones. If the obtained filtered environment is solvable it means that l is necessary and $\overline{l}_3 = \overline{l}_1 \cup \{l\}$ and $\overline{l}_4 = \overline{l}_2$. If it is unsolvable (solving the filtered environment failed and we obtained a new smaller error), it means that l is unnecessary for an error to occur and that any environment labelled by l can be completely filtered out in the next step. The label sets $\overline{l}_3$ and $\overline{l}_4$ are then restricted to the newly found error (see rule (M1)).

Environments (bindings, environment variables, ...) can be completely filtered out from one step to another because our constraint generator and solver, together ensure that if a binder is turned into a dummy one then none of its accessors will be part of any error. This invariant could explicitly be enforced during constraint solving by adding side conditions to rules (A1)-(A3) checking if the types of the accessed identifiers are not dummy variables (DumVar).

Enumeration algorithm. Fig. 9 also defines our enumeration algorithm: the relation $\rightarrow_{\mathsf{e}}$ where $\rightarrow^*_{\mathsf{e}}$ is its reflexive and transitive closure. Enumerating the minimal type errors in a piece of code consists of trying to solve diverse results of filtering the constraints generated for the piece of code. The tested filters (label sets) form the search space which is built while searching for errors. The enumeration algorithm starts with a unique filter: the empty set, to

Figure 7 Constraint solver

equality simplification

(S1) $\texttt{solve}(\Delta, \overline{d}, x{=}x) \rightarrow \texttt{succ}(\Delta)$

(S2) $\texttt{solve}(\Delta, \overline{d}, \tau\ \mu{=}\tau'\ \mu') \rightarrow \texttt{solve}(\Delta, \overline{d}, (\mu{=}\mu');(\tau{=}\tau'))$

(S3) $\texttt{solve}(\Delta, \overline{d}, \tau_1{\rightarrow}\tau_2{=}\tau_3{\rightarrow}\tau_4) \rightarrow \texttt{solve}(\Delta, \overline{d}, (\tau_1{=}\tau_3);(\tau_2{=}\tau_4))$

(S4) $\texttt{solve}(\Delta, \overline{d}, \tau_1{=}\tau_2) \rightarrow \texttt{solve}(\Delta, \overline{d}, \mu{=}\texttt{ar})$ if $\{\tau_1, \tau_2\} = \{\tau\ \mu, \tau_3{\rightarrow}\tau_4\}$ for some μ, τ, τ_3 and τ_4

(S5) $\texttt{solve}(\Delta, \overline{d}, \mu_1{=}\mu_2) \rightarrow \texttt{err}(\langle \texttt{tyConsClash}(\mu_1, \mu_2), \overline{d}\rangle)$, if $\mu_1 \neq \mu_2$ and $\{\mu_1, \mu_2\} \in \{\{\gamma, \gamma'\}, \{\gamma, \texttt{ar}\}\}$ for some γ and γ'

(S6) $\texttt{solve}(\Delta, \overline{d}, is_1{=}is_2) \rightarrow \texttt{err}(\langle \texttt{statusClash}(is_1, is_2), \overline{d}\rangle)$, if $\neg\mathsf{compatible}(is_1, is_2)$

(S7) $\texttt{solve}(\Delta, \overline{d}, is_1{=}is_2) \rightarrow \texttt{succ}(\Delta)$, if $\mathsf{compatible}(is_1, is_2)$ and $is_1, is_2 \in \mathsf{RawIdStatus}$

(S8) $\texttt{solve}(\Delta, \overline{d}, x^{\overline{d}'}{=}y) \rightarrow \texttt{solve}(\Delta, \overline{d} \cup \overline{d}', x{=}y)$

equality constraint reversing

(R) $\texttt{solve}(\Delta, \overline{d}, x{=}y) \rightarrow \texttt{solve}(\Delta, \overline{d}, y{=}x)$, if $y \in s$ and $x \notin s$, where $s = \mathsf{Var} \cup \mathsf{Dependent}$

unifier access/updating Rules (U1) through (U6) have also these common side conditions: $var \neq x$ and $y = \mathsf{build}(u, x)$

(U1) $\texttt{solve}(\langle u, e\rangle, \overline{d}, var{=}x) \rightarrow \texttt{err}(\langle \texttt{circularity}, \overline{d} \cup \mathsf{deps}(y)\rangle)$, if $var \in \mathsf{vars}(y) \setminus (\mathsf{dom}(u) \cup \mathsf{Env})$ and $var \neq \mathsf{strip}(y)$

(U2) $\texttt{solve}(\langle u, e\rangle, \overline{d}, var{=}x) \rightarrow \texttt{succ}(\langle u, e\rangle)$, if $var \in \mathsf{vars}(y) \setminus (\mathsf{dom}(u) \cup \mathsf{Env})$ and $var = \mathsf{strip}(y)$

(U3) $\texttt{solve}(\langle u, e\rangle, \overline{d}, var{=}x) \rightarrow \texttt{succ}(\langle u \uplus \{var \mapsto x^{\overline{d}}\}, e\rangle)$, if $var \notin \mathsf{vars}(y) \cup \mathsf{dom}(u) \cup \mathsf{Env}$

(U4) $\texttt{solve}(\langle u, e\rangle, \overline{d}, var{=}x) \rightarrow \texttt{succ}(\langle u' \uplus \{var \mapsto \mathsf{diff}(e, e')^{\overline{d}}\}, e\rangle)$, if $var \in \mathsf{Env} \setminus \mathsf{dom}(u)$ and $\texttt{solve}(\langle u, e\rangle, \overline{d}, x) \rightarrow^* \texttt{succ}(\langle u', e'\rangle)$

(U5) $\texttt{solve}(\langle u, e\rangle, \overline{d}, var{=}x) \rightarrow \texttt{err}(er)$, if $var \in \mathsf{Env} \setminus \mathsf{dom}(u)$ and $\texttt{solve}(\langle u, e\rangle, \overline{d}, x) \rightarrow^* \texttt{err}(er)$

(U6) $\texttt{solve}(\langle u, e\rangle, \overline{d}, var{=}x) \rightarrow \texttt{solve}(\langle u, e\rangle, \overline{d}, z{=}x)$, if $u(var) = z$

constrained environments

(C1) $\texttt{solve}(\Delta, \overline{d}, e_1;e_2) \rightarrow \texttt{solve}(\Delta', \overline{d}, e_2)$, if $\texttt{solve}(\Delta, \overline{d}, e_1) \rightarrow^* \texttt{succ}(\Delta')$

(C2) $\texttt{solve}(\Delta, \overline{d}, e_1;e_2) \rightarrow \texttt{err}(er)$, if $\texttt{solve}(\Delta, \overline{d}, e_1) \rightarrow^* \texttt{err}(er)$

dependent/empty/variables

(D) $\texttt{solve}(\Delta, \overline{d}, e^{\overline{d}'}) \rightarrow \texttt{solve}(\Delta, \overline{d} \cup \overline{d}', e)$

(N) $\texttt{solve}(\Delta, \overline{d}, \boxdot) \rightarrow \texttt{succ}(\Delta)$

(V) $\texttt{solve}(\langle u, e\rangle, \overline{d}, ev) \rightarrow \texttt{succ}(\langle u, e;\mathsf{build}(u, ev)\rangle)$

binders

(B1) $\texttt{solve}(\Delta, \overline{d}, {\downarrow}id{=}x) \rightarrow \texttt{succ}(\Delta;({\downarrow}id \overset{\overline{d}}{=} x))$

(B2) $\texttt{solve}(\Delta, \overline{d}, {\updownarrow}vid{=}\alpha) \rightarrow \texttt{solve}(\Delta, \overline{d}, {\uparrow}vid{=}\langle\alpha, \mathsf{ifNotDum}(\alpha, \mathtt{u})\rangle)$, if $\mathsf{strip}(\Delta[\![vid]\!]) \in \{\mathtt{c}, \mathtt{d}\}$

(B3) $\texttt{solve}(\Delta, \overline{d}, {\updownarrow}vid{=}\alpha) \rightarrow \texttt{succ}(\Delta;({\downarrow}vid \overset{\overline{d}\cup\overline{d}'}{=\!=} \alpha))$, if $\Delta[\![vid]\!] = is$ and $\mathsf{strip}(is) = \mathtt{v}$ and $\mathsf{deps}(is) = \overline{d}'$

(B4) $\texttt{solve}(\Delta, \overline{d}, {\updownarrow}vid{=}\alpha) \rightarrow \texttt{succ}(\Delta;({\downarrow}vid \overset{\overline{d}\cup\{vid\}}{=\!=} \langle\alpha, \mathsf{ifNotDum}(\alpha, \mathtt{u})\rangle))$, if $\mathsf{strip}(\Delta[\![vid]\!]) = \mathtt{u}$ or ($\neg\mathsf{hiding}(\Delta)$ and $\Delta[\![vid]\!]$ undefined)

(B5) $\texttt{solve}(\Delta, \overline{d}, {\updownarrow}vid{=}\alpha) \rightarrow \texttt{succ}(\Delta;({\downarrow}vid \overset{\overline{d}}{=} \langle\alpha_{\mathtt{dum}}, \mathsf{ifNotDum}(\alpha, \mathtt{p})\rangle))$, if $\mathsf{strip}(\Delta[\![vid]\!]) \in \{\mathtt{a}, \mathtt{p}\}$ or ($\mathsf{hiding}(\Delta)$ and $\Delta[\![vid]\!]$ undefined)

accessors

(A1) $\texttt{solve}(\Delta, \overline{d}, {\uparrow}vid{=}\alpha) \rightarrow \texttt{solve}(\Delta, \overline{d}, \tau[ren]{=}\alpha)$, if $\Delta(vid) = \forall\overline{\alpha}.\,\tau$ and $\mathsf{dom}(ren) = \overline{\alpha}$ and $\mathsf{dja}(\mathsf{vars}(\langle\Delta, \alpha\rangle) \setminus \{\alpha_{\mathtt{dum}}\}, \mathsf{ran}(ren))$

(A2) $\texttt{solve}(\Delta, \overline{d}, {\uparrow}vid{=}ris) \rightarrow \texttt{solve}(\Delta, \overline{d}, is{=}ris)$, if $\Delta[\![vid]\!] = is$

(A3) $\texttt{solve}(\Delta, \overline{d}, {\uparrow}id{=}var) \rightarrow \texttt{solve}(\Delta, \overline{d}, x{=}var)$, if $\Delta(id) = x$ and x is not of the form $\forall\overline{\alpha}.\,\tau$

(A4) $\texttt{solve}(\Delta, \overline{d}, {\uparrow}id{=}x) \rightarrow \texttt{succ}(\Delta)$, if ($x \in \mathsf{IdStatus}$ and $\Delta[\![id]\!]$ undefined) or ($x \notin \mathsf{IdStatus}$ and $\Delta(id)$ undefined)

polymorphic environments

(P1) $\texttt{solve}(\langle u_1, e_1\rangle, \overline{d}, \mathsf{poly}(e)) \rightarrow \texttt{succ}(\mathsf{toPoly}(\langle u_2, e_1\rangle, e'))$, if $\texttt{solve}(\langle u_1, e_1\rangle, \overline{d}, e) \rightarrow^* \texttt{succ}(\langle u_2, e_2\rangle)$ and $\mathsf{diff}(e_1, e_2) = \boxdot;e'$

(P2) $\texttt{solve}(\langle u_1, e_1\rangle, \overline{d}, \mathsf{poly}(e)) \rightarrow \texttt{succ}(\langle u_2, e_2\rangle)$, if $\texttt{solve}(\langle u_1, e_1\rangle, \overline{d}, e) \rightarrow^* \texttt{succ}(\langle u_2, e_2\rangle)$ and $\mathsf{diff}(e_1, e_2) = \boxdot$

(P3) $\texttt{solve}(\langle u_1, e_1\rangle, \overline{d}, \mathsf{poly}(e)) \rightarrow \texttt{err}(er)$, if $\texttt{solve}(\langle u_1, e_1\rangle, \overline{d}, e) \rightarrow^* \texttt{err}(er)$

Figure 8 Constraint filtering

$$\mathsf{filt}(e^l, \overline{l}_1, \overline{l}_2) = \begin{cases} e^l, & \text{if } l \in \overline{l}_1 \setminus \overline{l}_2 \\ \mathsf{dum}(e), & \text{if } l \in \overline{l}_2 \\ \boxdot, & \text{otherwise} \end{cases}$$

$\mathsf{filt}(ev{=}e, \overline{l}_1, \overline{l}_2) = (ev{=}\mathsf{filt}(e, \overline{l}_1, \overline{l}_2))$

$\mathsf{filt}(e_1;e_2, \overline{l}_1, \overline{l}_2) = \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2);\mathsf{filt}(e_2, \overline{l}_1, \overline{l}_2)$

$\mathsf{filt}(\mathtt{poly}(e), \overline{l}_1, \overline{l}_2) = \mathtt{poly}(\mathsf{filt}(e, \overline{l}_1, \overline{l}_2))$

$\mathsf{filt}(\boxdot, \overline{l}_1, \overline{l}_2) = \boxdot$

$\mathsf{dum}({\downarrow}id{=}x) = ({\downarrow}id{=}\mathsf{toDumVar}(x))$

$\mathsf{dum}({\updownarrow}id{=}x) = ({\updownarrow}id{=}\mathsf{toDumVar}(x))$

$\mathsf{dum}(ev) = ev_{\mathtt{dum}}$

$\mathsf{dum}(c) = \boxdot$

$\mathsf{dum}(acc) = \boxdot$

$\mathsf{toDumVar}(\sigma) = \alpha_{\mathtt{dum}}$

$\mathsf{toDumVar}(\mu) = \delta_{\mathtt{dum}}$

$\mathsf{toDumVar}(e) = ev_{\mathtt{dum}}$

$\mathsf{toDumVar}(\alpha) = \alpha_{\mathtt{dum}}$

$\mathsf{toDumVar}(is) = \mathtt{a}$

Figure 9 Minimisation and enumeration algorithms

minimisation

(M1) $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2\rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_1 \cap \overline{d}, \overline{l}_2 \cap \overline{d}\rangle$, if $\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\}) \xrightarrow{\mathsf{isErr}} \langle ek, \overline{d}\rangle$

(M2) $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2\rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_1 \cup \{l\}, \overline{l}_2\rangle$, if $\mathsf{solvable}(\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\}))$

(M3) $\langle e, er\rangle \xrightarrow{\mathsf{min}} er'$, if $\mathsf{IBinds}(e) = \overline{l}$ and $\langle e, \mathsf{labs}(er) \setminus \overline{l}, \mathsf{labs}(er) \cap \overline{l}\rangle \rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_1, \varnothing\rangle$ and $\langle e, \varnothing, \overline{l}_1\rangle \rightarrow^*_{\mathsf{test}} \langle e, \overline{l}_2, \varnothing\rangle$ and $\mathsf{filt}(e, \overline{l}_2, \varnothing) \xrightarrow{\mathsf{isErr}} er'$

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

(E1) $\mathtt{enum}(e) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \varnothing, \{\varnothing\})$

(E2) $\mathtt{enum}(e, \overline{er}, \varnothing) \rightarrow_{\mathsf{e}} \mathtt{errors}(\overline{er})$

(E3) $\mathtt{enum}(e, \overline{er}, \overline{\overline{l}} \uplus \{\overline{l}\}) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \overline{er}, \overline{\overline{l}})$, if $\mathsf{solvable}(\mathsf{filt}(e, \mathsf{labs}(e), \overline{l}))$

(E4) $\mathtt{enum}(e, \overline{er}, \overline{\overline{l}} \uplus \{\overline{l}\}) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \overline{er} \cup \{\langle ek, \overline{d}\rangle\}, \overline{\overline{l}}' \cup \overline{\overline{l}})$, if $\mathsf{filt}(e, \mathsf{labs}(e), \overline{l}) \xrightarrow{\mathsf{isErr}} er$ and $\langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \overline{d}\rangle$ and $\overline{\overline{l}}' = \{\overline{l} \cup \{l\} \mid l \in \overline{d} \wedge \forall\overline{l}_0 \in \overline{\overline{l}}.\ \overline{l}_0 \not\subseteq \overline{l} \cup \{l\}\}$

solve all the generated constraints. Then, when an error is found and minimised, the labels of the error are used to build new filters (see $\overline{\overline{l}}$ in rule (E4)). Once all the filters are exhausted the enumeration algorithm stops. The found errors are then all the minimal type errors of the analysed piece of code (see rule (E2)). For example, assume that $sdec \Rightarrow e$ for an untypable given piece of

code $sdec$. Then, the first enumeration state is (see rule (E1)): $\texttt{enum}(e, \varnothing, \{\varnothing\})$ where the first empty set is the set of found errors (empty at the beginning) and where the second empty set is the first filter. Because $sdec$ is untypable, the constraint solver fails and returns a type error er. The minimisation algorithm minimises er and returns a minimal error er' such that $er'(1) \subseteq er(1)$. The error er' can be er if it was already in a minimal form when found by the enumerator. New filters are then computed based on the filter used to find this new error ($\varnothing$ in our example) and the new error itself (er'): $\{\{l\} \mid l \in er'(1)\}$. The enumeration keeps searching for errors using this updated search space: the new state is $\texttt{enum}(e, \{er'\}, \{\{l\} \mid l \in er'(1)\})$. For the next step, one of the $\{l\}$ where $l \in er'(1)$ will be picked as the filter to try to find another error. When a filter leads to a solvable filtered environment, the filter is discarded (rule (E3)) otherwise the filter is used to update the search space as explained above (rule (E4))

4.6 Slicing. The last phase of our TES consists in the computation of a minimal type error slice from an untypable piece of code and a minimal error found by the enumeration algorithm. The nodes labelled by the labels not involved in the error are discarded and replaced by "dot" terms. For example, if we remove the node associated to the label l_2 (the unit expression) in $\lceil 1^{l_1}\ ()^{l_2} \rceil^{l_3}$ then we obtain $\lceil 1^{l_1}\ \texttt{dot-e}(\varnothing) \rceil^{l_3}$, displayed as `1 ⟨..⟩` in our implementation. Dots are used as a visually convenient way to show that information has been discarded. Fig. 10 extends our syntax and constraint generator to "dot" terms. Our constraint generator is extended to dot terms so that every piece of (our extended) syntax can be type checked (by generating constraints and by then solving the constraints), which is needed to state our minimality criteria in Sec. 4.8. We call *slice*, any syntactic form that can be produced using the grammar rules defined in Fig. 3 and Fig 10 combined. We call *type error slice*, any slice for which our constraint generation algorithm (defined in Fig 4 and Fig. 10 combined) only generates unsolvable constraints. Let us restrict our slice definitions to structure declarations. Formally, a slice is a $sdec$ and a type error slice is a $sdec$ such that $\neg\mathsf{solvable}(sdec)$.

Flattening. Turning nodes not participating in errors into dot nodes is not enough. Our slicing algorithm uses two tidying functions flat and tidy. The flattening function flat flattens sequences of parts (pt). For example, flattening $\langle \texttt{..1..}\langle \texttt{..()..}\rangle \texttt{..}\rangle$ results in $\langle \texttt{..1..()..}\rangle$. Not all nested dot terms are flattened: in order not to mix up bindings in a slice, we do not allow a declaration to be an ept (expression term as opposed to a declaration) and only allow ept's to be flattened, so that declarations cannot escape the scope defined by a dot term. For example, we do not flatten $\langle \texttt{..val x = ()..}\langle \texttt{..val x = 1..}\rangle \texttt{..x + 1..}\rangle$ to $\langle \texttt{..val x = ()..val x = 1..x + 1..}\rangle$ because they have different semantics: the first slice is not typable but the second is. Let flat be defined as follows (where x can be any of e, p, s, d):

$$\mathsf{flat}(\langle\rangle) = \langle\rangle$$
$$\mathsf{flat}(\langle pt\rangle @ \overrightarrow{pt}) = \begin{cases} \overrightarrow{ept} @ \mathsf{flat}(\overrightarrow{pt}), & \text{if } pt = \texttt{dot-}x(\overrightarrow{ept}) \\ \langle pt\rangle @ \mathsf{flat}(\overrightarrow{pt}), & \text{otherwise} \end{cases}$$

The function tidy tidies sequences of structure declarations ($sdec$) when slicing structure expressions:

$$\mathsf{tidy}(\langle\rangle) = \langle\rangle$$
$$\mathsf{tidy}(\langle \texttt{dot-d}(\overrightarrow{ept}), \texttt{dot-d}(\overrightarrow{pt})\rangle @ \overrightarrow{sdec}) = \mathsf{tidy}(\langle \texttt{dot-d}(\overrightarrow{ept} @ \overrightarrow{pt})\rangle @ \overrightarrow{sdec})$$
$$\mathsf{tidy}(\langle \texttt{dot-d}(\varnothing)\rangle @ \overrightarrow{sdec}) = \mathsf{tidy}(\overrightarrow{sdec}), \text{ if } \overrightarrow{sdec} \text{ not of the form } \texttt{dot-d}(\overrightarrow{ept}) @ \overrightarrow{sdec'}$$
$$\mathsf{tidy}(\langle sdec\rangle @ \overrightarrow{sdec}) = \langle sdec\rangle @ \mathsf{tidy}(\overrightarrow{sdec}), \text{ if none of the above applies}$$

Algorithm. Our slicing algorithm can be presented in a simple fashion if our syntax forms defined in Fig. 3 and Fig. 10 are regarded as abstract syntax trees. In such a tree $tree$, leaves are identifiers id and otherwise a node is labelled by a node kind $node$ and a label l (denoted $node^l\langle tree_1, \ldots, tree_n\rangle$). Using this notation, we define out slicing function sl in Fig. 11.

4.7 Overall algorithm for Type Error Slicing. First, given a SML structure declaration $sdec$, our constraint generation algorithm defined in Fig. 4 generates constraints structured in an environment e. Then, type errors of e are enumerated using the enumeration algorithm defined in Fig. 9. Once an error is found by the enumeration algorithm, it is minimised using the minimisation algorithm also defined in Fig. 9. Then a slice is computed from the minimised error and the original piece of code using the slicing algorithm defined in Fig. 11. Both enumeration and minimisation rely on the constraint solver defined in Fig. 7. The computed type error slices are finally reported to the user. In addition to a type error slice, a type error report also includes a highlighting of the slice in the SML user code, a message explaining the kind of the error (see Fig. 5), and a set of identifier status context dependencies. Formally, our overall algorithm tes is defined as follows:

$$\begin{aligned} \mathsf{tes}(sdec) = \{\langle sdec', ek, \overline{vid}\rangle \mid\ & sdec \triangleright e \\ & \wedge\ \texttt{enum}(e) \rightarrow^* \texttt{errors}(\overline{er}) \\ & \wedge\ \langle ek, \overline{l} \cup \overline{vid}\rangle \in \overline{er} \\ & \wedge\ \mathsf{sl}(sdec, \overline{l}) = sdec'\} \end{aligned}$$

4.8 Minimality. Let us informally define the function bindings on environments. This function uses a modified version of our constraint solver that keeps track of the bindings generated by the accessor rules ((A1)-(A3)). Given a piece of code, bindings, using the constraint generation algorithm, generates an environment, filters out all the labelled equality constraint in the generated environment, runs the modified constraint solver on the filtered environment, and finally returns the recorded bindings. For example, if exp is `let val x = true in let val x = 1 in x end end`, and the label l_i is associated to the ith occurrence of x then $\mathsf{bindings}(exp) = \{\langle l_2, l_3\rangle\}$.

We define the sub-slice relation as follows: $sdec_1 \sqsubseteq_{\overline{l}} sdec_2$ iff $\mathsf{sl}(sdec_2, \overline{l}) = sdec_1$ and $\mathsf{bindings}(sdec_1) \subseteq \mathsf{bindings}(sdec_2)$.

We say $sdec_2$ is a minimal type error slice of $sdec_1$ iff $sdec_2 \sqsubseteq_{\overline{l}} sdec_1$, $\neg\mathsf{solvable}(sdec_2)$ and for all $sdec'$ if $sdec' \sqsubseteq_{\overline{l}'} sdec_2$ and $sdec' \neq sdec_2$ for some $\overline{l}'$ then $\mathsf{solvable}(sdec')$.

We consider minimality as a design principle for our TES even though minimal slices do not always seem to be the correct answer to type error reporting. For example for record field name clashes, we do not want to provide minimal slices, as presented in Sec. 2.3.

For the subset of our TES presented in the present paper, we believe the following holds: a slice $sdec'$ is a minimal slice of $sdec$ iff $\langle sdec', ek, \overline{vid}\rangle \in \mathsf{tes}(sdec)$. We do not formally prove this statement for diverse reasons. First, our TES is constantly being updated and proving the minimality of one of its versions would not guaranty the minimality of the others. Then, as mentioned above, minimality is only a design principle. Let us finally stress that we feel improving the range and quality of our slices is more important than ensuring their minimality in particular.

4.9 Design principles. While developing our type error slicer we discovered, developed, and followed the following principles. *Constraint terms* are those pieces of syntax that can occur anywhere inside a constraint. In our system, this is any is, μ, τ, σ, or e.

1. Each syntactic sort of constraint terms should have a case ranging over an infinite variable set. This allows incomplete information in every possible place in constraints, which allows considering every possible way of slicing away parts of the program

Figure 10 Extension of our syntax and constraint generator to "dot" terms

extension of the syntax

$pt \in \mathsf{Part} ::= ept \mid sdec$ ConBind $::= \cdots \mid \texttt{dot-e}(\overrightarrow{pt})$ AtExp $::= \cdots \mid \texttt{dot-e}(\overrightarrow{pt})$ Pat $::= \cdots \mid \texttt{dot-p}(\overrightarrow{pat})$

$ept \in \mathsf{ExpPart} ::= exp \mid ty \mid sexp \mid pat$ DatName $::= \cdots \mid \texttt{dot-e}(\overrightarrow{pt})$ Exp $::= \cdots \mid \texttt{dot-e}(\overrightarrow{pt})$ StrDec $::= \cdots \mid \texttt{dot-d}(\overrightarrow{pt})$

Ty $::= \cdots \mid \texttt{dot-e}(\overrightarrow{pt})$ Dec $::= \cdots \mid \texttt{dot-d}(\overrightarrow{pt})$ AtPat $::= \cdots \mid \texttt{dot-p}(\overrightarrow{pat})$ StrExp $::= \cdots \mid \texttt{dot-s}(\overrightarrow{pt})$

extension of the constraint generator

Parts	$ept \triangleright e$	$\Leftarrow ept \triangleright \langle var, e\rangle$
Declarations	$\texttt{dot-d}(\langle pt_1, \ldots, pt_n\rangle) \triangleright [e_1;\cdots;e_n]$	$\Leftarrow pt_1 \triangleright e_1 \wedge \cdots \wedge pt_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n)$
Patterns	$\texttt{dot-p}(\langle pat_1, \ldots, pat_n\rangle) \triangleright \langle \alpha, e_1;\cdots;e_n\rangle$	$\Leftarrow pat_1 \triangleright e_1 \wedge \cdots \wedge pat_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \alpha)$
Structure expressions	$\texttt{dot-s}(\langle pt_1, \ldots, pt_n\rangle) \triangleright \langle ev, [e_1;\cdots;e_n]\rangle$	$\Leftarrow pt_1 \triangleright e_1 \wedge \cdots \wedge pt_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev)$
Expressions/Types/Constructor bindings/Datatype names		
	$\texttt{dot-e}(\langle pt_1, \ldots, pt_n\rangle) \triangleright \langle \alpha, [e_1;\cdots;e_n]\rangle$	$\Leftarrow pt_1 \triangleright e_1 \wedge \cdots \wedge pt_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \alpha)$

Figure 11 Slicing algorithm

$node \in \mathsf{Node} ::= \texttt{tyV} \mid \texttt{tyC} \mid \texttt{tyA} \mid \texttt{cbN} \mid \texttt{cbU} \mid \texttt{dn} \mid \texttt{decV} \mid \texttt{decD} \mid \texttt{decO} \mid \texttt{expI} \mid \texttt{expL} \mid \texttt{expF} \mid \texttt{expA} \mid \texttt{patI} \mid \texttt{patA} \mid \texttt{sdec} \mid \texttt{sexpI} \mid \texttt{sexpS}$

The node kinds correspond to the cases of the external syntax grammar. In particular, `sexpS` is the kind for a structure expression of the form $\texttt{struct}^l\ sdec_1 \cdots sdec_n\ \texttt{end}$. In (SL3) and (SL7), x is chosen depending on the argument provided to the slicing function. For example, in (SL3), if $node = \texttt{sexpS}$ then x is `s`.

(SL1) $\mathsf{sl}(node^l\ \langle tree_1, \ldots, tree_n\rangle, \overline{l}) = node^l\ \langle \mathsf{sl}_1(tree_1, \overline{l}), \ldots, \mathsf{sl}_1(tree_n, \overline{l})\rangle$, if $l \in \overline{l}$ and $node \neq \texttt{sexpS}$ or $\mathsf{sl}_1(tree_1, \overline{l}) = node'^{l'}\ \overrightarrow{tree}$ where $node' \in \{\texttt{patI}, \texttt{patA}\}$

(SL2) $\mathsf{sl}(node^l\ \langle tree_1, \ldots, tree_n\rangle, \overline{l}) = node^l\ \mathsf{tidy}(\langle \mathsf{sl}_1(tree_1, \overline{l}), \ldots, \mathsf{sl}_1(tree_n, \overline{l})\rangle)$, if $l \in \overline{l}$ and $node = \texttt{sexpS}$

(SL3) $\mathsf{sl}(node^l\ \langle tree_1, \ldots, tree_n\rangle, \overline{l}) = \texttt{dot-}x(\mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \overline{l}), \ldots, \mathsf{sl}_2(tree_n, \overline{l})\rangle))$, if none of the above applies

(SL4) $\mathsf{sl}_1(node^l\ \overrightarrow{tree}, \overline{l}) = \mathsf{sl}(node^l\ \overrightarrow{tree}, \overline{l})$ (SL6) $\mathsf{sl}_1(id, \overline{l}) = id$

(SL5) $\mathsf{sl}_2(node^l\ \overrightarrow{tree}, \overline{l}) = \mathsf{sl}(node^l\ \overrightarrow{tree}, \overline{l})$ (SL7) $\mathsf{sl}_2(id, \overline{l}) = \texttt{dot-}x(\langle\rangle)$.

syntax tree. This is essential to get precise type error slices that include all relevant details and exclude the irrelevant.

For us, this means the sorts μ, τ, and e have the variable cases δ α, and ev. Our implementation has a variable case of raw identifier statuses (ris) which is omitted from this paper to save space.

2. Each syntactic sort of constraint terms should have a dependency annotation case. This allows precise blame tracking, which in turn enables precise slicing, which we already motivate above.

For us, this means the sorts is, μ, τ, and e have the dependency cases $\langle is, \overline{d}\rangle$, $\langle \mu, \overline{d}\rangle$, $\langle \tau, \overline{d}\rangle$, and $\langle e, \overline{d}\rangle$. We omit type scheme dependencies (σ) because handling schemes is already complex and only dependencies on plain types (τ) are needed in this paper.

3. In our system, when processing a program syntax tree node, a constraint generation rule will return a main result (either a type or an environment) and in some cases also an environment result (used for constraints and bindings when the main result is not an environment). The rule may connect information from the results for the node's subtrees to the other subtrees or to the node's results.

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

A good example in our system is rule (G18) that handles structures (SML's modules). The environment for the structure is built by the unlabelled constraint $ev' = (e_1;\cdots;e_n)$. This "deep" constraint holds a complex structure in order to pack together a sequence of environments from the declarations making up a structure body. The structure environment is connected to the main result by the labelled shallow constraint $ev \overset{l}{=} ev'$.

4. Duplicating constraints should be unnecessary.

This seems obvious, but some previous constraint formalisms seem to be too weak to allow the needed sharing. Again, we use the example of rule (G18), where our system allows the environment for the structure can be written as the sequential composition of the environments for the component declarations: $e_1;\cdots;e_n$. Here the environment e_1 from the first declaration is available both in the subsequent declarations and also in the result (provided its bindings are not shadowed). A previous version of our system had a weaker constraint system with let-constraints similar to those of Pottier and Rémy [24], and the best solution we could devise required duplicating the environments for the pattern portion of each declaration, which resulted in an exponential slowdown.

5. Dependencies must be propagated during constraint solving exactly when needed. If dependencies are not propagated to places they should be, minimization will over-minimize producing non-errors. This can be detected. More insidiously, propagating dependencies where they are unneeded will keep alive unneeded parts of error slices much longer during minimization, resulting in serious slowdowns. Because correct results are eventually produced, detecting such bugs is harder so this issue requires great care.

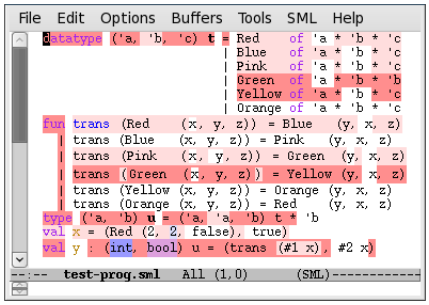
For example, an earlier version of our constraint solver copied dependencies from declarations in a structure to the structure's result, forcing the minimizer to remove each declaration one at a time. Debugging was hard because only speed suffered.

More generally, the constraint solver should be designed to produce error slices (before minimization) that are as close to minimal as can be reasonably achieved. If constraint solving yields a non-minimal error slice, then solving steps must have annotated a constraint with a location on which it does not uniquely depend.

5. Implementation details

5.1 Supported features. Our TES handles most SML features, e.g., structures, signatures, datatype replications (handled like *open*), operator and constant overloading, many uses of functors, etc. We also handle imperative features such as exceptions and the value polymorphism restriction. We also slice context-sensitive syntax errors, which comes naturally from handling identifier statuses and doing context-independent type checking, e.g., `x` occurring twice in the pattern in `fn (x, x) => x` is an error only if `x` has value variable status. We do not yet handle fixity changes. Type

Figure 12 Highlighting of an SML type error in Emacs



and structure sharing is incomplete. Some errors involving equality types and flexible record patterns are not reported.

5.2 Performance. Our implementation is currently usable for small projects (a few thousand lines) and is steadily improving. Our latest constraint system and solver is 10 to 100 times faster in many cases than before we switched to using constrained environments ($e_2; e_1$). Our previous TES version was already enormously faster than the original by Haack and Wells due to avoiding duplication of polymorphic types.

5.3 User interface. An Emacs interface (and a preliminary one for Vim) highlights slices in the edited source code. There is also a terminal command-line interface. Figure 12 presents a screenshot of the type error presented in Sec. 2.1 highlighted in Emacs. The light pink corresponds to slices other than the focused one.

5.4 The Standard ML basis library. Our examples have used operators like `::` and `+`. For now, we allow defining the Standard ML basis in a file, and we provide a file declaring a portion of the basis. For the future, we have begun implementing a way to use library types extracted from a running instance of SML/NJ, but there are still technical challenges to overcome.

6. Related work

6.1 Methods making use of slices. After the first version of TES presented by Haack and Wells, many researchers began to present type errors as program slices obtained from unsolvable sets of constraints.

Neubauer and Thiemann [21] use flow analysis to compute type dependencies for a small ML-like language to report type errors. Their system uses discriminative sum types and can analyze any term. Their first step ("collecting phase") labels the studied term and infers type information. This analysis generates a set of program point sets. These program points are directly stored in the discriminative sum types. A conflicting type ("multivocal") is then paired with the locations responsible for its generation. Their second step ("reporting phase") consists in generating error reports from the conflicts generated during the first phase. Slices are built from which highlighting are produced. An interesting detail is that a type derivation can be viewed as the description of all type errors in an untypable piece of code, from which another step then computes error reports.

Similar to ours is work by Stuckey, Sulzmann and Wazny [27, 29] (based on earlier work without slices [25, 26]). They do type inference, type checking and report type errors for the Chameleon language (a modified Haskell subset). Chameleon includes algebraic data types, type-class overloading, and functional dependencies. They code the typing problem into a constraint problem and attach labels to constraints to track program locations and highlight parts of untypable pieces of code. First they compute a minimal unsatisfiable set of generated constraints from which they select one of the type error locations to provide their type explanation. They finally provide a highlighting and an error message depending on the selected location. They provide slice highlighting but using a different strategy from ours. They focus on explaining conflicts in the inferred types at one program point inside the error location set. It is not completely clear, but they do not seem to worry much about whether the program text they are highlighting is exactly (no more and no less) a complete explanation of the type error. For example, they do not highlight applications because "they have no explicit tokens in the source code". It is then stated: "We leave it to the user to understand when we highlight a function position we may also refer to its application". This differs from our strategy because we think it is preferable to highlight all the program locations responsible for an error even if these are white spaces. Moreover, they do not appear to highlight the parts of datatype declarations relevant to type errors.

When running on a translation of the code presented in Sec. 2.1 into Haskell, ChameleonGecko outputs the error report partially displayed below (the rest of the output seems to be internal information computed during unification). This highlighting identifies the same error location as SML/NJ and would not help solve the error.

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

Significantly, because they handle a Haskell-like language, they face challenges for accurate type error location that are different from the ones for SML.

Gast [9] generates "detailed explanations of ML type errors in terms of data flows". His method is in three steps: generation of subtyping constraints annotated by reasons for their generation; gathering of reasons during constraint unification; transformation of the gathered reasons into explanations by data flows. He provides a visually convenient display of the data flows with arrows in XEmacs. Gast's method (which seems to be designed only for a small portion of OCaml) can be considered as a slicing method with data flow explanations.

Braßel [7] presents a generic approach (implemented for the language Curry) for type error reporting that consists in two different procedures. The first one tries to replace portions of code by dummy terms that can be assigned any type. If an untypable piece of code becomes typable when one of its subtrees has been replaced by a dummy term then the process goes on to apply the same strategy inside the subtree. The second procedure consists in the use of a heuristic to guide the search of type errors. The heuristic is based on two principles: it will always "prefer an inner correction point to an outer one" and will always "prefer the point which is located in a function farther away in the call graph from the function which was reported by the type checker as the error location". Braßel's method does not seem to compute proper slices but instead singles out different locations that might be the cause of a type error inside a piece of code.

6.2 Significant non-slicing type explanation methods. Heeren et al. designed a method used in the Helium project [15, 14, 16, 12] to provide error messages for the Haskell language relying on a constraint-based type inference. First, a constraint graph is generated from a piece of code. For an ill-typed piece of code, a conflicting path called an inconsistency is extracted from the constraint graph. Such a conflicting path is a structured unsolvable set of type constraints. Heuristics are used to remove inconsistencies. To each type constraint is associated a trust value and depending on these values and the defined heuristics, some constraints are discarded until the inconsistency is removed. They also propose some "program correcting heuristics" used to search for a typable piece of code from an untypable one. Such a heuristic is for example the permutation of parameters which is a common mistake in program-

ming. Their approach has been used with students learning functional programming. Using pieces of code written by students and their expertise of the language they refined their heuristics. This approach differs from ours by privileging locations over others by the use of some heuristics. They do not compute minimal slices and highlightings.

We present below the most interesting part of the error report obtained using Helium on a translation of the code presented in Sec. 2.1 into Haskell. It is reported that `x1` and `trans` don't have the expected types. The application, which is at the end of the code, is then blamed when our programming error is at the very beginning of the code.

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

Compilation failed with 1 error
```

They have also recently tackled the task to report type errors for Java [5, 6]. Error reports provided by usual compilers can be of little help, especially in the presence of generics. El Boustani and Haage try to do a better job by keeping track of more information during type checking. Having more information at hands when analysing an untypable piece of code allows a more global view of its type errors and leads to more informative error reports. The main difference between type error reporting for SML and for Java is that in Java "types are instantiated based on local information only and not through a long and complicated sequence of unifications" [5].

Lerner, Flower, Grossman and Chambers [19] present type error messages by constructing well-typed programs from ill-typed ones using different techniques (like Heeren et al. [12]), e.g., switching two parameters. Automatically conceived modifications to the ill-typed piece of code are checked for typability. They target Caml, and also developed a prototype for C++. The new typable generated code is presented as possible code that the programmer might have intended. It could be interesting to study the combination of this with TES.

7. Conclusion

7.1 Summary of contributions.

1. We solve a previous efficiency problem of TES (combinatorial explosion of the number of generated constraints) and also support features such as declaration sequences, structures, and *open* with the techniques of constrained environments and environment variables.

2. We solve SML's identifier status ambiguity (value variable vs. datatype constructor) while also computing minimal type error slices by using type constraints with context dependencies on identifier statuses.

3. We solve many other problems to provide clear and helpful type error slices for many different kinds of SML errors, which have been carefully designed to provide just the information the programmer needs.

4. This paper reports for a stripped-down core of SML the essential technical details of the TES machinery that solve the trickiest of the above-mentioned problems, and discusses some implementation issues.

5. We have an implementation that covers most of the SML language. A web demo is available, and there are downloadable packages for Ubuntu (covering also some other Debian-based systems) and CentOS (covering also some other Red-Hat-based systems). The web site is: `http://www.macs.hw.ac.uk/ultra/compositional-analysis/type-error-slicing`. We also have another implementation that faithfully implements just this paper.

7.2 Future work.

We have already implemented some merging of minimal slices and are extending this idea to other kind of errors than record clashes, such as for unmatched signature specifications.

In the near future, we plan to finish extending our TES to the full SML language. This includes finishing handling the key feature of functors and less vital features such as flexible records or equality types, which can cause errors we currently do not detect.

We also plan to extend our ideas to other languages such as the F# programming language or the C++ template language.

Finally, we have done user evaluations and begun designing proper scientific experiments to compare the effectiveness in improving the productivity of real users of TES vs. more traditional type error messages.

References

[1] A. W. Appel. A critique of Standard ML. *J. Funct. Program.*, 3(4), 1993.

[2] M. Beaven, R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4), 1993.

[3] M. Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, 1997.

[4] M. Blume. Dependency analysis for Standard ML. *ACM Trans. Program. Lang. Syst.*, 21(4), 1999.

[5] N. E. Boustani, J. Hage. Improving type error messages for generic java. In *PEPM*. ACM, 2009.

[6] N. E. Boustani, J. Hage. Corrective hints for type incorrect generic java programs. In *PEPM*. ACM, 2010.

[7] B. Braßel. TypeHope - there is hope for your type errors. In IFL04 [17].

[8] L. Damas, R. Milner. Principal type-schemes for functional programs. In *POPL82*, New York, NY, USA, 1982. ACM.

[9] H. Gast. Explaining ML type errors by data flows. In IFL04 [17].

[10] C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *ESOP*, vol. 2618 of *LNCS*. Springer, 2003.

[11] C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3), 2004.

[12] J. Hage, B. Heeren. Heuristics for type error discovery and recovery. In *18th Int'l Symp., IFL 2006*, vol. 4449 of *LNCS*. Springer, 2007.

[13] R. Harper. Programming in Standard ML, 2009. Working draft of August 20, 2009.

[14] B. Heeren, J. Hage. Type class directives. In *7th Int'l Symp., PADL 2005*, vol. 3350 of *LNCS*. Springer, 2005.

[15] B. Heeren, J. Jeuring, D. Swierstra, P. A. Alcocer. Improving type-error messages in functional languages. Technical report, Utrecht University, 2002.

[16] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005.

[17] *16th Int'l Workshop, IFL 2004*, vol. 3474 of *LNCS*. Springer, 2005.

[18] O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4), 1998.

[19] B. S. Lerner, M. Flower, D. Grossman, C. Chambers. Searching for type-error messages. In *ACM SIGPLAN 2007 Conference PLDI*. ACM, 2007.

[20] B. J. McAdam. On the unification of substitutions in type inference. In *10th Int'l Workshop, IFL'98*, vol. 1595 of *LNCS*. Springer, 1999.

[21] M. Neubauer, P. Thiemann. Discriminative sum types locate the source of type errors. In *8th ACM SIGPLAN Int'l Conference, ICFP 2003*. ACM, 2003.

[22] M. Odersky, M. Sulzmann, M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1), 1999.

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

[24] F. Pottier, D. Rémy. The essence of ML type inference. In B. C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, chapter 10. MIT Press, 2005.

[25] P. J. Stuckey, M. Sulzmann, J. Wazny. Interactive type debugging in haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2003. ACM.

[26] P. J. Stuckey, M. Sulzmann, J. Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2004. ACM.

[27] P. J. Stuckey, M. Sulzmann, J. Wazny. Type processing by constraint reasoning. In *4th Asian Symp., APLAS 2006*, vol. 4279 of *LNCS*. Springer, 2006.

[28] M. Wand. Finding the source of type errors. In *13th ACM SIGACT-SIGPLAN Symp., POPL'86*, New York, NY, USA, 1986. ACM.

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

[30] J. Yang. Explaining type errors by finding the source of a type conflict. In *1st Workshop, SFP'99*, Exeter, UK, 2000. Intellect Books.

[31] J. Yang, G. Michaelson, P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4), 2002.

[32] J. Yang, J. Wells, P. Trinder, G. Michaelson. Improved type error reporting. In *12th Int'l Workshop, IFL 2000*, vol. 2011 of *LNCS*. Springer, 2001.

A. Additional details for Section 4

A.1 Details for Section 4.1 (External syntax)

Remark about syntactic restrictions on our external syntax. Note that we do not enforce all the syntactic restrictions of the SML syntax. For example, in SML, in a recursive declaration such as `val rec` $pat \stackrel{l}{=} exp$, the expression exp must be a `fn`-expression.

A.2 Details for Section 4.2 (Constraint syntax)

Comparison with Pottier and Rémy's let-constraints. Our constraint system has evolved through many versions. One earlier version of our constraint system had a kind of constraint that was very close to the let-constraints[1] of systems of Pottier and Rémy [24, 23]. Pottier and Rémy define one system [24] that is an instance of HM(X) [22], and Pottier defines a very similar system [23] as a variation of the Damas/Milner type system. In our discussion, we will collectively refer to these two systems as the PR (Pottier/Rémy) system and ignore their technical differences, although our presentation will follow more closely the presentation of Pottier and Rémy [24].

In PR, a constraint can, among other things, be a let-constraint, a subtyping constraint, a type scheme instantiation constraint, a conjunction of constraints, or the constant (and satisfied) `true` constraint. A PR let-constraint looks like **let** $id{:}\dot{\sigma}$ **in** C where $\dot{\sigma}$ ranges over type schemes, and C ranges over constraints. In PR, type schemes are of the form $\forall\overline{X}[C].T$ where $\overline{X}$ is a type variable set, C is a constraint, and T is a type. We borrow for our discussion two abbreviations that Pottier and Rémy define: (1) the form $\forall\overline{X}.T$ stands for the type scheme $\forall\overline{X}[\texttt{true}].T$, and (2) the form **let** $id{:}T$ **in** C stands for **let** $id{:}\forall\varnothing.T$ **in** C.

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

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

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

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

The key point is that one can get the effect of making the appropriate number of copies of C and T while keeping the size of the constraint proportional to the program size. The constraints will need to be copied and each copy solved independently, but each copy can be solved immediately before the next copy is made, avoiding an exponential increase in the amount of memory used during constraint solving. To get the full benefit of this, an implementation should be eager in simplifying C and calculating T as much as possible before making any copies. (In our application, it could be good to also be lazy in simplifying and calculating only those portions of C and T that are actually needed by the uses of id, because our type error slicer needs to spend most of its time finding minimal portions of unsatisfiable constraints. We leave investigating this idea for future work.)

In our latest system, the equivalent of let-constraints can be represented as a special case of what our system supports. Informally, a let-constraint of the form **let** $id{:}\forall\overline{X}[C_1].T$ **in** C_2 generated for a SML recursive `let`-binding would be represented in our system by (using a combination of rules (G2) and (G12) in Fig. 4)

$$[\texttt{poly}((\downarrow id{=}\langle\tau, \texttt{v}\rangle); e_1); e_2]$$

where C_i is represented by e_i and T is represented by τ. (Let-constraints generated for other SML forms would not necessarily get the same representation.) There is no explicit representation of $\overline{X}$ in the representation in our system; instead the correct set of type variables that can be quantified is calculated by toPoly which generates type schemes when it handles environments of the form $\texttt{poly}(e)$ (see Fig. 6).

We now give an example comparing the constraints that would be generated for SML recursive value declarations in the PR system and our system. Consider the SML expression

$$\texttt{let val rec f = fn z =>}\ exp_1\ \texttt{in}\ exp_2$$

where exp_1 and exp_2 are two sub-expressions. The constraint generated in PR for this let-expression would be

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

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

Some of the differences between PR and our system can be seen when comparing how this example is handled. Our constraint generator builds roughly[2] the following constraint (technically, an environment) for the example let-expression:

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

In contrast to how PR handles this example, only one binder for `f` is needed in our system. Two features of our system interact to allow this. First, in a constrained environment (e_1;e_2), the bindings from e_1 are available in e_2, but also form part of the result (except where bindings in e_2 shadow them). Second, in an environment of the form $\texttt{poly}(e)$, the `poly` operator changes the status of binders in the result from the status they had internally. In the example constraint (environment) above, `f`'s binder is monomorphic within the scope of the `poly` operator (in e_1) and polymorphic outside (in e_2).

There is a sense in which what the PR system does is similar to what would happen in our system if the `poly` operator worked on just single types or single bindings rather than entire environments. It is significant that we can form environments of the form $\texttt{poly}(\downarrow vid{=}\langle\tau, is\rangle; e_1); e_2$, in which the type for vid is available monomorphically in e_1 and polymorphically in e_2.

The differences between the PR system and our system gain greater significance when we consider how to handle the SML module system. The most basic construct of the module system is what forms the body of a structure, namely a sequence of declarations $dec_1 \cdots dec_n$. For this discussion, assume each dec_i declares exactly one identifier x_i. Consider how declaration sequences can be handled by the PR system and our system. PR can handle such a sequence with nested let-constraints as follows:

$$\textbf{let } x_1{:}\dot{\sigma}_1 \textbf{ in } (\cdots \textbf{let } x_n{:}\dot{\sigma}_n \textbf{ in } C_0 \cdots)$$

The constraints must be nested as indicated because each x_i is only visible in the "**in**" part of the corresponding let-constraint, where an identifier binding occurrence is visible when constraints can refer to it. In contrast, our system handles the same declaration sequence with the environment

$$e_1; \cdots; e_n$$

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

[2] We have omitted labels and simplified a bit. The actual constraint that is generated (still omitting labels though) is

$$[(ev_2{=}\texttt{poly}(\downarrow\texttt{f}{=}\langle\alpha_1, \texttt{v}\rangle; [(ev_1{=}(\updownarrow\texttt{z}{=}\alpha_2)); ev_1; e_1; c_1]; c_2)); ev_2; e_2; c_3]$$

where $c_1 = (\alpha_3{=}\alpha_2{\to}\alpha_4)$, $c_2 = (\alpha_1{=}\alpha_3)$, $c_3 = (\alpha_5{=}\alpha_6)$, $\langle\alpha_4, e_1\rangle$ is generated for exp_1, $\langle\alpha_6, e_2\rangle$ is generated for exp_2, and α_5 is the type of the entire let-expression.

where e_i is the environment generated for the declaration dec_i for each $i \in \{1, \ldots, n\}$.

The importance of the difference becomes clearer when we consider how to represent full structures and structure bindings. Take the above example declaration sequence and wrap it up in a structure definition:

$$\texttt{structure}\ sid = \texttt{struct}\ dec_1 \cdots dec_n\ \texttt{end}$$

A structure expression packs into a unit a sequence of declarations. The normal scope of the declarations ends at the end of the structure, and subsequent access to the declarations must go through the structure itself, which must first be bound to a name via either a structure declaration like above or a functor application. When performing type inference for SML structure expressions, it is most natural and straightforward that the type inferred for a structure will be a sequence of individual mappings from declared names to their types[3]. Such sequences are often called *environments*. It seems clear that any type inference method will need to handle environments.

The PR system has never been extended to handle ML-style structures[4], but let us imagine how it might be extended to do this. First, let us point out that Pottier and Rémy allow abbreviating the above example of nested let-constraints as follows:

$$\textbf{let}\ \Gamma_{\mathrm{d}}\ \textbf{in}\ C_0, \text{ where } \Gamma_{\mathrm{d}} = x_1{:}\dot{\sigma}_1; \cdots; x_n{:}\dot{\sigma}_n$$

Let us call this constraint C_{d} where the "d" means "declarations". Given an SML structure definition, this kind of constraint can represent the constraints required for typability of the sequence of declarations in the structure body, and it is the only easy way to do so in the context of the PR system.

Now, how do we represent the connection of the structure's body to the structure's name? The immediately (and naively) obvious idea is to extend PR with let-constraints of a form similar to **let** $sid{:}\Gamma_{\mathrm{s}}$ **in** C, where sid is a structure identifier, and Γ_{s} is an environment (the type of a structure). Let us call this new constraint C_{s}. This is not enough, because there needs to be some way to connect the constraint C_{d} to the environment Γ_{s}. In fact, the environment Γ_{d} inside C_{d} is just what we need, but there is no easy way to get at it, because there is no mechanism in PR for generating an environment from a constraint. The easiest thing to do is to nest the entire constraint C_{s} inside the constraint C_0 inside of C_{d}, because the types of the x_i's are not accessible from outside C_{d}, but this seems like turning the program inside out, because the entire rest of the program must be nested inside the scope of the constraints for just the structure's body.

So one might then want to extend the PR constraint system with an exporting mechanism and generate a constrained environment of the form $[C_{\mathrm{d}}].\Gamma_{\mathrm{s}}$ for the structure expression where C_{d} would export the type schemes of the x_is and where Γ_{s} would refer to these exported type schemes. But, all this technicality really shouldn't be needed because Γ_{d} is already the environment that we would want to generate for the structure expression.

The way our constraint system achieves that is by instead of having only one mechanism (the let-constraints) to bind identifiers and to restrict their scope (let-constraints define a local scope), it has two separate mechanisms: one for bindings that does not restrict the scope of the binders (we obtain this behaviour by having binding constraints of form $\downarrow id{=}x$ and by having our general constrained environments forms $e_1;e_2$ where the accessors occurring in e_2 can depend on the binders occurring in e_1), and another one for constraining the scope of a type environment (obtained thanks to our environments of the form $[e]$). The environment we generate for the structure expression presented above is then similar to the environment Γ_{d}.

[3] The order of the sequence is important because a type scheme for one value identifier in a structure can refer to a type constructor name defined by the structure, while at the same time a type scheme for a different value identifier can use the same type constructor name to refer to a definition outside the structure.

[4] François Pottier told us this on 2010-08-09.

A.3 Details for Section 4.3 (Constraint generation)

An additional view of the constraints generated initially. Our constraint generator (Fig. 4) only generates restricted forms of environments (eg where "g" stands for "generation"). Let us present these restricted forms, where t is a restriction of τ, and the other forms are restrictions of e (where "p" stands for "poly" and "l" for "labelled"):

$$\begin{array}{lll}
t & \in \mathsf{ShallowTypes} & ::= \alpha \mid \alpha\,\delta \mid \alpha\,\gamma \mid \alpha_1{\to}\alpha_2 \\
lbind & \in \mathsf{LabBind} & ::= \downarrow tc \stackrel{l}{=} \gamma \mid \downarrow sid \stackrel{l}{=} ev \mid \downarrow tv \stackrel{l}{=} \alpha \\
 & & \quad \mid \downarrow vid \stackrel{l}{=} \alpha \mid \downarrow vid \stackrel{l}{=} ris \mid \updownarrow vid \stackrel{l}{=} \alpha \\
lc & \in \mathsf{LabCs} & ::= ev_1 \stackrel{l}{=} ev_2 \mid \alpha \stackrel{l}{=} t \\
lacc & \in \mathsf{LabAcc} & ::= acc^l \\
lev & \in \mathsf{LabEnvVar} & ::= ev^l \\
eip & \in \mathsf{InPolyEnv} & ::= lacc \mid lc \mid eip_1;eip_2 \\
ep & \in \mathsf{PolyEnv} & ::= \downarrow vid \stackrel{l}{=} \langle \alpha, ris \rangle \mid ep;eip \mid eip;ep \\
eg & \in \mathsf{GenEnv} & ::= \boxdot \mid lev \mid lbind \mid lacc \mid lc \mid ev{=}eg \\
 & & \quad \mid \mathtt{poly}(ep) \mid eg_1;eg_2
\end{array}$$

The rules of our constraint generator either return environments e (rules (G12)-(G14),(G16)) or constrained variables of the form $\langle var, e \rangle$ where e constrains var. In such a constrained variable, var is in some cases an internal type variable α (rules (G1)-(G11),(G15)) and in some other cases an environment variable ev (rules (G17)-(G18)). We chose not to have a constructor of constrained types that would build an internal type from an environment and an internal type (as $e_1;e_2$ builds a constrained environment from two environments), as it simplifies the presentation of our system by not having deep type structures. Such a system with constrained types could be investigated. Having chosen to return pairs of the form $\langle \alpha, e \rangle$ for expressions, we then decided to follow the same pattern for structure expressions and return pairs of the form $\langle ev, e \rangle$ instead of returning constrained environments of the form $e;ev$.

A.4 Details for Section 4.4 (Constraint solving)

An additional view of the environments generated at constraint solving. During constraint solving (see Fig. 7), a unification context of the form $\langle u, e \rangle$ is maintained. Such an environment e has a restricted form as follows (it is of the form es):

$$\begin{array}{lll}
sbind & \in \mathsf{SolvBind} & ::= \downarrow tc{=}\mu \mid \downarrow sid{=}es \mid \downarrow tv{=}\alpha \\
 & & \quad \mid \downarrow vid{=}\sigma \mid \downarrow vid{=}is \\
esrhs & \in \mathsf{SolvEnvRHS} & ::= ev \mid sbind^{\overline{d}} \mid es_1;es_2 \\
es & \in \mathsf{SolvEnv} & ::= \boxdot \mid \boxdot;esrhs
\end{array}$$

Moreover, if $\langle u, e \rangle$ occurs in a unification state $state$ and ev occurs in e then $ev \notin \mathsf{dom}(u)$. It is also the case that, for any environment variable ev, if $ev \in \mathsf{dom}(u)$ then $u(ev) \in \mathsf{SolvEnv}$.

We sometimes call an environment of the form es, a "solved" environment.

Improvement of the generation of polymorphic environments. Fig. 6 defines toPoly which is used by rule (P1) of our constraint solver to generate a polymorphic environment from a monomorphic one by quantifying the type variables not occurring in the types of the monomorphic bindings of the current unification context. In this figure $\overline{\tau}$ is the set of types of the monomorphic bindings from the current unification context. The set $\overline{\alpha}$ is the set

of type variables occurring in τ' (the type that we want to generalise to a *for all* type scheme) that can be generalised and quantified over. The dependencies in the dependency set $\overline{d}'$ are the reasons for not generalising the type variables occurring in τ' that are not in $\overline{\alpha}$ (these dependencies are the reasons why some type variables are not allowed to be quantified over).

The computation of $\overline{d}'$ and our constraining of τ' with $\overline{d}'$, even though a correct approximation (that cannot generate false errors and that will eventually allow obtaining minimal type errors), could be refined, thereby speeding up minimisation. We will now present how this can be done.

Let us first define two functions $\mathsf{getDepsVar}$ and $\mathsf{putDepsVar}$. The application $\mathsf{getDepsVar}(\alpha, \tau, \varnothing)$ will result in the dependency set occurring in τ on the paths from its root node to any occurrence of α. The application $\mathsf{putDepsVar}(\tau, \alpha, \overline{d})$ will result in the constraining of the occurrences of the type variable α in τ with the dependency set $\overline{d}$. The function $\mathsf{getDepsVar}$ is defined as follows:

$$\begin{array}{ll}
\mathsf{getDepsVar}(\alpha, \alpha', \overline{d}) & = \begin{cases} \overline{d}, \text{ if } \alpha = \alpha' \\ \varnothing, \text{ otherwise} \end{cases} \\
\mathsf{getDepsVar}(\tau\,\mu, \alpha, \overline{d}) & = \mathsf{getDepsVar}(\tau, \alpha, \overline{d}) \\
\mathsf{getDepsVar}(\tau_1 \to \tau_2, \alpha, \overline{d}) & = \cup_{i=1}^{2} \mathsf{getDepsVar}(\tau_i, \alpha, \overline{d}) \\
\mathsf{getDepsVar}(\tau^{\overline{d}}, \alpha, \overline{d}') & = \mathsf{getDepsVar}(\tau, \alpha, \overline{d} \cup \overline{d}')
\end{array}$$

The function $\mathsf{putDepsVar}$ is defined as follows:

$$\begin{array}{ll}
\mathsf{putDepsVar}(\alpha, \alpha', \overline{d}) & = \begin{cases} \alpha^{\overline{d}}, \text{ if } \alpha = \alpha' \\ \alpha, \quad \text{otherwise} \end{cases} \\
\mathsf{putDepsVar}(\tau\,\mu, \alpha, \overline{d}) & = \mathsf{putDepsVar}(\tau, \alpha, \overline{d})\,\mu \\
\mathsf{putDepsVar}(\tau_1 \to \tau_2, \alpha, \overline{d}) & = \tau_1' \to \tau_2' \\
\quad \text{where for } i \in \{1,2\},\ \tau_i' & = \mathsf{putDepsVar}(\tau_i, \alpha, \overline{d}) \\
\mathsf{putDepsVar}(\tau^{\overline{d}}, \alpha, \overline{d}') & = \mathsf{putDepsVar}(\tau, \alpha, \overline{d}')^{\overline{d}}
\end{array}$$

Let us now present another way of constraining τ' in Fig. 6 (different from constraining it with $\overline{d}'$). In the following, τ', $\overline{\tau}$ and $\overline{\alpha}$ are the same as in Fig. 6. First,

$$\{\alpha_1, \ldots, \alpha_n\} = (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus (\overline{\alpha} \cup \{\alpha_{\mathtt{dum}}\})$$

is the set of type variables that are not allowed to be quantified in the generated type scheme. Then,

$$\forall i \in \{1, \ldots, n\}.\ \overline{d}_i = \{d \mid \tau_0 \in \overline{\tau} \wedge d \in \mathsf{getDepsVar}(\tau_0, \alpha_i, \varnothing)\}$$

is the set of reasons for α_i for not being quantified over. Finally,

$$\forall i \in \{1, \ldots, n\}.\ \tau_i' = \mathsf{putDepsVar}(\tau_{i-1}', \alpha_i, \overline{d}_i)$$

where $\tau_0' = \tau'$. The function toPoly would then generate the following type scheme: $\forall \overline{\alpha}.\ \tau_n'$.

A.5 Details for Section 4.5 (Minimisation and enumeration)

Clarification on the domain of the constraint filtering function. Note that our filtering function (Fig. 8) is not defined on all environments. These forms on which the function is defined correspond to the ones generated by our constraint generator (defined in Sec. A.3). When applied to unlabelled equality constraints on environments, our filtering function is only applied to unlabelled equality constraints of the form $ev{=}e$ because our constraint generator only generates variables as the left-hand-side of an equality constraint on environments. Similarly, we only apply our filtering function on constrained environments of the form e^{l} (constrained by a unique label).

The intended meaning of a labelled constraint is that it only must hold if the condition represented by the label is true. The machinery of this paper is designed to implement this intended semantics.

Given that, we then allow our filtering function to entirely discard labelled equality constraints, bindings, accessors and environment variables because when generated, these forms are always shallow. As a matter of fact, by definition, the right-hand-side of an accessor can only be a variable (var) or a raw status (ris). When generated, the right-hand-side of a binding is either a variable (var), a type constructor name (γ), or a raw status (ris). Concerning the generated equality constraints, by shallow we mean a lc constraint as defined in Sec. A.3. The non-shallow generated equality constraints are the non-labelled ones generated by rules (G4), (G12), (G13), (G14), (G16) and (G18). Because these constraints are not labelled, they are then never filtered out but the filtering function is recursively called on the right-hand-sides of these constraints as they can be non shallow.

Further explanations on minimisation and binding discarding. A step of the first phase of our minimisation algorithm is as follows: we test if we can remove a label l associated to a binder $bind$ from the slice we want to minimise (and still obtain a type error slice) by first filtering the constraints of the original piece of code as follows: $\mathsf{filt}(e, \overline{l}, \{l\})$, to obtain e' and where e is the environment generated for the original piece of code and $\overline{l}$ is the label set labelling the current slice. In order not to mix up the bindings, the binder $bind$ associated to l is then replaced by a non labelled dummy binder that cannot participate to any error but that still acts as a binder. If we then solve e' and obtain an error then no label labelling (in e') an accessor to (the dummy version of) our binder $bind$ will occur in the found error (we give below an informal argument as why none of these accessors will be part of the new error). The bindings in this new error are then not mixed up. The found error is then the new slice to try to minimise further and next time the constraints will be filtered w.r.t. this slice, the binding $bind$ will be completely thrown away (as well as the other constraints not participating in the new error).

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

Let us now explain why when our unification algorithm returns an error, the error does not involve accessors to dummy binders or accessors without their corresponding binders.

According to rules (A1)-(A4), during unification the label labelling an accessor only gets recorded in a unification context if the accessed identifier is in the type environment stored in the unification context in the current state. In the environment (1) either the accessed identifier has a non labelled dummy static semantics (resulting from filtering) and then, according to rules (U3), (U4) and (S7), the label of the accessor does not get recorded into the unification environment. Given an accessor $\uparrow id{=}x$, according to rules (A1)-(A3), a constraint of the form $var{=}x$ where $var \in \mathsf{DumVar}$ comes from the corresponding binder and $x \in \mathsf{Var}$ or of the form $\mathsf{a}{=}x$ where $x \in \mathsf{RawIdStatus}$, is generated. Then (U3), (U4) or (S7) applies and the newly generated constraint is discarded without generating anything. (2) Or the accessed identifier has a labelled non dummy static semantics and both the label associated to the binder and the one associated the bound occurrence will always occur together in the unification context.

This is why we believe that an identifier occurring at a non-binding position in a piece of code (an accessor) only occurs in a slice if it is bound and its binder occurs in the slice as well.

This argument would be enough if only the rules (G1), (G7), and (G17) were generating accessors because each of them generates a unique labelled accessor. This is unfortunately not the case for the rules (G6) and (G8). These rules generate labelled accessors as well as labelled equality constraints. We might then think that

these labelled equality constraints can participate in an error without having the accessor and its binder participating. We could then potentially have the label of an identifier occurring at a bound position participate in an error without having its binding occurrence participate. We do not believe so. First, let us point out that this issue could be fixed by enforcing in our labelled syntax that each identifier must be labelled by a unique label that does not label any larger piece of syntax. This enforcement can be considered as a design principle concerning the labelled syntax, that was not followed for clarity purposes. Let us now explain why it works even without the enforcement described above. Let us first consider (G6). The type returned by the rule (the type α_2 in (G6)) does not directly occur on the left or right-hand-side of the generated constraint. For α_2 (which is the link with the context of the pattern) to be constrained to be equal to a type, first α_1 has to be constrained to be equal to a type. The only way for α_1 to be constrained is to solve the generated accessor which would result in the case of an error in having the binder of the identifier participate in the error. The same reasoning applies to (G8). The type returned by the rule (α') is constrained to be equal to $\alpha\,\delta$ which at this stage can be equal to any other type. This is true because we have the internal type constructor `ar` which corresponds to the arrow type constructor $\rightarrow$. Without this type constructor `ar`, during unification we could infer that $\alpha\,\delta$ cannot be an arrow type (of the form $\tau_1 \rightarrow \tau_2$) and generate an error. Because of that, we would obtain:

```
⟨..datatype 'a t = U of 'a
 ..datatype ⟨..⟩ = T of ⟨..⟩ t
 ..val rec g = fn ⟨..⟩ => ⟨..⟩
     ..⟨..val rec h = fn U x => T x
        ..h (U g)..⟩..⟩
```

as a minimal slice for:

```
structure S = struct
  datatype 'a t = U of 'a
  datatype 'a t = T of 'a t
  val rec g = fn v => v
  val rec f = fn v => let val rec h = fn U x => T x
                      in h (U g)
                      end
end
```

In the first slice `t`'s occurrence in `T`'s declaration is not bound to the same occurrence of `t` as in the original code.

But because we allow $\alpha\,\delta$ to be equal to any type as long as δ is not constrained to be equal to a type constructor name or to `ar` we then need to resolve the accessor generated in rule (G8) to obtain a type error if any. For the piece of code presented above we then obtain the following minimal type error slice instead of the one displayed above:

```
⟨..datatype 'a t = U of 'a
 ..datatype 'a t = T of ⟨..⟩ t
 ..val rec g = fn ⟨..⟩ => ⟨..⟩
     ..⟨..val rec h = fn U x => T x
        ..h (U g)..⟩..⟩
```

This type error slice differs from the previous one by the presence of the second binding occurrence of `t` in the slice.

Because of the invariant that if a binder is filtered out then its bound occurrences are also filtered out, we can then easily compute the free identifiers thanks to rule (A4) which is the rule for an accessor for which no binder exists in the current unification environment (free identifier) or for which the binder is hidden.

We could also enforce this invariant by (1) ensuring that identifiers are labelled independently from any other piece of syntax (as explained above), (2) ensuring that at constraint generation, if a label l labels an accessor then it does not label any other constraint, and (3) discarding accessors when the corresponding binders are dummy binders (binding a dummy variable or the status `a`).

Alternatively, we could enforce this invariant by adding an extra component to unification contexts as follows: $\langle u, e, \overline{l}\rangle$, where $\overline{l}$ indicates the labels that are not allowed to participate in an error. If an error is found involving a label in $\overline{l}$ then this error is not reported.

A.6 Details on Section 4.6 (Slicing)

An alternative formal presentation of the slicing algorithm. We will now provide an alternative generic definition of the external syntax presented in Fig. 3. We also extend our algorithm to "dot" terms. (This alternative definition is not the one presented in the main body of this paper for readability issues.) First we define abstract syntax trees as follows:

$$
\begin{array}{rcl}
class \in \mathsf{Class} & ::= & \texttt{ty} \mid \texttt{conbind} \mid \texttt{datname} \mid \texttt{dec} \mid \texttt{atexp} \\
 & & \mid \texttt{exp} \mid \texttt{atpat} \mid \texttt{pat} \mid \texttt{strdec} \mid \texttt{strexp} \\
prod \in \mathsf{Prod} & ::= & \texttt{tyVar} \mid \texttt{tyArr} \mid \texttt{tyCon} \\
 & & \mid \texttt{conbindOf} \mid \texttt{datnameCon} \\
 & & \mid \texttt{decRec} \mid \texttt{decDat} \mid \texttt{decOpn} \\
 & & \mid \texttt{atexpLet} \mid \texttt{expFn} \\
 & & \mid \texttt{strdecDec} \mid \texttt{strdecStr} \\
 & & \mid \texttt{strexpId} \mid \texttt{strexpSt} \\
 & & \mid \texttt{vid} \mid \texttt{app} \\
dot \in \mathsf{Dot} & ::= & \texttt{dotE} \mid \texttt{dotP} \mid \texttt{dotD} \mid \texttt{dotS} \\
node \in \mathsf{Node} & ::= & \langle class, prod\rangle \\
tree \in \mathsf{Tree} & ::= & \langle node, l, \overrightarrow{tree}\rangle \mid \langle dot, \overrightarrow{tree}\rangle \mid id
\end{array}
$$

A node in a tree $tree$ can either be a labelled node of the form $\langle node, l, \overrightarrow{tree}\rangle$, an unlabelled "dot" node of the form $\langle dot, \overrightarrow{tree}\rangle$, or a leaf of the form id.

Let a $term$ be any term that can be derived from Fig. 3:

$$
\begin{array}{rcl}
term \in \mathsf{Term} & ::= & ty \mid cb \mid dn \mid dec \mid atexp \mid exp \\
 & & \mid atpat \mid pat \mid sdec \mid sexp
\end{array}
$$

Fig. 13 defines the function toTree that associates a $tree$ to each $term$.

We are now going to redefine out flattening and tidying functions, and our slicing algorithm. First, we need a mechanism to distinguish between declarations and non-declarations, similar to the distinction between $\mathsf{ExpPart}$ and $\mathsf{Part} \setminus \mathsf{ExpPart}$. Let $\mathsf{isClass}(tree, \overline{class})$ be true iff $tree = \langle\langle class, prod\rangle, l, \overrightarrow{tree}\rangle$ and $class \in \overline{class}$, used to check the class of the root node of a tree. Let $\mathsf{declares}(tree)$ be true iff $\mathsf{isClass}(tree, \{\texttt{dec}, \texttt{strdec}\})$.

Let us redefine our flattening function flat:

$$
\mathsf{flat}(\langle\rangle) = \langle\rangle
$$

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

We slightly altered our flattening function from the one defined in Sec. 4.6, by adding the condition "or $\overrightarrow{tree} = \langle\rangle$". As a matter of fact, the condition "$\forall i \in \{1, \ldots, n\}.\ \neg\mathsf{declares}(tree_i)$" is there to ensure that bindings are not mixed up as explained in Sec. 4.6. However, flattening the last dot term (if it actually is a dot term) cannot mix up the bindings because there is no identifier left to bind. Therefore, flattening $\langle$`..val x = 1..`$\langle$`..val x = true..`$\rangle$`..`$\rangle$ would lead to $\langle$`..val x = 1..val x = true..`$\rangle$. We do not believe that this is an improvement of the function because we have not found a concrete example where this situation occurs.

We also redefine our function that tidies sequences of declarations in structure expressions as follows:

Figure 13 From terms to trees

Types	$\mathsf{toTree}(tv^l)$	$= \langle\langle \mathtt{ty}, \mathtt{tyVar}\rangle, l, \langle tv\rangle\rangle$
	$\mathsf{toTree}(ty_1 \xrightarrow{l} ty_2)$	$= \langle\langle \mathtt{ty}, \mathtt{tyArr}\rangle, l, \langle \mathsf{toTree}(ty_1), \mathsf{toTree}(ty_2)\rangle\rangle$
	$\mathsf{toTree}(ty\ tc^l)$	$= \langle\langle \mathtt{ty}, \mathtt{tyCon}\rangle, l, \langle \mathsf{toTree}(ty), tc\rangle\rangle$
Constructor bindings	$\mathsf{toTree}(vid^l_{\mathtt{c}})$	$= \langle\langle \mathtt{conbind}, \mathtt{vid}\rangle, l, \langle vid\rangle\rangle$
	$\mathsf{toTree}(vid\ \mathtt{of}^{\,l}\ ty)$	$= \langle\langle \mathtt{conbind}, \mathtt{conbindOf}\rangle, l, \langle vid, \mathsf{toTree}(ty)\rangle\rangle$
Datatype names	$\mathsf{toTree}(tv\ tc^l)$	$= \langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l, \langle tv, tc\rangle\rangle$
Declarations	$\mathsf{toTree}(\mathtt{val\ rec}\ pat \stackrel{l}{=} exp)$	$= \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l, \langle \mathsf{toTree}(pat), \mathsf{toTree}(exp)\rangle\rangle$
	$\mathsf{toTree}(\mathtt{datatype}\ dn \stackrel{l}{=} cb)$	$= \langle\langle \mathtt{dec}, \mathtt{decDat}\rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(cb)\rangle\rangle$
	$\mathsf{toTree}(\mathtt{open}^l\ sid)$	$= \langle\langle \mathtt{dec}, \mathtt{decOpn}\rangle, l, \langle sid\rangle\rangle$
Expressions	$\mathsf{toTree}(vid^l_{\mathtt{e}})$	$= \langle\langle \mathtt{atexp}, \mathtt{vid}\rangle, l, \langle vid\rangle\rangle$
	$\mathsf{toTree}(\mathtt{let}^l\ dec\ \mathtt{in}\ exp\ \mathtt{end})$	$= \langle\langle \mathtt{atexp}, \mathtt{atexpLet}\rangle, l, \langle \mathsf{toTree}(dec), \mathsf{toTree}(exp)\rangle\rangle$
	$\mathsf{toTree}(\mathtt{fn}\ pat \stackrel{l}{\Rightarrow} exp)$	$= \langle\langle \mathtt{exp}, \mathtt{expFn}\rangle, l, \langle \mathsf{toTree}(pat), \mathsf{toTree}(exp)\rangle\rangle$
	$\mathsf{toTree}(\lceil exp\ atexp\rceil^l)$	$= \langle\langle \mathtt{exp}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(exp), \mathsf{toTree}(atexp)\rangle\rangle$
Patterns	$\mathsf{toTree}(vid^l_{\mathtt{p}})$	$= \langle\langle \mathtt{atpat}, \mathtt{atpatVid}\rangle, l, \langle vid\rangle\rangle$
	$\mathsf{toTree}(vid^l\ atpat)$	$= \langle\langle \mathtt{pat}, \mathtt{app}\rangle, l, \langle vid, \mathsf{toTree}(atpat)\rangle\rangle$
Structure declarations	$\mathsf{toTree}(\mathtt{structure}\ sid \stackrel{l}{=} sexp)$	$= \langle\langle \mathtt{strdec}, \mathtt{strdecStr}\rangle, l, \langle sid, \mathsf{toTree}(sexp)\rangle\rangle$
Structure expressions	$\mathsf{toTree}(sid^l)$	$= \langle\langle \mathtt{strexp}, \mathtt{strexpId}\rangle, l, \langle sid\rangle\rangle$
	$\mathsf{toTree}(\mathtt{struct}^l\ sdec_1 \cdots sdec_n\ \mathtt{end})$	$= \langle\langle \mathtt{strexp}, \mathtt{strexpSt}\rangle, l, \langle \mathsf{toTree}(sdec_1), \ldots, \mathsf{toTree}(sdec_n)\rangle\rangle$
Dot terms	$\mathsf{toTree}(\mathtt{dot\text{-}e}(\langle pt_1, \ldots, pt_n\rangle))$	$= \langle \mathtt{dotE}, \langle \mathsf{toTree}(pt_1), \ldots, \mathsf{toTree}(pt_n)\rangle\rangle$
	$\mathsf{toTree}(\mathtt{dot\text{-}d}(\langle pt_1, \ldots, pt_n\rangle))$	$= \langle \mathtt{dotD}, \langle \mathsf{toTree}(pt_1), \ldots, \mathsf{toTree}(pt_n)\rangle\rangle$
	$\mathsf{toTree}(\mathtt{dot\text{-}p}(\langle pat_1, \ldots, pat_n\rangle))$	$= \langle \mathtt{dotP}, \langle \mathsf{toTree}(pat_1), \ldots, \mathsf{toTree}(pat_n)\rangle\rangle$
	$\mathsf{toTree}(\mathtt{dot\text{-}s}(\langle pt_1, \ldots, pt_n\rangle))$	$= \langle \mathtt{dotS}, \langle \mathsf{toTree}(pt_1), \ldots, \mathsf{toTree}(pt_n)\rangle\rangle$

$\mathsf{tidy}(\langle\rangle) = \langle\rangle$
$\mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{tree_1}\rangle, \langle \mathtt{dotD}, \overrightarrow{tree_2}\rangle\rangle @ \overrightarrow{tree})$
$\quad = \mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{tree_1} @ \overrightarrow{tree_2}\rangle\rangle @ \overrightarrow{tree})$,
$\quad$ if $\forall tree \in \mathsf{ran}(\overrightarrow{tree_1}).\ \neg\mathsf{declares}(tree)$
$\mathsf{tidy}(\langle\langle \mathtt{dotD}, \varnothing\rangle\rangle @ \overrightarrow{tree}) = \mathsf{tidy}(\overrightarrow{tree})$, if none of the above applies
$\mathsf{tidy}(\langle tree\rangle @ \overrightarrow{tree}) = \langle tree\rangle @ \mathsf{tidy}(\overrightarrow{tree})$, if none of the above applies

We also need the function getDot that generates a dot marker (a term in Dot) from a node kind $node$:

$$\begin{array}{ll}
\mathsf{getDot}(\langle \mathtt{ty}, prod\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{conbind}, prod\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{datname}, prod\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{dec}, prod\rangle) & = \mathtt{dotD} \\
\mathsf{getDot}(\langle \mathtt{atexp}, prod\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{exp}, prod\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{atpat}, prod\rangle) & = \mathtt{dotP} \\
\mathsf{getDot}(\langle \mathtt{pat}, prod\rangle) & = \mathtt{dotP} \\
\mathsf{getDot}(\langle \mathtt{strdec}, prod\rangle) & = \mathtt{dotD} \\
\mathsf{getDot}(\langle \mathtt{strexp}, prod\rangle) & = \mathtt{dotS}
\end{array}$$

This function is, among other things, used by rule (SL1) when discarding a labelled node and so generating a new dot node.

Fig. 14 redefines our slicing algorithm. Note that rule (SL9) generates the dot marker `dotE`, but we could have used any of the terms in Dot because the flattening function flat discards such terms. The functions sl_1 and sl_2 are defined on trees but also on sequences of trees in rules (SL6) and (SL7). Patterns are treated specially because in our system we do not add the label associated to the fn-expression to the following type error slice (the error being that `x` is declared as a unary datatype constructor and occurs at a nullary position in a pattern):

```
⟨..datatype ⟨..⟩ = x of ⟨..⟩
 ..fn x => ⟨..⟩..⟩
```

This is because the unconfirmed binder generated for `x`'s occurrence in the fn-expression turns into an accessor at constraint solving (`x` being declared as a datatype constructor) and this accessor can directly refer to `x`'s binder without using any constraint labelled by the label associated to the fn-expression.

B. Case study: modify user data types using TES.

Our TES is generally of great help when coding in SML. It is particularly helpful when one wants to modify a user data type in a well-typed program. Let us consider the very simple program provided in Fig. 15a where we define a structure `Id` to deal with labelled identifiers (see the type `idlab`). In this structure we define some functions to handle labelled identifiers such as a function to compare two labelled identifiers (`compare`), or a function to build a labelled identifier from a label and an identifier (`cons`).

Now, let us change the type `idlab`, for a more convenient type: `type idlab = {id : id, lab : lab}` which is a record type containing two fields, one named `id` of type `id` and a second one named `lab` of type `lab`. Records are usually preferred over tuples because they are more flexible and meaningful thanks to the field names.

For example, one can access the field named `id` in an expression `x` of type `idlab` (the new type `idlab`) as follows: `#id(x:idlab)`. Records are more flexible than tuples because the order of the fields does not matter in a record. For example, `{id = 0, lab = 0}` is equivalent to `{lab = 0, id = 0}`. Note that a tuple `(id, lab)` is equivalent to a record `{1 = id, 2 = lab}`.

First of all, let us mention that when compiling the updated code with SML/NJ v.110.72, one obtains a type error report for each function defined in the structure `Id`. The report concerning the `compare` function is as follows (where the first line has been split into two lines to fit in the column):

```
test-prog.sml:14.1-31.4
Error: value type in structure doesn't match signature spec
    name: compare
  spec:   ?.Id.idlab * ?.Id.idlab -> order
  actual: (int * int) * (int * int) -> order
```

Note that the reported region is the entire structure `Id`.

Figure 14 Slicing algorithm

$$\text{(SL1)}\ \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \overline{l}) = \begin{cases} \langle node, l, \mathsf{sl}_1(\overrightarrow{tree}, \overline{l})\rangle, & \text{if } (l \in \overline{l} \text{ and } \mathsf{getDot}(node) \neq \texttt{dotS}) \text{ or } \mathsf{isClass}(\mathsf{sl}_1(\overrightarrow{tree}(0), \overline{l}), \{\texttt{pat}, \texttt{atpat}\}) \\ \langle node, l, \mathsf{tidy}(\mathsf{sl}_1(\overrightarrow{tree}, \overline{l}))\rangle, & \text{if } l \in \overline{l} \text{ and } \mathsf{getDot}(node) = \texttt{dotS} \\ \langle \mathsf{getDot}(node), \mathsf{flat}(\mathsf{sl}_2(\overrightarrow{tree}, \overline{l}))\rangle, & \text{otherwise} \end{cases}$$

$$\text{(SL2)}\ \mathsf{sl}_1(\langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle, \overline{l}) = \langle dot, \mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \overline{l}), \ldots, \mathsf{sl}_2(tree_n, \overline{l})\rangle)\rangle \qquad \text{(SL4)}\ \mathsf{sl}_1(\langle node, l, \overrightarrow{tree}\rangle, \overline{l}) = \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \overline{l})$$

$$\text{(SL3)}\ \mathsf{sl}_2(\langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle, \overline{l}) = \langle dot, \mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \overline{l}), \ldots, \mathsf{sl}_2(tree_n, \overline{l})\rangle)\rangle \qquad \text{(SL5)}\ \mathsf{sl}_2(\langle node, l, \overrightarrow{tree}\rangle, \overline{l}) = \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \overline{l})$$

$$\text{(SL6)}\ \mathsf{sl}_1(\langle tree_1, \ldots, tree_n\rangle, \overline{l}) = \langle \mathsf{sl}_1(tree_1, \overline{l}), \ldots, \mathsf{sl}_1(tree_n, \overline{l})\rangle \qquad \text{(SL8)}\ \mathsf{sl}_1(id, \overline{l}) = id$$

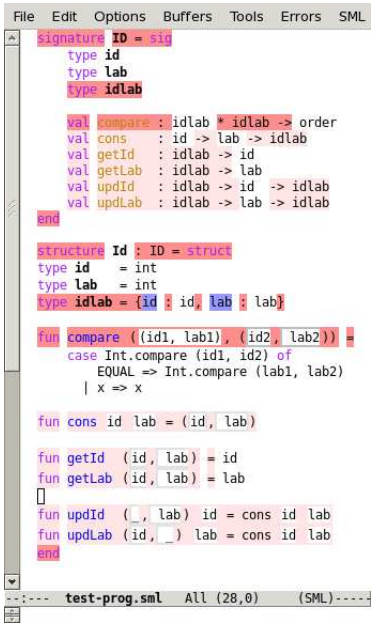
$$\text{(SL7)}\ \mathsf{sl}_2(\langle tree_1, \ldots, tree_n\rangle, \overline{l}) = \langle \mathsf{sl}_2(tree_1, \overline{l}), \ldots, \mathsf{sl}_2(tree_n, \overline{l})\rangle \qquad \text{(SL9)}\ \mathsf{sl}_2(id, \overline{l}) = \langle \texttt{dotE}, \langle\rangle\rangle$$

Figure 15 Using TES to modify user data types

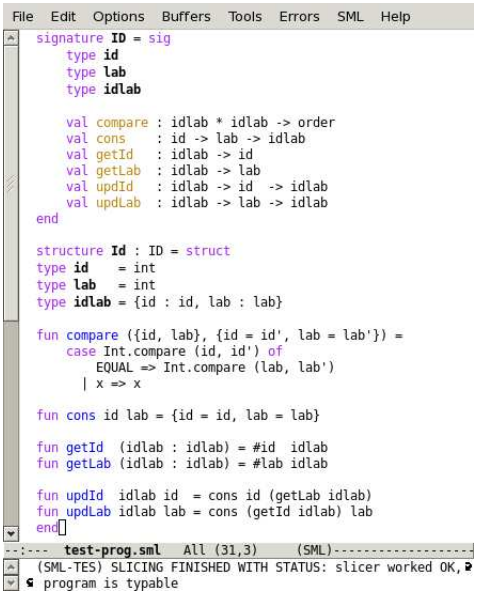
(a) Structure defining labelled identifiers

```
signature ID = sig
  type id
  type lab
  type idlab

  val compare : idlab * idlab -> order
  val cons    : id -> lab -> idlab
  val getId   : idlab -> id
  val getLab  : idlab -> lab
  val updId   : idlab -> id  -> idlab
  val updLab  : idlab -> lab -> idlab
end

structure Id : ID = struct
type id    = int
type lab   = int
type idlab = id * lab

fun compare ((id1, lab1), (id2, lab2)) =
  case Int.compare (id1, id2) of
      EQUAL => Int.compare (lab1, lab2)
    | x => x

fun cons id lab = (id, lab)

fun getId  (id, lab) = id
fun getLab (id, lab) = lab

fun updId  (_, lab) id = cons id lab
fun updLab (id, _) lab = cons id lab
end
```

(b) Type error obtained after a type change



(c) Program obtained after solving all the type errors



In contrast, Fig. 15b presents the highlighting that one obtains when running our TES on the updated piece of code. The error in focus (highlighted with a darker red) shows that the parameter of `compare` is a pair of pairs. The second pair (equivalent to a record with two fields named `1` and `2`) clashes with the type of `compare`'s second parameter given in the signature `ID`, which is `idlab`, declared as a record with field names `id` and `lab` in the structure `Id`. In the parameter of `compare`, the second pair has its elements surrounded by grey boxes. We do so, because tuples do not have explicitly written field names. The first grey box surrounds the first element of a pair that corresponds to a record where the element would be in field with field name `1` (and similarly for the second box). Note that the number of boxes indicates the arity of the tuple. In addition to the highlighting, we also report a type error slice and the following message for this type error:

Record clash, the fields {id,lab} conflict with {1,2}

This error is not context dependent, so no context dependency is reported.

The light pink corresponds to slices other than the focused one. One can then start solving the errors one at a time by just editing the highlighted portions of code, to get from a well-typed program to another well-typed program (see Fig. 15c).

C. Extensions to handle more of SML

Let us now present some extensions of our TES in order to handle features such as local declarations, type functions or signatures. Some of these features were already used in the examples provided so far. We will now formally present how to handle them.

Some syntactic forms will sometimes need to be redefined. In this section, we will sometimes write $x \xrightarrow{s} y$ to mean that in the set s, syntactic forms of the form x are replaced by syntactic forms of the form y.

C.1 Local declarations

External syntax. First, let us extend our external syntax with local declarations as follows:

$$dec ::= \cdots \mid \texttt{local}^{\,l}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$$

For example,

```
val x = true
local val x = 1 in val y = x end
val z = x + 1
```

is untypable because `x`'s last occurrence is bound to its first occurrence and not to its second (assuming that `+` is the one from the Standard ML basis library).

Let us present another example:

```
val x = true
local val x = 1 in val y = x end
val z = fn w => (w y, w x)
```

Only the declaration `z` differs from the previous example. This piece of code is also untypable because `w` has a monomorphic type and is applied to `y` which is an integer and `x` which is a Boolean. This example will be reused later in this section.

Constraint syntax. We extend environments with local environments as follows:

$$e ::= \cdots \mid \texttt{loc}\ e_1\ \texttt{in}\ e_2$$

The meaning of such an environment is that it builds an environment e_2 which depends on e_1 and only exports the binders of e_2 (only e_2's binders can be accessed from outside the local environment). Such environments differ from environments of the form $e_1;e_2$ because an environment of the form $e_1;e_2$ builds a new environment from both e_1 and e_2 and exports both the binders of e_1 (not shadowed by e_2) and e_2.

Environments of the form $[e]$ are not enough to handle local declarations because they do not allow partially exporting an environment. The requirement imposed by local declarations is that only e_1 and e_2 should be able to access e_1's binders. Unfortunately, $[e_1;e_2]$ does not export e_2's binders, and $[e_1];e_2$ does not allow e_2's accessors to refer to e_1's binders. The solution was to introduce environments of the form $\texttt{loc}\ e_1\ \texttt{in}\ e_2$.

Note that these environments are not only used to generate constraints for local declarations, they are also used to, e.g., handle bindings of external type variables (see Sec. C.2). In Sec. 4 we allow binding occurrences of explicit type variables to have a larger scope than they should, which is harmless in the tiny language of Sec. 4, but needs to be (and is) fixed to work for full SML in Sec. C.2.

Constraint generation. We extend our constraint generator with the following rule:

$$\begin{array}{l}(\mathsf{G19})\ \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \triangleright (ev{=}e_1);\texttt{loc}\ ev^l\ \texttt{in}\ e_2 \Leftarrow \\ \qquad dec_1 \triangleright e_1 \wedge dec_2 \triangleright e_2 \wedge \mathsf{dja}(e_1, e_2, ev)\end{array}$$

Because our initial constraint generation algorithm generates new forms of constraints, we extend the eg forms as follows (see Sec. A.3):

$$eg ::= \cdots \mid \texttt{loc}\ eg_1\ \texttt{in}\ eg_2$$

The forms generated by our initial constraint generator are in fact more restricted than that, but we already anticipate the forms generated by further extensions later, e.g., for type functions.

Constraint solving. We extend our constraint solver as follows:

$$\begin{array}{l}(\mathsf{L1})\ \texttt{solve}(\langle u, e\rangle, \overline{d}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{succ}(\langle u'', e_0\rangle), \\ \qquad \text{if } \texttt{solve}(\langle u, e\rangle, \overline{d}, e_1) \rightarrow^* \texttt{succ}(\langle u', e'\rangle) \\ \qquad \text{and } \texttt{solve}(\langle u', e'\rangle, \overline{d}, e_2) \rightarrow^* \texttt{succ}(\langle u'', e''\rangle) \\ \qquad \text{and } \mathsf{diff}(e', e'') = \boxdot; e'_1; \cdots; e'_n \\ \qquad \text{and } e_0 = e; e'_1; \cdots; e'_n \\ (\mathsf{L2})\ \texttt{solve}(\langle u, e\rangle, \overline{d}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{err}(er), \\ \qquad \text{if } \texttt{solve}(\langle u, e\rangle, \overline{d}, e_1) \rightarrow^* \texttt{succ}(\langle u', e'\rangle) \\ \qquad \text{and } \texttt{solve}(\langle u', e'\rangle, \overline{d}, e_2) \rightarrow^* \texttt{err}(er) \\ (\mathsf{L3})\ \texttt{solve}(\langle u, e\rangle, \overline{d}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{err}(er), \\ \qquad \text{if } \texttt{solve}(\langle u, e\rangle, \overline{d}, e_1) \rightarrow^* \texttt{err}(er)\end{array}$$

Constraint filtering (Minimisation and enumeration). We extend our filtering function as follows:

$$\mathsf{filt}(\texttt{loc}\ e_1\ \texttt{in}\ e_2, \overline{l}_1, \overline{l}_2) = \texttt{loc}\ \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2)\ \texttt{in}\ \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2)$$

Slicing. Finally, our slicing algorithm does not need to be extended but we need to update the tree syntax for programs as follows:

$$\mathsf{Prod} ::= \cdots \mid \texttt{decLoc}$$

We also need to extend the toTree function that associates trees of the form $tree$ to terms of the form $term$ as follows:

$$\begin{array}{l}\mathsf{toTree}(\texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}) \\ \quad = \langle\langle \texttt{dec}, \texttt{decLoc}\rangle, l, \langle \mathsf{toTree}(dec_1), \mathsf{toTree}(dec_2)\rangle\rangle\end{array}$$

Minimality. Let us illustrate what would happen if we were not generating an extra labelled environment variable in rule (G19). Consider the last example presented above. With our current system, we would obtain a type error slice involving the local declaration itself in addition to the nested declarations of `x` and `y`. If we were not to label the environment variable in rule (G19) or if we were to use e_1 instead of ev^l in the local constraint (and omit $ev{=}e_1$ which becomes useless), then we would obtain a type error slice that would look like:

```
⟨..val x = true
 ..val x = 1
 ..val y = x
 ..val z = fn w => ⟨..w y..w x..⟩..⟩
```

which is typable and therefore not a minimal type error slice of the piece of code presented above: both bound occurrences of `x` are bound to `x`'s second declaration.

C.2 Type declarations

External syntax. First, let us extend our external syntax with type functions as follows:

$$\mathsf{Dec} ::= \cdots \mid \texttt{type}\ dn \stackrel{l}{=} ty$$

For example,

```
type 'a t = 'a -> 'a -> 'a
datatype 'a u = U of 'a t
val x = U (fn x => x)
```

is untypable because `U` is applied to the identity function which cannot have the type `'a -> 'a -> 'a`.

Constraint syntax. We extend our constraint system with type functions:

$$\begin{array}{lll}\phi & \in \mathsf{TypFunVar} & \text{(type function variables)} \\ \theta & \in \mathsf{TypFun} & ::= \phi \mid \Lambda\alpha.\,\tau \mid \langle\theta, \overline{d}\rangle \\ \tau & \in \mathsf{ITy} & ::= \cdots \mid \theta \cdot \tau \\ c & \in \mathsf{Constraint} & ::= \cdots \mid \theta_1{=}\theta_2 \\ var & \in \mathsf{Var} & ::= \cdots \mid \phi\end{array}$$

Let $\phi_{\mathtt{dum}}$ be a distinguished dummy type function variable. We redefine DumVar to be $\mathsf{DumVar} = \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}, \phi_{\mathtt{dum}}\}$.

We then have to change the binders and accessors for type constructors:

$$\downarrow tc{=}\mu \xrightarrow{\mathsf{Bind}} \downarrow tc{=}\theta \qquad \uparrow tc{=}\delta \xrightarrow{\mathsf{Accessor}} \uparrow tc{=}\phi$$

Constraint generation. Fig. 16 modifies our rules for datatype names (G15), datatype declarations (G13) and type constructions (G8), and defines a new rule (G20) for type function declarations. Note the use of local environments (of the form $\texttt{loc}\ e_1\ \texttt{in}\ e_2$) in rules (G13) and (G20), used to handle binding occurrences of explicit type variables.

Because our initial constraint generation algorithm generates new forms of constraints, we extend the t forms as follows (see Sec. A.3):

$$t ::= \cdots \mid \phi \cdot \alpha$$

Figure 16 Constraint generation rules for type functions

(G8) $ty\ tc^l \twoheadrightarrow \langle \alpha', (\uparrow tc \stackrel{l}{=} \phi); (\alpha' \stackrel{l}{=} \phi \cdot \alpha); e \rangle$	$\Leftarrow\ ty \twoheadrightarrow \langle \alpha, e \rangle \wedge \mathsf{dja}(e, \alpha', \phi)$
(G15) $tv\ tc^l \twoheadrightarrow \langle \alpha, \alpha', \downarrow tc \stackrel{l}{=} \Lambda\alpha'.\, \alpha, \downarrow tv \stackrel{l}{=} \alpha' \rangle$	$\Leftarrow\ \alpha \neq \alpha'$
(G13) $\texttt{datatype}\ dn \stackrel{l}{=} cb \twoheadrightarrow ev = ((\alpha_1 \stackrel{l}{=} \alpha_1'\ \gamma); (\alpha_1 \stackrel{l}{=} \alpha_2); e_1; \texttt{loc}\ e_1'\ \texttt{in}\ \texttt{poly}(e_2)); ev^l$	$\Leftarrow\ dn \twoheadrightarrow \langle \alpha_1, \alpha_1', e_1, e_1' \rangle \wedge cb \twoheadrightarrow \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \gamma, ev)$
(G20) $\texttt{type}\ dn \stackrel{l}{=} ty \twoheadrightarrow ev = ((\alpha_1 \stackrel{l}{=} \alpha_2); \texttt{loc}\ e_1'\ \texttt{in}\ (e_2; e_1)); ev^l$	$\Leftarrow\ dn \twoheadrightarrow \langle \alpha_1, \alpha_1', e_1, e_1' \rangle \wedge ty \twoheadrightarrow \langle \alpha_2, e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

We also replace the initially generated type constructor binders as follows:

$$\downarrow tc \stackrel{l}{=} \gamma \xrightarrow{\mathsf{LabBind}} \downarrow tc \stackrel{l}{=} \Lambda\alpha.\, \alpha'$$

Constraint solving. Because we replaced our binders for type constructors, we need to modify our environment application as follows ((EA1) and (EA3) are the same as before but repeated here, and (EA2) differs by the replacement of μ by θ):

$$\begin{array}{ll}
(\mathsf{EA1})\ (e'; \downarrow vid \stackrel{\overline{d}}{=} \forall\overline{\alpha}.\, \tau)(vid) = \forall\overline{\alpha}.\, \tau^{\overline{d}} & \\
(\mathsf{EA2})\ (e'; \downarrow id \stackrel{\overline{d}}{=} x)(id) & = x^{\overline{d}}, \quad \text{if } x \text{ of the form } \tau,\ e, \text{ or } \theta \\
(\mathsf{EA3})\ (e'; \downarrow id' \stackrel{\overline{d}}{=} x)(id) & = e'(id), \text{ if } id \neq id' \text{ or } x \in \mathsf{IdStatus}
\end{array}$$

We also extend unifiers as follows:

$$\begin{array}{rl}
u \in \mathsf{Unifier} ::= \{f_1 \cup f_2 \cup f_3 \cup f_4 \mid & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\
& \wedge\ f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\
& \wedge\ f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\
& \wedge\ f_4 \in \mathsf{TypFunVar} \to \mathsf{TypFun}\}
\end{array}$$

Because we added types of the form $\theta \cdot \tau$, we need to update our type building function with the following cases:

$$\begin{array}{l}
\mathsf{build}(u, \theta \cdot \tau) = \begin{cases} \tau'^{\overline{d}}[\{\alpha \mapsto \mathsf{build}(u, \tau)\}], \\ \quad \text{if } \mathsf{build}(u, \theta) = (\Lambda\alpha.\, \tau')^{\overline{d}} \\ \alpha_{\mathtt{dum}}, \text{ otherwise} \end{cases} \\
\mathsf{build}(u, \Lambda\alpha.\, \tau) = \Lambda\alpha.\, \mathsf{build}(\{\alpha\} \lhd u, \tau)
\end{array}$$

Our building function builds internal types of the form $\theta \cdot \tau$ by first building the type function. If building the type function leads to a type function variable then our building function gives up building the application and returns the dummy type variable $\alpha_{\mathtt{dum}}$. This behaviour is correct in our system because the constraint generation rule (G8) is the only rule generating type function applications and it constrains ϕ before generating the application $\phi \cdot \alpha$. Thus, at constraint solving, when dealing with $\phi \cdot \alpha$, the constraints on ϕ will already have been dealt with.

Fig. 17 extends our constraint solver with two new rules to handle our new internal types of the form $\theta \cdot \tau$.

Constraint filtering (Minimisation and enumeration). We update our filtering function to handle the semantics of type constructors, by replacing $\mathsf{toDumVar}(\mu) = \delta_{\mathtt{dum}}$ by:

$$\mathsf{toDumVar}(\theta) = \phi_{\mathtt{dum}}$$

Slicing. Because we have changed our constraint generation rule for dn's, we need to replace the dot terms in DatName as follows:

$$\texttt{dot-e}(\overrightarrow{pt}) \xrightarrow{\mathsf{DatName}} \texttt{dot-n}(\overrightarrow{pt})$$

We define the new constraint generation rule for terms of the form $\texttt{dot-n}(\overrightarrow{pt})$ as follows:

$$\begin{array}{l}
\texttt{dot-n}(\langle pt_1, \ldots, pt_n \rangle) \twoheadrightarrow \langle \alpha, \alpha', [e_1; \cdots; e_n], \boxdot \rangle \Leftarrow \\
\quad pt_1 \twoheadrightarrow e_1 \wedge \cdots \wedge pt_n \twoheadrightarrow e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \alpha)
\end{array}$$

Finally, our slicing algorithm does not need to be extended but we need to update the tree syntax for programs as follows:

$$\begin{array}{rl}
\mathsf{Prod} ::= & \cdots \mid \texttt{decTyp} \\
\mathsf{Dot} ::= & \cdots \mid \texttt{dotN}
\end{array}$$

We also need to modify the getDot function that associates dot markers to node kinds as follows (the function now returns a `dotN` marker and not a `dotE` marker anymore when applied to a `datname` node):

$$\mathsf{getDot}(\langle \texttt{datname}, prod \rangle) = \texttt{dotN}$$

We also need to extend the toTree function that associates trees of the form $tree$ to terms of the form $term$ as follows:

$$\begin{array}{l}
\mathsf{toTree}(\texttt{type}\ dn \stackrel{l}{=} ty) \\
\quad = \langle \langle \texttt{dec}, \texttt{decTyp} \rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(ty) \rangle \rangle \\
\mathsf{toTree}(\texttt{dot-n}(\langle pt_1, \ldots, pt_n \rangle)) \\
\quad = \langle \texttt{dotN}, \langle \mathsf{toTree}(pt_1), \ldots, \mathsf{toTree}(pt_n) \rangle \rangle
\end{array}$$

C.3 Signatures

This section shows how to design a type error slicer that handles some signature related features. This section deals with value, type, datatype and structure specifications. It does not deal with include or sharing specifications, and does not deal with type realisations (where clauses) either. Type realisations and include specifications are "almost fully" supported by our implementation. We also partially support sharing specifications in our implementation.

Some kinds of errors are not handled by the system presented in this section. For example we do not handle unmatched errors: when an identifier is specified in a signature but not declared in a structure constrained by the signature. These errors will be dealt with in Sec. C.4. Another kind of error which is not dealt with in this section is when a type constructor is defined as a type function in a structure and as a datatype in the structure's signature. Even though this kind of error is handled by our implementation but we have not yet written the details.

External syntax. First, let us extend our external syntax with signatures as follows:

$$\begin{array}{llll}
sigid & \in \mathsf{SigId} & & \text{(signature identifiers)} \\
sigdec & \in \mathsf{SigDec} & ::= & \texttt{signature}\ sigid \stackrel{l}{=} sigexp \\
 & & \mid & \texttt{dot-d}(\overrightarrow{pt}) \\
sigexp & \in \mathsf{SigExp} & ::= & sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \\
 & & \mid & \texttt{dot-s}(\overrightarrow{pt}) \\
spec & \in \mathsf{Spec} & ::= & \texttt{val}\ vid\ :^l\ ty \\
 & & \mid & \texttt{type}\ dn^l \\
 & & \mid & \texttt{datatype}\ dn \stackrel{l}{=} cd \\
 & & \mid & \texttt{structure}\ sid\ :^l\ sigexp \\
 & & \mid & \texttt{dot-d}(\overrightarrow{pt}) \\
cd & \in \mathsf{ConDesc} & ::= & vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^l\ ty \\
 & & \mid & \texttt{dot-e}(\overrightarrow{pt}) \\
id & \in \mathsf{Id} & ::= & \cdots \mid sigid \\
sexp & \in \mathsf{StrExp} & ::= & \cdots \mid sexp\ :^l\ sigexp \mid sexp\ :>^l\ sigexp \\
topdec & \in \mathsf{TopDec} & ::= & sdec \mid sigdec \\
prog & \in \mathsf{Program} & ::= & topdec_1\,;\cdots;\,topdec_n, \text{ where } n \geq 0
\end{array}$$

The symbol :> is used for opaque constraints and : for translucent constraints. The structure $sexp\ :>^l\ sigexp$ is the structure $sexp$ constrained by the signature $sigexp$ where each of $sigexp$'s specifications has to be matched by one of $sexp$'s declarations (and similarly for $sexp\ :^l\ sigexp$). The structure $sexp$ can declare more identifiers than are specified in $sigexp$. In the structure $sexp\ :>^l\ sigexp$, only the identifiers specified in $sigexp$ can be

Figure 17 Constraint solving rules for type functions

(S9) $\texttt{solve}(\langle u, e\rangle, \overline{d}, \theta_1 \cdot \tau_1 = \theta_2 \cdot \tau_2) \rightarrow \texttt{solve}(\langle u, e\rangle, \overline{d}, \mathsf{build}(u, \theta_1 \cdot \tau_1) = \mathsf{build}(u, \theta_2 \cdot \tau_2))$

(S10) $\texttt{solve}(\langle u, e\rangle, \overline{d}, \tau_1 = \tau_2) \rightarrow \texttt{solve}(\langle u, e\rangle, \overline{d}, \tau = \mathsf{build}(u, \theta \cdot \tau'))$
if $\{\tau_1, \tau_2\} = \{\theta \cdot \tau', \tau\}$ and τ is of the form $\tau_3 \rightarrow \tau_4$ or $\tau_3\ \mu$

accessed from $sexp$ (only the $sigexp$ part from $sexp$ is visible to the outside world). The difference between $sexp$ `:>`^{l} $sigexp$ and $sexp$ `:`^{l} $sigexp$ is that in the first one if $sigexp$ specifies a type constructor tc then in $sexp$ `:>`^{l} $sigexp$ it is not constrained by its declaration in $sexp$, whereas in $sexp$ `:`^{l} $sigexp$ the type constructor would be constrained by its declaration in $sexp$. Opaque signatures are used to abstract types from structures and are usually preferred over translucent ones for this reason.

Let us now present an example involving an opaque signature:

(EX1)
```
signature s = sig val x : 'a end
structure S = struct val x = 1 end
structure T = S :> s
```

This piece of code is untypable because the type variable `'a` is more general than the type `int`. Types of declarations in structures have to be at least as general as the corresponding specifications in signatures. This kind of error will be referred as a *too general* error henceforth.

Constraint syntax. We now extend our constraint system to handle signatures:

$$\begin{array}{lll} bind \in \mathsf{Bind} & ::= \cdots \mid \downarrow sigid = e \\ acc \in \mathsf{Accessor} & ::= \cdots \mid \uparrow sigid = ev \\ \tau \in \mathsf{ITy} & ::= \cdots \mid \widehat{tv} \\ \mu \in \mathsf{ITyCon} & ::= \cdots \mid \widehat{tv} \\ e \in \mathsf{Env} & ::= \cdots \mid e_1 : e_2 \mid e_1 :> e_2 \end{array}$$

We also extend the form of the explicit type variable binders as follows:

$$\downarrow tv = \alpha \xrightarrow{\mathsf{Bind}} \downarrow tv = \tau$$

We add the explicit type variables to the internal type set and extend the explicit type variable binders to internal types because we want to allow explicit type variables to bind explicit type variables and not only internal type variables. This helps catching *too general* errors as presented above.

We also add the explicit type variables to the internal type constructor set because, in order to help catch *too general* errors, we do not generalise the external type variables occurring in a value specification in a signature until we match the signature against a structure. Inside a signature, an explicit type variable is then considered as a constant type. To such a constant type we associate an internal type constructor which is the explicit type variable itself. Let us explain our reason for doing so using the following piece of code (the same as (EX1) where we replaced `'a` by `bool` in `x`'s specification):

(EX2)
```
signature s = sig val x : bool end
structure S = struct val x = 1 end
structure T = S :> s
```

Given this piece of code, our enumeration algorithm would find the type error that `x` is specified as a Boolean in `s`, which is the signature constraining `S`, and that `x` is declared as an integer in `S`. The issue is that our minimisation algorithm would eventually try to slice out the type `bool` in `x`'s specification. This would result in `x` having a type scheme of the form $\forall\{\alpha\}.\,\alpha$ in its specification This type scheme is more general than `int` which is `x`'s type in its declaration. If we were to generalise the explicit type variables occurring in value specifications, we would also generate the type scheme $\forall\{\alpha\}.\,\alpha$ for `x`'s specification in (EX1). We then would not be able to distinguish between a type scheme which is genuinely too general (in (EX1)) and a type scheme which is too general because some information has been discarded (in (EX2) where `bool` has been filtered out). In order to avoid that, explicit type variables occurring in a signature are not generalised until we match the signature against a structure.

Because we extended our internal types, we also need to extend our building function as follows:

$$\mathsf{build}(u, \widehat{tv}) = \widehat{tv}$$

Constraint generation. Fig. 18 presents the new constraint generation rules for the syntactic forms introduced above. Rule (G24) uses the function tvBind which is defined as follows: $\mathsf{tvBind}(ty, l) = (\downarrow tv_1 \stackrel{l}{=} \widehat{tv_1}; \cdots; \downarrow tv_n \stackrel{l}{=} \widehat{tv_n})$ such that $\{tv_1, \ldots, tv_n\}$ is the set of external type variables occurring in ty and where if $i < j$ then tv_j does not occur on the left of tv_i in ty. This function is used to generate explicit type variable binders for explicit type variables occurring in value specifications such as in the specification `val f : 'a -> 'a`, for which we would generate a binder of the form $\downarrow$`'a`$=\widehat{\texttt{'a}}$.

Note that rules (G21), (G22) and (G23) for signature declarations and expressions are similar to rules (G16), (G17) and (G18), defined in Fig. 7, for structure declarations and expressions. Rule (G24) is a simplified version of rule (G12) for recursive value declarations (defined in Fig. 7), where the expression is replaced by an external type and where the pattern is reduced to a single value identifier. Note that even though explicit type variables occurring in a signature are not generalised until the signature is matched against a structure, rule (G24) generates a `poly` environment to generalise internal type variables that are unconstrained due to constraint filtering. The novelty in this rule, as described above is the necessity to bind the explicit type variables occurring in the external type. Rule (G25) is similar to rule (G20) defined in Fig. 16, but instead of binding the specified type constructor to an internal type computed from an external type, it generates a new type constructor name. Such a name might then be renamed during constraint solving when matching a signature against a structure. Rule (G27) is similar to rule (G13) defined in Fig. 16 and rule (G26) is similar to rule (G16) defined in Fig. 7. The constraint generation rules for constructor descriptions are the same as the ones for constructor declarations. Finally, rules (G28) and (G29) are the most interesting rules. They are the ones generating our new environments of the forms $e_1 : e_2$ (generated by rule (G28) for transparent signature constraints) and $e_1 :> e_2$ (generated by rule (G29) for opaque signature constraints).

Because our initial constraint generation algorithm generates new forms of constraints, we extend the $lbind$ and eg forms as follows (see Sec. A.3):

$$\begin{array}{lll} ietv & \in \mathsf{IETyVar} & ::= \alpha \mid \widehat{tv} \\ lbind & \in \mathsf{LabBind} & ::= \cdots \mid \downarrow sigid \stackrel{l}{=} ev \\ eg & \in \mathsf{GenEnv} & ::= \cdots \mid ev_1 : ev_2 \mid ev_1 :> ev_2 \end{array}$$

An $ietv$ (used below) can either be an internal ("I") or an external ("E") type variable.

We also replace the initially generated external type variable binders as follows:

$$\downarrow tv \stackrel{l}{=} \alpha \xrightarrow{\mathsf{LabBind}} \downarrow tv \stackrel{l}{=} ietv$$

Constraint solving. Let us extend unification states and error kinds as follows:

Figure 18 Constraint generation rules for signatures

Signature declarations

(G21) $\texttt{signature}\ sigid \stackrel{l}{=} sigexp \triangleright ev'=(e;(\downarrow sigid \stackrel{l}{=} ev));ev'^{l} \quad \Leftarrow\ sigexp \triangleright \langle ev, e\rangle \wedge \mathsf{dja}(e, ev')$

Signature expressions

(G22) $sigid^{l} \triangleright \langle ev, \uparrow sigid \stackrel{l}{=} ev\rangle$

(G23) $\texttt{sig}^{l}\ spec_1 \cdots spec_n\ \texttt{end} \triangleright \langle ev, (ev \stackrel{l}{=} ev');(ev'=(e_1;\cdots;e_n))\rangle$
$\quad \Leftarrow\ spec_1 \triangleright e_1 \wedge \cdots \wedge spec_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev, ev')$

Specifications

(G24) $\texttt{val}\ vid :^{l} ty \triangleright (ev=\texttt{poly}(\texttt{loc}\ \mathsf{tvBind}(ty, l)\ \texttt{in}\ (e;\downarrow vid \stackrel{l}{=} \langle \alpha, \mathtt{v}\rangle)));ev^{l} \quad \Leftarrow\ ty \triangleright \langle \alpha, e\rangle \wedge \mathsf{dja}(e, ev)$

(G25) $\texttt{type}\ dn^{l} \triangleright (ev=((\alpha \stackrel{l}{=} \alpha'\ \gamma);e));ev^{l} \quad \Leftarrow\ dn \triangleright \langle \alpha, \alpha', e, e'\rangle \wedge \mathsf{dja}(e, e', ev, \gamma)$

(G26) $\texttt{structure}\ sid :^{l} sigexp \triangleright (ev'=(e;(\downarrow sid \stackrel{l}{=} ev)));ev'^{l} \quad \Leftarrow\ sigexp \triangleright \langle ev, e\rangle \wedge \mathsf{dja}(e, ev')$

(G27) $\texttt{datatype}\ dn \stackrel{l}{=} cd \triangleright (ev=((\alpha_1 \stackrel{l}{=} \alpha_1'\ \gamma);(\alpha_1 \stackrel{l}{=} \alpha_2);e_1;\texttt{loc}\ e_1'\ \texttt{in}\ \texttt{poly}(e_2)));ev^{l}$
$\quad \Leftarrow\ dn \triangleright \langle \alpha_1, \alpha_1', e_1, e_1'\rangle \wedge cd \triangleright \langle \alpha_2, e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \gamma, ev)$

Structure expressions

(G28) $sexp :^{l} sigexp \triangleright \langle ev, e_2;e_1;(ev \stackrel{l}{=} ev_1{:}ev_2)\rangle \quad \Leftarrow\ sexp \triangleright \langle ev_1, e_1\rangle \wedge sigexp \triangleright \langle ev_2, e_2\rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

(G29) $sexp :\!>^{l} sigexp \triangleright \langle ev, e_2;e_1;(ev \stackrel{l}{=} ev_1{:}\!>ev_2)\rangle \quad \Leftarrow\ sexp \triangleright \langle ev_1, e_1\rangle \wedge sigexp \triangleright \langle ev_2, e_2\rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

Programs

(G30) $topdec_1\,;\cdots;\,topdec_n \triangleright e_1;\cdots;e_n \quad \Leftarrow\ topdec_1 \triangleright e_1 \wedge \cdots \wedge topdec_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev)$

$$
\begin{array}{lll}
tfm & \in \mathsf{TypFunMap} & = \mathsf{TyConName} \to \mathsf{TypFun} \\
state & \in \mathsf{State} & ::= \cdots \\
 & & \mid \texttt{prematch}(\Delta, \overline{d}, e_1, e_2) \\
 & & \mid \texttt{match}(\Delta, \overline{d}, tfm, e_1, e_2) \\
 & & \mid \texttt{succ}(\Delta, tfm) \\
ek & \in \mathsf{ErrKind} & ::= \cdots \\
 & & \mid \texttt{TyVarClash}(tv_1, tv_2) \\
 & & \mid \texttt{TooGeneral}(\mu_1, \mu_2)
\end{array}
$$

Type functions of the form tfm are used to gather the type functions defined in a structure, to then apply them to the types generated for a signature constraint. Roughly speaking, when solving an environment of the form $e_1{:}e_2$ or of the form $e_1{:}\!>e_2$, the type functions defined in e_1 (related to a structure) are gathered and applied to e_2 (related to a signature) using the appTFM function defined below. This step is required because our initial constraint generation algorithm might generates different type constructor names for two type constructors that might turn our to be the same type constructor. For example, in the following (typable) piece of code

```
signature s = sig datatype 'a t = T end
structure S = struct datatype 'a t = T end : s
```

Our initial constraint generation algorithm will generate two distinct type constructor names for the two occurrence of `t`. But, when checking that the signature `s`, matches the structure `S`, these two type constructor names have to be equated. This is done by extracting the one defined in the structure and by then renaming the one from the signature into the one from the structure.

Error kinds of the form $\texttt{TooGeneral}(\mu_1, \mu_2)$ are for type errors as the one described above (*too general* errors), where a signature constrains a structure and is more general than the structure. Error kinds of the form $\texttt{TyVarClash}(tv_1, tv_2)$ are for type errors such that the one in the following piece of code:

```
signature s = sig val f : 'a -> 'b end
structure S = struct val rec f = fn x => x end
structure T = S :> s
```

In this piece of code, `f` is specified in the signature `s` as a function where its argument's type can differ from its body's type. In the structure `S`, the function `f` is declared as the identity function and so its argument's type has to be the same as its body's type. Finally `S` is constrained by `s`. Therefore, we report an explicit type variable clash between `'a` and `'b`. This is a special kind of *too general* errors.

Rules (SM4) and (SM5) of the extension of our constraint solver defined below in Fig. 20, make use of the function appTFM that applies type functions (extracted from a structure) to a type or an environment (related to a signature) and which is defined as follows:

$$
\begin{array}{lll}
\mathsf{appTFM}(\tau\,\mu, tfm) & = & \begin{cases} \tau'[\{\alpha \mapsto \mathsf{appTFM}(\tau, tfm)\}], \\ \quad \text{if } tfm(\mu) = \Lambda\alpha.\,\tau' \\ \alpha_{\mathtt{dum}}, \text{if } tfm(\mu) \in \mathsf{Var} \\ \text{undefined, if } \mu \text{ is of the form } \mu'^{\overline{d} \cup d} \\ \mathsf{appTFM}(\tau, tfm)\,\mu, \text{otherwise} \end{cases} \\
\mathsf{appTFM}(\tau_1 \to \tau_2, tfm) & = & \mathsf{appTFM}(\tau_1, tfm) \to \mathsf{appTFM}(\tau_2, tfm) \\
\mathsf{appTFM}(\forall\overline{\alpha}.\,\tau, tfm) & = & \forall\overline{\alpha}.\,\mathsf{appTFM}(\tau, tfm) \\
\mathsf{appTFM}(\Lambda\alpha.\,\tau, tfm) & = & \Lambda\alpha.\,\mathsf{appTFM}(\tau, tfm) \\
\mathsf{appTFM}(e_1;e_2, tfm) & = & \mathsf{appTFM}(e_1, tfm);\mathsf{appTFM}(e_2, tfm) \\
\mathsf{appTFM}(\downarrow id{=}x, tfm) & = & \downarrow id{=}\mathsf{appTFM}(x, tfm) \\
\mathsf{appTFM}(x^{\overline{d}}, tfm) & = & \mathsf{appTFM}(x, tfm)^{\overline{d}} \\
\mathsf{appTFM}(x, tfm) & = & x, \text{ if none of the above applies}
\end{array}
$$

Let us define tyvars and $\mathsf{nonDumVars}$ that are used by some functions and predicates defined below. The function tyvars is defined as follows: $\mathsf{tyvars}(x)$ is the set of syntactic forms belonging to TyVar and occurring in x whatever x is. The function $\mathsf{nonDumVars}$ is defined as follows: $\mathsf{nonDumVars}(x) = \mathsf{vars}(x) \setminus \mathsf{DumVar}$.

Rule (SC2) of the extension of our constraint solver defined below in Fig. 20 uses the predicate $\mathsf{abstract}$ which is used to rename the type constructor names declared in an environment and defined as follows:

$$
\begin{array}{l}
\langle \Delta, e, \{\gamma_1\} \uplus \cdots \uplus \{\gamma_n\}\rangle \xrightarrow{\mathsf{abstract}} \mathsf{appTFM}(e, tfm) \\
\quad \text{if } \mathsf{dja}(\mathsf{nonDumVars}(\Delta), \gamma_1', \ldots, \gamma_n', \alpha_1, \ldots, \alpha_n) \\
\quad \text{and } tfm = \cup_{i \in \{1,\ldots,n\}} \{\gamma_i \mapsto \Lambda\alpha_i.\,\alpha_i\,\gamma_i'\}
\end{array}
$$

Fig. 19 defines our algorithm $\mathsf{genExTyVar}$ that generalises explicit type variables in environments. It also uses renamings of the form $rentv$ defined as follows:

$$
\begin{array}{lll}
rentv & \in \mathsf{RenTv} & ::= \{rentv \in \mathsf{TyVar} \to \mathsf{ITyVar} \mid \\
 & & \quad rentv \text{ is injective} \\
 & & \quad \wedge\ \mathsf{dja}(\mathsf{dom}(rentv), \mathsf{ran}(rentv))\}
\end{array}
$$

These renamings are applied using the renTv function which is defined as follows:

$$
\begin{array}{ll}
\mathsf{renTv}(\alpha, rentv) & = \alpha \\
\mathsf{renTv}(\widehat{tv}, rentv) & = rentv(tv) \\
\mathsf{renTv}(\tau\,\mu, rentv) & = \mathsf{renTv}(\tau, rentv)\,\mu \\
\mathsf{renTv}(\tau_1 \to \tau_2, rentv) & = \mathsf{renTv}(\tau_1, rentv) \to \mathsf{renTv}(\tau_2, rentv) \\
\mathsf{renTv}(\tau^{\overline{d}}, rentv) & = \mathsf{renTv}(\tau, rentv)^{\overline{d}}
\end{array}
$$

This function is partially defined. It is not defined on types of the form $\theta \cdot \tau$ because theses forms cannot occur in environments in "solved" forms. Moreover when applying renTv to an internal type and a renaming of explicit type variables, then the explicit type variables occurring in the internal type have to be the domain of the renaming. This is always the case when calling renTv in Fig. 19.

Fig. 20 extends our constraint solver to deal with our new constraint terms.

Figure 19 Generalisation of explicit type variables

$$
\begin{array}{ll}
\langle \Delta, e_1;e_2\rangle \xrightarrow{\mathsf{genExTyVar}} e_2';e_2' & \Leftrightarrow \langle \Delta, e_1\rangle \xrightarrow{\mathsf{genExTyVar}} e_1' \text{ and } \langle \Delta;e_1', e_2\rangle \xrightarrow{\mathsf{genExTyVar}} e_2' \\
\langle \Delta, \downarrow id{=}e\rangle \xrightarrow{\mathsf{genExTyVar}} \downarrow id{=}e' & \Leftrightarrow \langle \Delta, e\rangle \xrightarrow{\mathsf{genExTyVar}} e' \\
\langle \Delta, \downarrow vid{=}\tau\rangle \xrightarrow{\mathsf{genExTyVar}} \downarrow vid{=}\forall\overline{\alpha}_0.\,\mathsf{renTv}(\tau, rentv) & \Leftrightarrow \mathsf{tyvars}(\tau) = \mathsf{dom}(rentv) \text{ and } \overline{\alpha}_0 = \mathsf{ran}(rentv) \text{ and } \mathsf{dja}(\mathsf{nonDumVars}(\Delta), \overline{\alpha}_0) \\
\langle \Delta, \downarrow vid{=}\forall\overline{\alpha}.\,\tau\rangle \xrightarrow{\mathsf{genExTyVar}} \downarrow vid{=}\forall(\overline{\alpha}\cup\overline{\alpha}_0).\,\mathsf{renTv}(\tau, rentv) & \Leftrightarrow \mathsf{tyvars}(\tau) = \mathsf{dom}(rentv) \text{ and } \overline{\alpha}_0 = \mathsf{ran}(rentv) \text{ and } \mathsf{dja}(\mathsf{nonDumVars}(\Delta), \overline{\alpha}_0) \\
\langle \Delta, e^{\overline{d}}\rangle \xrightarrow{\mathsf{genExTyVar}} e'^{\overline{d}} & \Leftrightarrow \langle \Delta, e\rangle \xrightarrow{\mathsf{genExTyVar}} e' \\
\langle \Delta, e\rangle \xrightarrow{\mathsf{genExTyVar}} e, & \text{if none of the above applies}
\end{array}
$$

Rule (S10), originally defined in Sec. C.2, is updated to handle explicit type variables as internal types.

Rules (S11)-(S13) are to handle our new cases of internal types and internal type constructors.

Rule (B1) is redefined so that it builds up the semantics of binders. The reason is that when checking if a signature matches a structure, we want the corresponding environments fully built up. Building the binders when solving them allows having a shallow building function, as the one we currently have, that does not need to go down structure or signature binders.

Rules (SM1)-(SM13) check whether a signature matches a structure. These rules are used for both translucent and opaque constraints. If $\mathtt{match}(\Delta, \overline{d}, tfm, e_1, e_2) \rightarrow^* \mathtt{match}(\Delta', \overline{d}', tfm', e_1', e_2')$ using rules (SM1)-(SM13) then $e_1 = e_1'$.

Rule (SM4) checks that a type scheme from the signature does not allow generating types that the corresponding type scheme in the structure cannot generate. Before checking that, the gathered type functions are applied to the type extracted from the binding coming from the signature (τ_1). This is where the instantiation of a signature is performed in our system. Finally, explicit type variables occurring in the generated binder are generalised. The generated type scheme is built from τ_1 and not from τ_2 in case the binding from the signature is a dummy binding. If the binding from the signature was a dummy binding and the corresponding binding from the structure was not a dummy binding, then we do not want to generate an unlabelled non-dummy binding that could therefore lead to a type error because this type error might then not be dependent on the label associated to the specification for which the binder has been generated (the specification would not be part of the reported error).

Rules (SM6) and (SM11) gather type functions defined by the structure. These type functions are applied to the signature during the process of checking the matching (in rules (SM4) and (SM5)). Once again, extra care has to be taken when the binder from the signature is a dummy binder, so that the algorithm does not generate a non-dummy binder.

The other (SMi) rules are fairly straightforward.

Rule (SC5) is just so that the same mechanism can be used for opaque and translucent signatures.

Rule (SC1) for translucent signature just checks that the signature matches the structure. It does not need further computation because the resulting structure is computed while checking the matching.

Rule (SC3) for opaque signatures checks that the signature matches the structure and generates a new structure based on the signature. The generated structure is the signature where the internal type constructor names are renamed and where the explicit type variables are generalised.

Constraint filtering (Minimisation and enumeration). We extend our filtering function as follows:

$$
\begin{array}{ll}
\mathsf{filt}(e_1{:}e_2, \overline{l}_1, \overline{l}_2) & = \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2){:}\mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2) \\
\mathsf{filt}(e_1{:>}e_2, \overline{l}_1, \overline{l}_2) & = \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2){:>}\mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2) \\
\mathsf{filt}(ev, \overline{l}_1, \overline{l}_2) & = ev
\end{array}
$$

We now need the filtering of unlabelled environment variables because we now allow unlabelled environment variables to occur within environments of the form $e_1{:}e_2$ or $e_1{:>}e_2$.

Because explicit type variables can now bind internal types and not only internal type variables, we also need to update our filtering function by replacing $\mathsf{toDumVar}(\alpha) = \alpha_{\mathtt{dum}}$ by:

$$\mathsf{toDumVar}(\tau) = \alpha_{\mathtt{dum}}$$

Slicing. We extend our tree syntax for programs as follows:

$$
\begin{array}{l}
\mathsf{Class} ::= \cdots \mid \mathtt{sigdec} \mid \mathtt{sigexp} \mid \mathtt{spec} \\
\mathsf{Prod} ::= \cdots \\
\qquad \mid \mathtt{sigdecDec} \\
\qquad \mid \mathtt{sigexpSig} \\
\qquad \mid \mathtt{specVal} \mid \mathtt{specTyp} \mid \mathtt{specDat} \mid \mathtt{specStr} \\
\qquad \mid \mathtt{strexpTr} \mid \mathtt{strexpOp}
\end{array}
$$

We also extend our function getDot that associates dot markers to node kinds as follows:

$$
\begin{array}{l}
\mathsf{getDot}(\langle \mathtt{sigdec}, prod\rangle) = \mathtt{dotD} \\
\mathsf{getDot}(\langle \mathtt{sigexp}, prod\rangle) = \mathtt{dotS} \\
\mathsf{getDot}(\langle \mathtt{spec}, prod\rangle) \quad = \mathtt{dotD}
\end{array}
$$

Finally, Fig. 21 extends our function toTree that transforms a term $term$ into a tree $tree$.

C.4 Reporting unmatched errors

There is a kind of error involving signatures that is not handled by the constraint solver as defined above: the "unmatched" errors.

For example, in

```
signature s = sig val fool : int end
structure S = struct val foo = 1 val bar = 2 end
structure T = S :> s
```

the specification `fool` from the signature `s` is not matched in the structure `S`, but `s` constrains `S` in `T`. This error could be solved in many ways, such as: (1) one could replace `fool` by `foo` in `s`, (2) one could replace `foo` by `fool` in `S`, (3) one could constrain `S` using a different signature, (4) one could bind `s` or `S` to other expressions.

For this error we would like to report that `fool` specified in `s` is not any of `foo` or `bar` declared in `S`, but `s` constrains `S`.

For that we need to be able to check that indeed `fool` is not any of the declarations of `S`.

With the system as described above, we cannot report such errors because we do not have any way of knowing whether an environment is constituted by the binders corresponding to *all* the declarations of a structure. As a matter of fact, this is not possible with the current system because of the way constraint filtering can replace environment variables and binders by $\boxdot$.

We will now show how to extend our system to report such errors.

Constraint syntax. We extend our environment with a new empty environment as follows:

$$\mathsf{Env} ::= \cdots \mid \odot$$

The meaning of the environment $\odot$ lies in between the meaning of $\boxdot$ and the meaning of environment variables.

Figure 20 Constraint solving for signature related constraints

Some kinds of errors are not handled by the system presented in this section, although our implementation handles them. For more information please refer to the introductory paragraph of this section (Sec. C.3).

equality simplification

(S10) $\texttt{solve}(\langle u, e\rangle, \overline{d}, \tau_1{=}\tau_2) \to \texttt{solve}(\langle u, e\rangle, \overline{d}, \tau{=}\mathsf{build}(u, \theta \cdot \tau'))$ if $\{\tau_1, \tau_2\} = \{\theta \cdot \tau', \tau\}$ and τ is of the form $\tau_3{\to}\tau_4$, $\tau_3\ \mu$, or $\widehat{tv}$

(S11) $\texttt{solve}(\Delta, \overline{d}, \widehat{tv_1}{=}\widehat{tv_2}) \to \texttt{err}(\langle \texttt{TyVarClash}(tv_1, tv_2), \overline{d}\rangle)$, if $tv_1 \neq tv_2$

(S12) $\texttt{solve}(\Delta, \overline{d}, \tau_1{=}\tau_2) \to \texttt{solve}(\Delta, \overline{d}, \widetilde{tv}{=}\mu)$, if $(\{\tau_1, \tau_2\} = \{\widehat{tv}, \tau{\to}\tau'\}$ and $\mu = \texttt{ar})$ or $\{\tau_1, \tau_2\} = \{\widehat{tv}, \tau\ \mu\}$), for some tv, τ, τ', μ

(S13) $\texttt{solve}(\Delta, \overline{d}, \mu_1{=}\mu_2) \to \texttt{err}(\langle \texttt{TooGeneral}(\mu_1, \mu_2), \overline{d}\rangle)$, if $\{\mu_1, \mu_2\} \in \{\{\widetilde{tv}, \texttt{ar}\}, \{\widetilde{tv}, \gamma\}\}$, for some tv and γ

binders

(B1) $\texttt{solve}(\langle u, e\rangle, \overline{d}, {\downarrow}id{=}x) \to \texttt{succ}(\langle u, e\rangle; ({\downarrow}id \overset{\overline{d}}{=} \mathsf{build}(u, x)))$

signature constraints

(SC1) $\texttt{solve}(\langle u, e\rangle, \overline{d}, e_1{:}e_2) \to \texttt{succ}(\Delta)$, if $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \to^* \texttt{succ}(\Delta, tfm)$,

(SC2) $\texttt{solve}(\langle u, e\rangle, \overline{d}, e_1{:}e_2) \to \texttt{err}(er)$, if $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \to^* \texttt{err}(er)$,

(SC3) $\texttt{solve}(\langle u, e\rangle, \overline{d}, e_1{:}{>}e_2) \to \texttt{succ}(\langle u', e; e_2''\rangle)$, if $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \to^* \texttt{succ}(\langle u', e'\rangle, tfm)$ and $\langle\langle u', e'\rangle, \mathsf{build}(u, e_2), \mathsf{dom}(tfm)\rangle \xrightarrow{\mathsf{abstract}} e_2'$ and $\langle\langle u', e'\rangle, e_2'\rangle \xrightarrow{\mathsf{genExTyVar}} e_2''$

(SC4) $\texttt{solve}(\langle u, e\rangle, \overline{d}, e_1{:}{>}e_2) \to \texttt{err}(er)$, if $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \to^* \texttt{err}(er)$

(SC5) $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \to state$, if $\texttt{match}(\langle u, e\rangle, \overline{d}, \varnothing, \mathsf{build}(u, e_1), \mathsf{build}(u, e_2)) \to^* state$

structure/signature matching

(SM1) $\texttt{match}(\Delta, \overline{d}, tfm, e, \boxdot) \to \texttt{succ}(\Delta, tfm)$

(SM2) $\texttt{match}(\Delta, \overline{d}, tfm, e, e_1; e_2) \to \texttt{match}(\Delta', \overline{d}, tfm', e, e_2)$, if $\texttt{match}(\Delta, \overline{d}, tfm, e, e_1) \to^* \texttt{succ}(\Delta', tfm')$

(SM3) $\texttt{match}(\Delta, \overline{d}, tfm, e, e_1; e_2) \to \texttt{err}(er)$, if $\texttt{match}(\Delta, \overline{d}, tfm, e, e_1) \to^* \texttt{err}(er)$

(SM4) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}vid{=}\sigma_1) \to \texttt{succ}(\Delta; e_0, tfm)$, if $e(vid) = \sigma_2$ and $\forall i \in \{1, 2\}$. ($\sigma_i = \forall\overline{\alpha}_i.\ \tau_i$ or ($\sigma_i = \tau_i$ and $\overline{\alpha}_i = \varnothing$)) and $\tau_1' = \mathsf{appTFM}(\tau_1, tfm)$ and $\texttt{solve}(\Delta, \overline{d}, \tau_1'{=}\tau_2) \to^* \texttt{succ}(\langle u', e'\rangle)$ and $\tau = \mathsf{build}(u', \tau_1')$ and $\langle\langle u', e'\rangle, {\downarrow}vid \overset{\overline{d}}{=} \forall(\overline{\alpha}_1 \cup \overline{\alpha}_2) \cap \mathsf{vars}(\tau).\ \tau\rangle \xrightarrow{\mathsf{genExTyVar}} e_0$

(SM5) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}vid{=}\sigma_1) \to \texttt{err}(er)$, if $e(vid) = \sigma_2$ and $\forall i \in \{1, 2\}$. ($\sigma_i = \forall\overline{\alpha}_i.\ \tau_i$ or ($\sigma_i = \tau_i$ and $\overline{\alpha}_i = \varnothing$)) and $\texttt{solve}(\Delta, \overline{d}, \mathsf{appTFM}(\tau_1, tfm){=}\tau_2) \to^* \texttt{err}(er)$

(SM6) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}tc{=}\theta_1) \to \texttt{succ}(\Delta; ({\downarrow}tc \overset{\overline{d}}{=} \theta_2'), tfm')$, if $e(tc) = \theta_2$ and (if $\theta_1 \in \mathsf{DumVar}$ then $\theta_2' = \theta_1$ else $\theta_2' = \theta_2^{\mathsf{deps}(\theta_1)}$) and (if $\theta_1 = \Lambda\alpha.\ (\alpha\ \gamma)^{\overline{d}'}$ then $tfm' = tfm \boxplus \{\gamma \mapsto \theta_2'\}$ else $tfm' = tfm$)

(SM7) $\texttt{match}(\langle u_1, e_1\rangle, \overline{d}, tfm, e, {\downarrow}sid{=}e_0) \to \texttt{succ}(\langle u_2, e_1; ({\downarrow}sid \overset{\overline{d}}{=} \mathsf{diff}(e_1, e_2))\rangle, tfm')$, if $e(sid) = e_0'$ and $\texttt{match}(\langle u_1, e_1\rangle, \overline{d}, tfm, e_0', e_0) \to^* \texttt{succ}(\langle u_2, e_2\rangle, tfm')$

(SM8) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}sid{=}e_0) \to \texttt{err}(er)$, if $e(sid) = e_0'$ and $\texttt{match}(\Delta, \overline{d}, tfm, e_0', e_0) \to^* \texttt{err}(er)$

(SM9) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}vid{=}is_1) \to \texttt{succ}(\Delta; ({\downarrow}vid \overset{\overline{d}'}{=} vid), tfm)$, if $e[\![vid]\!] = is_2$ and $\mathsf{deps}(is_2) = \overline{d}'$ and ($\texttt{solve}(\Delta, \overline{d}, is_1{=}is_2) \to^* \texttt{succ}(\Delta')$ or $\mathsf{strip}(is_1) = \mathsf{v}$)

(SM10) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}vid{=}is_1) \to \texttt{err}(er)$, if $e[\![vid]\!] = is_2$ and $\mathsf{strip}(is_1) \neq \mathsf{v}$ and $\texttt{solve}(\Delta, \overline{d}, is_1{=}is_2) \to^* \texttt{err}(er)$

(SM11) $\texttt{match}(\Delta, \overline{d}, tfm, e, {\downarrow}id{=}x) \to \texttt{succ}(\Delta; ({\downarrow}id{=}\mathsf{toDumVar}(x)), tfm')$, if $e(id)$ is undefined and (if $x = \Lambda\alpha.\ (\alpha\ \gamma)^{\overline{d}'}$ then $tfm' = tfm \boxplus \{\gamma \mapsto \alpha_{\texttt{dum}}\}$ else $tfm' = tfm$)

(SM12) $\texttt{match}(\Delta, \overline{d}, tfm, e, ev) \to \texttt{succ}(\Delta; ev, tfm)$

(SM13) $\texttt{match}(\Delta, \overline{d}, tfm, e, e'^{\overline{d}'}) \to \texttt{match}(\Delta, \overline{d} \cup \overline{d}', tfm, e, e')$

Figure 21 Extension of our slicing algorithm with signatures

Signature declarations	$\mathsf{toTree}(\texttt{signature}\ sigid \stackrel{l}{=} sigexp)$	$= \langle\langle \texttt{sigdec}, \texttt{sigdecDec}\rangle, l, \langle sigid, \mathsf{toTree}(sigexp)\rangle\rangle$
Signature expressions	$\mathsf{toTree}(sigid^l)$	$= \langle\langle \texttt{sigexp}, \texttt{sigexpId}\rangle, l, \langle sigid\rangle\rangle$
	$\mathsf{toTree}(\texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end})$	$= \langle\langle \texttt{sigexp}, \texttt{sigexpSig}\rangle, l, \langle \mathsf{toTree}(spec_1), \ldots, \mathsf{toTree}(spec_n)\rangle\rangle$
Specifications	$\mathsf{toTree}(\texttt{val}\ vid :^l ty)$	$= \langle\langle \texttt{spec}, \texttt{specVal}\rangle, l, \langle vid, \mathsf{toTree}(ty)\rangle\rangle$
	$\mathsf{toTree}(\texttt{type}\ dn^l)$	$= \langle\langle \texttt{spec}, \texttt{specTyp}\rangle, l, \langle \mathsf{toTree}(dn)\rangle\rangle$
	$\mathsf{toTree}(\texttt{datatype}\ dn \stackrel{l}{=} cd)$	$= \langle\langle \texttt{spec}, \texttt{specDat}\rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(cd)\rangle\rangle$
	$\mathsf{toTree}(\texttt{structure}\ sid :^l sigexp)$	$= \langle\langle \texttt{spec}, \texttt{specStr}\rangle, l, \langle sid, \mathsf{toTree}(sigexp)\rangle\rangle$
Structure expressions	$\mathsf{toTree}(sexp :^l sigexp)$	$= \langle\langle \texttt{strexp}, \texttt{strexpTr}\rangle, l, \langle \mathsf{toTree}(sexp), \mathsf{toTree}(sigexp)\rangle\rangle$
	$\mathsf{toTree}(sexp :>^l sigexp)$	$= \langle\langle \texttt{strexp}, \texttt{strexpOp}\rangle, l, \langle \mathsf{toTree}(sexp), \mathsf{toTree}(sigexp)\rangle\rangle$
Programs	$\mathsf{toTree}(topdec_1\,;\cdots;\,topdec_n)$	$= \langle \texttt{dotD}, \langle \mathsf{toTree}(topdec_1), \ldots, \mathsf{toTree}(topdec_n)\rangle\rangle$

The difference between $\boxdot$ and $\odot$ is that the second one will be used to indicate that we filtered out an environment which has the potential to bind (either an environment variable or a binder) and not just, say, an equality constraint.

The difference between $\odot$ and an environment variable is that in an environment of the form $\odot;e$, $\odot$ does not shadow e.

Constraint solving. The extra environment $\odot$ will be allowed to exist within unification contexts. Given a unification state, if Δ occurs in it, then Δ is of the form $\langle u, e\rangle$ where $e = \boxdot;e_1;\cdots;e_n$ where each e_i can either be an environment variable, a labelled binder or $\odot$.

Because $\odot$ can occur in unification contexts, we extend our environment application functions as follows:

$$(e;\odot)(id) = e(id)$$
$$(e;\odot)[\![id]\!] = e[\![id]\!]$$

Let us extend error kinds as follows:

$$ek \in \mathsf{ErrKind} ::= \cdots \mid \texttt{unmatched}(id, \overline{id})$$

Fig. 22 extends our constraint solver with rules to handle unmatched errors: rule (SM11) replaces the previous rule (SM11) from Fig. 20 and rules (SM14) and (N2) are new.

Rules (SM11) and (SM14) make use of the predicate complete (similar to hiding) which is defined as follows:

$$\mathsf{complete}(e) \Leftrightarrow \begin{cases} (e \text{ of the form } \downarrow id \stackrel{\overline{d}}{=} x \\ \quad \text{and } x \notin \mathsf{DumVar} \cup \{\texttt{a}\}) \\ \text{or } (e \text{ of the form } e_1;e_2 \\ \quad \text{and } \forall i \in \{1,2\}.\ \mathsf{complete}(e_i)) \\ \text{or } e = \boxdot \end{cases}$$

A "solved" environment (occurring in a unification context) is said to be complete if it is not composed by an environment variable, a filtered binder or a dummy binder.

Rule (SM14) makes use of the function getBinders which gathers the identifiers bound in its argument:

$$\begin{aligned} &\mathsf{getBinders}(e_1;e_2) &&= \mathsf{getBinders}(e_1) \cup \mathsf{getBinders}(e_2) \\ &\mathsf{getBinders}(\boxdot) &&= \varnothing \\ &\mathsf{getBinders}(\downarrow id \stackrel{\overline{d}}{=} x) &&= \{id\} \end{aligned}$$

Constraint filtering (Minimisation and enumeration). We add a new rule to filter $\odot$ and update the filtering of labelled environment as follows:

$$\mathsf{filt}(e^l, \overline{l}_1, \overline{l}_2) = \begin{cases} e^l, & \text{if } l \in \overline{l}_1 \setminus \overline{l}_2 \\ \mathsf{dum}(e), & \text{if } l \in \overline{l}_2 \\ \odot, & \text{if } l \notin \overline{l}_1 \cup \overline{l}_2 \text{ and } e \in \mathsf{Var} \cup \mathsf{Bind} \\ \boxdot, & \text{otherwise} \end{cases}$$
$$\mathsf{filt}(\odot, \overline{l}_1, \overline{l}_2) = \odot$$

Slicing. We also need to modify our slicing algorithm. Consider the following piece of code:

```
signature s = sig val x : int val y : bool end
structure S :  s = struct val x = 1 val y = true end
structure T :> s = struct val x = 1 val y = true end
val u = let open T val z = y open S
        in fn w => (w z, w x)
        end
```

where in the fn-expression, `z` is the `y` from `T` and `x` comes from `S` via the structure opening. The structures `S` and `T` have the same structure body constrained by the same signature `s`, but `S` has a translucent signature while `T`'s signature is opaque.

This piece of code is untypable because `w` has a monomorphic type and is applied to `z` which is the Boolean `y` defined in `T`, and it is also applied to `x` which is an integer defined in `S`.

With our current slicing algorithm, one of the type error slice we obtain would be as follows:

```
⟨..signature s = sig val x : ⟨..⟩ val y : bool end
 ..structure S :  s = struct val x = 1 end
 ..structure T :> s = ⟨..⟩
 ..⟨..open T..val z = y..open S..fn w => ⟨..w z..w x..⟩..⟩..⟩
```

which is not minimal: `s` does not match `S` because `y` is not declared in `S`.

The problem comes from our tidying of declarations in structure expressions. We therefore need to update our tidying function so that it does not discard empty dot declarations:

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

With this new tidy function, we would then obtain a slice as follows:

```
⟨..signature s = sig val x : ⟨..⟩ val y : bool end
 ..structure S :  s = struct ⟨..⟩ val x = 1 end
 ..structure T :> s = ⟨..⟩
 ..⟨..open T..val z = y..open S..fn w => ⟨..w z..w x..⟩..⟩..⟩
```

We also have to replace our constraint generation rule for dot declarations, in order to generate markers of discarded binders:

$$\texttt{dot-d}(\langle pt_1, \ldots, pt_n\rangle) \triangleright [e_1;\cdots;e_n];\odot \Leftarrow$$
$$pt_1 \triangleright e_1 \wedge \cdots \wedge pt_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n)$$

However, this modification is not enough because binders are generated for cb's, pat's, and dn's.

For example, we would like to generate a marker of discarded binder for the following declaration: `datatype 'a t = ⟨..⟩`.

Figure 22 Constraint solving rules handling unmatched errors

Some kinds of errors are not handled by the system presented in this section, although our implementation handles them. For more information please refer to the introductory paragraph of Sec. C.3.

structure/signature matching

(SM11) $\texttt{match}(\Delta, \overline{d}, tfm, e, \downarrow id{=}x) \rightarrow \texttt{succ}(\Delta;(\downarrow id \stackrel{\overline{d}}{=} y), tfm')$, if $e(id)$ is undefined and $\neg\mathsf{complete}(e)$ and $y = \mathsf{toDumVar}(x)$ and (if $x = \Lambda\alpha.\,\alpha\,\gamma$ then $tfm' = tfm \boxplus \{\gamma \mapsto \alpha_{\texttt{dum}}\}$ else $tfm' = tfm$)

(SM14) $\texttt{match}(\Delta, \overline{d}, tfm, e, \downarrow id{=}x) \rightarrow \texttt{err}(\langle \texttt{unmatched}(id, \overline{id}), \overline{d}\rangle)$, if $e(id)$ is undefined and $\mathsf{complete}(e)$ and where $\overline{id} = \mathsf{getBinders}(e)$

(SM15) $\texttt{match}(\Delta, \overline{d}, tfm, e, \odot) \rightarrow \texttt{succ}(\Delta;\odot, tfm)$

empty

(N2) $\texttt{solve}(\Delta, \overline{d}, \odot) \rightarrow \texttt{succ}(\Delta;\odot)$

First, let us replace the dot terms for cb's. We need to do so because we want to generate markers of discarded binders only for cb dot terms, but not for expressions and types. We replace these dot terms as follows:

$$\texttt{dot-e}(\overrightarrow{pt}) \xrightarrow{\mathsf{ConBind}\triangleright} \texttt{dot-c}(\overrightarrow{pt})$$

We redefine the constraint generation rules for the forms $\texttt{dot-n}(\overrightarrow{pt})$ and $\texttt{dot-p}(\overrightarrow{pt})$, and we introduce a new constraint generation rule for the forms $\texttt{dot-c}(\overrightarrow{pt})$ as follows:

$$\begin{array}{l}\texttt{dot-n}(\langle pt_1, \ldots, pt_n\rangle) \triangleright \langle \alpha, \alpha', [e_1;\cdots;e_n];\odot, \boxdot\rangle \\ \quad \Leftarrow\ pt_1 \triangleright e_1 \wedge \cdots \wedge pt_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \alpha, \alpha') \\ \texttt{dot-p}(\langle pat_1, \ldots, pat_n\rangle) \triangleright \langle \alpha, e_1;\cdots;e_n;\odot\rangle \\ \quad \Leftarrow\ pat_1 \triangleright e_1 \wedge \cdots \wedge pat_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \alpha) \\ \texttt{dot-c}(\langle pt_1, \ldots, pt_n\rangle) \triangleright \langle \alpha, [e_1;\cdots;e_n];\odot\rangle \\ \quad \Leftarrow\ pt_1 \triangleright e_1 \wedge \cdots \wedge pt_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \alpha)\end{array}$$

We add a new dot marker to the set Dot as follows:

$$\mathsf{Dot} ::= \cdots \mid \texttt{dotC}$$

Finally, we extend the toTree function as follows:

$$\begin{array}{c}\mathsf{toTree}(\texttt{dot-n}(\langle pt_1, \ldots, pt_n\rangle)) = \\ \langle \texttt{dotC}, \langle \mathsf{toTree}(pt_1), \ldots, \mathsf{toTree}(pt_n)\rangle\rangle\end{array}$$

C.5 Further extensions

We are currently extending the formal presentation of our TES to handle features such as functors, non-recursive value declarations, type annotations, or long identifiers. These features are already handled by our implementation and we invite the reader to try it and read its source code for more details on how the features are handled.

D. Extensions for a better error handling

D.1 Merged minimal slices

With the constraint solver as defined above, our TES would report two minimal *unmatched* type error slices for the following piece of code:

```
structure S = struct val (fool, barr, x, y) = (1, 2, 3, 4) end
signature s = sig val foo : int val bar : int val x : int end
structure T = S :> s
```

One of the type error is that the specification `foo` in `s` is not matched in the structure `S` (that declares `fool`, `barr`, `x` and `y`), but `s` constrains `S` in `T`. The other error is similar but concerns the specification `bar`.

This is another typical example where finding and reporting merged minimal error slices would be useful (see Sec. 2.3, that presents another example of merged minimal error slices). For the example above, instead of the two reports described above, we would prefer a highlighting that would look like:

```
structure S = struct val (fool, barr, x, y) = (1, 2, 3) end
signature s = sig val foo : int val bar : int val x : int end
structure T = S :> s
```

This highlighting shows that `foo` and `bar` are not matched in the structure `S`, but also suppose that `x` might not be the matching for `foo` or `bar` as `x` is specified in the signature `s`. Note that `x` is still reported because we can't know if `x` in the structure `S` is definitely not the matching of, e.g., `foo` in the signature `s`.

Note that we do not want to find the two minimal error reports and then merge them into a single report, but we directly want to generate the merged error.

We could obtain this slice by altering the part of our constraint solver defined in Fig. 20 and Fig. 22.

First, we want unmatched error kinds to be as follows instead (we replace the previous form by this new one):

$$ek \in \mathsf{ErrKind} ::= \cdots \mid \texttt{unmatched}(\overline{id}_1, \overline{id}_2, \overline{id}_3)$$

For the highlighting presented above, the error kind would then be $\texttt{unmatched}(\overline{id}_1, \overline{id}_2, \overline{id}_3)$, where $\overline{id}_1$ is the set of identifiers highlighted in dark grey (the identifiers specified in `s` that are not declared in `S`), $\overline{id}_2$ is the set of identifiers highlighted with the darkest grey (the identifiers declared in `S` that are not specified in `s`) and $\overline{id}_3$ is the set of identifiers highlighted in light grey (the identifiers both specified in `s` and declared in `S`).

Then, when checking if a signature matches a structure, in order to gather (1) the identifiers that are specified in the signature but not declared in the structure, (2) the identifiers that are declared in the structure but not specified in the signature, and (3) the identifiers that are both specified in the signature and declared in the structure, we extend our "match" states as follows:

$$\begin{array}{lll} unm \in \mathsf{Unmatched} & ::= \langle \overline{id}_1, \overline{id}_2\rangle \\ state \in \mathsf{State} & ::= \cdots \\ & \quad \mid \texttt{match}(\Delta, \overline{d}, tfm, unm, e_1, e_2) \\ & \quad \mid \texttt{succ}(\Delta, tfm, unm)\end{array}$$

Finally, Fig. 23 updates the rules defined in Fig. 20 and Fig. 22 to handle the reporting of merged unmatched errors. Rules (SC1), (SC2), (SC3) and (SC4) are as before and are not repeated here. Rules (SC6) and (SM16) are new and replace rule (SM14).

The main difference between this new algorithm and the one presented in Fig. 20 and Fig. 22, is that our new algorithm gathers the identifiers that are both specified in the signature and declared in the structure (rules (SM4), (SM6), and (SM7)) and also gathers the identifier that are not matched in the structure (rule (SM11)). If there exists such an identifier, it means that there is an unmatched error. We then wait to check the matching of the entire signature against the structure to finally report all such unmatched identifiers in a single error report (rules (SC6) and (SM16)).

Note that such type error reports (for unmatched errors) are still imperfect. For example, the highlighting above does not show that {`fool`, `barr`, `x`, `y`} is precisely the set of identifiers declared in the structure `S`. Similarly the highlighting does not show that {`foo`, `bar`, `x`} is precisely the set of identifiers specified in the signature `s`. We could then consider the following convention when highlighting a type error: if all the identifiers declared in a structure or specified in

Figure 23 Constraint solving to handle merged unmatched errors

Some kinds of errors are not handled by the system presented in this section, although our implementation handles them. For more information please refer to the introductory paragraph of Sec. C.3.

signature constraints

(SC5) $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \rightarrow \texttt{succ}(\Delta', tfm)$, if $\mathsf{build}(u, e_1) = e_1''$ and $\mathsf{build}(u, e_2) = e_2''$ and $\texttt{match}(\langle u, e\rangle, \overline{d}, \varnothing, \langle\varnothing, \varnothing\rangle, e_1'', e_2'') \rightarrow^* \texttt{succ}(\Delta', tfm, unm)$ and $(unm = \langle\varnothing, \overline{id}_2\rangle$ or $\neg\mathsf{complete}(e_1''; e_2''))$

(SC6) $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \rightarrow \texttt{err}(\langle ek, \overline{d}\rangle)$, if $\mathsf{build}(u, e_1) = e_1''$ and $\mathsf{build}(u, e_2) = e_2''$ and $\texttt{match}(\langle u, e\rangle, \varnothing, \varnothing, \langle\varnothing, \varnothing\rangle, e_1'', e_2'') \rightarrow^* \texttt{succ}(\Delta', tfm, unm)$ and $unm = \langle \overline{id}_1, \overline{id}_2\rangle$ and $\overline{id}_1 \neq \varnothing$ and $\mathsf{complete}(e_1''; e_2'')$ and $ek = \texttt{unmatched}(\overline{id}_1, \mathsf{getBinders}(e_1'') \setminus \overline{id}_2, \overline{id}_2)$

(SC7) $\texttt{prematch}(\langle u, e\rangle, \overline{d}, e_1, e_2) \rightarrow \texttt{err}(er)$, if $\texttt{match}(\langle u, e\rangle, \overline{d}, \varnothing, \langle\varnothing, \varnothing\rangle, \mathsf{build}(u, e_1), \mathsf{build}(u, e_2)) \rightarrow^* \texttt{err}(er)$

structure/signature matching

(SM1) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, \boxdot) \rightarrow \texttt{succ}(\Delta, tfm, unm)$

(SM2) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, e_1;e_2) \rightarrow \texttt{match}(\Delta', \overline{d}, tfm', unm', e, e_2)$,
if $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, e_1) \rightarrow^* \texttt{succ}(\Delta', tfm', unm')$

(SM3) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, e_1;e_2) \rightarrow \texttt{err}(er)$,
if $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, e_1) \rightarrow^* \texttt{err}(er)$

(SM4) $\texttt{match}(\Delta, \overline{d}, tfm, \langle \overline{id}_1, \overline{id}_2\rangle, e, {\downarrow}vid{=}\sigma_1) \rightarrow \texttt{succ}(\Delta; e_0, tfm, \langle \overline{id}_1, \overline{id}_2 \cup \{vid\}\rangle)$,
if $e(vid) = \sigma_2$ and $\forall i \in \{1, 2\}.\ (\sigma_i = \forall\overline{\alpha}_i.\ \tau_i$ or $(\sigma_i = \tau_i$ and $\overline{\alpha}_i = \varnothing))$
and $\tau_1' = \mathsf{appTFM}(\tau_1, tfm)$ and $\texttt{solve}(\Delta, \overline{d}, \tau_1'{=}\tau_2) \rightarrow^* \texttt{succ}(\langle u', e'\rangle)$
and $\tau = \mathsf{build}(u', \tau_1')$ and $\langle\langle u', e'\rangle, {\downarrow}vid \stackrel{\overline{d}}{=} \forall(\overline{\alpha}_1 \cup \overline{\alpha}_2) \cap \mathsf{vars}(\tau).\ \tau\rangle \xrightarrow{\mathsf{genExTyVar}} e_0$

(SM5) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, {\downarrow}vid{=}\sigma_1) \rightarrow \texttt{err}(er)$,
if $e(vid) = \sigma_2$ and $\forall i \in \{1, 2\}.\ (\sigma_i = \forall\overline{\alpha}_i.\ \tau_i$ or $(\sigma_i = \tau_i$ and $\overline{\alpha}_i = \varnothing))$
and $\texttt{solve}(\Delta, \overline{d}, \mathsf{appTFM}(\tau_1, tfm){=}\tau_2) \rightarrow^* \texttt{err}(er)$

(SM6) $\texttt{match}(\Delta, \overline{d}, tfm, \langle \overline{id}_1, \overline{id}_2\rangle, e, {\downarrow}tc{=}\theta_1) \rightarrow \texttt{succ}(\Delta; ({\downarrow}tc \stackrel{\overline{d}}{=} \theta_2'), tfm', \langle \overline{id}_1, \overline{id}_2 \cup \{tc\}\rangle)$,
if $e(tc) = \theta_2$ and (if $\theta_1 \in \mathsf{DumVar}$ then $\theta_2' = \theta_1$ else $\theta_2' = \theta_2^{\mathsf{deps}(\theta_1)}$)
and (if $\theta_1 = \Lambda\alpha.\ (\alpha\,\gamma)^{\overline{d}'}$ then $tfm' = tfm \boxplus \{\gamma \mapsto \theta_2'\}$ else $tfm' = tfm$)

(SM7) $\texttt{match}(\langle u_1, e_1\rangle, \overline{d}, tfm, \langle \overline{id}_1, \overline{id}_2\rangle, e, {\downarrow}sid{=}e_0) \rightarrow \texttt{succ}(\langle u_2, e_1; ({\downarrow}sid \stackrel{\overline{d}}{=} \mathsf{diff}(e_1, e_2))\rangle, tfm', \langle \overline{id}_1, \overline{id}_2 \cup \{sid\}\rangle)$,
if $e(sid) = e_0'$
and $\texttt{match}(\langle u_1, e_1\rangle, \overline{d}, tfm, \langle\varnothing, \varnothing\rangle, e_0', e_0) \rightarrow^* \texttt{succ}(\langle u_2, e_2\rangle, tfm', unm')$
and $(unm' = \langle\varnothing, \overline{id}_2\rangle$ or $\neg\mathsf{complete}(e_0'; e_0))$

(SM16) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, {\downarrow}sid{=}e_0) \rightarrow \texttt{err}(\langle\texttt{unmatched}(\overline{id}_1, \mathsf{getBinders}(e_0') \setminus \overline{id}_2, \overline{id}_2), \overline{d}\rangle)$,
if $e(sid) = e_0'$
and $\texttt{match}(\Delta, \overline{d}, tfm, \langle\varnothing, \varnothing\rangle, e_0', e_0) \rightarrow^* \texttt{succ}(\Delta', tfm', \langle \overline{id}_1, \overline{id}_2\rangle)$ and $\overline{id}_1 \neq \varnothing$ and $\mathsf{complete}(e_0'; e_0)$

(SM8) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, {\downarrow}sid{=}e_0) \rightarrow \texttt{err}(er)$,
if $e(sid) = e_0'$ and $\texttt{match}(\Delta, \overline{d}, tfm, e_0', e_0) \rightarrow^* \texttt{err}(er)$

(SM9) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, {\downarrow}vid{=}is_1) \rightarrow \texttt{succ}(\Delta; ({\downarrow}vid \stackrel{\overline{d}'}{=} vid), tfm, unm)$,
if $e[\![vid]\!] = is_2$ and $\mathsf{deps}(is_2) = \overline{d}'$ and $(\texttt{solve}(\Delta, \overline{d}, is_1{=}is_2) \rightarrow^* \texttt{succ}(\Delta')$ or $\mathsf{strip}(is_1) = \mathsf{v})$

(SM10) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, {\downarrow}vid{=}is_1) \rightarrow \texttt{err}(er)$,
if $e[\![vid]\!] = is_2$ and $\mathsf{strip}(is_1) \neq \mathsf{v}$ and $\texttt{solve}(\Delta, \overline{d}, is_1{=}is_2) \rightarrow^* \texttt{err}(er)$

(SM11) $\texttt{match}(\Delta, \overline{d}, tfm, \langle \overline{id}_1, \overline{id}_2\rangle, e, {\downarrow}id{=}x) \rightarrow \texttt{succ}(\Delta; ({\downarrow}id{=}\mathsf{toDumVar}(x)), tfm', \langle \overline{id}_1 \cup \{id\}, \overline{id}_2\rangle)$,
if $e(id)$ is undefined and (if $x = \Lambda\alpha.\ (\alpha\,\gamma)^{\overline{d}'}$ then $tfm' = tfm \boxplus \{\gamma \mapsto \alpha_{\texttt{dum}}\}$ else $tfm' = tfm$)

(SM12) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, ev) \rightarrow \texttt{succ}(\Delta; ev, tfm, unm)$

(SM13) $\texttt{match}(\Delta, \overline{d}, tfm, unm, e, e'^{\overline{d}'}) \rightarrow \texttt{match}(\Delta, \overline{d} \cup \overline{d}', tfm, unm, e, e')$

a signature are involved in the reported error and this information is necessary for the error to occur then we highlight the blank spaces (if any) preceding the corresponding `val`, `type`, `datatype` and `structure` keywords.

We would then obtain the following highlighting which is a bit more informative than the one presented above:

```
structure S = struct val (fool, barr, x, y) = (1, 2, 3) end
signature s = sig val foo : int val bar : int val x : int end
structure T = S :> s
```