New Developments to Skalpel: A Type Error Slicing Method for Explaining Errors in Type and Effect Systems

John Pirie

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University

School of Mathematical and Computer Sciences

11th September, 2014

Abstract

Type error reports provide programmers with a description of type errors which exist in their code. Such descriptions are frequently of poor quality, as they often present just one point in the program, rather than all locations in the code which contribute to that type error.

Skalpel is a type error report system for the Standard ML language which tackles this problem, by presenting all and only the locations in the program which contribute to the type error. While the original Skalpel gives substantially better error reports than comparable systems, it has a number of limitations such as a lack of support for language features and poor efficiency.

In this research we have made a number of contributions, including a full critique of both the Skalpel core theoretical system and its extensions, support for the remaining features of Standard ML, an analysis and improvements to the efficiency, and an investigation for the first time on Skalpel's theoretical properties.

Acknowledgements

There are many people who I must thank in this space, who have given me their time, effort, and steadfast support in the duration of this research project.

- Professor Fairouz Kamareddine who spent valuable time reading my thesis and paper drafts. The technical inner workings of Skalpel are certainly time-consuming to learn, and the time you have spent pouring over that theory, and indeed all aspects of this thesis, are appreciated more than I can say.
- Doctor Joe Wells for his great attention to detail and personal enthusiasm in this project, and for the generous amount of time you have given me in our time together.
- Professor Greg Michaelson for your role as my internal examiner. I thank you greatly for the time you have spent in this capacity.
- My ever-loving and supportive parents. You have both supported me in your own way throughout the course of my time at university, both in my undergraduate and postgraduate years, and that support will never be forgotten.
- Vincent Rahli for the excellent work you have done in your own thesis, which this one has built directly on top of. I hope to meet you again one day, and wish you good luck with all the (no doubt successful!) years that are to come.
- William Gatens. It's astonishing how much chatting we have done over the past however many years it has been. Through Kiwibot and rough calculations I estimate since we started chatting years ago, we have spent more than **360** hours (each!) *just typing* something at one another, which is amazing. I've likely talked your ear off about Skalpel more than anyone, so thanks!
- Peter Gatens, who has eyed over parts of my thesis, and who I would miss if he really did get out!
- My flatmate Dodds, who gives me just the right amount of distractions.
- Lavinia Burski for being a motivated tutorial student, and who is now a member of our research group. I truly wish you all the best with your thesis.
- All the my friends in Edinburgh Jitsu, and also in music groups across Scotland. Life without music is empty to me, thank you all for you dedication to meet and make all that wonderful music together.

Contents

List of Figures

Chapter 1

Overview

Skalpel is a tool for the Standard ML (SML) [MTHM98] programming language designed to solve the problem of low quality reporting of type errors which is currently present in the available compilers for the language. This document outlines progress made by a research project designed to combat these issues of low quality reporting of type errors. This thesis used and extends the work of previous research projects which have led the Skalpel project as a whole to its current state.

In section 1.1, we outline the structure of this thesis, and list contributions in section 1.2. In section 1.3 we present the history of the project up to the point before this thesis began.

1.1 Outline of this thesis

Below, we describe each of the following chapters which are present in this thesis.

- Chapter 2 gives a review of some of the more important literature relevant to this project.
- Chapter 3 discusses the original presentation of the Skalpel core as described in [Rah10], with minor edits made to fix any bugs that existed in the theory.
- Chapter 4 presents the newest version of the Skalpel core and a discussion of the relevant changes.
- Chapter 5 gives some fully worked examples of the Skalpel machinery as presented in chapter 4.
- Chapter 6 describes some extensions to the original version of the Skalpel theory (chapter 3) that were present in [Rah10], and updates them to work

with the new version of the Skalpel core. We use these in the next chapter (chapter 7).

- Chapter 7 presents further new extensions to the theory to enhance Skalpel's capability to detect and present errors.
- Chapter 8 discusses some of the properties of Skalpel, such as the termination of algorithms.
- Chapter 9 lays out ideas for future work which could be done on the Skalpel project extending from the work in this thesis, and concludes this thesis.
- Appendix A lists symbols used in this thesis, with a brief description and gives a point of definition.
- Appendix B describes how a new tool has been developed in order to document ML code and gives a critique of the other existing alternatives that were unsuitable.
- Appendix C discusses the Skalpel implementation, and some of the changes that were made to it.
- Appendix D Gives an extract of the LaTeX version of the documentation that is generated from the documentation tool described in appendix B.
- Appendix E shows the progression of both the external and constraint syntax that is specified in the presentation of the Skalpel core and all of the Skalpel extensions.

1.2 Contributions

This document presents the following contributions:

- **Critique of the initial Skalpel presentation**. In chapter 3 we discuss the initial presentation of Skalpel and in chapter 6 we critique the extensions that were made prior to this project.
- **Several new extensions to the theory.** We present several new extensions to the existing theory surrounding Skalpel in chapter 7, adding support for features such as equality types and type sharing. We build this theory on top of the existing theory but we attempt to keep as much simplicity as possible.

- **Work on implementation efficiency.** We present an approach in section 4.7 which shows how the Skalpel implementation can be modified to execute faster by representing label sets in a new way, and discuss some other possible means of representation. This is particularly important for our implementation as shortening the time we spend computing label sets is an attractive area for optimization. We review the profiling state of the analysis engine, the core Skalpel implementation component, and discuss the ways in which the implementation can be enhanced using this information. As the Skalpel implementation is very much a real project, and not a toy implementation, implementation details such as these are extremely important. We use the findings here to locate opportunities for future work.

- **Developed meta-theory.** We list properties of the theoretical parts of Skalpel in chapter 8 which have never previously been made and justifications on why such properties hold.

1.3 History of Skalpel

The start of an implementation which was designed to help programmers understand and solve type errors in their Standard ML code took place in the duration of Christian Haack's work as a postdoctoral assistant with the ULTRA group at Heriot-Watt University working with Dr. Joe Wells. Shortly after this, Sebastien Carlier created the first version of a web demonstration of this implementation, which allowed users to try the tool online.

In 2008, Dr. Vincent Rahli began work on the Skalpel implementation as part of his Ph.D. studies. He started to extend the work already completed so that it was capable of handling datatypes, type functions, and other various features of the Standard ML programming language as described in the definition of Standard ML [MTHM98].

Chapter 2

Literature Review

In this chapter a discussion is presented of the other literature related to type checking and type error slicing.

2.1 Type checking

A type is a role that can be assigned to different forms of data, (such as an `integer` type representing numbers) which dictate what sort of values are available for that type, how that type is used, and operations that be performed on values which are defined to be of that type. A *type checking algorithm* verifies that all of the constraints imposed by these types in a user program are solvable.

A type checker will check that all of the values of a program are used correctly and will check the types of all expressions in the user program. Not all languages have type checkers, and those that do are generally classified into two categories - strongly typed and weakly typed. In the general case, programs which do not allow loss of information when altering existing types are considered strongly typed, with those that do considered weakly typed. Statically type checking languages can bring benefits to the compiler as it knows more information about the program, and can lead to more efficient code. Another major benefit is that it will catch errors at compile time that would otherwise not be caught.

Type checking can either be performed statically, at compile time, or dynamically, at run time. The approach to static vs dynamic type checking, and strong vs weak typing has been a source of some debate [Han10]. Doing type checking at compile time will mean that the incorrect use of types is detected by the programmer, whereby dynamic type checking will inevitably mean that some of these errors will not be detected until run-time, possibly by end-users of software.

When a type system detects the use of types incorrectly, it will report a *type error*. In the next section we discuss the importance of the explanation of these type errors.

2.2 Importance of the explanation of type errors

The explanation of type errors in mainstream programming languages is generally poor, especially when large amounts of type inference is involved. An example of this can be seen in C++ in the template feature, and in the ML family of languages where explicit typing is not mandatory and a sophisticated type inference algorithm is used. The type checking algorithms used by these languages highlight only one location in the program as the source of error, yet type errors reported during compilation are composed of *conflicting* information at *multiple* points in the program. Any explanation of a type error should contain multiple program points.

A poor description of a type error will mean that the programmer may have to locate all the points in their program that contributed to the type error they received from the compiler manually, and this may be extremely time consuming in complex cases. It is therefore important that type errors are explained in the clearest and most precise way possible, as otherwise the real location of the error can be situated a great distance away from the reported error location.

In the next section, the descriptions of [Yan01] are presented, which state what factors of a type error report make that report of a high quality.

2.3 Criteria of Yang et al.

In [YMTW00], properties are discussed which make up a good error report. In this section, we look at each of the properties that were highlighted in turn, and discuss why we think these are good properties to have in a type error report.

Correct. This property means that errors are generated when the program is illegal, and that all the locations of the user program present in the error actually contribute to the type error. This is a property we believe should hold, as generating errors for a legal program will waste programmer time looking for an error that doesn't actually exist, and having bogus locations in the error report would mean that the programmer is looking at more program locations than they have to. By minimizing the number of locations that the programmer should look at, their time

and effort is focused only on areas of the program that can be modified in order to repair the type error.

Precise. Yang et. al. state that a precise type error should include only the smallest possible amount of programmer code and that the relationship between that and conflicting type error information should be simple. This property is important, as showing extra information to the programmer than they need to understand the error is undesirable.

Succinct. A succinct error is said to be an error that maximizes useful information, preferring short explanations to longer ones. We believe succinct errors are also desirable, to shorten the time the programmer needs to spend to understand the type error fully. Any additional information will spend extra programmer time reading the error.

A-mechanical. An error report meeting this property is that which does not contain inferred type information as a result of the internal unification process. This is a particularly important property as breaking it can mean the programmer is shown parts of a program which is artificial in nature, and does not directly correspond to the programmer's code. This leads to confusion as to what the error report is actually demonstrating and causing overhead for the programmer to fix the error, as they have to understand how this inferred portion of the error report corresponds to the program.

Source-based. A source-based report gives the property that all the programmer should need to understand the error report is the error report itself (so that no internals of the compiler syntax should be dumped upon the programmer). Breaking this property would mean that the programmer could get different error reports depending on which compiler they used, as each have their own internal mechanisms, which is unacceptable.

Unbiased. A report which contains a bias is that which reports a different location when the order that certain features in the user program is changed. This is of importance as the reported error locations should be identical, irrespective of the order that unification is done.

Comprehensive. This property means that any error report should report *all* locations that contribute to the type error. This property should hold for type error reports, as breaking it would mean that the programmer must inspect other portions of the program that are not contained in the error report in order to understand it, wasting programmer time.

2.4 Monomorphic type checking

In monomorphic checking, each expression has a unique type, or alternatively no type at all if it is not well-typed [Pie04]. This means that there are no need for type variables when doing monomorphic type checking. Only expressions or functions with a fixed type are supported [PRO].

The rules when using monomorphic type checking are straight-forward. If there is a function defined `F`, which is of type `Bool` $\rightarrow$ `Bool`, then whenever an application to that function is used `f t`, then `t` must be of type `Bool` and the type of the whole expression must be of type `Bool`. If it is not, a type error is generated.

There are three stages which are present in monomorphic type checking, firstly to derive type constraints from the program, to solve those constraints, and to extract type definitions and type signatures from those solved constraints [SBG09a]. In monomorphic type checking, there are only two kinds of constraint - either the types are equal, or in e.g. function application, one type is a case of another (so they are not strictly identical, but still compatible). Given these rules, solving is straight-forward - if types are not equal, and one is not a case of the other (e.g. in function application), then checking halts with an error.

2.5 Polymorphic type checking and the W algorithm

A polymorphic type system relaxes the rules for a monomorphic one, so that the type signature of a function call is an instantiation of the signature [SBG09b]. The way that the W algorithm works is to destructively modify (i.e. use substitution on) the type information known at parts of the program, which gives high performance, but poor quality error reporting information.

2.5.1 The Hindley/Milner type system

The Hindley/Milner type system is a type system for the lambda calculus, with parametric polymorphism. This system was first written by Hindley [Hin69], and later discussed by Milner [Mil78].

The syntax that makes up the various expressions that are to be types is the λ-calculus, extended with `let` declarations. The syntax for this is shown in figure 2.1.

Figure 2.1 Syntax of Hindley/Milner

exp	::=	x	Variable case
		$exp_1\ exp_2$	Application
		$\lambda x.exp$	Abstraction
		`let` exp_1 `in` exp_2	Let expression

Monomorphic types are represented as τ (referred to as *primitive types* in the original paper), and can be either variables (α) or applications. Polymorphic forms σ (referred to as *type schemes* in the original paper) are either a τ or a universally quantified term $\forall\alpha.\sigma$.

The typing rules for Hindley/Milner is shown in figure 2.2 [Pie04]. The variable case is where if a variable x is typed to some type τ, then it is said to be of type τ. Rule `App` is for application, where one expression is applied to another. If exp_0 takes a expression of type τ, and returns something of type τ', then the result of the application is of type τ'.

Figure 2.2 Typing rules for Hindley/Milner [Pie04]

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ v : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ v\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ v : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ v = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

The abstraction rule, `Abs`, describes that if we have a variable x of type τ, and an expression e of type τ', then the function which could be constructed with the variable x and the body e would be of type $\tau \to \tau'$. The rule for let expressions states that for some expressions e_0 and e_1, if the result of e_0 is set to some variable x which is used in e_2 and is of type τ, then the resulting type is τ.

Rule `Inst` handles instantiations, which deals with subtyping. If we have an expression e of type σ', and σ' is a subtype of σ, then e is of type σ. Finally `Gen` handles the generalization case, where if an expression e is of type σ, and the

variable x is not in the free variables of the environment (where free is a function which computes the free variables in its argument) γ, then x must be bound in e.

2.5.2 Algorithm W

In order to define algorithm W, Milner references the unification algorithm of Robinson [Rob65] to define W, given in figure 2.3. The W algorithm itself is provided in figure 2.4 [DM82], and the algorithm fails whenever any of these rules are not met (where A is a set of assumptions, and Id^2 is the empty substitution).

Figure 2.3 Proposition of Robinson [Rob65]

There is an algorithm U which, given a pair of types, either returns a substitution V or fails; further

- If $U(\tau, \tau')$ returns V, then V unifies τ and τ', i.e. $V\tau = \tau'$.
- If S unifies τ and τ' then $U(\tau, \tau')$ returns some V and there is another substitution R such that S = RV.

Moveover, V involves only variables in τ and τ'.

Figure 2.4 Algorithm W [DM82]

Algorithm W.

$W(A, e) = (S, \tau)$ where

(i) If e is x and there is an assumption $x : \forall\alpha_1, \ldots, \alpha_n\tau'$ in A then $S = Id^2$ and $\tau = [\beta_i/\alpha_i]\tau'$ where the β_is are new.

(ii) If e is $e_1\, e_2$ then let $W(A, e_2) = (S_1, \tau_2)$ and $W(S_1A, e_2) = (S_2, \tau_2)$ and $U(S_2\tau_1, \tau_2 \to \beta) = V$ where β is new; then $S = VS_2S_1$ and $\tau = V\beta$.

(iii) If e is $\lambda x.e_1$ then let β be a new type variable and $W(A_x \cup \{x : \beta\}, e_1) = (S_1, \tau_1)$; then $S = S_1$ and $\tau = S_1\beta \to \tau_1$.

(iv) If e is `let` $x = e_1$ `in` e_2 then let $W(A, e_1) = (S_1, \tau_2)$ and $W(S_1A_x \cup \{x : \overline{S_1A}(\tau_1)\}, e_2) = (S_2, \tau_2)$; then $S = S_2S_1$ and $\tau = \tau_2$.

The reason W gives poor error messages, although it is efficient, is that the substitution causes loss of information with respect to where types have been inferred. It is also biased, as it operates left to right and halts whenever a problem occurs, and the location where type checking stopped is reported. The same issues are true with other algorithms such as M, where similar substitution approaches are used.

2.5.3 Other systems

Dowek [Dow01] gives a description of unification, and demonstrates that it is undecidable by looking at Hilbert's 10th problem stated below.

There is no algorithm that takes as arguments two polynomials $P(x_1, ..., x_n)$ and $Q(x_1, ..., x_n)$ whose coefficients are of type $\mathbb{N}$ and answers if there exists $m_1, ..., m_n \in \mathbb{N}$ such that $P(m_1, ..., m_n) = Q(m_1, ..., m_n)$.

It is also discussed how decidable sub-cases of the problem of unification can be identified, whereby they can then be solved.

A unification algorithm is also presented by Martelli and Montanari [MM82], which is presented as a non-deterministic algorithm which approaches the unification problem as finding the solution to sets of equations. They define a *multiequation* to be the generalization of an equation, which allows for the grouping of many terms which should be unified. Any unifier to a multiequation is a solution that makes the terms in the left and right hand side of the multiequation identical. Orderings can they be applied to these multiequations with the goal of making the unification algorithm more efficient. The algorithm presented is said to be more efficient than Patterson and Wegman's algorithm [Pur91], which achieves O(n) computational complexity.

Hage and Heeren present a constraint based approach to the problem of type checking in type systems [HH09]. In their presentation, constraint generation and constraint solving phases are kept separate, and not interleaved.

The authors do not annotate program points with labels. It is for this reason that their system will also locate a single point of error in a given untypable program. However the way in which the authors allow the programmer to edit their constraint solving approach allows for the detection of multiple points where the error takes place by running the constraint solving algorithm with different configurations, as explained later in this section.

The interesting part about the way that these constraints are represented is that they are essentially trees. These trees typically have the same shape as the abstract syntax tree, but the structure of the tree can be edited if chosen by the programmer. By doing this, the bias in the ordering of how the program is traversed can be changed at will, simply by rearranging these trees.

It is in this way that the system of [HH09] is able to replicate algorithms such as algorithm W, but it is also able to represent algorithm M by rearranging the constraint tree. It is reported in the paper in question that other more advanced algorithms can also be used with this technique, such as $\mathcal{H}$ by Lee and Yi [LY00].

The programmer can specify in which order the attempt to solve constraints should be made, and by selecting different ordering strategies they can locate different points where unification fails, and gain more information about the true cause

of the error.

2.6 Attempts to provide better explanations from the W algorithm

Several attempts have been made to provide better explanations from the W algorithm, such as the work of [LG06], which describes how error suggestions may even be presented by calling the compiler (i.e. running the W algorithm) multiple times on a piece of user code. Their system is split up into three different pieces:

- Changer. The changer takes as input a program which contains a type error and outputs changes that make the program well typed. It verifies such changes make the program typable by replacing sections of user code with dummy expressions that typecheck, then calling the compiler again to verify that the code is then typable. This process is broken down further into an enumerator which suggests syntax changes and a searcher, which guides where changes should be attempted.
- Type checker. This remains unchanged from the original compiler implementation but is used to detect which changes that are output from the Changer actually succeed in making the program typable.
- Ranker. The ranker will prioritize the changes that have been checked and verified by the type checker and will display messages accordingly.

The reporting system described here is a wrapper around the compiler that intercepts the type checker's error messages and then produces better ones. This approach is not entirely dissimilar to the approach taken by Braßel [Bra04], as discussed below. As this is a patch on the Ocaml sources which simply looks at the untypable program as reported by the Ocaml compiler, attempts some changes by taking the point of error and replacing the code surrounding it by dummy expressions, runs the type inference algorithm of the Ocaml compiler again, analyses the results, and repeats as necessary, this approach has the following properties:

- Installation is trivial. The patch is simply added to the compiler and then the build process is completed as normal.
- Any new features which are added to the language will automatically be supported, at least in some sense. Depending on the complexity and properties

of the new language feature, it may be the case that errors involving this new feature will perhaps not be of an excellent quality, but it is likely that it would be able to provide the programmer with some assistance.

- If the type system is edited in some minor way, the error reporting of would likely be unaffected.
- Use of the implementation is trivial, a command line argument is simply added to the invocation of the Ocaml compiler.

In the approach given in [LG06], a better error is located in an untypable program by identifying *interesting* nodes in the abstract syntax tree, by taking the root of the tree and replacing a node with a wildcard character. If the code typechecks after this is put in place, then this node is marked as interesting and recursion moves to the children of that node in order to find further interesting nodes. Otherwise, that node and its children are deemed uninteresting. It is pointed out that this has some nice properties because it will always find interesting nodes (the root is always going to be interesting, because replacing that with the wildcard operator will typecheck, naturally), and no recursion is done into subtrees which don't contain a type error (if the father of a given node is not interesting, then the given node can't be interesting either).

Given this approach however, there are the drawbacks that the error locations located by this approach as when run of programs with multiple programming errors, the results may be of a lesser quality. This is the case because these approaches look for *any* way to make the program typable, rather than looking for *all* the locations that a program can be made typable.

Other techniques to locate possible fixes are used, such as replacing variables with other variables which are in the same scope, swapping, removing or inserting some arguments of functions involved in an error and removing type annotations, etc. All of these techniques when used will attempt to locate better suggested fixes for the user to look at.

Another related system is that of Braßel [Bra04] named TypeHope, which is a system which runs on programs written in the logic language Curry with the objective of locating and providing possible solutions to type errors in those programs. It is designed to operate in a similar way to human programmers when attempting to locate the source of an error by replacing entire functions by dummy definitions and then checking whether the program is now typable.

When this system is run on an untypable section of code, it locates regions where a fix may be possible and assigns dummy types to expressions inside those

locations. It then checks whether the code is made typable after making such a change by repeating a call to the compiler, and if it is, repeats the process on any sub-expressions of the expression that was turned into a dummy expression in order to locate a more accurate source of error. In TypeHope, writing a constraint generator and constraint solver is not necessary, as one can just send modified code to the compiler to test if it is typable after a dummy expression has been added to the program.

In order that the user is not flooded with lots of possible fixes for the program, TypeHope assigns a weight to each possible solution it finds based on a compound of factors (including how many types would need to be changed etc), and attempts to show the most promising correction points first. To find these most promising correction points, TypeHope prefers inner correction points to outer ones in a given expression, and prefers fix points which are located in a function further away in the call graph from the function which is deemed to contain an error.

Figure 2.5 shows the code example presented in the [Bra04], where

```
reverse (x:xs) = reverse xs ++ x
```

is mistakenly entered in the program instead of of

```
reverse (x:xs) = reverse xs ++ [x]
```

at line 2.

Figure 2.5 Example code for TypeHope to analyze

```
1  reverse [] = []
2  reverse (x:xs) = reverse xs ++ x
3
4  last ls = head (reverse ls)
5
6  init = reverse . tail . reverse
7
8  rotate ls = last ls : init ls
```

TypeHope begins searching for the error in the location that the compiler reported, line 8, by replacing the right hand side with a dummy function. As this removes the previous type error, TypeHope inspects the call graph for the right hand side expression and notices that the farthest away function is `reverse` (as both the `last` and `init` functions call reverse, and reverse calls no other function).

The right hand side definition of reverse is then replaced with a dummy function call and it is noticed that the program is now typable, so the definition of this

function is inspected. The deepest innermost expression here is the term `xs`, which is replaced with a dummy expression but the error still exists, and so TypeHope then does the same for `x` in line 2, and notices that the program is now made typable. TypeHope uses this information to present a possible solution to the user as shown in figure 2.6.

Figure 2.6 TypeHope output for figure 2.5

```
1  Function reverse, line 2
2  Change in reverse (x:xs) = reverse xs ++ x
     affects       reverse       init            last        ...
3    current type  [[a]] -> [a]  [[[a]]] -> [a]  [[a]] -> a  ...
     to type       [a] -> [b]    [a] -> [b]      [a] -> b    ...
4
5  by replacing the term x
     with term of type              [b]
6    type of x (after replacement)  a
     where                          b should depend on a
```

It is interesting to note that in both [LG06] and [Bra04], the criteria laid out by [YMTW00] are not met, as they are not comprehensive, nor are they precise. Note that the work of [Rah10] is entirely distinct from the work presented here, as it is designed as a complete replacement to the W algorithm.

2.7 Pre-slice attempts to fix type errors

Several attempts to produce more helpful error messages are described in [McA02], e.g. looking at the order used by type systems to infer types, and representing an entire program as a graph first before later inspecting it in order to locate errors (instead of just performing substitutions immediately).

McAdam notes that by changing the order that substitutions are made during unification in algorithm W, different end points of error can be reported. This is also backed up by the findings of Hage and Heeren [HH09]. This bias is called an *asymmetry* in this text.

Also covered in this text is the idea of error repair suggestions instead of or in addition to conventional type error reports. McAdam states that such information helps programmers as it directly attempts to address their needs.

Again, the work here still falls into the category of focusing the programmer on a particular point in the program in order to fix the error, which we believe to be an insufficient way to report a type error. While this work is nevertheless

important, as finding more accurate ways to give a "correct" error point is useful, any one location reported by the compiler is insufficient to fully understand, and often to fix, the type error that is present in the user program.

2.8 An overview of slicing

The more general approach to reviewing slicing techniques is undertaken by Tip [Tip95], which includes a thorough description of the methods with which slices can be generated, a discussion on how a higher accuracy of slices can be gained using compiler optimization and symbolic execution techniques, and a description on how slicing can be used in development areas such as debugging and data flow testing.

Tip gives an overview of static and dynamic slicing techniques. In this work, the text on static slicing is of the most interest. Tip describes how program slices can be represented as a graph, with program points being represented as nodes, and the connections between them representing data flow. This approach however can give a confusing overall picture of the program slice, as the position of the nodes may bear no resemblance to their position in the program text, and with any significant amount of data flow, the number of lines which must then be made will make the slice difficult to interpret. Furthermore, the format of this display would mean that in practice the slice would be difficult to view in any source editor unless it has image viewing capabilities, or unless it is somehow extended to render such diagrams in the viewer.

System dependence graphs are also analyzed, whereby the slice is represented as a directed graph. Such slices also suffer from the same problems are previously described.

This work also discussed various other aspects of slicing such as distributed slicing, concurrency, slices involving pointers, a range of topics for dynamic slicing etc. which are not relevant to this thesis; the main benefit this text has for us is demonstrating that graph-based approaches to program slicing are unsuitable for us, as our slices can be large and would then be unreadable for the user.

2.9 Type error slicing

The work presented in [HW03] was the base used from which the work of [Rah10] was built, which is in many ways the base for this document, so a number of the

concepts introduced in this paper are discussed in chapter 4.

For example, the definitions of a type error slice, what it means for a slice to be minimal, the assignment of constraints to program points (labels), the finding of minimal unsolvable constraints sets, and the concept of having a minimiser and enumerator are all present in both the paper but in this document, discussion on these concepts does not take place here but instead in chapter 4.

The concept of overlapping error regions in type error slices are also discussed, and the text on that discussion is still very much relevant, namely that if there are two slices with overlapping regions the likely fix location is somewhere in the overlapping region and this should somehow be presented differently so as not to be hard to notice. The Skalpel analysis engine may generate multiple type error slices for a given piece of source code in a file and it is entirely possible that two or more different program slices will overlap. In this case it is indeed likely, though not necessarily the case, that the place in the file where the slices overlap is a cause of error, and fixing that spot in the while will often fix multiple problems that Skalpel detected.

This work was extended by Rahli [Rah10], and that work is is the work from which this thesis directly builds upon, and so the work present in that thesis is discussed at length in this one in various places throughout this thesis. Here, a discussion is given on the coverage of the Standard ML language that is present in that document. The Skalpel core, as described in that document, is similar to the core presented in this document and discussion is left until later in chapter 4. The thesis includes a number of extensions that are not discussed here, namely:

- Identifier statuses;
- Local declarations;
- Type declarations;
- Non-recursive value declarations;
- Value polymorphism restriction;
- Type annotations;
- Signatures;
- Reporting unmatched errors;
- Functors;
- Arity clash errors.

Local declarations, type declarations, type annotations, and signatures are discussed at length in chapter 7. A brief analysis is given here on the other sections.

The section on identifier statuses gives a more complete mechanism in which to handle the distinction between value variables and datatype constructors. As such a distinction is not possible to determine syntactically, it must be determined by analyzing the user program. That section describes how errors made involving this distinction can be correctly detected and presented to the user.

The section on non-recursive value declarations describe how Skalpel does constraint generation and solving on such declarations, the theory involved with this is uncomplicated.

Value polymorphism restriction describes how features such as references are handled in Standard ML. This is not discussed in this thesis as there is no need to make use of any of this theory for other extensions to the Skalpel core.

The section on reporting unmatched errors describes how Skalpel handles errors that can arise when a structure is constrained by a signature, but the structure does not declare all of the variables that the signature dictates it must declare.

The sections on functors and arity clash errors describe as the titles suggest. We do not include this theory in this thesis as the mechanisms that were constructed to be able to handle such SML constructs are not needed in the new extensions presented in this thesis.

2.10 An overview of Standand ML

This brief chapter is written for those to whom Standard ML (SML) is unfamiliar and gives a short description of some of the language features. Readers wishing for a thorough description of the language should read the definition of Standard ML [MTHM98]. This chapter does not cover all features of Standard ML but describes the features which are used in the core presentation of Skalpel.

Standard ML is a functional programming language which is strictly evaluating and supports parametric polymorphism. This chapter will look at the following features of Standard ML:

- Type variables;
- Function declarations;
- Structure declarations;

- The `open` feature;
- Datatype declarations;
- Let bindings.

First, an overview of the history of Standard ML is given.

2.10.1 History of Standard ML

ML was created by Robin Milner among others in the 1970s at the University of Edinburgh and was used to develop tactics in the LCF (Milner, Gordon and Wadsworth, 1979) theorem prover [Gor00]. There are now a couple of major dialects - such as Standard ML, Ocaml, and F#. In 1984 the module system was outlined for Standard ML by McQueen [Mac84]. ML has influenced many different languages since then such as Haskell, and more recently Rust.

2.10.2 Function declarations

In the Skalpel core presented in this thesis (in chapter 4), functions in Standard ML can be defined using the following syntax:

```
val rec <name> = <fn expression>
```

The `<fn expression>` is of the form `fn <pattern> = <expression>`, which can be seen in the example below demonstrating how the identity function is written, where `arg1` is an *argument* to the function `myFunction`:

```
val rec myFunction = fn arg1 => arg1
```

An alternative definition (which we do not use in the core) uses the `fun` keyword in order to define functions. The equivalent identity function with the `fun` keyword used is shown below:

```
fun myFunction arg1 = arg1
```

Note that functions can be partially applied in Standard ML (where a function is not given all of its arguments) and that we do this in some of our examples.

2.10.3 Let declarations

A let statement allows a programmer to define some declarations within a scope of expressions (*exp*). A let statement has the form

`let` *declarations* `in` *exp* `end`

where the scope of the declarations ends at the `end` keyword, at which point they are no longer accessible. In the core presentation of Skalpel, declarations can be either:

- Function declarations (discussed in section 2.10.2);
- Open statements (see section 2.10.6);
- Datatype declarations (see section 2.10.4).

and expressions can be either:

- An application;
- An anonymous function (`fn`-expression);
- A value identifier (such as a datatype constructor, which we explain below);

An example use of a program containing a `let` statement can be seen in section 2.10.4.

2.10.4 Datatypes

Datatypes in the Skalpel core have exactly one constructor and exactly one type variable (type variables in SML can be represented by one or more primes followed by one or more alphanumeric characters) in order to simplify the core presentation. An example of a datatype declaration and a use in a let statement is given below. In the let statements shown here, we allow an expression to also be a sequence of expressions separated with the `;` operator and surrounded with parentheses (exp_1;exp_2;...;exp_n), or a boolean.

```
let
    datatype 'a myDatatype = MyConstructor of 'a
in
    (MyConstructor (fn x => x);
    true)
end
```

Note that as the constructor `MyConstructor` takes a type variable argument in its definition, we can supply anything to it as its argument. For example, instead of giving a function as an argument, we could give a boolean such as shown below.

```
let
    datatype 'a myDatatype = MyConstructor of 'a
in
    (MyConstructor true;
    true)
end
```

2.10.5 Ref types

Standard ML is not a pure functional programming language, and this can be seen by looking at the *ref* feature of the language. This feature allows the programmer to keep track of state, something which is not possible in some other functional programming languages. An example of a ref type being declared and then printed is shown below.

val myref = ref 5; myref := (!myref) + 1; print(Int.toString(!myref));

The '!' operator allows the programmer to access the value that is stored in the ref cell, and the ':=' operator allows assignment. Using this construct, programmers may store state in SML.

2.10.6 Structures

Structures are part of what is known as the Standard ML *module system*. Structures allow for any declaration, local declaration, structure declaration, or sequence of any of these (which are simply space separated).

The `open` keyword

The `open` keyword takes a structure identifier as an argument and can be used to bring declarations in the body of a structure definition into the current environment. An example of this is shown below.

```
structure S = struct val rec f = fn x => x end
open S
f (fn x => x)
```

When we open the structure `S`, the declarations inside it, in this case the function `f`, become available for use in the current environment. Were we not to open the structure `f`, there would be an error in the *expression* (`f (fn x => x)`), as the function `f` would be undefined.

2.10.7 BNF Grammar

The BNF grammar for the Standard ML language is given in this section for reference. This includes everything in the language, and not just the syntax described in the core. This is included so that in later chapters, when discussing parts of the syntax, it is clearer to see how this fits in to the SML language as a whole.

$$
\begin{array}{lcl}
atexp & ::= & scon \\
 & & \langle op \rangle \\
 & & \{\langle exprow \rangle\} \\
 & & \#lab \\
 & & () \\
 & & (exp_1, ..., exp_n) \\
 & & [exp_1, ..., exp_n] \\
 & & (exp_1; ...; exp_n) \\
 & & \texttt{let}\ dec\ \texttt{in}\ exp_1; ...; exp_n\ \texttt{end} \\
 & & (exp)
\end{array}
$$

$$
exprow \quad ::= \quad lab = exp\ \langle, exprow \rangle
$$

$$
\begin{array}{lcl}
appexp & ::= & atexp \\
 & & appexp\ atexp
\end{array}
$$

$$
\begin{array}{lcl}
infexp & ::= & appexp \\
 & & infexp_1\ vid\ infexp_2
\end{array}
$$

$$
\begin{array}{lcl}
exp & ::= & infexp \\
 & & exp \ : \ ty \\
 & & exp_1 \ \texttt{andalso} \ exp_2 \\
 & & exp_1 \ \texttt{orelse} \ exp_2 \\
 & & exp \ \texttt{handle} \ match \\
 & & \texttt{raise} \ exp \\
 & & \texttt{if} \ exp_1 \ \texttt{then} \ exp_2 \ \texttt{else} \ exp_3 \\
 & & \texttt{while} \ exp_1 \ \texttt{do} \ exp_2 \\
 & & \texttt{case} \ exp \ \texttt{of} \ match \\
 & & \texttt{fn} \ match
\end{array}
$$

$$
match \ ::= \ mrule \ \langle | \ match \rangle
$$

$$
mrule \ ::= \ pat \rightarrow exp
$$

$$
\begin{array}{lcl}
dec & ::= & \texttt{val} \ tyvarseq \ valbind \\
 & & \texttt{fun} \ tyvarseq \ fvalbind \\
 & & \texttt{type} \ typbind \\
 & & \texttt{datatype} \ datbind \ \langle \texttt{withtype} \ typbind \rangle \\
 & & \texttt{datatype} \ tycon = \texttt{datatype} \ longtycon \\
 & & \texttt{abstype} \ datbind \ \langle \texttt{withtype} \ typbind \rangle \ \texttt{with} \ dec \ \texttt{end} \\
 & & \texttt{exception} \ exbind \\
 & & \texttt{local} \ dec_1 \ \texttt{in} \ dec_2 \ \texttt{end} \\
 & & \texttt{open} \ longstrid_1, ..., longstrid_n
\end{array}
$$

$$
\begin{array}{lcl}
valbind & ::= & pat = exp \ \langle \texttt{and} \ valbind \rangle \\
 & & \texttt{rec} \ valbind
\end{array}
$$

$$
\begin{array}{lcl}
fvalbind & ::= & \langle \texttt{op} \rangle vid \ atpat_{11} ... atpat_{1n} \langle : \ ty \rangle = exp_1 \\
 & & | \langle \texttt{op} \rangle vid \ atpat_{21} ... atpat_{2n} \langle : \ ty \rangle = exp_2 \\
 & & | ... \quad ... \\
 & & | \langle \texttt{op} \rangle vid \ atpat_{m1} ... atpat_{mn} \langle : \ ty \rangle = exp_m \ \langle \texttt{and} \ fvalbind \rangle
\end{array}
$$

$$
typbind \ ::= \ tyvarseq \ tycon = ty \ \langle \texttt{and} \ typbind \rangle
$$

$$
datbind \ ::= \ tyvarseq \ tycon = conbind \ \langle \texttt{and} \ datbind \rangle
$$

$$
conbind \ ::= \ \langle \ \texttt{op} \ \rangle \ vid \ \langle \texttt{of} \ ty \rangle \ \langle | \ conbind \rangle
$$

$$
\begin{array}{lcl}
exbind & ::= & \langle \ \texttt{op} \ \rangle \ vid \ \langle \texttt{of} \ ty \rangle \ \langle \texttt{and} \ exbind \rangle \\
 & & \langle \ \texttt{op} \ \rangle \ vid \ = \langle \texttt{op} \rangle \ longvid \ \langle \texttt{and} \ exbind \rangle
\end{array}
$$

$$
\begin{array}{lll}
atpat & ::= & \_ \\
 & & scon \\
 & & \langle \texttt{op} \rangle\ longvid \\
 & & \{\ \langle patrow \rangle\ \} \\
 & & () \\
 & & (pat_1, ..., pat_n) \\
 & & [pat_1, ..., pat_n] \\
 & & (pat)
\end{array}
$$

$$
\begin{array}{lll}
patrow & ::= & ... \\
 & & lab = pat\ \langle , patrow \rangle \\
 & & vid\ \langle : ty \rangle\ \langle \texttt{as}\ pat \rangle\ \langle , patrow \rangle
\end{array}
$$

$$
\begin{array}{lll}
atpat & ::= & \langle \texttt{op} \rangle\ longvid\ atpat \\
 & & pat_1\ vid\ pat_2 \\
 & & pat\ :\ ty \\
 & & \langle \texttt{op}\ \rangle\ vid\ \langle :\ ty \rangle\ \texttt{as}\ pat
\end{array}
$$

$$
\begin{array}{lll}
strexp & ::= & \texttt{struct}\ strdec\ \texttt{end} \\
 & & longstrid \\
 & & strexp : sigexp \\
 & & strexp :> sigexp \\
 & & funid(strexp) \\
 & & \texttt{let}\ strdec\ \texttt{in}\ strexp\ \texttt{end}
\end{array}
$$

$$
\begin{array}{lll}
strdec & ::= & dec \\
 & & \texttt{structure}\ strbind \\
 & & \texttt{local}\ strdec_1\ \texttt{in}\ strdec_2\ \texttt{end} \\
 & & strdec_1 \langle ; \rangle\ strdec_2
\end{array}
$$

$$
strbind \quad ::= \quad strid = strexp\ \langle \texttt{and}\ strbind \rangle
$$

$$
\begin{array}{lll}
sigexp & ::= & \texttt{sig}\ spec\ \texttt{end} \\
 & & sigid \\
 & & sigexp\ \texttt{where type}\ tyvarseqlongtycon = ty
\end{array}
$$

$$
sigdec \quad ::= \quad \texttt{signature}\ sigbind
$$

$$
sigbind \quad ::= \quad sigid = sigexp\ \langle \texttt{and}\ sigbind \rangle
$$

$$
\begin{array}{lcl}
spec & ::= & \texttt{val}\ valdesc \\
 & & \texttt{type}\ typdesc \\
 & & \texttt{eqtype}\ typdesc \\
 & & \texttt{datatype}\ datdesc \\
 & & \texttt{datatype}\ tycon = \texttt{datatype}\ longtycon \\
 & & \texttt{exception}\ exdesc \\
 & & \texttt{structure}\ strdesc \\
 & & \texttt{include}\ sigexp \\
 & & spec_1\ \langle ; \rangle\ spec_2 \\
 & & spec\ \texttt{sharing type}\ longtycon_1 = ... = longtycon_n
\end{array}
$$

$$valdesc ::= vid\ :\ ty\ \langle \texttt{and}\ valdesc \rangle$$

$$typdesc ::= tyvarseq\ tycon\ \langle \texttt{and}\ typdesc \rangle$$

$$datdesc ::= tyvarseq\ tycon = condesc\ \langle \texttt{and}\ datdesc \rangle$$

$$condesc ::= vid\ \langle \texttt{of}\ ty \rangle\ \langle |\ condesc \rangle$$

$$exdesc ::= vid\ \langle \texttt{of}\ ty \rangle\ \langle \texttt{and}\ exdesc \rangle$$

$$strdesc ::= strid\ :\ sigexp\ \langle \texttt{and}\ strdesc \rangle$$

$$fundec ::= \texttt{functor}\ funbind$$

$$funbind ::= funid\ (strid : sigexp) = strexp\ \langle \texttt{and}\ funbind \rangle$$

$$
\begin{array}{lcl}
topdoc & ::= & strdec\ \langle topdec \rangle \\
 & & sigdec\ \langle topdec \rangle \\
 & & fundec\ \langle topdec \rangle
\end{array}
$$

Chapter 3

Prior Technical Design of Skalpel Core

Included in this chapter is a description of the Skalpel core prior to the start of this research project similar to that presented in [Rah10], with some corrections made to any problems that were noticed during analysis of the work for the purpose of its extension (computation of polymorphic binders - this is discussed in the new version of the core in section 4.4). First, the motivation for this research project is given.

3.1 Motivation for the Skalpel Project

The existing compilers for Standard ML often give confusing type error messages to the programmer. The original type checking algorithm for ML is Milner's W algorithm [DM82], which will usually show the programmer some information about a node in the syntax tree which it was visiting when unification failed. Although other algorithms exist which were developed to try to improve this (such as M [OL98], W' [Mca98] and UAE [Yan00]) all of these algorithms blame only one node in the syntax tree (often one far away from the real source of the error) for a type error which is really composed of conflicting information given by the programmer at different parts of the program.

Rahli, Wells and Kamareddine [RWK10] pointed out that "The confusion is worsened because these algorithms usually exhibit in error messages (1) an internal representation of the program subtree at the blame[d] 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".

The aim of the Skalpel project is to combat this problem by locating type errors in untypable code and then showing the user *type error slices*, which are a set of program points which represent a type error. It is somewhere in this set of program points that the user will want to make a change which will fix the type error, which will make the code typable. These program fragments are then displayed to the programmer by highlighting portions of the source file and by printing out type error slices to the user.

The overall process works as follows. The user can write their Standard ML source code in their editor of choice, then run a binary program which is currently still in active development by the Skalpel project team and is referred to as the *analysis engine* on this code which will output type error slices for untypable code. The different stages of the analysis engine can be seen in figure 3.1.

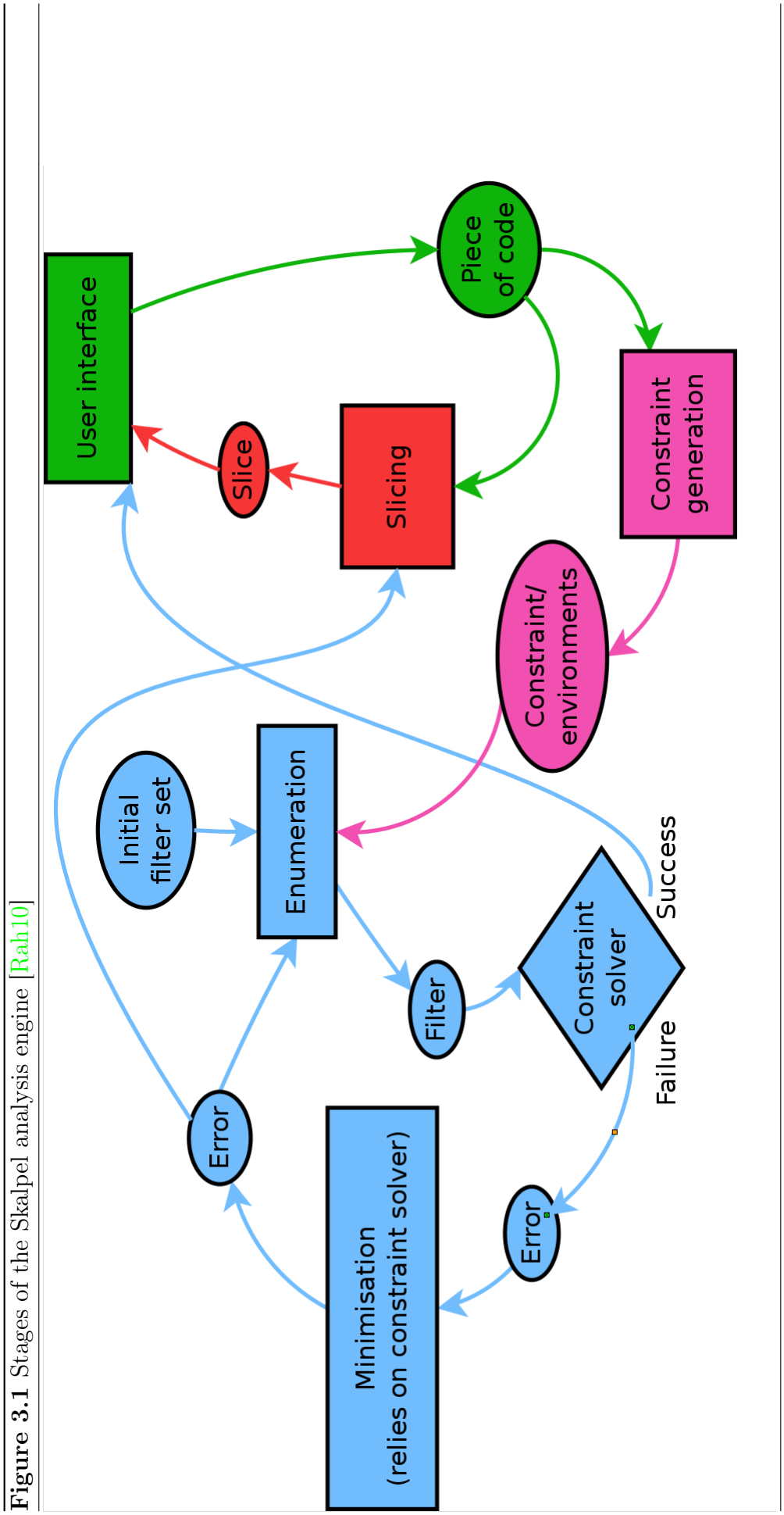


Figure 3.1 Stages of the Skalpel analysis engine [Rah10]

The green elements of the picture are external to the the analysis engine, and represent the piece of code being sent to the analysis engine. After the code is sent in, constraints are generated (see below for why a constraint-based approach is used) for the piece of code (the pink elements in the picture), and then an attempt is made to solve these constraints (the blue elements in the picture). If an error is detected, then type error slices are generated (the red elements of the picture) and send those back to the user interface. Some discussion of how this works in our implementation can be seen in appendix C.

We use constraints to represent conditions that are applied by some parts of the program to other parts of the program. For example, for the program

```
fun myFunction arg1 arg2 = arg1 + arg2
```

we will create internal *constraints* representing the fact that `arg1` and `arg2` *must* be numbers. These constraints are annotated with *labels*, which are locations of the program, so that it is known where constraints come from. For example, a constraint representing that `arg1` is a number will be annotated with the program location of the `arg1` keyword and the `+` operator. The compilers for Standard ML prefer a substitution-based approach, where the information representing the user program is destructively modified to be updated with new information, such an approach is not used as destructive modification of such information means that blame cannot be tracked for conditions imposed by different parts of the program effectively. A formal treatment on how we represent constraints is given in chapter 4.

There is support so that the user can use Emacs [EMA] as a front-end to the analysis engine back-end, which can be done by loading some Emacs lisp files which various Skalpel developers have helped prepare in previous projects as covered in section 1.3. This is released as a separate package that the user can install at their discretion. Skalpel is implemented in the Standard ML language itself.

3.1.1 An example demonstrating the use of Skalpel

An untypable program can be seen in figure 3.2.

In line 20, the number `1` (which is of type `int`), is passed to the function `find_best`. `1` then becomes the value of the `weight` argument in line 8, which is then passed to the function `average` in line 9. This then becomes the value of the `weight` argument in line 1, but an error can be seen occurring at line 2. The variable `weight` is constrained to be a function in line 2. The `weight` variable

Figure 3.2 Untypable SML Program

```
1  fun average weight list =
2   let fun iterator (x,(sum,length)) = (sum + weight x,
3                                        length + 1)
4       val (sum,length) = foldl iterator (0,0) list
5   in sum div length
6   end
7
8  fun find_best weight lists =
9   let val average1 = average weight
10      fun iterator (list,(best,max)) =
11        let val avg_list = average1 list
12        in if avg_list > max
13           then (list,avg_list)
14           else (best,max)
15        end
16      val (best,_) = foldl iterator (nil,0) lists
17  in best
18   end
19
20 val find_best_simple = find_best 1
```

cannot be both a function and an integer, and so a Standard ML compiler (such as SML/NJ [SMLc] or PolyML [POLeb]) will generate a type error for this.

SML/NJ (version 110.72) [SMLc], a compiler for Standard ML, reports the error shown in figure 3.3. This is confusing because at line 6 of the compilation output the compiler draws attention to line 20 of the code, but this might not be the source of the error. Perhaps at line 2, the user meant to write `sum + weight + x` instead of `sum + weight x`. Perhaps instead in line 9 they meant to write `average (fn _ => weight)` instead of `average weight`.

Figure 3.3 Compiler output for figure 3.2

```
1  code1.sml:20.5-20.35 Error: operator and operand don't agree
2                       [literal]
3    operator domain: 'Z -> int
4    operand:         int
5    in expression:
6      find_best 1
```

It is impossible to know where the user made the error so instead the user should be presented with all of the places where they could possibly have made a mistake. Skalpel has been designed to do this.

Figure 3.4 shows the highlighting Skalpel produces when run on this untypable program. The highlighted regions show the minimum amount of information that is responsible for the type error in the code. By looking at the highlighted regions,

the user can be confident that they need to make a change in one of these places in order to make the code typable, and in addition, that there are no other locations in the file which are not highlighted where they could make changes in order to fix the problem.

Figure 3.4 Highlighting shown for the code of figure 3.2

```
1  fun average weight list =
2    let fun iterator (x,(sum,length)) = (sum + weight x,
3                                         length + 1)
4        val (sum,length) = foldl iterator (0,0) list
5    in sum div length
6    end
7
8  fun find_best weight lists =
9    let val average1 = average weight
10       fun iterator (list,(best,max)) =
11         let val avg_list = average1 list
12         in if avg_list > max
13            then (list,avg_list)
14            else (best,max)
15         end
16       val (best,_) = foldl iterator (nil,0) lists
17   in best
18   end
19
20 val find_best_simple = find_best 1
```

3.1.2 A second example

We can see another untypable program in figure 3.5

Figure 3.5 A second untypable SML Program

```
1  let
2      datatype ('a,'b) T = C of ('a * 'b)
3      fun f (C (x,_)) (C (y,_)) z = (z x, z y)
4  in if true
5     then f (C (1,2)) (C ((),4))
6     else
7         let
8             datatype U = f
9             fun x _ = f
10        in fn z => (z 1, z 2)
11        end
12 end;
```

In this example, at line 2 in can be seen that a datatype constructor `C` is defined which takes a pair of any two values. At line 3, a function is defined which takes

three arguments, the first two of them a pair wrapped in the constructor `C`, and a final argument `z`, which is a function (we can see that it is a function from how it is used in the function body). In the body of this function, it can be seen that this function returns a pair, where the first element of the pair is the function `z` applied to `x`, and the second element of the pair is a function `z` applied to `y`, where `x` and `y` are bound to the first part of the pair of the first and second constructor arguments to the function `f` respectively. This means that `x` and `y` **must** be of the same type. However, at line 5, it can be seen that the function `f` is given the first two arguments, where the first parts of the pair wrapped in the `C` constructor are of **different** types. This causes a type error when attempting to compile this program.

SML/NJ reports the error shown in figure 3.6. This presents only the location at line 5, but this may not be not helpful for the user particularly if they made the error at line 3 (for example if they wish the `y` in the body of the function `f` to actually be `x` instead).

Figure 3.6 Compiler output for figure 3.5

```
1 [opening code13.sml]
2 code13.sml:7.9-7.31 Error: operator and operand don't agree
3                       [literal]
4   operator domain: (int,'Z) ?.T
5   operand:         (unit,int) ?.T
6   in expression:
7     (f (C (1,2))) (C ((),4))
```

Again, it is not known where the user made the error, and so we must present all the possible locations where the programmer may have made their error.

We present the error reported by Skalpel in figure 3.7. Here, it is shown again the minimum amount of information that is responsible for the error in the program, including the declaration at line 2, the definition of `f` in line 3, and the application in line 5.

We can see in line 5 that each of the arguments which cause a problem in line 3 are highlighted in gray and blue individually. This highlighting of what is called *endpoints* can help the user to discern how types of `int` and `unit` were deduced which then caused a problem at line 3.

It should be noted that it is very possible that type error slices can be presented to the user which involve more than one file of source code. In this event, every file which is attributed to the type error is shown in the type error slice, and highlighting is given in all affected files.

Figure 3.7 Highlighting shown for code in figure 3.2

```
1  let
2      datatype ('a,'b) T = C of ('a * 'b)
3      fun f (C (x,_)) (C (y,_)) z = (z x, z y)
4  in if true
5     then f (C (1,2)) (C ((),4))
6     else
7         let
8             datatype U = f
9             fun x _ = f
10        in fn z => (z 1, z 2)
11        end
12 end;
```

There are some cases where the code that is fed to Skalpel may contain multiple separate type errors. In this case, multiple program slices are given. It is perfectly possible that some of these program slices might overlap at a common point which indicates that such a point is likely to be the spot in the program that the user wishes to change to fix the type error (although this is not always the case). More discussion about this is found in section 2.9 of the literature review, which discusses a paper written by Haack and Wells [HW03].

We now discuss the notation used in Skalpel.

3.2 Definitions, notations and other symbol information

This section gives some basic mathematical definitions which are used throughout this thesis, followed by some information which may help the reader in recalling the meaning of symbols defined anywhere in this document.

3.2.1 Definitions

We use symbols $\mathbb{N}$ and $\mathbb{P}(Q)$ to represent the set of natural numbers and a power set of some set Q, respectively. Let i, j, k, l range over $\mathbb{N}$, and if a metavariable v ranges over Q, let $\bar{v}$ range over $\mathbb{P}(Q)$, v_x range over Q, where x is anything, and v', v'' etc. range over Q.

Let S range over sets.

Relations

Let $(\!|x, y|\!)$ be a pairing of x and y. If rel is a set of pairs, let (x rel y) iff $(\!|x, y|\!) \in rel$. Let the following operations be defined on relations:

$$\begin{array}{lcl} \mathsf{dom}(rel) & = & \{x \mid (\!|x, y|\!) \in rel\} \\ \mathsf{ran}(rel) & = & \{y \mid (\!|x, y|\!) \in rel\} \\ rel^{-1} & = & \{(\!|y, x|\!) \mid (\!|x, y|\!) \in rel\} \\ S \lhd rel & = & \{(\!|x, y|\!) \in rel \mid x \in S\} \\ S \rhd rel & = & \{(\!|x, y|\!) \in rel \mid y \in S\} \\ S \lhd\!\!\!\cdot\, rel & = & \{(\!|x, y|\!) \in rel \mid x \notin S\} \end{array}$$

Functions

Let f range over functions, and the notation $x \mapsto y$ be an alternative to $(\!|x, y|\!)$ used when writing functions.

Let $S_1 \rightarrow S_2 = \{f \mid \mathsf{dom}(f) \subseteq S_1 \wedge \mathsf{ran}(f) \subseteq S_2\}$.

Disjoint sets

Let $\mathsf{dj}(S_1, S_2, ..., S_n)$ mean that $S_i \cap S_j = \varnothing$ for all i $\neq$ j where $1 \leq i, j \leq n$. Let $S_1 \uplus S_2$ be $S_1 \cup S_2$ if $\mathsf{dj}(S_1, S_2)$.

Tuples

A tuple t is a function where $\mathsf{dom}(t) \subset \mathbb{N}$ where for some $i > 1$, if $(i, x) \in t$ then there exists an x' such that $(i - 1, x') \in t$. Let t range over tuples. If v ranges over S then let $\overrightarrow{v}$ range over $\mathsf{tuple}(S) = \{t \mid \mathsf{ran}(t) \subseteq S\}$. Let $\langle x_{i0}, x_{i1}, ..., x_{in}\rangle$ be an abbreviation for $\{0 \mapsto x_{i0}, 1 \mapsto x_{i1}, ..., n \mapsto x_{in}\}$. Let the @ operator append tuples such that $\langle x_1, x_2\rangle$ @ $\langle x_5, x_6\rangle = \langle x_1, x_2, x_5, x_6\rangle$. Let $S_1 \times S_2 \times ... \times S_n = \{\langle x_1, x_2, ...x_n\rangle \mid \forall i \in \{1, 2, ..., n\}.\ x_i \in S_i\}$.

3.2.2 Symbol Look-up

To ease the difficulty that the reader may face in recalling the definitions of certain sets, functions, metavariables and various other concepts used throughout this document, appendix A lists symbols used, sets they belong to (section A.1 is this information sorted by symbol, with section A.2 sorted by set), a brief description and location in the document where it was first defined. Sections A.3 and A.4 give brief descriptions and location in the document of definition for functions and other abbreviations used, respectively.

A thorough index is also present at the back of this document, which can be used to locate where symbols and other concepts occur anywhere in this thesis.

The Skalpel system is made up of several pieces:

- **Labelling**. Firstly, the program that the user submits to Skalpel is annotated with labels, which are used to represent program points and track blame of errors.
- **Constraint generation**. The constraint generator takes a labelled Standard ML program as input and generates a set of constraints. The constraints that are generated have program labels embedded into them, so it is known what parts of the program a certain constraint has been generated for.
- **Enumeration**. The enumerator creates *filters* for the next phase, which are effectively program points which should not be considered during constraint solving. Initially, the empty filter set is used, so all labels in the program are considered.
- **Constraint solving**. The constraints constructed during the constraint generation phase are taken and an attempt is then made to solve them. In the event that all constraints can be solved, the program the user submitted is deemed typable, otherwise, the program contains an error.
- **Minimisation**. In this phase an attempt is made to increase the precision of the reported error locations which can be more precise in some cases than the locations initially reported by the constraint solving stage. Less precise locations are never during this phase.
- **Slicing**. Creates a description of any located errors known as a *type error slice* (defined in 3.9) from the errors reported by the enumerator.

3.3 External (Input) Syntax

The external syntax is presented that is understood by the Skalpel core (formally called the SML-TES core) in figure 3.8. Many of the forms in this figure are annotated with labels, written l. These labels are created so that blame can be tracked for errors that are generated. In some cases $\lceil$ and $\rceil$ symbols will be used to surround a labelled term (parentheses are not used as they could be confused with those of the Standard ML language syntax).

In this core, datatypes have exactly one constructor and exactly one type argument. In the implementation however, the cases for multiple or nullary constructors and multiple or no type arguments are handled. This can be seen from DatName in figure 3.8.

The declarations supported by the core can be seen by looking at Dec in figure 3.8. There exists support for recursive value declarations (functions), open declarations,

and datatype declarations. Note that again in the implementation, far more declarations are handled than just this subset which are used in order to present the core.

Figure 3.8 External labelled syntax

external syntax (what the programmer sees, plus labels)

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

extra metavariables

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

3.4 Constraint syntax

The constraint syntax for this presentation of the core is defined in figure 3.9.

Forms annotated with dependencies (labels) are called 'dependent forms'. By attaching label sets to these forms it is possible to represent which program fragments such a constraint is involved in.

In this presentation, type constructors take exactly one argument. This singular argument is represented in the external syntax by tc and in the constraint syntax by μ. This can be seen in the definition of τ (the $\tau\mu$ form).

The `arr` piece of constraint syntax is never generated during the initial constraint generation phase. It can however be generated later, in constraint solving, to allow us to represent constraints between the arrow type constructor and an unary type constructor. This can be seen in rule (S6) of the constraint solver in this chapter.

Figure 3.9 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
d	$\in$	Dependency	$::=$	l
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \langle \mu, \overline{d} \rangle$
τ	$\in$	ITy	$::=$	$\alpha \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle \tau, \overline{d} \rangle$
σ	$\in$	Scheme	$::=$	$\tau \mid \forall \overline{\alpha}.\,\tau \mid \langle \sigma, \overline{d} \rangle$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}\mu \mid \downarrow strid{=}e \mid \downarrow tv{=}\alpha \mid \downarrow vid{=}\sigma$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid e_2;e_1 \mid \langle e, \overline{d} \rangle$
v	$\in$	Var	$::=$	$\alpha \mid \delta \mid ev$
dep	$\in$	Dependent	$::=$	$\langle \tau, \overline{d} \rangle \mid \langle \mu, \overline{d} \rangle \mid \langle \delta, \overline{d} \rangle$

The type schemes (σ) are subject to alpha conversion.

The functions strip and collapse are defined here. strip is a function that strips off dependencies from a dependent term, and collapse unions all nested dependencies. Both functions are defined in figure 3.10.

Figure 3.10 Definition of strip and collapse

$\mathsf{strip}(x) = \mathsf{strip}(y)$, if $x = \langle y, \overline{d} \rangle$, x otherwise.
$\mathsf{collapse}(x) = \mathsf{collapse}(\langle y, \overline{d}_1 \cup \overline{d}_2 \rangle)$ if $x = \langle (\langle y, \overline{d}_1 \rangle), d_2 \rangle, x$ otherwise.

3.4.1 The constraint/environment form (e)

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

- **The empty environment / satisfied constraint**. This is represented by the symbol $\top$.
- **An environment variable**. To abbreviate, $[e]$ is written for $(ev_{dum} = e)$, where ev does not occur in e. This is a constraint which enforces the logical constraint nature of e while limiting the scope of its bindings. Note that the bindings can still have an effect if e constrains an environment variable.
- **A composition environment**. The operator ; is used to compose environments, which is associative. Note that $e;\top$, $\top;e$, and e are considered to be equivalent.
- **A binder/accessor**. A binder is of the form $\downarrow id{=}\sigma$, and an accessor is of the form $\uparrow id{=}v$. Binders represent program occurrences of an identifier id

that are being bound, and accessors represent a place where that binding is used. For example, in the environment

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

the internal type variable α is constrained through the binding of *vid* to be an instance of x. It is often also the case that binders and accessors can be connected without being next to each other. In an environment such as

$$\downarrow vid{=}x;...;\uparrow vid{=}\alpha$$

it is *possible* that the binder and accessor of *vid* are connected. There are some environment forms that can be in the omitted (...) section which will mean that the accessor and the binder will be disconnected. Section 3.5.1 on *shadowing* specifies which forms would cause this.

To abbreviate, $\downarrow vid \stackrel{y}{=} ct$ is written for $\langle \downarrow vid{=}ct, y \rangle$ (similarly for accessors).

- **An equality constraint**. A constraint which represents that two pieces of constraint syntax are somehow equal.
- **A polymorphic environment** $\texttt{poly}(e)$. This environment promotes binders in the argument to `poly` to be polymorphic. This is discussed further in section 3.7.
- **Dependent form**. An environment annotated with dependencies acts like an environment only when the dependencies are satisfied.

3.4.2 Syntactic forms

The set $\mathsf{atoms}(x)$ is defined to be a syntactic form set belonging to $\mathsf{Var} \cup \mathsf{Label} \cup \mathsf{Dependency}$ and occurring in x for some x.

Function are defined to extract variables, labels, and dependencies from a constraint term in figure 3.11.

Figure 3.11 Definition of sets of variables, labels and dependencies

$\mathsf{vars}(x)$	$=$	$\mathsf{atoms}(x)$	$\cap$	Var	(set of variables)
$\mathsf{labs}(x)$	$=$	$\mathsf{atoms}(x)$	$\cap$	Label	(set of labels)
$\mathsf{deps}(x)$	$=$	$\mathsf{atoms}(x)$	$\cap$	$\mathsf{Dependency}$	(set of dependencies)

3.4.3 Freshness

In this version of the core presentation dummy variables are used, which act like fresh variables. The definition of the dummy set is defined below:

$$\mathsf{Dum} ::= \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}\}$$

Consider the environment $(\alpha_{\mathtt{dum}} = \alpha_1); (\alpha_{\mathtt{dum}} = \alpha_2)$. In such an expression the dummy variables are considered to be distinct from each other, and indeed from α_1 and α_2. A function $\mathsf{nonDums}$ is also defined which extracts all the variables in a term while excluding the dummy variables. This is defined as follows:

$$\mathsf{nonDums}(x) = \mathsf{vars}(x) \backslash \mathsf{Dum}$$

3.5 Semantics of constraints/environments

The set of unifiers, renamings, and substitutions are defined in figure 3.12. Note that $\mathsf{Ren} \subset \mathsf{Unifier} \subset \mathsf{Sub}$. Renamings are used to instantiate type schemes, and the $\mathsf{Unifier}$ set contains unifiers generated by the constraint solver (see section 3.7). Substitution is defined in figure 3.13, where given a constraint term and a substitution, a resulting constraint term is produced.

Figure 3.12 Renaming, unifiers, and substitutions

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

Defined also are constraint solving contexts Δ. Such contexts are used during constraint solving and consist of the unifiers and an environment as a tuple. Also, $\langle u, e\rangle(v)$ is written to represent $u(v)$ and $\langle u, e\rangle; e'$ to represent $\langle u, e; e'\rangle$.

3.5.1 Shadowing

In the environment portion of a constraint solving context it may be the case that some parts are inaccessible. For example, in the following constraint solving context:

Figure 3.13 Substitution semantics

$$
\begin{array}{lcl}
v[sub] & = & \begin{cases} x, & \text{if } sub(v) = x \\ v, & \text{otherwise} \end{cases} \\
(\tau\ \mu)[sub] & = & \tau[sub]\ \mu[sub] \\
(\tau_1 \to \tau_2)[sub] & = & \tau_1[sub] \to \tau_2[sub] \\
x^{\overline{d}}[sub] & = & x[sub]^{\overline{d}} \\
(x_1 = x_2)[sub] & = & (x_1[sub] = x_2[sub]) \\
(e_1;e_2)[sub] & = & e_1[sub];e_2[sub] \\
(\forall \overline{v}.\, x)[sub] & = & \forall \overline{v}.\, x[sub] \text{ s.t. } \mathsf{dj}(\overline{v}, \mathsf{atoms}(sub)) \\
(\uparrow id{=}v)[sub] & = & \begin{cases} (\uparrow id{=}v[sub]), & \text{if } v[sub] \in \mathsf{Var} \\ \text{undefined}, & \text{otherwise} \end{cases} \\
(\downarrow id{=}x)[sub] & = & (\downarrow id{=}x[sub]) \\
\mathtt{poly}(e)[sub] & = & \mathtt{poly}(e[sub]) \\
x[sub] & = & x, \text{ otherwise}
\end{array}
$$

$$\langle u, bind_1; ev; bind_2 \rangle$$

if $ev \notin \mathsf{dom}(u)$, it is said that ev *shadows* $bind_1$ because ev could potentially be bound to an environment which rebinds $bind_1$. The predicate $\mathsf{shadowsAll}$ is defined below.

$$
\begin{array}{lcl}
\mathsf{shadowsAll}(\langle u,\ e\rangle) & \iff & \left\{ \begin{array}{llll}
 & (e = ev & \wedge & (\mathsf{shadowsAll}(\langle u,\ u(ev)\rangle) \vee \\
 & & & \quad ev \notin \mathsf{dom}(u))) \\
\vee & (e = (e_1;e_2) & \wedge & (\mathsf{shadowsAll}(\langle u,\ e_1\rangle) \vee \\
 & & & \quad \mathsf{shadowsAll}(\langle u,\ e_2\rangle))) \\
\vee & (e = \langle e', \overline{d}\rangle & \wedge & \mathsf{shadowsAll}(\langle u,\ e'\rangle))
\end{array} \right. \\
\mathsf{shadowsAll}(e) & \iff & \mathsf{shadowsAll}(\langle \emptyset,\ e\rangle)
\end{array}
$$

Below presents how to access the semantics of an identifier in a constraint solving context.

$$
\begin{array}{lcl}
\langle u, (\downarrow id{=}x)\rangle(id) & = & x \\
\langle u, (e^{\overline{d}})\rangle(id) & = & \mathsf{collapse}(\langle u, e\rangle(id)^{\overline{d}}) \\
\langle u, (e_1;e_2)\rangle(id) & = & \begin{cases} x, \text{ if} \langle u, e_2\rangle(id) = x \text{ or } \mathsf{shadowsAll}(\langle u, e_2\rangle) \\ \langle u, e_1\rangle(id), \text{otherwise} \end{cases} \\
\langle u, ev\rangle(id) & = & \begin{cases} \langle u, e\rangle(id), \text{if } u(ev) = e \\ \text{undefined}, \text{otherwise} \end{cases} \\
e(id) & = & \langle \varnothing, e\rangle(id)
\end{array}
$$

3.5.2 Relations

Two $\mathsf{instance}$ relations are defined here, the use of which can be seen in constraint solving.

$$x \xrightarrow{\mathsf{instance}} y^{\overline{d}}[sub], \quad \text{if} \quad \mathsf{collapse}(x^{\varnothing}) = (\forall v_0.y)^{\overline{d}} \text{ and } \mathsf{dom}(sub) = \overline{v}_0$$
$$x \xrightarrow{\mathsf{instance}} x, \quad \text{if} \quad \mathsf{collapse}(x^{\varnothing}) \text{ is not of the form } (\forall \overline{v}_0.y)^{\overline{d}}$$

3.6 Constraint generation

The *initial constraint generator* is defined in figure 4.17. This is referred to as the *initial* constraint generator because constraints are also generated during the constraint solving process (section 4.4).

In this presentation the relation $\twoheadrightarrow$ is used which is the smallest relation satisfying the rules in the constraint generator.

The rules in figure 3.14 return either an environment e, or something of the form $\langle v, e \rangle$, where e constraints the variable v.

It can be seen that datatype declarations only have one constructor by looking at rules (G18), (G14) and (G16). The core has been defined in this manner in order to reduce its complexity. In rule (G13) the datatype names are defined to have exactly one type variable argument.

Structure declarations are handled in rule (G20). In the core, signatures to constrain these structures are not presented, but this extension to the core can be found in section 6.4.

In order that environments can be sliced out correctly, environment variables are annotated with labels, such as in rule (G4). Such environment variables must be annotated with a label otherwise they could not be sliced out, and that environment variable would then shadow any following environment, even if the program point the label was assigned to was itself sliced out.

Figure 3.14 Initial constraint generator

All rules of the form $P \Leftarrow Q$ have to be read as $P \Leftarrow (Q \wedge \mathsf{dja}(e, e_1, e_2, \alpha, \alpha', ev, ev'))$

Expressions $(exp \triangleright \langle \alpha,\ e \rangle)$

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

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

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

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

Labelled datatype constructors $(ldcon \triangleright \langle \alpha,\ e \rangle)$

(G5) $dcon^l \triangleright \langle \alpha, \uparrow dcon \stackrel{l}{=} \alpha \rangle$

Patterns$(pat \triangleright \langle \alpha,\ e \rangle)$

(G6) $vvar^l_{\mathsf{p}} \triangleright \langle \alpha, \downarrow vvar \stackrel{l}{=} \alpha \rangle$

(G7) $dcon^l_{\mathsf{p}} \triangleright \langle \alpha, \uparrow dcon \stackrel{l}{=} \alpha \rangle$

(G8) $\lceil ldcon\ atpat \rceil^l \triangleright \langle \alpha, e_1; e_2; (\alpha_1 \stackrel{l}{=} \alpha_2 \to \alpha) \rangle \Leftarrow ldcon \triangleright \langle \alpha_1, e_1 \rangle \wedge atpat \triangleright \langle \alpha_2, e_2 \rangle$

Labelled type constructors $(ltc \triangleright \langle \delta,\ e \rangle)$

(G9) $tc^l \triangleright \langle \delta, \uparrow tc \stackrel{l}{=} \delta \rangle$

Types$(ty \triangleright \langle \alpha,\ e \rangle)$

(G10) $tv^l \triangleright \langle \alpha, \uparrow tv \stackrel{l}{=} \alpha \rangle$

(G11) $\lceil ty\ ltc \rceil^l \triangleright \langle \alpha', e_1; e_2; (\alpha' \stackrel{l}{=} \alpha\delta) \rangle \Leftarrow ty_1 \triangleright \langle \alpha_1, e_1 \rangle \wedge ltc \triangleright \langle \delta, e_2 \rangle$

(G12) $\alpha_1 \to \alpha_2 \triangleright \langle \alpha, e_1; e_2; (\alpha \stackrel{l}{=} \alpha_1 \to \alpha_2) \rangle \Leftarrow ty_1 \triangleright \langle \alpha_1, e_1 \rangle \wedge ty_2 \triangleright \langle \alpha_2, e_2 \rangle$

Datatype names$(dn \triangleright \langle \alpha,\ e \rangle)$

(G13) $\lceil tv\ tc \rceil^l \triangleright \langle \alpha', (\alpha' \stackrel{l}{=} \alpha\gamma); (\downarrow tc \stackrel{l}{=} \gamma); (\downarrow tv \stackrel{l}{=} \alpha) \rangle \Leftarrow \alpha \neq \alpha'$

Constructor bindings $(cb \triangleright \langle \alpha,\ e \rangle)$

(G14) $dcon^l_{\mathsf{c}} \triangleright \langle \alpha, \downarrow dcon \stackrel{l}{=} \alpha \rangle$

(G16) $dcon\ \texttt{of}^{\,l}\ ty \triangleright \langle \alpha, e_1; (\alpha' \stackrel{l}{=} \alpha_1 \to \alpha); (\alpha' \stackrel{l}{=} \alpha_1 \to \alpha); (\downarrow dcon \stackrel{l}{=} \alpha') \rangle \Leftarrow ty \triangleright \langle \alpha_1, e_1 \rangle$

Declarations $(dec \triangleright e)$

(G17) $\texttt{val rec}\ pat \stackrel{l}{=} exp \triangleright (ev = \texttt{poly}(e_1; e_2; (\alpha_1 \stackrel{l}{=} \alpha_2))); ev^l \Leftarrow pat \triangleright \langle \alpha_1, e_1 \rangle \wedge exp \triangleright \langle \alpha_2, e_2 \rangle$

(G18) $\texttt{datatype}\ dn \stackrel{l}{=} cb \triangleright (ev = ((\alpha_1 \stackrel{l}{=} \alpha_2); e_1; \texttt{poly}(e_2))); ev^l \Leftarrow dn \triangleright \langle \alpha_1, e_1 \rangle \wedge cb \triangleright \langle \alpha_2, e_2 \rangle$

(G19) $\texttt{open}^l\ strid \triangleright (\uparrow strid \stackrel{l}{=} ev); ev^l$

Structure declarations $(strdec \triangleright e)$

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

Structure expressions $(strexp \triangleright \langle ev,\ e \rangle)$

(G21) $strid^l \triangleright \langle ev, \uparrow strid \stackrel{l}{=} ev \rangle$

(G22) $\texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end} \triangleright \langle ev, (ev \stackrel{l}{=} ev'); (ev' = (e_1; ...; e_n)) \rangle$
$\Leftarrow strdec_1 \triangleright e_1 \wedge ... \wedge strdec_n \triangleright e_n \wedge \mathsf{dja}(e_1; ...; e_n, ev, ev')$

Each of the constraint generation rules are now discussed in turn.

3.6.1 Expressions

Rule (G1) generates accessors for value identifiers. For example, where a function previously defined is applied to an argument, the accessor which connects the use of this function to its definition is generated.

Rule (G2) handles let expressions, where some declarations are to be defined in the scope of some expression. We accomplish this with the $[e]$ notation - by encasing the environments from the declaration inside square brackets, upon closing of these brackets that environment is not exported (as described previously) and so binders inside are not available to following expressions.

Applications are handled with rule (G3). Here the $\rightarrow$ piece of constraint syntax is used to represent the expression part being used as a function.

Nameless functions are handled in rule (G4). We label the environment variable in this rule (and similarly in other rules) in order that declarations which have been sliced out do not shadow their context. If the environment variable was not labelled, then it would shadow the context it was in irrespective if some environment was sliced out or not.

3.6.2 Labelled datatype constructors

Labelled datatype constructors are handled in rule (G5). With this rule, an accessor is created to a datatype constructor in the same way as in (G1). The way datatype constructors and value identifiers are differentiated is enhanced in section 14.1 of [Rah10].

3.6.3 Patterns

Rule (G6) creates bindings for value variables occurring in patterns (such as in function declarations), while rule (G7) creates accessors to datatype constructors occurring in patterns. Rule (G8) handles the use of datatype constructors in patterns which have an argument.

3.6.4 Labelled type constructors

Labelled type constructors can occur in rule (G9). An accessor is created for this type constructor. This constraint can be generated for example in datatype constructor bindings.

3.6.5 Types

Rule (G10) handles the case where an external type variable occurring in a constructor binding is being dealt with. As a result, an accessor is generated (this should be connected during solving to the binder occurring in the declaration of the datatype which declares the constructor in which this explicit type variable is used). Rule (G11) is for labelled datatype constructors occurring in definitions of datatype constructors, and rule (G12) handles the case where an arrow type is specified in the user program.

3.6.6 Datatype names

Datatype names are handled with rule (G13). By looking at this rule it can be seen that datatype declarations have exactly one explicit type variable argument. Binders are created for both the name of the datatype and for the specified type variable argument in this rule.

3.6.7 Constructor bindings

Rules (G14) and (G16) give support for datatype constructor bindings to the constraint generator. (G14) is for a constructor which doesn't take an argument and rule (G16) is for constructors defined with the `of` keyword, where the type of the argument for the datatype constructor is defined. In both cases, binders are created for the name of the constructor.

3.6.8 Declarations

Functions are supported in the constraint generator with rule (G17). Function declarations are not supported using the `fun` keyword in this presentation of the core but the implementation does handle this, which is simply a syntactic variation. In this rule, as in rule (G18), the novel `poly` environment is used to make function

Figure 3.15 Syntactic forms used by the constraint solver

ek	$\in$	ErrKind	$::=$	$\texttt{clash}(\mu_1, \mu_2) \mid \texttt{circularity}$
er	$\in$	Error	$::=$	$\langle ek, \overline{d} \rangle$
$state$	$\in$	State	$::=$	$\texttt{slv}(\Delta, \overline{d}, e) \mid \texttt{succ}(\Delta, e) \mid \texttt{err}(er)$

bindings and datatype declarations polymorphic. The `open` feature is handled with rule (G19), where an accessor constraint is created using the structure identifier.

3.6.9 Structure declarations

Rule (G20) handles structure declarations. The environment generated for that structure is wrapped with $[e]$ to limit the scope of bindings occurring in that structure.

3.6.10 Structure expressions

Structure expressions are handled in rules (G21), which handles the case where the structure expression is some identifier which an accessor is then created for, and (G22) in the case of a `struct` expression, in which case environments are generated for each structure declaration and compose the results using the environment composition operator (`;`).

3.7 Constraint solving

3.7.1 Syntax

Additional syntactic forms which are used by the constraint solver are defined in figure 3.15. A constraint solving step is defined by the relation $\rightarrow$, and where $\rightarrow^*$ is its reflexive and transitive closure with respect to State.

Given an environment e to solve, the constraint solver starts in the state

$$\texttt{slv}(\langle \emptyset, \top \rangle, \emptyset, e)$$

and either terminates in one of two states:

- A success state $\texttt{succ}(\Delta)$ returning its current constraint solving context Δ.

- A failure state $\texttt{err}(er)$. In this case an error is returned of the one of the kinds discussed in figure 3.15.

3.7.2 The build function

A function build is defined which is similar to the substitution function but with the important change that it is recursively called in the variable case. Note that this function also collapses dependencies and is undefined on universally quantified terms and environments. Note that in the final case of the build defined below, the build function is allowed to be defined on constraint solving contexts. Types are built up in this way to avoid duplicating constraints in the system.

$$
\begin{array}{lcl}
\mathsf{build}(u, v) & = & \begin{cases} \mathsf{build}(u, x), & \text{if } u(v) = x \\ v, & \text{otherwise} \end{cases} \\
\mathsf{build}(u, \tau\,\mu) & = & \mathsf{build}(u, \tau)\,\mathsf{build}(u, \mu) \\
\mathsf{build}(u, \tau_1 \to \tau_2) & = & \mathsf{build}(u, \tau_1) \to \mathsf{build}(u, \tau_2) \\
\mathsf{build}(u, x^{\overline{d}}) & = & \mathsf{collapse}(\mathsf{build}(u, x)^{\overline{d}}) \\
\mathsf{build}(u, x) & = & x, \text{ otherwise} \\
\mathsf{build}(\langle u,\ e\rangle, x) & = & \mathsf{build}(u, x)
\end{array}
$$

3.7.3 Environment difference

In this presentation of the core the notion of *environment difference* exists, which is a way to extract certain environments which have been generated in the constraint solving machinery. This function is used in the constraint solver to determine what has been added to an environment since a constraint solving call was made.

For example, let us assume that the constraint solver started with constraint solving context $\langle \top, e\rangle$ and ended in a success state with constraint solving context $\langle \top, e'\rangle$. The environment difference operation computes the *new* parts of the environment, by taking e' and discarding the pre-existing environment (the environment e) portion of it.

This operation is defined below.

$$
\begin{array}{lcll}
e \backslash e & = & \top & \\
e_1 \backslash (e_2; e_3) & = & (e_1 \backslash e_2); e_3 & \text{if } e_1 \neq (e_2; e_3)
\end{array}
$$

Figure 3.16 Monomorphic to polymorphic environment

$$
\begin{array}{lcll}
\mathsf{toPoly}(\Delta, {\downarrow}vid{=}\tau) & = & ({\downarrow}vid \stackrel{\overline{d}}{=} \forall\overline{\alpha}.\,\tau'), & \text{if } \tau' = \mathsf{build}(\Delta, \tau) \\
 & & \wedge\ \overline{\alpha} = (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\Delta)) \cup \{\alpha_{\mathtt{dum}}\}) & \\
 & & \qquad \wedge\{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\Delta) \wedge \alpha \in \mathsf{vars}(\tau') \setminus \overline{\alpha}\} & \\
\mathsf{toPoly}(\langle u, e\rangle, e_0^{\overline{d}}) & = & \langle u', (e; e\backslash e'^{\overline{d}})\rangle & \text{if } \mathsf{toPoly}(\langle u, e\rangle, e_0) = \langle u', e'\rangle \\
\mathsf{toPoly}(\Delta, e_1{;}e_2) & = & \mathsf{toPoly}(\Delta', e_2), & \text{if } \mathsf{toPoly}(\Delta, e_1) = \Delta' \\
\mathsf{toPoly}(\Delta, e) & = & \Delta; e, & \text{if none of the above applies}
\end{array}
$$

3.7.4 Polymorphic environments

Here the function monos is introduced, which computes the set of dependent monomorphic type variables which occur in some environment given as an argument with respect to a unifier.

$$\mathsf{monos}(\Delta) = \{\alpha^{\mathsf{deps}(\tau)} \mid \exists vid.\tau = \mathsf{build}(\Delta, \Delta(vid)) \wedge \alpha \in \mathsf{nonDums}(\tau)\}$$

It is ensured that the value identifier in the definition of monos is monomorphic by building it after looking the value identifier up in the current environment, as the build function is undefined on quantified type schemes, which are present in the case of a polymorphic binder.

The toPoly function is now defined, which is responsible for taking a monomorphic binder and creating from it a polymorphic binder by quantifying the variables which do not occur in the monomorphic binder. This function is defined in figure 3.16.

There are four cases presented in the toPoly function. In the case where the environment has been annotated with dependencies, the dependencies are added on to the polymorphic version of the environment given as the argument by recursing. In the case of environment composition, toPoly is called on the first environment, add it to the constraint solving context, and then use that when calling toPoly with the the second environment. In the case of an environment where no rules apply, the environment is returned that was given as the argument. This leaves the case where there is a monomorphic binder as an argument.

In the case where a monomorphic binder is being dealt with as the argument, three things are calculated:

1. The built version of the type given in the monomorphic binder. This is τ'.
2. The variables to be quantified over. To do this all of the variables are gathered in the previously computed type τ', and take the intersection of those variables with the set of the internal type variables. Discarded from this set are all of

the monomorphic variables in the constraint solving context (computed by calling monos), and also the dummy variables. This is written $\overline{\alpha}$.

3. The dependencies with which the new polymorphic binder should be annotated. This is done by taking the dependencies which are annotated on the internal type variables that are returned from the call to the monos where those type variables occur in the built-up type τ' and are not in the computed set $\overline{\alpha}$.

Given these three computed items, a polymorphic binder is created by quantifying τ' with the set $\overline{\alpha}$ and place the dependencies $\overline{d}$ on that binder.

3.7.5 Constraint solving rules

The constraint solver is presented in figure 3.17, and discuss some of the important rules in the constraint solver below.

Discussion of some important rules

- Rule (S6) generates a type constructor clash if it notices two type constructor names have an equality constraint constraining them, where the type constructors are not actually equal.
- Rule (U1) generates a circularity error. This error is generated where a variable is constrained to be equal to some term where the variable occurs in that term.
- Rule (U4) handles constraints where environment variables are assigned to environments. In this case the environment is solved first, then a mapping is added from the environment variable to this solved environment to the unifier in the constraint solving context.
- Rule (C1) handles composition environments. In the case of such composition environments the first environment that is composed with the second is solved, then use the constraint solving context that was returned from the `slv` call in order to call `slv` again with the second environment.
- Rules (A1) and (A3) deal with accessors. In the case of rule (A1) the accessors are connected to their binders by looking the variable up in the constraint solving context. In rule (A3) success is generated, but this can be used to report free identifiers. If $\texttt{slv}(\Delta, \overline{d}, \uparrow id{=}v) \rightarrow \texttt{succ}(\Delta)$ and $\neg\mathsf{shadows}(\Delta)$ then there is no binder for id.

Figure 3.17 Constraint solver

equality constraint reversing

(R) $\mathtt{slv}(\Delta, \overline{d}, x = y) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, y = x), \quad$ if $s = \mathsf{Var} \cup \mathsf{Dependent} \wedge y \in s \wedge x \notin s$

equality simplification

(S1) $\mathtt{slv}(\Delta, \overline{d}, x = x) \rightarrow \mathtt{succ}(\Delta)$

(S2) $\mathtt{slv}(\Delta, \overline{d}, x^{\overline{d}'} = y) \rightarrow \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', x = y)$

(S3) $\mathtt{slv}(\Delta, \overline{d}, \tau\ \mu = \tau'\ \mu') \rightarrow \mathtt{slv}(\Delta, \overline{d}, (\mu = \mu');(\tau = \tau'))$

(S4) $\mathtt{slv}(\Delta, \overline{d}, \tau_1 \to \tau_2 = \tau_3 \to \tau_4) \rightarrow \mathtt{slv}(\Delta, \overline{d}, (\tau_1 = \tau_3);(\tau_2 = \tau_4))$

(S5) $\mathtt{slv}(\Delta, \overline{d}, \tau = \tau') \rightarrow \mathtt{slv}(\Delta, \overline{d}, \mu = \mathtt{arr})$,
if $\{\tau, \tau'\} = \{\tau_1\ \mu, \tau_2 \to \tau_3\}$

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

unifier access

Rules (U1) through (U6) have also the side condition $v \neq x$ and $y = \mathsf{build}(u, x^{\overline{d}})$.

(U1) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, v = x) \rightarrow \mathtt{err}(\langle \mathtt{circularity}, \mathsf{deps}(y) \rangle)$,
if $v \in \mathsf{vars}(y) \backslash (\mathsf{dom}(u) \cup \mathsf{Env} \cup \mathsf{Dum}) \wedge \mathsf{strip}(y) \neq v$

(U2) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, v = x) \rightarrow \mathtt{succ}(\langle u, e \rangle)$,
if $v \in \mathsf{vars}(y) \backslash (\mathsf{dom}(u) \cup \mathsf{Env}) \wedge \mathsf{strip}(y) \neq v$

(U3) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, v = x) \rightarrow \mathtt{succ}(\langle u \oplus \{v \mapsto y\}, e \rangle)$,
if $v \notin (\mathsf{vars}(y) \backslash \mathsf{Dum}) \cup \mathsf{dom}(u) \cup \mathsf{Env}$

(U4) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, v = x) \rightarrow \mathtt{succ}(\langle u' \oplus \{v \mapsto e \backslash e'\}, e \rangle)$,
if $v \in \mathsf{Env} \backslash \mathsf{dom}(u) \wedge \mathtt{slv}(\langle u, e \rangle, \overline{d}, x) \to^* \mathtt{succ}(\langle u', e' \rangle)$

(U5) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, v = x) \rightarrow \mathtt{err}(er)$,
if $v \in \mathsf{Env} \backslash \mathsf{dom}(u) \wedge \mathtt{slv}(\langle u, e \rangle, \overline{d}, x) \to^* \mathtt{err}(er)$

(U6) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, v = x) \rightarrow \mathtt{slv}(\langle u, e \rangle, \overline{d}, z = x)$,
if $u(v) = z$

binders

(B1) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, {\downarrow}id{=}x) \rightarrow \mathtt{succ}(\langle u, (e; {\downarrow}id \stackrel{\overline{d}}{=} x) \rangle)$

empty/dependent/variables

(E) $\mathtt{slv}(\Delta, \overline{d}, \top) \rightarrow \mathtt{succ}(\Delta)$

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

(V) $\mathtt{slv}(\langle u, e \rangle, \overline{d}, ev) \rightarrow \mathtt{succ}(\langle u, e; ev^{\overline{d}} \rangle)$

composition environments

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

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

accessors

(A1) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', v = \tau[ren])$,
if $\Delta(id) = (\forall \overline{\alpha}.\, \tau)^{\overline{d}'} \ \wedge\ \mathsf{dom}(ren) = \overline{\alpha} \ \wedge\ \mathsf{dj}(\mathsf{vars}(\langle \Delta, v \rangle), \mathsf{ran}(ren))$

(A2) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \mathtt{slv}(\Delta, \overline{d}, v = x)$,
if $\Delta(id) = x \ \wedge\ \mathsf{strip}(x)$ is not of the form $\forall \overline{\alpha}.\, \tau$

(A3) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \mathtt{succ}(\Delta)$, if $\Delta(id)$ undefined

polymorphic environments

(P1) $\mathtt{slv}(\langle u_1, e_1 \rangle, \overline{d}, \mathtt{poly}(e)) \rightarrow \mathtt{succ}(\mathsf{toPoly}(\langle u_2, e_1 \rangle, e_1 \backslash e_2))$,
if $\mathtt{slv}(\langle u_1, e_1 \rangle, \overline{d}, e) \to^* \mathtt{succ}(\langle u_2, e_2 \rangle)$

(P2) $\mathtt{slv}(\langle u_1, e_1 \rangle, \overline{d}, \mathtt{poly}(e)) \rightarrow \mathtt{err}(er)$,
if $\mathtt{slv}(\langle u_1, e_1 \rangle, \overline{d}, e) \to^* \mathtt{err}(er)$

- Rule (P1) deals with the promotion of monomorphic binders to a polymorphic status by calling the toPoly function.

The relations isErr and solvable are defined as shown below, which are used in the definition of the minimisation and enumeration algorithms.

$$
\begin{array}{lcl}
e \xrightarrow{\mathsf{isErr}} er & \iff & \mathtt{slv}(\langle \emptyset,\ \top \rangle, \emptyset, e) \rightarrow^* \mathtt{err}(er) \\
\mathsf{solvable}(e) & \iff & \exists \Delta.\mathtt{slv}(\langle \emptyset,\ \top \rangle, \emptyset, e) \rightarrow^* \mathtt{succ}(\Delta) \\
\mathsf{solvable}(\mathit{strdec}) & \iff & \exists e.\mathit{strdec} \triangleright e \wedge \mathsf{solvable}(e)
\end{array}
$$

3.8 Minimisation and enumeration

3.8.1 Dummy binders

In the first phase of the minimisation algorithm, an attempt is made to try to remove potentially large sections of code (such as structures, datatype definitions etc). This is done by replacing binders with dummy binders. Let us define the set lBinds, which will gather the labels of all binders in a given environment e be defined as follows :

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

3.8.2 Constraint filtering

The definition of the constraint filtering function filt can be seen in figure 3.18. This function is used to check the solvability of constraints in the case that some have been discarded, and label sets are used to accomplish this. In $\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)$, e is the environment to be filtered and $\overline{l}_1$ contains the labels that are to be kept. Any accessors or equality constraints which are annotated with a label which is present in the $\overline{l}_2$ set which is the set of labels not to be kept, if a binder label is in that set then it is turned into a dummy binder. Any environment labelled with a label which is not in $\overline{l}_1 \cup \overline{l}_2$ is discarded. Note that binders that are discarded (any binder with a label not in $\overline{l}_1 \cup \overline{l}_2$) and the binders that are turned into dummy binders (those in $\overline{l}_2$) are distinguished in order that when throwing away an environment, care is taken so that accessors in the resulting environment do not get captured by a different binder with the same name.

The set DepStatus is defined as follows:

Figure 3.18 Constraint filtering

filtering function

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

$$\begin{array}{lcl} \mathsf{filt}(ev = e, \overline{l}_1, \overline{l}_2) & = & (ev = \mathsf{filt}(e, \overline{l}_1, \overline{l}_2)) \\ \mathsf{filt}(e_1;e_2, \overline{l}_1, \overline{l}_2) & = & \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2);\mathsf{filt}(e_2, \overline{l}_1, \overline{l}_2) \\ \mathsf{filt}(\texttt{poly}(e), \overline{l}_1, \overline{l}_2) & = & \texttt{poly}(\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)) \\ \mathsf{filt}(\top, \overline{l}_1, \overline{l}_2) & = & \top \end{array}$$

conversion of environments into dummy environments

$$\begin{array}{lcl} \mathsf{dum}(\downarrow id{=}x) & = & (\downarrow id{=}\mathsf{toDumVar}(x)) \\ \mathsf{dum}(ev) & = & ev_{\texttt{dum}} \\ \mathsf{dum}(e_1;e_2) & = & \mathsf{dum}(e_1);\mathsf{dum}(e_2) \\ \mathsf{dum}(e^{\overline{d}}) & = & \mathsf{dum}(e) \\ \mathsf{toDumVar}(\sigma) & = & \alpha_{\texttt{dum}} \\ \mathsf{toDumVar}(\mu) & = & \delta_{\texttt{dum}} \\ \mathsf{toDumVar}(e) & = & ev_{\texttt{dum}} \end{array} \qquad \begin{array}{lcl} \mathsf{dum}(c) & = & \top \\ \mathsf{dum}(acc) & = & \top \\ \mathsf{dum}(\top) & = & \top \end{array}$$

$$ds \in \mathsf{DepStatus} ::= \texttt{keep} \mid \texttt{drop} \mid \texttt{keep-only-binders}$$

and also the set DepEnv, which will map a dependency to a dependency status:

$$de \in \mathsf{DepEnv} ::= \mathsf{Dependency} \to \mathsf{DepStatus}$$

Defined also is *de*s on dependency sets as follows:

$$de(\overline{d}) = \{de(d) \mid d \in \overline{d}\}$$

Figure 3.19 Minimisation and enumeration algorithms

minimisation

$$
\begin{array}{lllll}
\text{(MIN1)} & \langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2 \rangle & \rightarrow_{\text{test}} & \langle e, \overline{l}_1 \cap \overline{d}, \overline{l}_2 \cap \overline{d} \rangle, & \\
 & & & \text{if } \mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\}) \xrightarrow{\mathsf{isErr}} \langle ek, \overline{d} \rangle & \\
\text{(MIN2)} & \langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2 \rangle & \rightarrow_{\text{test}} & \langle e, \overline{l}_1 \cup \{l\}, \overline{l}_2 \rangle, \quad \text{if } \mathsf{solvable}(\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\})) & \\
\text{(MIN3)} & \langle e, er \rangle \xrightarrow{\mathsf{min}} er', & \text{if} & \mathsf{lBinds}(e) = \overline{l} & \\
 & & \wedge & \langle e, \mathsf{labs}(er) \setminus \overline{l}, \mathsf{labs}(er) \cap \overline{l} \rangle \rightarrow^*_{\text{test}} \langle e, \overline{l}_1, \emptyset \rangle & \text{(phase1)} \\
 & & \wedge & \langle e, \emptyset, \overline{l}_1 \rangle \rightarrow^*_{\text{test}} \langle e, \overline{l}_2, \emptyset \rangle & \text{(phase2)} \\
 & & \wedge & \mathsf{filt}(e, \overline{l}_2, \emptyset) \xrightarrow{\mathsf{isErr}} er' &
\end{array}
$$

enumeration

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

3.8.3 Justification of the minimisation algorithm

Here a variation on the example given by Rahli [Rah10] is presented, where the notion of tuples have been removed so an example can be shown which is fully supported by the core presented in this chapter.

Consider the program in figure 3.20.

Figure 3.20 Example program showing the need for minimisation

```
1 datatype 'a mydt = DUMMY
2 val rec e = fn x => fn y => DUMMY
3 val rec f = fn x => e (x (fn z => z)) (x (fn DUMMY => DUMMY))
4 val rec g = fn y => y true
5 val u = f g
```

For this program, after the first constraint solving run the following error is produced:

```
let
    datatype 'a mydt = DUMMY
    val rec e = fn x => fn y => DUMMY
    val rec f = fn x => e (x (fn z => z)) (x (fn DUMMY => DUMMY))
    val rec g = fn y => y true
in
    f g
end
```

After the minimisation process however, the following error is generated:

```
let
    datatype 'a mydt = DUMMY
    val rec e = fn x => fn y => DUMMY
    val rec f = fn x => e (x (fn z => z)) (x (fn DUMMY => DUMMY))
    val rec g = fn y => y true
in
    f g
end
```

As `x` is monomorphic, it is constrained by both `z` and by the datatype constructor `DUMMY`. There is no known way yet to find this type of error without the help of a minimiser, and investigation on this is left for future work.

3.8.4 Minimisation

There are some cases where when an error is discovered during constraint solving it may not be minimal, that is, that some of the labels attributed to the error are extraneous. The minimisation algorithm can be seen in figure 3.19.

The $\rightarrow_{\mathsf{test}}$ relation determines whether a given label can be removed from the set of labels associated with an error without causing the error to no longer exist in the user program. Let $\rightarrow^{*}_{\mathsf{test}}$ be the reflexive and transitive closure of $\rightarrow_{\mathsf{test}}$.

The process of minimisation is separated into two phases (labelled phase1 and phase2 in figure 3.19 respectively). In phase one binders are turned into dummy binders which can potentially remove large sections of code, then in phase two labels are removed one at a time until the minimal amount of labels are found that are attributed to the error.

A minimisation step is represented as $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2 \rangle \xrightarrow{\mathsf{test}} \langle e, \overline{l}_3, \overline{l}_4 \rangle$, where $\overline{l}_3$ and $\overline{l}_4$ depend upon the solvability of $\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\})$, referred to now as e'. The full set of labels for the error the minimiser is working on is the set $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$, and $\{l\} \uplus \overline{l}_2$ is the label set where discard attempts must still be made. The new environment e' is obtained from e by filtering out the constraints not labelled by $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$, by

filtering out out accessors and equality constraints annotated with the label l, and by creating dummy binders and environment variables for binders and environment variables which were annotated with this label. If the new environment is solvable, then label l must be in the error's label set for the error to occur, and so $\bar{l}_3 = \bar{l}_1 \cup \{l\}$ and $\bar{l}_4 = \bar{l}_2$, otherwise this label is extraneous and can be removed.

3.8.5 Enumeration

An enumeration step is denoted with the relation $\rightarrow_{\mathsf{e}}$, with $\rightarrow_{\mathsf{e}}^{*}$ being its reflexive (w.r.t. EnumState) and transitive closure. Enumeration states are defined below:

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

The enumeration process always starts in the state $\mathsf{enum}(e)$ and ends in the state $\texttt{errors}(\overline{er})$. The enumeration algorithm creates *filters*, which form the search space built when searching for errors, and starts with the empty filter (which causes all constraints to be considered).

After an error has been found and the minimisation process has been completed, the labels of the error that has been located are used to build new filters (see $\bar{\bar{l}'}$ in rule (ENUM4)). When all filters are exhausted the enumeration algorithm stops.

After the enumeration algorithm has stopped, the errors that have been found are all the minimal type errors in the analyzed piece of code.

3.9 Slicing

3.9.1 Dot Terms

After an error has been located in the user program, a type error slice is made from the labels and the error kind ek. This is done by the slicing function sl defined in figure 3.25. Any program nodes which are annotated with labels not occurring in the set of labels as part of the error are replaced by "dot" terms, which are used to show that some program nodes have been thrown away as they do not contribute to the error. As an example if a node is removed annotated with label l_2 in $\lceil 1^{l_1}()^{l_2} \rceil^{l_3}$, then $\lceil 1^{l_1} \texttt{dot-e}(\varnothing) \rceil^{l_3}$ results. This is displayed as 1 $\langle .. \rangle$.

Any syntactic form that can be produced using the grammar rules defined in the combination of figures 3.8 and 3.21 is referred to as a *slice*, and a *type error slice*

Figure 3.21 Extension of the syntax and constraint generator to "dot" terms

extension of the constraint syntax

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{LabTyCon} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & \mathsf{DatName} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) \\
\mathsf{LabDatCon} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & \mathsf{Dec} & ::= & \cdots \mid \texttt{dot-d}(\overrightarrow{term}) \\
\mathsf{Ty} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & \mathsf{AtExp} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) \\
\mathsf{ConBind} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & \mathsf{Exp} & ::= & \cdots \mid \texttt{dot-e}(\overrightarrow{term}) \\
\mathsf{AtPat} & ::= & \cdots \mid \texttt{dot-p}(\overrightarrow{pat}) & & & \\
\mathsf{Pat} & ::= & \cdots \mid \texttt{dot-p}(\overrightarrow{pat}) & & & \\
\mathsf{StrDec} & ::= & \cdots \mid \texttt{dot-d}(\overrightarrow{term}) & & & \\
\mathsf{StrExp} & ::= & \cdots \mid \texttt{dot-s}(\overrightarrow{term}) & & &
\end{array}
$$

extension of the constraint generator

(G24) $[\![\texttt{dot-d}(\langle term_1, \ldots, term_n \rangle)]\!] = [[\![term_1]\!]; \cdots; [\![term_n]\!]]$

(G25) $[\![\texttt{dot-p}(\langle pat_1, \ldots, pat_n \rangle), \alpha]\!] = [\![pat_1]\!]; \cdots; [\![pat_n]\!]$

(G26) $[\![\texttt{dot-s}(\langle term_1, \ldots, term_n \rangle), ev]\!] = [[\![term_1]\!]; \cdots; [\![term_n]\!]]$

(G27) $[\![\texttt{dot-e}(\langle term_1, \ldots, term_n \rangle), \alpha]\!] = [[\![term_1]\!]; \cdots; [\![term_n]\!]]$

Figure 3.22 Labelled abstract syntax trees

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

any slice for which the constraint generation algorithm (which has been extended to dot terms) only generates unsolvable constraints.

An alternative definition of the external labelled syntax presented in 3.8 is given here. In figure 3.22 the labelled abstract syntax trees are defined, where 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*. toTree is defined in figure 3.23 which associates a *tree* with every *term* (toTree is also defined on a sequence of *terms*).

Figure 3.24 also defines the function getDot which generates terms in Dot from nodes. This function is used in the slicing algorithm to generate dot nodes from labelled nodes.

3.9.2 Tidying

Defined here is flat, which flattens a series of terms. For example, flattening $\langle..1..\langle..()..\rangle..\rangle$ becomes $\langle..1..()..\rangle$. Note that nested dot terms are not always flat-

Figure 3.23 From terms to trees

$$
\begin{array}{lcl}
\mathsf{toTree}(tc^{l}) & = & \langle\langle \mathtt{lTc}, \mathtt{id}\rangle, l, \langle tc\rangle\rangle \\
\mathsf{toTree}(dcon^{l}) & = & \langle\langle \mathtt{lDcon}, \mathtt{id}\rangle, l, \langle dcon\rangle\rangle \\
\mathsf{toTree}(tv^{l}) & = & \langle\langle \mathtt{ty}, \mathtt{id}\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}(\lceil ty\ ltc \rceil^{l}) & = & \langle\langle \mathtt{ty}, \mathtt{tyCon}\rangle, l, \langle \mathsf{toTree}(ty), \mathsf{toTree}(ltc)\rangle\rangle \\
\mathsf{toTree}(dcon^{l}_{\mathsf{c}}) & = & \langle\langle \mathtt{conbind}, \mathtt{id}\rangle, l, \langle dcon\rangle\rangle \\
\mathsf{toTree}(dcon\ \mathtt{of}^{\,l}\ ty) & = & \langle\langle \mathtt{conbind}, \mathtt{conbindOf}\rangle, l, \langle dcon, \mathsf{toTree}(ty)\rangle\rangle \\
\mathsf{toTree}(\lceil tv\ tc \rceil^{l}) & = & \langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l, \langle tv, tc\rangle\rangle \\
\mathsf{toTree}(\mathtt{val\ rec}\ pat \stackrel{l}{=} exp) & = & \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l, \langle \mathsf{toTree}(pat), \mathsf{toTree}(exp)\rangle\rangle \\
\mathsf{toTree}(\mathtt{datatype}\ dn \stackrel{l}{=} cb) & = & \langle\langle \mathtt{dec}, \mathtt{decDat}\rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(cb)\rangle\rangle \\
\mathsf{toTree}(\mathtt{open}^{l}\ strid) & = & \langle\langle \mathtt{dec}, \mathtt{decOpn}\rangle, l, \langle strid\rangle\rangle \\
\mathsf{toTree}(vid^{l}_{\mathsf{e}}) & = & \langle\langle \mathtt{atexp}, \mathtt{id}\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 \\
\mathsf{toTree}(vid^{l}_{\mathsf{p}}) & = & \langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l, \langle vid\rangle\rangle \\
\mathsf{toTree}(\lceil ldcon\ atpat \rceil^{l}) & = & \langle\langle \mathtt{pat}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(ldcon), \mathsf{toTree}(atpat)\rangle\rangle \\
\mathsf{toTree}(\mathtt{structure}\ strid \stackrel{l}{=} strexp) & = & \langle\langle \mathtt{strdec}, \mathtt{strdecStr}\rangle, l, \langle strid, \mathsf{toTree}(strexp)\rangle\rangle \\
\mathsf{toTree}(strid^{l}) & = & \langle\langle \mathtt{strexp}, \mathtt{id}\rangle, l, \langle strid\rangle\rangle \\
\mathsf{toTree}(\langle term_1, \ldots, term_n\rangle) & = & \langle \mathsf{toTree}(term_1), \ldots, \mathsf{toTree}(term_n)\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}e}(\overrightarrow{term})) & = & \langle \mathtt{dotE}, \mathsf{toTree}(\overrightarrow{term})\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}d}(\overrightarrow{term})) & = & \langle \mathtt{dotD}, \mathsf{toTree}(\overrightarrow{term})\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}p}(\overrightarrow{pat})) & = & \langle \mathtt{dotP}, \mathsf{toTree}(\overrightarrow{pat})\rangle \\
\mathsf{toTree}(\mathtt{dot\text{-}s}(\overrightarrow{term})) & = & \langle \mathtt{dotS}, \mathsf{toTree}(\overrightarrow{term})\rangle \\
\mathsf{toTree}(\mathtt{struct}^{l}\ strdec_1 \cdots strdec_n\ \mathtt{end}) & = & \langle\langle \mathtt{strexp}, \mathtt{strexpSt}\rangle, l, \mathsf{toTree}(\langle strdec_1, \ldots, strdec_n\rangle)\rangle
\end{array}
$$

tened as different semantics can sometimes be produced, for example

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

is not flattened to become

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

as the semantics have changed- the first is a typable slice and the second is not.

The predicates below check the classes of trees:

Figure 3.24 Definition of getDot

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

Figure 3.25 Slicing algorithm

$$
\begin{array}{lll}
\text{(SL1)} & \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \bar{l}) = & \\
 & \quad \begin{cases} \langle node, l, \mathsf{sl}_1(\overrightarrow{tree}, \bar{l})\rangle, & \text{if } l \in \bar{l} \text{ and } \mathsf{getDot}(node) \neq \texttt{dotS} \\ \langle node, l, \mathsf{tidy}(\mathsf{sl}_1(\overrightarrow{tree}, \bar{l}))\rangle, & \text{if } l \in \bar{l} \text{ and } \mathsf{getDot}(node) = \texttt{dotS} \\ \langle \mathsf{getDot}(node), \mathsf{flat}(\mathsf{sl}_2(\overrightarrow{tree}, \bar{l}))\rangle, & \text{otherwise} \end{cases} \\
\text{(SL2)} & \mathsf{sl}_1(\langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle, \bar{l}) = & \langle dot, \mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \bar{l}), \ldots, \mathsf{sl}_2(tree_n, \bar{l})\rangle)\rangle \\
\text{(SL3)} & \mathsf{sl}_2(\langle dot, \langle tree_1, \ldots, tree_n\rangle\rangle, \bar{l}) = & \langle dot, \mathsf{flat}(\langle \mathsf{sl}_2(tree_1, \bar{l}), \ldots, \mathsf{sl}_2(tree_n, \bar{l})\rangle)\rangle \\
\text{(SL4)} & \mathsf{sl}_1(\langle node, l, \overrightarrow{tree}\rangle, \bar{l}) = & \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \bar{l}) \\
\text{(SL5)} & \mathsf{sl}_2(\langle node, l, \overrightarrow{tree}\rangle, \bar{l}) = & \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \bar{l}) \\
\text{(SL6)} & \mathsf{sl}_1(\langle tree_1, \ldots, tree_n\rangle, \bar{l}) = & \langle \mathsf{sl}_1(tree_1, \bar{l}), \ldots, \mathsf{sl}_1(tree_n, \bar{l})\rangle \\
\text{(SL7)} & \mathsf{sl}_2(\langle tree_1, \ldots, tree_n\rangle, \bar{l}) = & \langle \mathsf{sl}_2(tree_1, \bar{l}), \ldots, \mathsf{sl}_2(tree_n, \bar{l})\rangle \\
\text{(SL8)} & \mathsf{sl}_1(id, \bar{l}) = & id \\
\text{(SL9)} & \mathsf{sl}_2(id, \bar{l}) = & \langle \texttt{dotE}, \langle\rangle\rangle
\end{array}
$$

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

These can be used to check whether a given tree has any binders (declares), is a pattern (pattern). With these in place, a formal definition of flat is defined below:

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

A function tidy is also defined which merges dot terms containing declarations in structures. It is defined below.

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

3.9.3 Algorithm

Figure 3.25 formally defines the slicing algorithm. In this figure let $\mathsf{sl}(strdec, \overline{l})$ be an abbreviation for $\mathsf{sl}(\mathsf{toTree}(strdec), \overline{l})$.

3.9.4 Meeting criteria of Jun [YMTW00] et al.

Below, we discuss the properties of [YMTW00] with respect to program slicing, and we show how Skalpel meets such properties.

Correct We believe that for an erroneous program as defined by the Definition of Standard ML [MTHM98] where syntax supported by the core is used we will always generate unsolvable constraints from the constraint generator. The correctness of the Skalpel implementation is evaluated by the test framework discussed in appendix section C.7.

Precise Before reporting final errors to the user, Skalpel runs its minimisation algorithm which removes all extraneous labels (that is, labels which were initially reported to be part of the error but were discovered later to be irrelevant), so Skalpel is precise in the errors that it generates. Some work has been done with experimenting in the implementation with a phase where Skalpel shows errors to the user where those errors are not minimal, but this is present for implementation reasons, as the time taken to generate a minimal error can be in some cases quite high.

Succinct With respect to the formal type error slice, flattening Skalpel's program slices allows us to be succinct in our reports. With the highlighting, as we highlight the code relevant to the error anyway and do not highlight anything in the program which is irrelevant to the error, we are as succinct here as possible. For these reasons, Skalpel is succinct in its error reporting.

A-mechanical. Unlike how the current compilers for Standard ML report error messages, Skalpel will never report as part of its error some internal representation which has been generated.

Source-based Skalpel can present to the user either highlighting or a formal representation of a type error slice. The highlighting presented is entirely source-based, however the type error slice contains characters which are used to show that some portion of code has been sliced out ($\langle..\rangle$). Removing this notation would not allow us to be as succinct as we are now, and as the highlighting is entirely source based, we meet this property.

Unbiased Skalpel does not prioritise any error that it reports over any other error, and so it is unbiased. The enumeration algorithm is responsible for this, as it continues to search for unique errors when some have already been found. While some regions differently to others (e.g. end points of errors, which may be highlighted in another colour), no region is prioritised over any other region, and so also in that sense Skalpel is unbiased.

Comprehensive Skalpel presents *all and only* those pieces of code which are responsible for an error, and so it follows that the user will never need to modify any code not presented by Skalpel in order to solve an error. For this reason, Skalpel is comprehensive in nature.

3.10 Other Implementations

Some of the authors of papers reviewed in chapter 2 have prepared implementations which are at least largely based on the work they have presented in their papers. This section aims to give an overview of some implementations that are currently available, whether they be simply improvements on algorithm W or more sophisticated implementations where for example type error corrections are suggested to the user.

The implementation of Goto and Sasano [GS12] can be found at the following URL[1]:

http://www.cs.ise.shibaura-it.ac.jp/lambda-mode/

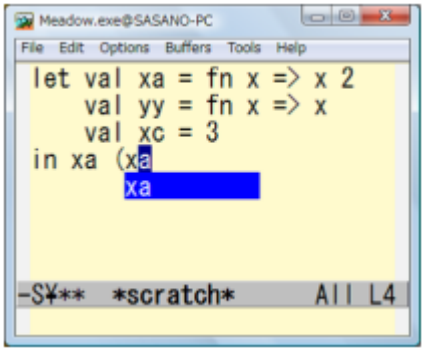
In this implementation the user writes their Standard ML code in Emacs (the implementation is an Emacs mode called Lambda-mode), as users of Skalpel are able to do, but rather than calling an external binary as Skalpel does, everything in this implementation is done in Emacs Lisp. A screenshot of what the software looks like can be seen in figure 3.26.

While they gain the advantage that their software is extremely easy to install they tie their users to Emacs, so if the user doesn't use that editor they will not be able to use the software. An advantage to implementing Skalpel in the current way is that users are not tied to any specific editor.

There are two implementations of type inference algorithms in this work, one is Milner's type inference algorithm W [DM82], and the other is described as a type inference algorithm V, which was an algorithm created by the authors. The W

[1] In the README located in the tarball at that location, Goto and Sassano invite users to try sample programs located in the 'sample' folder, but this directory does not exist. It is recommended to enter the program shown in figure 8 of their paper as a sample program.

Figure 3.26 Goto and Sasano Implementation



algorithm is used in the case of function applications, and in the case where there is a let expression which has a body (in the case where there is a let expression without a body the V algorithm is used).

In this implementation the way that candidates are checked whether they will make the code typable is that they are inserted into the code and then the unification algorithm is run with that candidate inserted. If unification succeeds, then this candidate is presented to the user, if it does not, then it is removed from the set of possible candidates[2] that the user will be presented with. The authors essentially check for valid candidates in a similar way that the Skalpel analysis engine finds minimal type error slices. Each approach (ours and theirs) simply repeatedly run its own unification algorithm and looks at the results.

Another interesting implementation is that of Seminal, written in Ocaml, which detects type errors in untypable pieces of Ocaml code and provides suggestions to the user.

A basic example is shown here. Suppose the user has the basic program shown in figure 3.27.

Figure 3.27 A simple ill-typed Ocaml program

```
1  let
2    concatFunction s1 s2 = s1 ^ s2
3  in
4    print_string (concatFunction "hello," 10)
```

This example is a simple case of a type error, as the user in this case has a function `concatFunction` which takes two strings and concatenates them together. In line 4 however, the user has mistakenly given `concatFunction` a `string` and an `int` as arguments.

[2]This can be seen in the `enum-candidates-from-v-result-set` and `enum-candidates` functions in the `lambda-candidates.el` file of the implementation.

Figure 3.28 shows the output of `ocamlc` without the use of seminal, and figure 3.29 shows the output with Seminal output enabled.

Figure 3.28 Output of `ocamlc` for code in figure 3.27

```
File "/tmp/test.ml", line 4, characters 40-42:
This expression has type int but is here used with type string
```

Figure 3.29 Seminal output for code in figure 3.27

```
File "/tmp/test.ml", line 4, characters 40-42:
This expression has type int but is here used with type string
Relevant code: 10
```

```
File "/tmp/test.ml", line 2, characters 30-32:
Try replacing
    s2
with
    s1
of type
    string
within context
    let concatFunction s1 s2 = s1 ^ s1 in
        (print_string (concatFunction "hello," 10))
    ;;
==============================
```

From these figures it be seen here that the compiler simply states that there is a type clash between `int` and `string`, on line 4 of the file, between characters 40 and 42. While the figure containing the Seminal output does not attempt to describe to the user precisely *why* this is an error (which is what Skalpel tries to do), it does make an effort to look at other lines and suggest changes that could be made.

In this case, Seminal recommends that in line 2 of the code, `s2` should be replaced by `s1`. This would indeed make the code typable.

There is no support for handling syntax errors in Seminal, so no suggestions are given for syntactic changes that the user might wish to make, and suggesting changes is available for type errors only.

The Seminal patch changes a few of the existing files in places (the Makefile, the configuration script and so on), but the bulk of the implementation comes in the form of a structure called `Seminal`. This is certainly desirable when implementing a system like this as a patch on compiler sources, if as little changes are made to the current compiler sources as possible then the less has to be changed when a new compiler release comes out and the code changes.

The `Seminal` structure can be thought of as a function, with kinds of errors on the left hand side and options to try out to fix the error on the right hand side. The implementation seems to follow very closely what the paper describes. While the implementation for Seminal is not currently active, it is suspected that there would not be many changes needed in the implementation to make this work for the latest version of the Ocaml compiler.

Whether Skalpel would eventually be extended to give the user code suggestions as to what the user might change in order to fix their code is unknown at this time and outside the scope of this research project, but it is certainly an area for potential future work. It is an interesting aspect to consider however, as so much information is generated by Skalpel about the program that the user supplies already as it is.

The Easy OCaml [EAS] implementation tarball can be downloaded from the below URL:

http://forge.ocamlcore.org/frs/download.php/94/easyocaml-0.49.tar.bz2

This tarball contains a patched version of the Ocaml compiler (reported to be version 3.10.2) in order to run EasyOCaml. A webdemo is also available at the following URL at this time of writing:

http://easyocaml.forge.ocamlcore.org/demo.php

The bulk of the implementation files for EasyOCaml can be found in the ocaml-3.10.2/easyocaml/ subdirectory of the downloaded tarball. In here are a number of ML modules. A description of some of the more important ones is given here.

- ezyFeatures.ml. Contains many booleans allowing the user to restrict what feature set EasyOCaml is to handle. The below is a short extract of the file.

```
....
e_if_then : bool;
e_if_then_else : bool;
e_let_in : let_feats option;
e_let_rec_in : letrec_feats option;
...
```

  These flags would be enabled or disabled depending on the requested language feature set that is to be supported as specified by the user.

- ezyAst.ml. Contains the abstract syntax tree.

- ezyConstraints.ml. Somewhat comparable to the Env.sml implementation file present in Skalpel, which states how constraints can be used and manipulated.

Also has some code which implements techniques from [HW03], such as applying a derived environment to a type. A fragment of the code which handles this is given in figure 3.30.

Figure 3.30 Code extract showing application of derived environment to a type

```
let rec apply_to_ty e ty =
  match ty with
    | Ty.Var tyvar ->
        begin try
          let ty', _ = TyVarMap.find tyvar e in
          apply_to_ty e ty'
        with Not_found ->
          ty
        end
    | Ty.Constr (l, k, tys) ->
        Ty.Constr (l, k, List.map (apply_to_ty e) tys)
    | Ty.Tuple (l, tys) ->
        Ty.Tuple (l, List.map (apply_to_ty e) tys)
    | Ty.Arrow (l, tx, tx') ->
        Ty.Arrow (l, apply_to_ty e tx, apply_to_ty e tx')
```

- ezyTypingCoreTypes.ml. Comparable to the Ty.sml file in the Skalpel implementation which contains definitions for how constraints are represented. The type `t` is shown below:

```
type t =
  | Var of TyVar.t
  | Tuple of ExtLocation.t * t list
  | Constr of ExtLocation.t * Path.t * t list
  | Arrow of ExtLocation.t * t * t
```

A TyVar.t is represented with an integer as shown below (some definitions at the bottom have been omitted with '...' characters).

```
module TyVar : sig
  type tyvar
  include PrintableOrderedType with type t = tyvar
  val fresh: unit -> t
  val id: t -> int
  val none: t
end = struct
  type tyvar = int
  type t = tyvar
  let counter = ref 0
  let fresh () = incr counter; !counter
```

```
    ..............
end
```

- ezyTyping. Contains the unification algorithm, unifies constraints as generated by the constraint generation algorithm. An example where two tuples are compared is shown below.

```
(* (...), (...) *)
| Ty.Tuple (_, tys1), Ty.Tuple (_, tys2) ->
begin try
let aux ds_rest ty1 ty2 =
 ConstrSet.add ds_rest (Constr.create ty1 locs ty2)
in let
 new_ds = List.fold_left2 aux ConstrSet.empty tys1 tys2
 in
 (* ..... *)
 unify_full env cs e (ConstrSet.union ds_rest new_ds) l
 with Invalid_argument "List.fold_left2" ->
  Result.Error (EzyErrors.ArityClash (tx', ty',
         List.length tys1, List.length tys2), locs, l)
end
```

Inside the implementation is also some work completed towards some language packs, which is a framework designed to make OCaml easier to learn (and is a motivation for this work as a whole anyway).

3.11 Critique of the Skalpel core

The next chapter demonstrates a number of improvements over the version of the Skalpel core presented here. These are described below.

- Dummy variables are used in this presentation of the core, and care is always needed when using such a concept in our machinery such that dummy variables are never constrained in the set of unifiers. The next chapter demonstrates how the use of existential constraints eliminates that additional machinery.
- The rules handling polymorphism have a greater number of side conditions than any other rule, and in the next chapter it is shown how part of that machinery can be simplified.

- This version of the Skalpel core has the concept of a constraint solving context, but when updated constraint solving contexts are received from recursive constraint solver calls, there are also by design different versions of the unifier set. This is not desirable, as there should only ever be one version of the set of unifiers.
- The build function must be updated with every new extension to the theory, but a better representation of the unifier set would mean that such updates do not need to be presented so laboriously. A new representation is shown in the next chapter.
- The concept of environment difference does not need to exist if environments are handled correctly as a stack. This would mean that again so many external functions need to be called in the constraint solver.
- Monomorphic variable computation requires an external call to a function from the constraint solver. In the next chapter, it is shown how such a call is unnecessary, as the information can be carried in the constraint solver as a parameter.

In the next chapter an updated version of the Skalpel core is shown, and all the stages of the process are investigated which were initially shown in this chapter with the updated version of the theoretical presentation.

Chapter 4

Current Technical Design of Skalpel Core

This chapter discusses the most recent version of the Skalpel core, which includes changes which were made to it during the course of this research project.

Directions are given at the start of each section to readers who have knowledge of the original presentation of the core where the material in that section is unchanged.

Note that lemmas about this presentation are not given here, but occur instead in chapter 8.

As in the original presentation of the core in 3, the system is made up of multiple stages. These stages are repeated below:

- **Labelling**. Firstly, the program that the user submits to Skalpel is annotated with labels, which are used to represent program points and track blame of errors.
- **Constraint generation**. The constraint generator takes a labelled Standard ML program as input and generates a set of constraints. The constraints that are generated have program labels embedded into them, so it is known what parts of the program a certain constraint has been generated for.
- **Enumeration**. The enumerator creates *filters* to run the next phase with, which are effectively program points which should not be considered during constraint solving. Initially, the empty filter set is used, so all labels in the program are considered.
- **Constraint solving**. The constraints constructed during the constraint generation phase are taken and an attempt is made to solve them. In the event

that all constraints can be solved, the program the user submitted is deemed typable, otherwise, the program contains an error.

- **Minimisation**. In this phase an attempt is made to increase the precision of the reported error locations which can be more precise in some cases than the locations initially reported by the constraint solving stage. We will never generate less precise locations during this phase.
- **Slicing**. Creates a description of any located errors known as a *type error slice* (defined in 3.9) from the errors reported by the enumerator.

In our figure titles, we write $x \to y$ to give domain and range information of defined functions, where appropriate (where the domain is x and the range is y).

4.1 External Syntax

The external syntax for the core has not changed since its definition in chapter 3 (the same subset of SML is supported for this core presentation), but it is repeated here for readers unfamiliar with that chapter.

The external syntax that is understood by the Skalpel core is presented (formally called the SML-TES core) in figure 4.1. Many of the forms in this figure are annotated with labels, written l. These labels are created so that blame can be tracked for errors that are generated. In some cases the ⌈ and ⌉ symbols will be used to surround a labelled term (parenthesis are not used as they could be confused with those of the Standard ML language syntax).

In this core, datatypes have exactly one constructor and exactly one type argument. In the implementation however, cases are handled for multiple or nullary constructors and multiple or no type arguments. We can see this from DatName in figure 4.1.

The declarations supported by the core can be seen by looking at Dec in figure 4.1. Here it can be seen there is support for recursive value declarations (functions), open declarations, and datatype declarations. Note that again in the implementation, far more declarations are handled than just this subset that is used in order to present the core.

Figure 4.1 External labelled syntax: The subset of SML that Skalpel handles

external syntax (what the programmer sees, plus labels)

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

extra metavariables

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

4.2 Constraint syntax

The constraint syntax for the Skalpel core is defined in figure 4.1. The syntax presented here is used throughout this document in order to represent constraints, for example in the initial constraint generator where some initial constraints are built (section 4.3) and in the constraint solver (section 4.4).

From the original definition in section 3.4, type schemes have been introduced along with an existential constraint environment. Every binder produced by this syntax is a kind of type scheme. This change brings us closer to removing the dummy variable machinery as all variables are now quantified.

The following sub-sections explain the various parts of this syntax. Note the novel hybrid constraint/environment forms e where binders, accessors and composition environments interact. The motivation is to build environments that avoid duplication at initial constraint generation or during constraint solving. The binders and accessors are also novel. Earlier systems (e.g.
([DPR05])) are too restrictive to represent module systems because they only support very limited cases of binders. With the constraints, a compositional constraint

generation algorithm can be defined.

During analysis, a dependent form $\langle \mathcal{C}^L, \bar{l} \rangle$ depends on the program nodes with labels in $\bar{l}$. For example, the dependent equality constraint $\langle \tau_1 = \tau_2, \bar{l} \cup \{l\} \rangle$ might be generated for the labelled function application $\lceil exp\ atexp \rceil^l$, indicating the equality constraint $\tau_1 = \tau_2$ need only be true if node l has not been sliced out. In order to manipulate labels, two functions are defined strip and collapse below, which respectively allow us to take all labels off any given term, and to union nested labels of terms. Note that dom(strip) = dom(collapse) = IntLabSynt, and ran(strip) is any piece of syntax which is not a dependent form, while ran(collapse) = IntLabSynt.

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

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

Note that $\langle ct, l \rangle$ can be written for $\langle ct, \{l\} \rangle$. Given a label or a set of labels y, ct^y is written to abbreviate $\langle ct, y \rangle$, and $ct_1 \stackrel{y}{=} ct_2$ for $\langle ct_1 = ct_2, y \rangle$.

4.2.1 Internal types (τ) and their constructors (μ)

The ITy and ITyCon sets contain internal types and internal type constructors respectively. In order to maintain some simplicity for the core, only unary type constructors are supported[1]. A special kind of type constructor `arr` exists, which is used to create a constraint in the constraint solving process (rule (S5)) between a unary type constructor and an arrow ($\rightarrow$) type.

4.2.2 Quantification and Schemes (σ)

In this presentation of the Skalpel core, there is no longer the concept of dummy variables and instead all variables are quantified. This has the advantage of removing the machinery surrounding freshness that was present in the original presentation of the core. There is no longer any use for the Dum set as all variables are now quantified, and so it has been removed in this chapter. This means that dummy internal type variables, dummy environment variables, and dummy type constructor variables no longer exist.

There are three kinds of universally quantified schemes in this system - internal

[1] Section 14.10 in ([Rah10]) presents a solution whereby type constructors can have any arity.

Figure 4.2 Syntax of constraint terms

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

type schemes, type constructor schemes and environment schemes. All schemes are subject to alpha-conversion (e.g. the schemes $\forall\alpha_1.\,\alpha_1$ and $\forall\alpha_2.\,\alpha_2$ are equivalent).

By not having dummy variables any longer, the issue of erroneously creating accidental bindings to dummy variables and checking for dummy variables in conditions of e.g. monomorphic variable computation (further discussion of this given in section 4.6).

4.2.3 The constraint/environment form (e)

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

- **The empty environment / satisfied constraint**. This is represented by the symbol $\top$.
- **An environment variable**. To abbreviate, $[e]$ is written instead of $(\exists ev.ev = e)$, where ev does not occur in e. This is a constraint which enforces the logical constraint nature of e while limiting the scope of its bindings. Note that the bindings can still have an effect if e constrains an environment variable. This used to be a dummy variable assignment in the old presentation of the core.

- **A composition environment**. The operator ';' is used to compose environments, which is associative. Note that $e;\top$, $\top;e$, and e are considered to be equivalent.

- **A binder/accessor**. A binder is of the form $\downarrow id{=}\sigma$, and an accessor is of the form $\uparrow id{=}v$. Binders represent program occurrences of an identifier id that are being bound, and accessors represent a place where that binding is used. For example, in the environment

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

  the internal type variable α is constrained through the binding of vid to be an instance of x. It is often also the case that binders and accessors can be connected without being next to each other. In an environment such as

  $$\downarrow vid{=}x;...;\uparrow vid{=}\alpha$$

  it is *possible* that the binder and accessor of vid are connected. There are some environment forms that can be in the omitted (...) section which will mean that the accessor and the binder will be disconnected. Section 4.2.6 on *shadowing* specifies which forms would cause this.

  To abbreviate, $\downarrow vid{=}ct$ is written instead of $\downarrow vid{=}\forall\varnothing.\, ct$, and $\downarrow vid \stackrel{y}{=} ct$ to abbreviate $\langle\downarrow vid{=}ct, y\rangle$ (similarly for accessors).

- **An equality constraint**. A constraint which creates the condition of two pieces of constraint syntax that they should be in some way equal.

- **Existential environment**. The form $\exists v.e$, binds all free occurrences of v that occur free in e. The notation $\exists\langle v_1, v_2, v_3.\cdots, v_n\rangle.e$ is used to abbreviate $\exists v_1.\exists v_2.\exists v_3.\cdots\exists v_n.e$.

- **A polymorphic environment** $\texttt{poly}(e)$. This environment promotes binders in the argument to `poly` to be polymorphic.

- **Dependent form**. An environment annotated with dependencies acts like an environment only when the dependencies are satisfied.

4.2.4 Syntactic forms

Defined here is $\mathsf{atoms}(\mathcal{C}^L)$, which is a syntactic form set belonging to $\mathsf{Var} \cup \mathsf{Label} \cup \mathsf{Dependency}$ and occurring in $\mathcal{C}^L$.

Functions are defined to extract variables, labels, and dependencies from a constraint term in figure 4.3.

Figure 4.3 Definition of sets of variables, labels and dependencies

$$
\begin{array}{lcl}
\mathsf{vars} & : & \mathcal{C}^L \to \mathbb{P}(\mathsf{Var}) \\
\mathsf{vars}(\mathcal{C}^L) & = & \mathsf{atoms}(\mathcal{C}^L) \cap \mathsf{Var} \text{ (set of variables)} \\
 & & \\
\mathsf{labs} & : & \mathcal{C}^L \to \mathbb{P}(\mathsf{Label}) \\
\mathsf{labs}(\mathcal{C}^L) & = & \mathsf{atoms}(\mathcal{C}^L) \cap \mathsf{Label} \text{ (set of labels)} \\
 & & \\
\mathsf{deps} & : & \mathcal{C}^L \to \mathbb{P}(\mathsf{Dependency}) \\
\mathsf{deps}(\mathcal{C}^L) & = & \mathsf{atoms}(\mathcal{C}^L) \cap \mathsf{Dependency} \text{ (set of dependencies)}
\end{array}
$$

4.2.5 Semantics of constraints/environments

The set of renamings and substitutions are defined in figure 4.4. Note that $\mathsf{Ren} \subset \mathsf{Sub}$. Renamings are used to instantiate type schemes, and substitution is defined in figure 4.5, where given a constraint term and a substitution, a resulting constraint term is produced.

Figure 4.4 Renaming and substitutions

$$
\begin{array}{lcll}
\mathsf{Ren} & : & \mathsf{ITyVar} \to \mathsf{ITyVar} & \\
\mathit{ren} & \in & \mathsf{Ren} & = \{\mathit{ren} \in \mathsf{ITyVar} \to \mathsf{ITyVar} \mid \mathit{ren} \text{ is injective} \\
 & & & \quad \wedge\ \mathsf{dj}(\mathsf{dom}(\mathit{ren}), \mathsf{ran}(\mathit{ren}))\} \\
 & & & \\
\mathsf{Sub} & : & \mathsf{SubstTerm} \to \mathsf{SubstTerm} & \\
\mathit{sub} & \in & \mathsf{Sub} & = \{f_1 \cup f_2 \mid f_1 \in \mathsf{Unifier} \wedge f_2 \in \mathsf{TyConName} \to \mathsf{TyConName}\}
\end{array}
$$

The unifier set is defined as a directed acyclic graph as follows, where $\mathsf{V} = \mathsf{ITyVar} \cup \mathsf{ITy} \cup \mathsf{ITyCon}$ and $\mathsf{E} = \mathbb{P}(\mathsf{V} \times \mathsf{V})$ which specify directional edges:

$$\mathsf{Unifier} : \mathsf{ITyVar} \to \mathsf{ITyVar}$$

$$\mathcal{U} \in \mathsf{Unifier} = \{\mathsf{V}, \mathsf{E}\}$$

Note that for each $V_x \in \mathsf{V}$, the edge $V_x \mapsto V_x'$ occurs at most once, and so $\mathcal{U}$ is also considered as a function. When using an application $\mathcal{U}(V_x)$, vertex V_x' will be returned where a path from V_x to V_x' exists (if it does not, $V_x = V_x'$) and $V_x' \mapsto V_x''$ does not exist. For example, where $\mathcal{U} = \{\{V_1, V_2, V_3, V_4, V_5, V_6\}, \{V_1 \mapsto V_3, V_3 \mapsto V_2, V_4 \mapsto V_5, V_2 \mapsto V_6\}\}, \mathcal{U}(V_1) = V_6$. During application, if $\mathcal{U}(v) = {\mathcal{C}^L}_x$ and $\mathsf{vars}(\mathcal{C}^L) \neq \{\}$, then for each $v' \in \mathsf{vars}(\mathcal{C}^L)$ if $\mathcal{U}(v') \neq v'$ then it is replaced by $\mathcal{U}(v')$.

Figure 4.5 Substitution semantics

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

4.2.6 Shadowing

In an environment it may be the case that some parts are shadowed and so inaccessible. Consider the environment $bind_1; ev; bind_2$. In the event that $ev \notin \mathsf{dom}(\mathcal{U})$, it can also be said that ev *shadows* $bind_1$ because ev could potentially be bound to an environment which rebinds $bind_1$. The predicate $\mathsf{shadowsAll}$ is defined in figure 4.6. We write $\mathsf{shadowsAll}(e)$ for $\mathsf{shadowsAll}(\langle \emptyset,\ e \rangle)$.

Figure 4.6 Shadowing function : $\mathsf{Unifier} \times \mathsf{Env} \to \{true, false\}$

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

Figure 4.7 shown the semantics of accessing an identifier in an environment , in the context where there is access to a unifier set $\mathcal{U}$ during constraint solving.

Figure 4.7 Accessing the Semantics

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

Note that the application of an existential environment to an identifier is undefined as such an environment represents incomplete information.

4.2.7 Relations

Two instance relations are defined here, the use of which can be seen in constraint solving. These have been updated from the definition in the original presentation of the core to deal with the new use of type schemes and existential quantification of terms.

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

4.3 Constraint generation

The *initial constraint generator* is defined in figure 4.17. This is referred to as the *initial* constraint generator because constraints are also generated during the constraint solving process (section 4.4).

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

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

Structure declarations are handled in rule (G20). In the core, signatures to constrain these structures are not presented, but this extension to the core can be found in section 6.4.

In order that environments can be sliced out correctly, environment variables are annotated with labels, such as in rule (G4). Such environment variables must be annotated with a label otherwise it could not be sliced out, and that environment variable would then shadow any following environment, even if the program point the label was assigned to was itself sliced out.

Readers should see chapter 8 for lemmas and proofs concerning this constraint generator. Each of the constraint generation rules are now discussed in turn.

4.3.1 Expressions

Figure 4.8 Initial constraint generator (ExtLabSynt → Env)

Expressions (*exp*)

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

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

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

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

Rule (G1) generates accessors for value identifiers. For example, where a function previously defined is applied to an argument, the accessor which connects the use of this function to its definition is generated in this rule.

Rule (G2) handles let expressions, where some declarations are to be defined in the scope of some expression. We accomplish this with the $[e]$ notation in this rule - by encasing the environments from the declaration inside square brackets, upon closing of these brackets that environment is not exported (as described previously) and so binders inside are not available to following expressions.

Applications are handled with rule (G3). Here the $\to$ piece of constraint syntax is used to represent the expression part being used as a function.

Nameless functions are handled in rule (G4). The environment variable is labelled in this rule (and similarly in other rules) in order that declarations which have been sliced out do not shadow their context. If the environment variable was not labelled, then it would shadow the context it was in irrespective if some environment was sliced out or not.

4.3.2 Labelled datatype constructors

Figure 4.9 Initial constraint generator (ExtLabSynt → Env)

Labelled datatype constructors (*ldcon*)

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

Labelled datatype constructors are handled in rule (G5). With this rule, an accessor is created to a datatype constructor in the same way as in (G1). The way datatype constructors and value identifiers are differentiated is enhanced in section 14.1 of [Rah10].

4.3.3 Patterns

Figure 4.10 Initial constraint generator (ExtLabSynt → Env)

Patterns (*pat*)

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

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

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

Rule (G6) creates bindings for value variables occurring in patterns (such as in function declarations), while rule (G7) creates accessors to datatype constructors occurring in patterns. Rule (G8) handles the use of datatype constructors in patterns which have an argument.

4.3.4 Labelled type constructors

Figure 4.11 Initial constraint generator (ExtLabSynt → Env)

Labelled type constructors (*ltc*)

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

Labelled type constructors can occur in rule (G9). An accessor is created for this type constructor. This constraint can be generated for example in datatype constructor bindings.

4.3.5 Types

Figure 4.12 Initial constraint generator (ExtLabSynt → Env)

Types (*ty*)

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

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

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

Rule (G10) handles the case where an external type variable is being dealt with which occurs in a constructor binding. As a result, an accessor is generated (this should be connected during solving to the binder occurring in the declaration of the datatype which declares the constructor in which this explicit type variable

is used). Rule (G11) is for labelled datatype constructors occurring in definitions of datatype constructors, and rule (G12) handles the case where an arrow type is specified in the user program.

4.3.6 Datatype names

Figure 4.13 Initial constraint generator (ExtLabSynt → Env)

Datatype names (dn)

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

Datatype names are handled with rule (G13). By looking at this rule it can be seen that datatype declarations have exactly one explicit type variable argument. Binders are created for both the name of the datatype and for the specified type variable argument in this rule.

4.3.7 Constructor bindings

Figure 4.14 Initial constraint generator (ExtLabSynt → Env)

Constructor bindings (cb)

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

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

Rules (G14) and (G16) give support for datatype constructor bindings to the constraint generator. (G14) is for a constructor which doesn't take an argument and rule (G16) is for constructors defined with the `of` keyword, where the type of the argument for the datatype constructor is defined. In both cases, binders are created for the name of the constructor.

4.3.8 Declarations

Figure 4.15 Initial constraint generator (ExtLabSynt → Env)

Declarations (*dec*)

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

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

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

Functions are supported in the constraint generator with rule (G17). Function declarations are not supported using the `fun` keyword in this presentation of the core but the implementation does handle this, which is simply a syntactic variation. In this rule, as in rule (G18), the novel `poly` environments are used to make function bindings and datatype declarations polymorphic. The `open` feature is handled with rule (G19), where an accessor constraint using the structure identifier is created.

4.3.9 Structure declarations

Figure 4.16 Initial constraint generator (ExtLabSynt → Env)

Structure declarations (*strdec*)

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

Rule (G20) handles structure declarations. The environment generated for that structure is wrapped with $[e]$ to limit the scope of bindings occurring in that structure.

4.3.10 Structure expressions

Figure 4.17 Initial constraint generator (ExtLabSynt → Env)

Structure expressions (*strexp*)

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

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

Structure expressions are handled in rules (G21), which handles the case where the structure expression is some identifier which an accessor is created for, and (G22) in

the case of a `struct` expression, in which case environments are generated for each structure declaration and compose the results using the environment composition operator (`;`).

4.4 Constraint solving

This section discusses the constraint solver, which takes constraints which were generated during the previous section and terminates in either one of two states:

- A success state `succ`.
- A failure state `err`(er). In this case an error is returned of the one of the kinds discussed in section 4.18.

Readers familiar with the earlier presentation of the Skalpel core in chapter 3 will notice that there is no longer a constraint solving context returned by the constraint solver (Δ) when it terminates in success. The reason for this simplification is that the set of unifiers is no longer bundled along with an environment (written $\langle u, e \rangle$) as firstly the set of unifiers is now considered a global entity of the constraint solver (discussed further in section 4.4.1), and secondly because of the new stack mechanism to replace what used to be multiple constraint solving calls in the same rule. The stack mechanism is discussed in section 4.6.3.

Let the set of monomorphic type variables be defined as follows:

$$m \in \mathsf{Monomorphic} ::= \langle \alpha, \overline{d} \rangle$$

Figure 4.18 Additional syntactic forms used by constraint solver

er	$\in$	Error	$::=$	$\langle ek, \overline{d} \rangle$
ek	$\in$	ErrKind	$::=$	`clash`(μ_1, μ_2) | `circularity`
$state$	$\in$	State	$::=$	`slv`$(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e')$ | `succ` | `err`(er)

Additional syntactic forms that are used by the constraint solver (defined in figure 4.19) are shown in figure 4.18. The symbol $\overrightarrow{st}$ is defined in section 4.6.3.

The constraint solving process starts in the form `slv`$(\langle \top \rangle, \varnothing, \varnothing, \langle \rangle, e)$, and ends either in the form `succ`, which indicates success, or in the state `err`(er) where er is either a type constructor clash or a circularity error (figure 4.18).

The relations isErr and solvable are defined below, where $\rightarrow$ indicates a constraint solving step and $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. The Solved is also redefined as shown below.

$$solvT \in \mathsf{Solved} ::= \mathsf{Env} \cup \mathsf{StrDec}$$

$$\begin{array}{lcl}
\mathsf{isErr} & : & \mathsf{Env} \to \mathsf{Error} \\
e \overset{\mathsf{isErr}}{\to} er & \Leftrightarrow & \mathtt{slv}(\top, \overline{d}, \varnothing, \varnothing, e) \to^* \mathsf{err}(er)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{solvable} & : & \mathsf{State} \to \{true, false\} \\
\mathsf{solvable}(e) & \Leftrightarrow & \mathtt{slv}(\top, \overline{d}, \varnothing, \varnothing, e) \to^* \mathtt{succ}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{solvable} & : & \mathsf{Solved} \to \{true, false\} \\
\mathsf{solvable}(strdec) & \Leftrightarrow & \exists e.strdec \to e \wedge \ \mathsf{solvable}(e)
\end{array}$$

4.4.1 Unifiers

In this constraint solving presentation (figure 4.19), unifiers are stored in $\mathcal{U}$.

When constraint solving is started, $\mathcal{U} = \varnothing$. During the constraint solving process, nothing is ever subtracted from $\mathcal{U}$, this set is only added to. The set of unifiers is available for use by the constraint solver, and any associated functions called by it.

While $\mathcal{U}$ is a function, it can be considered as a special kind of directed acyclic graph {V,E}, where V are vertices and E are edges, and where for each vertex V_x, the edge $V_x \mapsto V'_x$ occurs at most once. When using an application $\mathcal{U}(V_x)$, vertex V'_x will be returned where a path from V_x to V'_x exists (if it does not, $V_x = V'_x$) and $V'_x \mapsto V''_x$ does not exist. For example, where $\mathcal{U} = \{\{V_1, V_2, V_3, V_4, V_5, V_6\}, \{V_1 \mapsto V_3, V_3 \mapsto V_2, V_4 \mapsto V_5, V_2 \mapsto V_6\}\}, \mathcal{U}(V_1) = V_6$. In the case where the term applied to is an arrow type $V_1 \to V_2$, this is computed as $\mathcal{U}(V_1) \to \mathcal{U}(V_2)$ (and similarly for an application).

The unifier set is defined in this way now so that it not hidden inside another constraint solving form, which led to awkward expressions in the old presentation where the Δ had to be broken down into both its components in order to use each of them, and when there were new and old versions of Δ (e.g. Δ and Δ'), there then exists u and u', u would never be used and always use u'. For these reasons it was decided to break up the coupling of unifier sets with environment information.

Figure 4.19 Constraint solver (1 of 2) : State → State

`equality constraint reversing`

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

`equality simplification`

(S1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ct = ct) \rightarrow \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st})$

(S2) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ct^{\overline{d}'} = ct') \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d} \cup \overline{d}', \overline{m}, \overrightarrow{st}, ct = ct')$

(S3) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1\ \mu_1 = \tau_2\ \mu_2) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, (\mu_1 = \mu_2);(\tau_1 = \tau_2))$

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

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

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

`unifier access`

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

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

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

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

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

(U6) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, v = ct) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, z = ct)$,
if $\mathcal{U}(v) = z$

`binders/empty/dependent/variables`

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

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

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

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

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

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

`composition environments`

(C1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e_1;e_2) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st} @ \langle \langle \texttt{new}, \overline{d}, \texttt{new}, e_2 \rangle \rangle, e_1)$

Figure 4.20 Constraint solver (2 of 2) : State → State

`accessors`

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

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

`polymorphic environments`

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

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

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

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

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

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

4.5 Discussion: Unifier representation

In the previous version of the core the set of unifiers were bundled along with an environment in a *constraint solving context*. This complicated the constraint solving rules, as this argument to the constraint solver was not atomic in nature.

It did not make sense to pass around the constraint solver a copy of the unifier set, which we then kept track of multiple copies of during the constraint solving rules for environment difference, when the unifier set is only ever added to and we do not need to keep track of older (smaller) versions. We do not care about any version of the unifier set other than the very latest version. Furthermore, it meant that any constraint solving rule which needed to access portions inside the constraint solving context first needed to decompose it and we wanted to be free of this complication so that our theory could be presented as clearly as possible. By having the set of unifiers considered to be a global entity that is available to the constraint solving rules and any functions called by it, we remove the need for decomposition of constraint solving contexts and indeed constraint solving contexts

Figure 4.21 Monomorphic to polymorphic environment

$$
\begin{array}{lcll}
\mathsf{toPoly}(\Delta, {\downarrow}vid{=}\tau) & = & ({\downarrow}vid \stackrel{\overline{d}}{=} \forall\overline{\alpha}.\,\tau'), & \text{if } \left\{ \begin{array}{lcl} \tau' & = & \mathsf{build}(\Delta,\tau) \\ \overline{\alpha} & = & (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus \\ & & (\mathsf{vars}(\mathsf{monos}(\Delta)) \cup \{\alpha_{\mathtt{dum}}\}) \\ \overline{d} & = & \{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\Delta) \wedge \\ & & \alpha \in \mathsf{vars}(\tau') \setminus \overline{\alpha}\} \end{array} \right. \\
\mathsf{toPoly}(\Delta, e_0^{\overline{d}}) & = & e_1^{\overline{d}}, & \text{if } \mathsf{toPoly}(\Delta, e_0) = e_1 \\
\mathsf{toPoly}(\Delta, e_1{;}e_2) & = & e_2', & \text{if } \mathsf{toPoly}(\Delta, e_1) = e_1' \text{ and} \\
 & & & \mathsf{toPoly}(\Delta{;}e_1', e_2) = e_2' \\
\mathsf{toPoly}(\Delta, e) & = & e, & \text{if none of the above applies}
\end{array}
$$

themselves (instead they are replaced by the only remaining environment portion).

Furthermore, the set of unifiers is considered a graph as described in the new presentation of the core in order to remove the complicated `build` machinery.

4.6 Discussion: Computation of monomorphic variables and changes to rules for polymorphism

In the previous version of the core given in [Rah10], we had the following constraint solving rule for polymorphism:

(P1) $\mathtt{slv}(\langle u_1, e_1\rangle, \overline{d}, \mathtt{poly}(e)) \rightarrow \mathtt{succ}(\mathsf{toPoly}(\langle u_2, e_1\rangle, e_1 \backslash e_2))$,
if $\mathtt{slv}(\langle u_1, e_1\rangle, \overline{d}, e) \rightarrow {}^*\mathtt{succ}(\langle u_2, e_2\rangle)$

The environment difference is not used any longer in the new presentation due to reasons discussed previously, so we do not discuss that here and instead look at the definition of toPoly which is given again in figure 4.21.

Firstly, we need a new constraint solving rule in this version of the core to deal with existential constraints. The new rule (P6) deals with this situation.

Secondly, the way we compute monomorphic variables has changed. In [Rah10], we calculate which variables to quantify over with the following side condition:

$$\overline{\alpha} = (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\Delta)) \cup \{\alpha_{\mathtt{dum}}\})$$

The definition of the monos function is also repeated below:

$$\mathsf{monos}(\Delta) = \{\alpha^{\mathsf{deps}(\tau)} \mid \exists vid.\tau = \mathsf{build}(\Delta, \Delta(vid)) \wedge \alpha \in \mathsf{nonDums}(\tau)\}$$

Firstly, it was noticed that the monos function returns a set of internal type variables

annotated with dependencies. However, the elements of the set resulting in the computation on the left hand side of the set subtraction in the computation of $\overline{\alpha}$ in the toPoly function can only contain variables. It is therefore the case that these labels are thrown away, and after some analysis it can be seen that they do not need to be computed, as the set of dependencies computed by another side condition of toPoly are sufficient.

Secondly, it was verified that the only place where we generate α variables which are on the right hand side of binders is in rule (B) (figure 4.19). With this in mind, we simply add these internal type variables to a parameter of the constraint solver, to save complexity while solving rules supporting polymorphism.

It would not however be acceptable to subtract just these internal type variables from the internal type variables in the type we are to make polymorphic (τ' in figure 4.21), as we first need to check further constraints involving these variables were made and added to the set of unifiers. To solve this, we look up each of the internal type variables that we keep track of in the constraint solver in the set of unifiers, then take those away from the internal type variables of the internal type variables in the result of looking up the right hand side of the binder to be made polymorphic.

Notice also that instead of matching $\downarrow vid{=}\tau$ as the environment in the associated rule for dealing with polymorphism, we now match the environment $\downarrow vid{=}\alpha$. We do this in order to be more precise and simplify the constraint solving rules, as we only ever generate binders with internal type variables. (The definition in [Rah10] is still valid, as internal types can be internal type variables as seen in figure 4.2).

Note also that additional rules have been added into the Skalpel core constraint solver for forms such as environment composition inside a polymorphic environment. This is a bug in [Rah10] where polymorphism on some generated environments could be undefined, which is fixed with these new rules.

4.6.1 The Environment Tuple Parameter of the Constraint Solver

In this presentation of the constraint solving mechanism, the first argument is now a tuple. The reason for this is to remove the mechanism behind environment difference which was present in the initial presentation of the core. This environment difference is present inside the original constraint solver, causing additional complications to understanding the solver.

With this new way to handle different scopes, the environment argument is a tuple, and to enter a new scope, in order to deal with e.g. bindings assigned to an environment variable, a new tuple element is simply created. When resulting bindings are to be mapped to an environment variable, a mapping is added from the environment variable to the bindings in the last element of the tuple, then join the last two elements of $\overrightarrow{e}$ together with the environment composition operator (;).

4.6.2 Discussion: The Environment Tuple

Instead of having a single environment e in the first argument to the constraint solving rules, we now have a tuple of environments. The reason for this change was to remove the notion of environment difference, which complicated the rules of constraint solving.

It was previously the case that we needed to keep track of two environments, the one which resulted after solving some constraint term(s), and the one which existed prior to when this solving took place. We would then take the *difference* of these environments in order to see what had changed in the environment. As this environment was bundled inside a constraint solving context, we actually needed to keep track of two constraint solving contexts.

This is no longer computed, as this information is now available in the constraint solver immediately. With the environment being represented as a tuple, the new portion of the environment which resulted in some constraint term(s) being solved is the last element of the environment tuple, with the initial environment before solving being in the second-last environment tuple element. This makes it easier to map the new portion of the environment which exists to a value in the set of unifiers - we create a mapping from some variable to whatever is in the last element of the environment tuple rather than computing it first using environment difference that was present in the old version of the core.

Another reason for removing such a complexity is that we needed to use environment difference when handling rules involved with polymorphic calculations. As these rules in the constraint solver are by far the most complicated, finding a way to simplify this process is a necessary and valuable goal.

4.6.3 The stack parameter

The fourth argument to the `slv` function of the constraint solver, denoted as $\overrightarrow{st}$, is used as a stack of environments or other tasks which are still to be solved/completed

in addition to the environment that is in the last parameter of `slv`, which is the environment currently being solved.

Figure 4.22 Definitions for the environment stack

stackEv	$\in$	StackEv	$=$	$e \mid \texttt{new}$
stackMono	$\in$	StackMono	$=$	$\overline{m} \mid \texttt{new}$
stackAction	$\in$	StackAction	$=$	$e \mid v \mid \overline{d} \mid \texttt{done}$

This stack is a tuple where each element is itself a tuple which has four components. Figure 4.22 defines some new metavariables used in the description of each below:

1. *stackEv*. This is used to represent which environment should be used when taking action on the *stackAction* parameter. This can either be the symbol `new`, in which case the environment of the constraint solver can be used when the `isSucc` function was called which deals with handling stack items, or instead it can be a specified environment e, in which case the environment pushed to the stack at the time when this stack item was created is used.
2. $\overline{d}$. A set of dependencies.
3. *stackMono*. Same as *stackEv*, but with monomorphic variable sets instead of environments.
4. *stackAction*. What this argument contains affects which operation should be performed. What is done in each case of *stackAction* can be seen in the definition of `isSucc`$'$ in figure 4.23.

When solving the environment in the last position of the `slv` argument tuple is completed, `isSucc` is called which solves the argument at the top of $\overrightarrow{st}$ stack, otherwise if the stack is empty the constraint solver terminates in the success state.

The definition of `isSucc` is seen in figure 4.23, where given an tuple of environments, a set of monomorphic variables and a stack of remaining environments still to process, will either recurse, return the constraint solver success state, or run the constraint solver on some environment.

The reason for using this new stack mechanism is to reduce the complexity needed in the constraint solver when declaring rules. For example, in the extension on local declarations in section 6.1, it can be seen that only one rule is used where in the old presentation (section 6.1), it was necessary to use three rules to give the same semantics. To give an example in the Skalpel core, in the old presentation of the core (figure 3.17), there are rules (C1) and (C2) for handling composition environments, but only one is needed in the new presentation in figure 4.19.

Figure 4.23 Definition of `isSucc` and `isSucc′`

$\texttt{isSucc} : \mathsf{tuple}(\mathsf{Env}) \times \mathsf{Monomorphic} \times \mathsf{tuple}(\mathsf{StackEv} \times \mathsf{Dependency} \times \mathsf{StackMono} \times \mathsf{StackAction}) \rightarrow \mathsf{ran}(\texttt{isSucc}')$

$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \langle\rangle) \rightarrow \texttt{succ}$
$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}@\langle\langle \texttt{new}, \overline{d}, \texttt{new}, x\rangle\rangle) \rightarrow \texttt{isSucc}'(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, x)$
$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}@\langle\langle \overrightarrow{e}_1, \overline{d}, \texttt{new}, x\rangle\rangle) \rightarrow \texttt{isSucc}'(\overrightarrow{e}_1, \overline{d}, \overline{m}, \overrightarrow{st}, x)$
$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}@\langle\langle \texttt{new}, \overline{d}, \overline{m}', x\rangle\rangle) \rightarrow \texttt{isSucc}'(\overrightarrow{e}, \overline{d}, \overline{m}', \overrightarrow{st}, x)$
$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}@\langle\langle \overrightarrow{e}_1, \overline{d}, \overline{m}', x\rangle\rangle) \rightarrow \texttt{isSucc}'(\overrightarrow{e}_1, \overline{d}, \overline{m}', \overrightarrow{st}, x)$

$\texttt{isSucc}' : \mathsf{Env} \times \mathsf{Dependency} \times \mathsf{Monomorphic} \times \mathsf{tuple}(\mathsf{StackEv} \times \mathsf{Dependency} \times \mathsf{StackMono} \times \mathsf{StackAction}) \times \mathsf{StackAction} \rightarrow \mathsf{State} \backslash \texttt{err}(er)$

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

The advantages of this new representation are even clearer when looking at [Rah10] in the extension of the old version of the core to support functors. These rules have a great number of side conditions attached to them, and these are duplicated as both success and error cases need to be handled which might arise. By handling both cases in a more linear style with this stack, the need for additional constraint solving rules to handle error cases is bypassed. The stack also makes the constraint solving rules much easier to follow; with the old presentation it was necessary to know whether any future environments to be solved solve resulted in success or not in order to know what to do next and this meant the reader needed to mentally maintain the success status of any future environment to be solved, jumping back to rules where necessary. This is no longer the case with this new stack representation.

Discussion: The Stack

The recent version of the Skalpel core presented in this thesis uses a stack in order to hold information on which tasks are still to be completed by the constraint solver. For example it was impossible to know which rule to use in some cases without looking ahead.

As an example, consider ten environments all composed with one another using the '`;`' operator, all of which can be arbitrarily complicated. It was previously the case that for each of these we would not know whether to use rule (C1) or (C2) of the constraint solver, as these rules required knowledge of whether solving the first environment (which the constraint solver was run on in a recursive manner) was

solved successfully, or whether it resulted in an error.

We can see how a reduction in the number of constraint solving rules can be made by looking at the constraint solving rules involving the theory for functors from [Rah10] is given below.

Consider the constraint solving rules for functors given in figures 4.24 and 4.25.

Figure 4.24 Constraint solving rules for functors (1 of 2)

subtyping constraints

(SU1) $\mathtt{slv}(\Delta, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2) \rightarrow \mathtt{succ}\langle u_1 \oplus u_2 \oplus u_3,\ e'; \downarrow vid \stackrel{\overline{d}}{=} \forall \overline{\rho}.\ \overline{cap}'_1 \diamond \tau \rangle,$
if $\sigma'_1 = \mathsf{head}(\sigma_1) \wedge \sigma'_2 = \sigma_2$
$\wedge\ \forall i \in \{1,2\}.(\sigma'_i = \forall \overline{\rho}_i.\ \overline{cap}_i \diamond \tau_i$
or $(\sigma'_i = \tau_i$ and $\overline{\rho}_i = \overline{cap}_i = \emptyset$ and $\tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\} \wedge \mathsf{dj}(\mathsf{vars}(\Delta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2), \{sv'\})$
$\wedge\ \mathtt{slv}(\Delta, \overline{d}, \tau_1[ren_1] = \tau_2[ren_2]) \rightarrow^* \mathtt{succ}\langle u_1,\ e' \rangle$
$\wedge\ \tau = \mathsf{build}(u_1, \tau'_2[ren_2])$
$\wedge\ \overline{\rho} = (\overline{\rho}_1[ren_1] \cup \overline{\rho}_2[ren_2]) \cap \mathsf{svars}(\tau)$
$\wedge\ \langle \langle u_1,\ e' \rangle, \overline{d}, \overline{cap}_1[ren_1] \rangle \xrightarrow{\text{duplicate}} u_2 \wedge sv' \notin \mathsf{vars}(u_2)$
$\wedge$ (if $\mathsf{tail}(\sigma_1, u_1 \oplus u_2) = sv$ then $u_3 = \{sv \mapsto \tau \cap sv'\} \wedge \overline{cap} = \{\langle \tau, sv' \rangle\}$ else $u_3 = \overline{cap} = \emptyset$)
$\wedge\ \overline{cap}'_1 = \overline{cap} \cup \{\langle \tau'_0, sv_0 \rangle \mid \langle \tau_0, sv_0 \rangle \in \overline{cap}_1[ren_1] \wedge \tau'_0 = \mathsf{build}(u_1, \tau_0) \wedge \neg \mathsf{dja}(\mathsf{vars}(\tau'_0), \overline{\rho})\}$

(SU2) $\mathtt{slv}(\Delta, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2) \rightarrow \mathtt{err}(er),$
if $\sigma'_1 = \mathsf{head}(\sigma_1) \wedge \sigma'_2 = \sigma_2$
$\wedge\ \forall i \in \{1,2\}.(\sigma'_i = \forall \overline{\rho}_i.\ \overline{cap}_i \diamond \tau_i$ or $(\sigma'_i = \tau_i$ and $\overline{\rho}_i = \overline{cap}_i = \emptyset$ and $\tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\} \wedge \mathsf{dj}(\mathsf{vars}(\Delta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\Delta, \overline{d}, \tau_1[ren_1] = \tau_2[ren_2]) \rightarrow^* \mathtt{err}(er)$

(SU6) $\mathtt{slv}(\langle u,\ e \rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2 \cap \sigma_3) \rightarrow \mathtt{slv}(\langle u',\ e \rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2),$
if $\mathtt{slv}(\langle u,\ e \rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_3) \rightarrow^* \mathtt{succ}\langle u',\ e' \rangle$

(SU7) $\mathtt{slv}(\langle u,\ e \rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2 \cap \sigma_3) \rightarrow \mathtt{err}(er),$
if $\mathtt{slv}(\langle u,\ e \rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_3) \rightarrow^* \mathtt{err}(er)$

Figure 4.25 Constraint solving rules for functors (2 of 2)

`binders`

(B) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, {\downarrow}id{=}x) \rightarrow \texttt{succ}\langle u,\ e\rangle;({\downarrow}id \stackrel{\overline{d}}{=} x)$,
if $id \notin \mathsf{FunId} \cup \mathsf{SigId} \cup \mathsf{TyCon}$

(B8) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, {\downarrow}funid{=}fctsem) \rightarrow \texttt{succ}\langle u,\ e\rangle;({\downarrow}funid \stackrel{\overline{d}}{=} fctsem')$,
if $\langle \mathsf{build}(u, fctsem), \langle u,\ e\rangle\rangle \xrightarrow{\mathsf{abstract}} fctsem'$

`accessors`

(A1) $\texttt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \texttt{slv}(\Delta, \overline{d} \cup \overline{d}', v = x[ren])$,
if $\Delta(id) = (\forall \overline{svar}.\ x)^{\overline{d}'} \wedge \mathsf{dom}(ren) = \overline{svar} \wedge \mathsf{dj}(\mathsf{vars}(\langle \Delta, v\rangle), \mathsf{ran}(ren))$
$\wedge\ id \notin \mathsf{VId}$

(A2) $\texttt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \texttt{slv}(\Delta, \overline{d}, v = x)$,
if $\Delta(id) = x \wedge id \in \mathsf{StrId} \cup \mathsf{TyVar}$

(A5) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, {\uparrow}vid{=}\alpha) \rightarrow \texttt{succ}\langle u',\ e\rangle$,
if $\Delta(vid) = \sigma \wedge \texttt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma \preceq_{vid} \alpha) \rightarrow^* \texttt{succ}\langle u',\ e'\rangle$

(A6) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, {\uparrow}vid{=}\alpha) \rightarrow \texttt{err}(er)$,
if $\Delta(vid) = \sigma \wedge \texttt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma \preceq_{vid} \alpha) \rightarrow^* \texttt{err}(er)$

`functor parameters`

(FP1) $\texttt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, \texttt{lazy}(e)) \rightarrow \texttt{succ}\langle u_2,\ e_1;e'\rangle$,
if $\texttt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, e) \rightarrow^* \texttt{succ}\langle u_2,\ e_2\rangle \wedge e_1 \backslash e_2 \xrightarrow{\mathsf{toLazy}} e'$

(FP2) $\texttt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, \texttt{lazy}(e)) \rightarrow \texttt{err}(er)$,
if $\texttt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, e) \rightarrow^* \texttt{err}(er)$

`functor applications`

(FA1) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, fct \cdot e) \rightarrow \texttt{succ}\Delta';\mathsf{genLazy}(\Delta', e_2^{\overline{d}'})$,
if $\mathsf{build}(u, fct) = (e_1 \rightsquigarrow e_2)^{\overline{d}'} \wedge \texttt{slv}(\langle u,\ e\rangle, \overline{d} \cup \overline{d}', e{:}e_1) \rightarrow^* \texttt{succ}\Delta'$

(FA2) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, fct \cdot e) \rightarrow \texttt{err}(er)$,
if $\mathsf{build}(u, fct) = (e_1 \rightsquigarrow e_2)^{\overline{d}'} \wedge \texttt{slv}(\langle u,\ e\rangle, \overline{d}, e{:}e_1) \rightarrow^* \texttt{err}(er)$

(FA3) $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, fct \cdot e) \rightarrow \texttt{succ}\langle u,\ e\rangle$,
if $\mathsf{strip}(\mathsf{build}(u, fct)) \in \mathsf{Var}$

The stack can be used as shown in the core to remove rules which eventually result in error. This means that in figures 4.24 and 4.25 presented above, we can remove rules `(A6)`, `(SU2)`, `(SU7)`, `(FP2)`, and `(FA2)` by pushing elements to solve after success has been generated on to the stack. If we use any of these rules during constraint solving and we terminate in an error state, then we do not need to think back any number of steps to a previous state in the constraint solver in order to know what rule it turned out we needed to use. Instead, the constraint solver will terminate with an error, and the future actions to be completed on the stack will be thrown away as they are at that point irrelevant.

4.6.4 Discussion of some important rules of the constraint solver

- Rule (S6) generates a type constructor clash if it notices two type constructor names have an equality constraint constraining them, where the type constructors are not equal.
- Rule (U1) generates a circularity error. This error is generated where a variable is seen constrained to be equal to some term where the variable occurs in that term.
- Rule (U4) handles constraints where environment variables are assigned to environments. In this case environment is first solved, then add a mapping from the environment variable to this solved environment to the unifier set. This is done using the stack parameter.
- Rule (C1) handles composition environments. The solving of the second environment e_2 is pushed on to the stack and then afterward recurse with the new stack and solve e_1.
- Rules (A1) and (A3) deal with accessors. In the case of rule (A1) accessors are connected to their binders by looking the variable up in the constraint solving context. In rule (A3) success is generated, but this can be used to report free identifiers.
- Rule (P1) deals with the promotion of monomorphic binders to a polymorphic status by calling the toPoly function using the new mechanism discussed in this section.

4.7 Efficiency

This section will discuss the efficiency of Skalpel and shows areas where improvements have been made. In section 4.7.1, we look at how profiling information is generated. In section 4.7.2, we look at how the master branch of the Skalpel source code repository represents label sets, and its performance information using the profiling output of the MLton compiler when compiled on the master branch. Section 9.3.2 looks at a proposal to have a variable amount of granularity with respect to specificity of program labels. Section 4.7.3 does the same but for an alternative implementation of labels sets. Finally, in section 4.7.4 we show how the speed of the analysis engine can be sped up more than 70% in some cases by allowing the user to focus Skalpel on specific kinds of error detection.

4.7.1 Profiler choice and how profiling is generated

The profiler that has been used to generate profile information for Skalpel is that which comes bundled with the MLton compiler installation. We chose this compiler because as we use MLton to prepare our Skalpel builds (MLton produces the fastest binaries) it is advantageous to us to use the same compiler to generate profiling information. This is principally because support may be dropped for other compilers at some point in the near future and making improvements suggested after profiling with the profiler bundled with e.g. the SML/NJ compiler may not give us performance gains in the MLton binary, which is almost always used.

The output of this version of the analysis engine binary which supports profiling generates output such as that shown in figure 4.26.

Figure 4.26 Example of generated profile information

9.57 seconds of CPU time (1.90 seconds GC)

function	cur (raw)	stack (raw)	GC (raw)
<gc>	16.6% (1.90s)	0.0% (0.00s)	16.6% (1.90s)
getValOneState.<case onestate x> StateEnv.sml: 323	3.9% (0.45s)	6.9% (0.79s)	2.2% (0.25s)
splay.adj smlnj-lib/Util/splaytree.sml: 23	3.7% (0.42s)	3.7% (0.43s)	2.2% (0.25s)
unionExtLab.<case (x1, labs1 ...> ExtLab.sml: 82	2.9% (0.33s)	4.2% (0.48s)	0.0% (0.00s)
List.@ $(SML_LIB)/basis/list/list.sml: 55	2.8% (0.32s)	2.8% (0.32s)	2.2% (0.25s)
List.foldl.loop $(SML_LIB)/basis/list/list.sml: 40	2.8% (0.32s)	3.8% (0.44s)	5.8% (0.66s)
...			

The first line specifies the total time taken for the binary to execute and how much time was spent doing garbage collection (this is represented in figure 4.26 by the initials `GC`). Following this functions are listed in descending order of how much time was dedicated to them, including information of time taken, stack information and garbage collection.

We currently do not generate profiling information on a daily basis, but this is an area recommended for future work. It would be interesting to produce graphs of profiling output from each month in order to track how developing features and other changes to the implementation affect Skalpel performance.

4.7.2 Current performance on the master branch of the Skalpel repository

In the master branch of the Skalpel repository, we represent label sets with binary trees. In this section we look at profiling information for Skalpel which shows where the implementation spends its time. We can see the profiling information for the master branch in figure 4.27.

In this figure we can see that a significant amount of time is spent doing garbage

collection, and also spent doing operations involving labels (the BinarySetFn functor and SplayTree structure are both involved in representing program points). The parts of the analysis engine which perform heavy operations involving label sets are the portions of figure 3.1 which are highlighted in blue - these are (and for the foreseeable future will be) the areas are interest with respect to improvement of implementation efficiency.

The analysis engine for the Standard ML programming language takes a long time to solve constraints for large programs and, with each new extension, Skalpel takes longer to find errors as it generates more constraints which must then be solved. Equality types are an example of an extension which has a noticeable impact on performance. It is therefore important to look at the analysis engine in terms of performance, and in other sections of this chapter we will look at different approaches to handle the representation of program points with the goal of speeding up the analysis engine.

Figure 4.27 Profile of the master branch

9.66 seconds of CPU time (1.22 seconds GC)

function	cur (raw)	stack (raw)	GC (raw)
<gc>	11.2% (1.22s)	0.0% (0.00s)	11.2% (1.22s)
SplayTree.splay.adj $(SML_LIB)/smlnj-lib/Util/splaytree.sml: 23	4.8% (0.52s)	4.8% (0.52s)	2.4% (0.26s)
Label.union.<case set1 set2> sets/Label.sml: 141	3.3% (0.36s)	12.8% (1.39s)	2.7% (0.29s)
StateEnv.getValOneState.<case onestate x> enum/StateEnv.sml: 323	2.8% (0.31s)	6.3% (0.69s)	0.6% (0.07s)
ExtLab.unionExtLab.<case (x1, labs1 ...> constraint/ExtLab.sml: 78	2.8% (0.30s)	5.7% (0.62s)	0.0% (0.00s)
StateEnv.updateOneState.<case onestate x y> enum/StateEnv.sml: 406	2.3% (0.25s)	2.8% (0.30s)	0.0% (0.00s)
Unif.unif.fsimplify enum/Unification.sml: 3634	2.3% (0.25s)	71.2% (7.75s)	10.5% (1.14s)
BinarySetFn.uni_bd $(SML_LIB)/smlnj-lib/Util/binary-set-fn.sml: 202	2.3% (0.25s)	2.8% (0.30s)	0.0% (0.00s)
BinarySetFn.uni_lo $(SML_LIB)/smlnj-lib/Util/binary-set-fn.sml: 236	1.8% (0.20s)	5.1% (0.55s)	0.0% (0.00s)
BinarySetFn.add $(SML_LIB)/smlnj-lib/Util/binary-set-fn.sml: 144	1.6% (0.17s)	1.6% (0.17s)	0.0% (0.00s)
BinarySetFn.uni_hi $(SML_LIB)/smlnj-lib/Util/binary-set-fn.sml: 229	1.6% (0.17s)	2.7% (0.29s)	0.0% (0.00s)
BinarySetFn.concat3 $(SML_LIB)/smlnj-lib/Util/binary-set-fn.sml: 152	1.5% (0.16s)	1.5% (0.16s)	2.7% (0.29s)
StateEnv.getValStateId.<case E.ROW_ENV ...> enum/StateEnv.sml: 562	1.3% (0.14s)	5.4% (0.59s)	1.7% (0.18s)
StateEnv.getValStateId.<case (NONE, y, ...> enum/StateEnv.sml: 563	1.3% (0.14s)	1.3% (0.14s)	0.0% (0.00s)
SplayMapFn.map.ap $(SML_LIB)/smlnj-lib/Util/splay-map-fn.sml: 249	1.0% (0.11s)	5.1% (0.56s)	0.0% (0.00s)
Unif.buildfieldType.<val ty'> enum/Unification.sml: 1012	0.9% (0.10s)	5.6% (0.61s)	1.9% (0.21s)
List.map $(SML_LIB)/basis/list/list.sml: 66	0.9% (0.10s)	16.5% (1.79s)	3.1% (0.34s)
Unif.unif.fsimplify.<case false> enum/Unification.sml: 4976	0.8% (0.09s)	6.8% (0.74s)	0.0% (0.00s)

4.7.3 Hash tables to represent label sets

A new branch of the source repository has been created which represents the label sets using hash tables (other representations were attempted, but these proved the fastest results when running our analysis engine test database). In order to attach labels to constraints, we now use a mapping from labels to booleans using a hash table[2] (the reason for having booleans in the map is discussed later).

In order to initially test this design, a small implementation was built which generated integer sets and integer to boolean maps, and then performed the union operation for each approach many times. The results for that initial test can be seen in figure 4.28.

Figure 4.28 Data for union operations of integer to boolean maps (1,000,000,000 iterations)

Total number of labels	Time taken (original)	Time taken (new)
5	$<$ 0:01	$<$ 0:01
30	$<$ 0:01	$<$ 0:01
1000	0:34	0:06
3000	1:58	0:19
15000	12:40	2:04

It was discovered however, that the implementation of hash tables in the SML/NJ library is done using ref cells which caused problems, as when passing hash table values to functions, the function did not get a *copy* of the hash table, but rather the hash table itself. This meant that in order to be certain the old semantics were preserved, one of the hash tables given as an argument to the union function inside the analysis engine must be copied to an empty table of the same size (this will give a new, unique hash table and avoid destructive modification of the original). The time spent copying these hash tables is somewhat expensive.

Even with the time spent copying hash tables, this implementation still proved faster while solving many of Skalpel's test cases. Given the use of hashing, this implementation will perform better than the original implementation in terms of time taken to generate errors as the program input becomes more complex.

4.7.4 Targeting Skalpel to find Specific Error Types

Another branch has been created in the Skalpel implementation which allows Skalpel to focus itself on targeting specific kinds of errors. For example, if the user runs

[2] A data structure mapping keys to value, where in this case here the labels are keys. A hash tables uses a hash function (unique identifier generation) which can access the value associated with a given key.

their compiler and sees that a specific error has occurred, and they know no other errors are occurring (or they just want answers much faster), by configuring Skalpel to look at specific kinds of error, they will get results far faster.

As an example we look at equality types. Consider the case where a user has made a small change to their program and their compiler reports that they have an equality type error in their code, if the user wishes an answer quickly, we should be able to target Skalpel to look for that kind of error specifically. By doing this, Skalpel will run far faster than it otherwise would. In this branch that has been created, by focusing Skalpel to find errors involving type constructors, and ignoring constraints for e.g. equality types, the time taken for the implementation to run the first 104 tests in our analysis engine test database was reduced by **more that 70%**. This is currently implemented by ignoring the relevant generated constraints at constraint solving time, but by making the constraint generator modular and allowing the user to toggle the language features we generate constrains for at will, we will be able to see even bigger gains than this and Skalpel will become more effective for the user.

Chapter 5

Worked examples of Skalpel Algorithms

This chapter shows two examples where we take some piece of SML code, we present the labelled version of that code, generate constraints for it, and then solve those constraints with the machinery described in chapter 4. In both cases, an error exists in the code presented, and so the constraints generated are unsolvable, and we demonstrate this.

This chapter is quite verbose in nature, and it is hoped that our system can be better understood by analyzing worked examples such as these.

Note that we only demonstrate worked examples of the *new* version of the Skalpel core described in chapter 4. We do not demonstrate examples on the old version of the Skalpel core (chapter 3), as this new core is designed to supersede the old one.

These examples demonstrate the following features of the new core design:

- It is always known from the current state (environments left to solve, the current set of unifiers, and current environment e), which rule must be used next *without* the concept of having to look ahead.
- The stack of environments in the first parameter to `slv`.
- Our use of the existential forms, without any dummy variables.

We will show in this following examples some places where the new changes to the core have been made and how this affects the solving of constraints. Note also that throughout both examples variables are existentially quantified, so we no longer need to remember which variables are dummy variables, and which are not and we

can constrain. This improvement is not discussed explicitly in the examples as they apple to every step.

5.1 Example One

Now let us consider an example for the program given below.

$\texttt{datatype 'a a}^{l_2} =^{l_1} \texttt{A}_{\texttt{c}}^{l_3}$
$\texttt{datatype 'a b}^{l_5} =^{l_4} \texttt{B}_{\texttt{c}}^{l_6}$
$\lceil(\texttt{fn A}_{\texttt{p}}^{l_9} =>^{l_8} \texttt{A}^{l_{10}})\ \texttt{B}_{\texttt{e}}^{l_{11}}\rceil^{l_7}$

5.1.1 Constraint generation

- We first generate constraints for the datatype `a`. We can see from figure 4.17 that rule (G18) handles datatype so we first apply that rule. Note that here our label l_1 is used as the label l. Then we see that the datatype name (`'a a`) is matched with rule (G13). The constraint generation for the datatype constructor is handled with rule (G14). From these rules, we generate the following constraints:

  $\exists\langle\alpha_1, \alpha_2, ev\rangle.ev = ((\alpha_1 \overset{l_1}{=} \alpha_2)); (\exists\langle\alpha_3, \gamma\rangle.(\alpha_1 \overset{l}{=} \alpha_3\ \gamma); (\downarrow\texttt{a} \overset{l_2}{=} \gamma); (\downarrow\texttt{'a} \overset{l_2}{=} \alpha_3));$
  $\texttt{poly}(\downarrow\texttt{A} \overset{l_3}{=} \alpha_2); ev^{l_1}$

- We also generate constrains for the datatype `b`, which is done the same way as `a`. The generated constraints are as follows:

  $\exists\langle\alpha_4, \alpha_5, ev_2\rangle.ev_2 = ((\alpha_4 \overset{l_1}{=} \alpha_5));$
  $(\exists\langle\alpha_6, \gamma_2\rangle.(\alpha_4 \overset{l}{=} \alpha_6\ \gamma_2); (\downarrow\texttt{b} \overset{l_2}{=} \gamma_2); (\downarrow\texttt{'a} \overset{l_2}{=} \alpha_6)); \texttt{poly}(\downarrow\texttt{B} \overset{l_3}{=} \alpha_5); ev_2^{l_1}$

- Finally, we generate constraints for the application portion. Application is handled with (G3), where the left hand side of the expression becomes the anonymous function, and the right becomes the datatype constructor. We use rules (G6) and (G1) to generate constraints for the function and rule (G5) to handle the `B` constructor. The fully built constraints are shown below.

  $\exists\langle\alpha_7, \alpha_8\rangle.[\exists\langle\alpha_9, \alpha_{10}, ev_3\rangle.(ev_3 = \downarrow\texttt{A} \overset{l_9}{=} \alpha_9); ev_3^{l_8}; \uparrow\texttt{A} \overset{l_{10}}{=} \alpha_{10}; (\alpha_7 \overset{l_8}{=} \alpha_9 \to \alpha_{10})];$
  $(\uparrow\texttt{B} \overset{l_{11}}{=} \alpha_8); (\alpha_7 \overset{l_7}{=} \alpha_8 \to \alpha)$

  The fully generated constraints are therefore:

  $\exists\langle\alpha_1, \alpha_2, ev\rangle.ev = ((\alpha_1 \overset{l_1}{=} \alpha_2)); (\exists\langle\alpha_3, \gamma\rangle.(\alpha_1 \overset{l}{=} \alpha_3\ \gamma); (\downarrow\texttt{a} \overset{l_2}{=} \gamma); (\downarrow\texttt{'a} \overset{l_2}{=} \alpha_3));$
  $\texttt{poly}(\downarrow\texttt{A} \overset{l_3}{=} \alpha_2); ev^{l_1};$
  $\exists\langle\alpha_4, \alpha_5, ev_2\rangle.ev_2 = ((\alpha_4 \overset{l_1}{=} \alpha_5));$

$(\exists\langle\alpha_6, \gamma_2\rangle.(\alpha_4 \overset{l}{=} \alpha_6\ \gamma_2); (\downarrow\mathtt{b} \overset{l_2}{=} \gamma_2); (\downarrow\mathtt{'a} \overset{l_2}{=} \alpha_6)); \mathtt{poly}(\downarrow\mathtt{B} \overset{l_3}{=} \alpha_5); ev_2^{l_1};$
$\exists\langle\alpha_7, \alpha_8\rangle.[\exists\langle\alpha_9, \alpha_{10}, ev_3\rangle.(ev_3 = \downarrow\mathtt{A} \overset{l_9}{=} \alpha_9); ev_3^{l_8}; \uparrow\mathtt{A} \overset{l_{10}}{=} \alpha_{10}; (\alpha_7 \overset{l_8}{=} \alpha_9 \to \alpha_{10})];$
$(\uparrow\mathtt{B} \overset{l_{11}}{=} \alpha_8); (\alpha_7 \overset{l_7}{=} \alpha_8 \to \alpha)$

5.1.2 Constraint solving

- We use rule (C1) to get the constraints for the first datatype declaration, then rule (X) to remove the existential quantification. This results in:

  $ev' = ((\alpha'_1 \overset{l_1}{=} \alpha'_2); (\exists\langle\alpha_3, \gamma\rangle.(\alpha'_1 \overset{l}{=} \alpha_3\ \gamma); (\downarrow\mathtt{a} \overset{l_2}{=} \gamma); (\downarrow\mathtt{'a} \overset{l_2}{=} \alpha_3));$
  $\mathtt{poly}(\downarrow\mathtt{A} \overset{l_3}{=} \alpha'_2)); ev'^{l_1}$

- We use rule (C1), pushing ev'^{l_1} on the stack, use rule (U4) to handle the environment variable mapping, then use rules (C1) and (U3) to handle $\alpha'_1 \overset{l_1}{=} \alpha'_2$. Note that rule (U4) no longer has the notion of environment difference, so there is no need to compute that. Instead, we just add a new environment on to the environment stack. We then use (C1) to push the `poly` form to the stack and then (X) to deal with the following existential quantifier.

$$(\alpha'_1 \overset{l}{=} \alpha'_3\ \gamma'); (\downarrow\mathtt{a} \overset{l_2}{=} \gamma'); (\downarrow\mathtt{'a} \overset{l_2}{=} \alpha'_3)$$

- Rules (C1), (U6) and (U3) are used to handle the first constraint, and rules (C1), (D) and (B) are used twice to handle the second and third constraints. We then have the `poly` constraint to handle from the stack:

$$\mathtt{poly}(\downarrow\mathtt{A} \overset{l_3}{=} \alpha'_2)$$

- We handle this with rule (P1). Again, note here that there is no longer any need to use the build function as there is no need to call that function with the new unifier representation described in chapter 4. The resulting expression is mapped to the environment variable ev', and we then handle the variable still on the stack, ev'^{l}_1, with rule (V).

  At this point, the set of unifiers, $\mathcal{U}$, is as follows:

$$\mathcal{U} = \{\alpha'_1 \mapsto \alpha'_2, \alpha'_2 \mapsto \alpha'_3\ \gamma, ev' \mapsto \downarrow\mathtt{a}\ \overset{\{l_1,l_2\}}{=}\ \gamma'; \downarrow\mathtt{'a}\ \overset{\{l_1,l_2\}}{=}\ \alpha'_3; \downarrow\mathtt{A}\ \overset{\{l_1,l_2,l_3\}}{=}\ \forall\alpha_2.\,\alpha_2\}$$

  and the environment tuple is:

$$ev'^{\{l_1,l_2,l_3\}}$$

As the second datatype declaration is the same as the first but with the name of constructors and labels changed, we execute the same rules as before to leave us with the unifier set:

$$\mathcal{U} = \{\alpha'_1 \mapsto \alpha'_2, \alpha'_2 \mapsto \alpha'_3\ \gamma', ev' \mapsto \downarrow\mathtt{a}\ ^{\{l_1,l_2\}}\!\!=\ \gamma'; \downarrow\mathtt{'a}\ ^{\{l_1,l_2\}}\!\!=\ \alpha'_3; \downarrow\mathtt{A}\ ^{\{l_1,l_2,l_3\}}\!\!=\ \forall\alpha_2.\,\alpha_2,$$

$$\alpha'_4 \mapsto \alpha'_5, \alpha'_5 \mapsto \alpha'_6\ \gamma'_2, ev'_2 \mapsto \downarrow\mathtt{b}\ ^{\{l_4,l_5\}}\!\!=\ \gamma'_2; \downarrow\mathtt{'a}\ ^{\{l_1,l_5\}}\!\!=\ \alpha'_6; \downarrow\mathtt{B}\ ^{\{l_4,l_5,l_6\}}\!\!=\ \forall\alpha_5.\,\alpha_5\}$$

and the environment tuple is:

$$ev'^{\{l_1 l_2, l_3\}}_1\, ev'^{\{l_4,l_5,l_6\}}_2$$

- We then continue, solving the constraints for the application. We use rules (X), (C1) and (U4) to deal with existential quantification and square brackets. Note that here, we use the stack to handle the sequence of environments and do not just compose them with the ';' operator.

$$(ev'_3 = \uparrow\mathtt{A} \stackrel{l_9}{=} \alpha'_9); ev'^{l_8}_3; \uparrow\mathtt{A} \stackrel{l_{10}}{=} \alpha'_{10}; (\alpha'_7 \stackrel{l_8}{=} \alpha'_9 \to \alpha'_{10})$$

- We handle the first constraint with (C1), (U3), (D) and then (A1), and the second with (D) and (V).

$$\uparrow\mathtt{A} \stackrel{l_{10}}{=} \alpha'_{10}; (\alpha'_7 \stackrel{l_8}{=} \alpha'_9 \to \alpha'_{10})$$

- The accessor is handled with constraints (C1), (D) and (A1), and rule (U3) handles the second constraint. We pop the remaining constraints from the stack to leave us with:

$$(\uparrow\mathtt{B} \stackrel{l_{11}}{=} \alpha'_8); (\alpha'_7 \stackrel{l_7}{=} \alpha'_8 \to \alpha)$$

- We handle the accessor again with rules (C1), (D) and (A1) to leave us with our last constraint:

$$\alpha'_7 \stackrel{l_7}{=} \alpha'_8 \to \alpha$$

- We use rule (U6) as α'_7 already exists in the unifier set. This gets us the following constraint:

$$\alpha'_3\gamma \to \alpha'_3\gamma = \alpha'_6\ \gamma_2 \to \alpha$$

- We then use rules (S4) and (C1).

$$\alpha'_3\gamma = \alpha'_6\ \gamma_2$$

- Finally, we use rules (S3) and (S6), which generates a type constructor clash.

$$\texttt{err}(\langle \texttt{clash}, \{l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8\}\rangle)$$

5.2 Example Two

Let us consider the following labelled program, and then generate and solve constraints for it with all steps shown.

$\texttt{fn y}^{l_2} \overset{l}{\Rightarrow} \texttt{let}^{l_3}\ \texttt{val rec f}^{l_8} =^{l_7} \texttt{fn x}^{l_9} \overset{l_{10}}{\Rightarrow} \lceil \texttt{x}^{l_{12}}\ \texttt{y}^{l_{13}} \rceil^{l_{11}}\ \texttt{in}\ \lceil \texttt{f}^{l_4}\ \texttt{y}^{l_5} \rceil^{l_6}\ \texttt{end}$

5.2.1 Constraint generation

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

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

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

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

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

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

where the expression *exp* is the nameless function taking the pattern `x`. To generate constraints for this we apply rules (G4) and (G6) of the constraint generator again. This produces the below.

$[\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = \downarrow\mathtt{x} \stackrel{l_9}{=} \alpha_6); ev_3^{l_{10}}; [\![exp, \alpha_7]\!]; \alpha_5 \stackrel{l_{10}}{=} \alpha_6 \to \alpha_7]$

The expression *exp* here is the application `x y`. This is handled with rule (G3) for application and rules (G7) and (G1) to create accessors for `x` and `y` respectively. After applying these rules we generate the following constraints:

$\exists\langle\alpha_8, \alpha_9\rangle.\uparrow\mathtt{x} \stackrel{l_{12}}{=} \alpha_8; \uparrow\mathtt{y} \stackrel{l_{13}}{=} \alpha_9; (\alpha_8 \stackrel{l_{11}}{=} \alpha_9 \to \alpha_7)$

Now we have finished dealing with the declaration portion of the `let` statement in this program, we deal with the expression portion. In this portion of the let statement, we see that we have the application `f y`. We use rules (G3), (G7) and (G1) to handle this which produces:

$\exists\langle\alpha', \alpha''\rangle.\uparrow\mathtt{f} \stackrel{l_4}{=} \alpha'; \uparrow\mathtt{y} \stackrel{l_5}{=} \alpha''; (\alpha' \stackrel{l_6}{=} \alpha'' \to \alpha_2)$

We have now generated all of the constraints for this program. The final generated constraints for this program are therefore:

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

5.2.2 Constraint Solving

We now show the start form of the constraint generator and proceed from there. We start with the function call:

$\mathtt{slv}(\langle\top\rangle, \varnothing, \varnothing, \langle\rangle, e_1)$ where e_1 is the environment returned from the initial constraint generator, shown below.
$[\exists\langle\alpha_1, \alpha_2, ev\rangle.(ev = \downarrow\mathtt{y} \stackrel{l_2}{=} \alpha_1); ev^l; [\exists\alpha_3.\exists\langle\alpha_4, \alpha_5, ev_2\rangle.(ev_2 =$
$\mathtt{poly}(\downarrow\mathtt{f} \stackrel{l_8}{=} \alpha_4; [\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = \downarrow\mathtt{x} \stackrel{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.\uparrow\mathtt{x} \stackrel{l_{12}}{=} \alpha_8; \uparrow\mathtt{y} \stackrel{l_{13}}{=}$
$\alpha_9; (\alpha_8 \stackrel{l_{11}}{=} \alpha_9 \to \alpha_7); \alpha_5 \stackrel{l_{10}}{=} \alpha_6 \to \alpha_7]; (\alpha_4 \stackrel{l_7}{=} \alpha_5))); ev_2^{l_7}; \exists\langle\alpha', \alpha''\rangle.\uparrow\mathtt{f} \stackrel{l_4}{=} \alpha'; \uparrow\mathtt{y} \stackrel{l_5}{=}$
$\alpha''; (\alpha' \stackrel{l_6}{=} \alpha'' \to \alpha_2); (\alpha_2 \stackrel{l_3}{=} \alpha_3)]; (\alpha \stackrel{l}{=} \alpha_1 \to \alpha_2)]$

In this step we apply rules (U4) and (X) to remove the [] notation and existential quantification, renaming α_1, α_2 and ev to α_0, α_1 and ev' respectively.

$\mathtt{slv}(\langle\top\rangle, \varnothing, \varnothing, \langle\rangle, e_2)$ where e_2 is
$(ev' = \downarrow\mathtt{y} \stackrel{l_1}{=} \alpha_0); ev'^l; [\exists\alpha_3.\exists\langle\alpha_4, \alpha_5, ev_2\rangle.(ev_2 =$
$\mathtt{poly}(\downarrow\mathtt{f} \stackrel{l_8}{=} \alpha_4; [\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = \downarrow\mathtt{x} \stackrel{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.\uparrow\mathtt{x} \stackrel{l_{12}}{=} \alpha_8; \uparrow\mathtt{y} \stackrel{l_{13}}{=}$
$\alpha_9; (\alpha_8 \stackrel{l_{11}}{=} \alpha_9 \to \alpha_7); \alpha_5 \stackrel{l_{10}}{=} \alpha_6 \to \alpha_7]; (\alpha_4 \stackrel{l_7}{=} \alpha_5))); ev_2^{l_7}; \exists\langle\alpha', \alpha''\rangle.\uparrow\mathtt{f} \stackrel{l_4}{=} \alpha'; \uparrow\mathtt{y} \stackrel{l_5}{=}$
$\alpha''; (\alpha' \stackrel{l_6}{=} \alpha'' \to \alpha_1); (\alpha_1 \stackrel{l_3}{=} \alpha_3)]; (\alpha \stackrel{l}{=} \alpha_0 \to \alpha_1)$

We now apply rules (C1) to break up the environment composition, then rules (U4), (D) to strip off the dependency on the binder, and (B) to handle the binder. Rules (C1), (D) and (V) are applied to handle the ev'^{l} expression. Note that this the representaiton shown in the next step demonstrates how we have now kept a monomorphic variable in the parameter of `slv`. Later, when a polymorphic environment needs to be handled, it is the variables in this set that are used in order to compute the labels that are annotated on to a polymorphic binder.

$\texttt{slv}(\langle ev'^{\{l,l_2\}}\rangle, \{l, l_2\}, \{\alpha_0\}, \langle\rangle, e_3)$ where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0\}$ and e_3 is
$[\exists\alpha_3.\exists\langle\alpha_4, \alpha_5, ev_2\rangle.(ev_2 =$
$\texttt{poly}(\downarrow\texttt{f} \stackrel{l_8}{=} \alpha_4; [\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = \downarrow\texttt{x} \stackrel{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.\uparrow\texttt{x} \stackrel{l_{12}}{=} \alpha_8; \uparrow\texttt{y} \stackrel{l_{13}}{=} \alpha_9; (\alpha_8 \stackrel{l_{11}}{=} \alpha_9 \to \alpha_7); \alpha_5 \stackrel{l_{10}}{=} \alpha_6 \to \alpha_7]; (\alpha_4 \stackrel{l_7}{=} \alpha_5))); ev_2^{l_7}; \exists\langle\alpha', \alpha''\rangle.\uparrow\texttt{f} \stackrel{l_4}{=} \alpha'; \uparrow\texttt{y} \stackrel{l_5}{=} \alpha''; (\alpha' \stackrel{l_6}{=} \alpha'' \to \alpha_1); (\alpha_1 \stackrel{l_3}{=} \alpha_3)]; (\alpha \stackrel{l}{=} \alpha_0 \to \alpha_1)$

Rule (C1) is used, pushing the expression $(\alpha \stackrel{l}{=} \alpha_0 \to \alpha_1)$ onto the stack $\overrightarrow{st}$, then rules (U4) and (X) are used to remove the dummy environment variable expression and existential quantification of α_3. We then apply rules (C1) and (X) again to remove further existential quantification, then apply rule (C1).

$\texttt{slv}(\langle ev'^{\{l,l_2\}}\rangle, \{l, l_2\}, \{\alpha_0\}, \overrightarrow{st}, e_4)$ where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0\}$ and e_4 is
$ev_1 = \texttt{poly}(\downarrow\texttt{f} \stackrel{l_8}{=} \alpha_3; [\exists\langle\alpha_6, \alpha_7, ev_3\rangle.(ev_3 = \downarrow\texttt{x} \stackrel{l_9}{=} \alpha_6); ev_3^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.\uparrow\texttt{x} \stackrel{l_{12}}{=} \alpha_8; \uparrow\texttt{y} \stackrel{l_{13}}{=} \alpha_9; (\alpha_8 \stackrel{l_{11}}{=} \alpha_9 \to \alpha_7); \alpha_4 \stackrel{l_{10}}{=} \alpha_6 \to \alpha_7]; (\alpha_3 \stackrel{l_7}{=} \alpha_4))$

We apply rule (U4), (P4) (computing the inside of the polymorphic expression first), rules (C1), (D) and (B) to deal with the binder of `f`. Rules (C1), (U4) and (X) remove quantification, and rules (C1), (U4), (D) and (B) are used to deal with the binder of `x`.

$\texttt{slv}(\langle ev'^{\{l,l_2\}}, \downarrow\texttt{f} \stackrel{\{l,l_1,l_8\}}{=} \alpha_3\rangle, \{l, l_1, l_2, l_8, l_9\}, \{\alpha_0, \alpha_3\}, \overrightarrow{st}, e_5)$
where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0, ev_2 \mapsto \downarrow\texttt{x} \stackrel{\{l,l_1,l_8,l_9\}}{=} \alpha_5\}$
and e_5 is $ev_2^{l_{10}}; \exists\langle\alpha_8, \alpha_9\rangle.\uparrow\texttt{x} \stackrel{l_{12}}{=} \alpha_8; \uparrow\texttt{y} \stackrel{l_{13}}{=} \alpha_9; (\alpha_8 \stackrel{l_{11}}{=} \alpha_9 \to \alpha_6); \alpha_4 \stackrel{l_{10}}{=} \alpha_5 \to \alpha_6$

We use rules (C1), (D), and (V) to deal with the environment variable, (C1) and (X) to deal with quantification, (C1), (D), (A1) and (U3) twice to deal with the accessors.

$\texttt{slv}(\langle ev'^{\{l,l_2\}}, \downarrow\texttt{f} \stackrel{\{l,l_1,l_8\}}{=} \alpha_3; ev_2^{\{l,l_1,l_8,l_9,l_{10}\}}\rangle, \{l, l_1, l_2, l_8, l_9, l_{10}, l_{12}, l_{13}\}, \{\alpha_1, \alpha_3\}, \overrightarrow{st}, e_6)$
where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0, ev_2 \mapsto \downarrow\texttt{x} \stackrel{\{l,l_1,l_8,l_9\}}{=} \alpha_5, \alpha_7 \mapsto \alpha_5, \alpha_8 \mapsto \alpha_0\}$
and e_6 is $(\alpha_7 \stackrel{l_{11}}{=} \alpha_8 \to \alpha_6)$

Rule (D), (U6) and then (U3) deal with the remaining environment, and we solve

the constraints $\alpha_4 \stackrel{l_{10}}{=} \alpha_5 \to \alpha_6$ and $\alpha_3 \stackrel{l_7}{=} \alpha_4$ from the stack, for which we use rules (D), (U3) for the first rule, and (D), (U6) and (U3) for the second case. Now we have completed the inside of the `poly` environment, so we restore the initial environment and monomorphic variable set from the stack.

$\texttt{slv}(\langle ev'^{\{l,l_2\}} \rangle, \{l, l_1, l_2, l_8, l_9, l_{10}, l_{12}, l_{13}\}, \alpha_0, \overrightarrow{st}, e_7)$
where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0, ev_2 \mapsto \downarrow\texttt{x} \stackrel{\{l,l_1,l_8,l_9\}}{=} \alpha_5, \alpha_7 \mapsto \alpha_5, \alpha_8 \mapsto \alpha_0, \alpha_5 \mapsto \alpha_8 \to \alpha_6, \alpha_4 \mapsto \alpha_5 \to \alpha_6, \alpha_3 \to (\alpha_0 \to \alpha_6) \to \alpha_6\}$
and e_7 is $\texttt{poly}(\downarrow\texttt{f} \stackrel{l_8}{=} \alpha_3)$

We apply rule (P1), and add the result to the unifier set. Here, when we need to know the labels of all of the monomorphic binders that we have encountered, there is no need for computation as they are stored in a set which we have access to via a parameter of the constraint solver. We then resume to solve the final part of the environment that is left on the stack.

$\texttt{slv}(\langle ev'^{\{l,l_2\}} \rangle, \{l, l_1, l_2, l_8, l_9, l_{10}, l_{12}, l_{13}\}, \alpha_0, \overrightarrow{st}, e_8)$
where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0, ev_2 \mapsto \downarrow\texttt{x} \stackrel{\{l,l_1,l_8,l_9\}}{=} \alpha_5, \alpha_7 \mapsto \alpha_5, \alpha_8 \mapsto \alpha_0, \alpha_5 \mapsto \alpha_8 \to \alpha_6, \alpha_4 \mapsto \alpha_5 \to \alpha_6, \alpha_3 \to (\alpha_0 \to \alpha_6) \to \alpha_6\}, ev_1 \mapsto \downarrow\texttt{f} \stackrel{l_8}{=} \forall\alpha_6.\,(\alpha_0 \to \alpha_6) \to \alpha_6\}$
and e_8 is $ev_1^{l_7}; \exists\langle\alpha', \alpha''\rangle.\uparrow\texttt{f} \stackrel{l_4}{=} \alpha'; \uparrow\texttt{y} \stackrel{l_5}{=} \alpha''; (\alpha' \stackrel{l_6}{=} \alpha'' \to \alpha_1); (\alpha_1 \stackrel{l_3}{=} \alpha_2); (\alpha \stackrel{l}{=} \alpha_0 \to \alpha_1)$

Rules (C1), (D), (V) handle the environment variable, and rules (C1) and (X) handle quantification.

$\texttt{slv}(\langle ev'^{\{l,l_2\}}; ev_1^{\{l,l_1,l_2,l_7,l_8,l_9,l_{10},l_{12},l_{13}\}} \rangle, \{l, l_1, l_2, l_7, l_8, l_9, l_{10}, l_{12}, l_{13}\}, \alpha_0, \overrightarrow{st}, e_9)$
where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0, ev_2 \mapsto \downarrow\texttt{x} \stackrel{\{l,l_1,l_8,l_9\}}{=} \alpha_5, \alpha_7 \mapsto \alpha_5, \alpha_8 \mapsto \alpha_0, \alpha_5 \mapsto \alpha_8 \to \alpha_6, \alpha_4 \mapsto \alpha_5 \to \alpha_6, \alpha_3 \to (\alpha_0 \to \alpha_6) \to \alpha_6\}, ev_1 \mapsto \downarrow\texttt{f} \stackrel{l_8}{=} \forall\alpha_6.\,(\alpha_0 \to \alpha_6) \to \alpha_6\}$
and e_9 is $\uparrow\texttt{f} \stackrel{l_4}{=} \alpha_9; \uparrow\texttt{y} \stackrel{l_5}{=} \alpha_{10}; (\alpha_9 \stackrel{l_6}{=} \alpha_{10} \to \alpha_1); (\alpha_1 \stackrel{l_3}{=} \alpha_2); (\alpha \stackrel{l}{=} \alpha_0 \to \alpha_1)$

Rules (C1), (D), and (A1) are applied twice to handle the accessors.

$\texttt{slv}(\langle ev'^{\{l,l_2\}}; ev_1^{\{l,l_1,l_2,l_7,l_8,l_9,l_{10},l_{12},l_{13}\}} \rangle, \{l, l_1, l_2, l_4, l_5, l_7, l_8, l_9, l_{10}, l_{12}, l_{13}\}, \alpha_0, \overrightarrow{st}, e_{10})$
where the set of unifiers $\mathcal{U}$ is $\{ev' \mapsto \downarrow\texttt{y} \stackrel{l}{=} \alpha_0, ev_2 \mapsto \downarrow\texttt{x} \stackrel{\{l,l_1,l_8,l_9\}}{=} \alpha_5, \alpha_7 \mapsto \alpha_5, \alpha_8 \mapsto \alpha_0, \alpha_5 \mapsto \alpha_8 \to \alpha_6, \alpha_4 \mapsto \alpha_5 \to \alpha_6, \alpha_3 \to (\alpha_0 \to \alpha_6) \to \alpha_6\}, ev_1 \mapsto \downarrow\texttt{f} \stackrel{l_8}{=} \forall\alpha_6.\,(\alpha_0 \to \alpha_6) \to \alpha_6, \alpha_9 \mapsto (\alpha_0 \to \alpha_{11}) \to \alpha_{11}, \alpha_{10} \mapsto \alpha_0\}$
and e_{10} is $(\alpha_9 \stackrel{l_6}{=} \alpha_{10} \to \alpha_1); (\alpha_1 \stackrel{l_3}{=} \alpha_2); (\alpha \stackrel{l}{=} \alpha_0 \to \alpha_1)$

We now apply rules (C1), (D), (U6), (S4), (C1), (R), (U6), and rule (U1), which generates a circularity error. Below, we show the error that is produced.

$\texttt{err}(\langle \texttt{circularity}, \{l, l_2, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11}, l_{12}, l_{13}\} \rangle)$

Chapter 6

Modifications to Pre-existing Extensions of Skalpel Core

This chapter presents pre-existing extensions which were present in [Rah10] to the original version of the Skalpel core to increase support for the Standard ML language, and shows how these extensions are modified to fit with the new version of the Skalpel core and to fix any bugs. There are four extensions presented here, which add support for the following Standard ML features:

- Local declarations.
- Type declarations.
- Type annotations.
- Signatures.

6.1 Local declarations

6.1.1 External Syntax

In order to extend Skalpel with a new feature, first additions to the external syntax must be made, to include any new program keywords that this extension is designed to support.

The external syntax used by this extension to the existing theory is ExtLabSynt with the additional extensions in this section. This will be referred to as $\mathsf{ExtLabSynt}_{\mathtt{loc}}$, which ${\mathcal{P}^L}_{\mathtt{loc}}$ ranges over.

In order to deal with local declarations, the external syntax should be extended as follows:

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

Let us consider the following example, which gives an example of an error that can occur when using local declarations in a program:

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

This code is erroneous, as `w` is applied to `y` which is an integer and `x` which is a boolean, but `w` is a monomorphic type.

6.1.2 Constraint Syntax

In the same way that the external syntax has been extended, the constraint syntax also needs to be extended (recall that the *external* syntax is the input (SML) syntax, and that the *constraint* syntax is used to represent constraints internally).

The constraint syntax used in this extension is IntLabSynt with the extensions listed in this section. This set is referred to as $\mathsf{IntLabSynt}_{\mathtt{loc}}$, which ${\mathcal{C}^L}_{\mathtt{loc}}$ ranges over.

The extension to the constraint syntax to support local declarations is given below:

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

This new piece of constraint syntax is introduced to allow us to have greater control of the scope of the binders. In this new form, the binders that are defined in the environment e_1 are available to the environment e_2, but are not exported outside of the `loc` environment. The current mechanisms that exist for controlling the availability of binders under certain environments is insufficient to cover this new usage:

- Using $e_1;e_2$ is insufficient because although the binders in e_1 are accessible in e_2, the binders of e_1 are exported along with the binders of e_2;
- Using $[e_1];e_2$ is insufficient because the binders in e_1 are not accessible in e_2;
- Using $[e_1; e_2]$ is insufficient because neither of the binders from e_1 nor e_2 are exported.

Applications of a substitution to a constraint term are extended with the following rule:

$$(\texttt{loc}\ e_1\ \texttt{in}\ e_2)[sub]{=}\texttt{loc}\ (e_1[sub])\ \texttt{in}\ (e_2[sub]).$$

This allows substation to take place inside the **loc** environments, by applying the substation to both environments contained inside.

6.1.3 Constraint Generation

In figure 6.1, the constraint generator is extended with three rules to handle local declarations.

Figure 6.1 Constraint generation rule for local declarations

Declaration$(dec \triangleright e)$

(G29) $\texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \triangleright (ev = e_1);\texttt{loc}\ ev^l\ \texttt{in}\ e_2$
$\Leftarrow dec_1 \triangleright e_1 \wedge dec_2 \triangleright e_2 \wedge \mathsf{dja}(e_1, e_2, ev)$

This new constraint rule takes the two declarations from the external syntax containing a `local` expression, and creates an internal representation using the `loc` environment defined previously.

These rules have been modified so that they can work correctly with the new version of the Skalpel core outlined in chapter 4, these updated rules can be seen in figure 6.2.

Figure 6.2 Constraint generation rules for local declarations (new core) : $\mathsf{ExtLabSynt}_{\mathtt{loc}} \times \mathbb{P}(\mathsf{Var}) \to \mathsf{Env}$

Declaration

(G29) $[\![\mathtt{local}^{l}\ \mathtt{dec}_1\ \mathtt{in}\ \mathtt{dec}_2\ \mathtt{end}]\!] = \exists ev.ev = [\![dec_1]\!]; \mathtt{loc}\ ev^{l}\ \mathtt{in}\ [\![dec_2]\!]$

6.1.4 Constraint Solving

In figure 6.3, the constraint solver is extended with three rules to handle local declarations. (Due to the new stack mechanism in the new version of the Skalpel core defined in section 4.6.3, some of the rules dealing with the error case can be removed as they are no longer necessary.)

Figure 6.3 Extension to constraint solving rules to support local declarations

local environments

(L1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \mathtt{loc}\ e_1\ \mathtt{in}\ e_2) \quad \to \quad \mathtt{succ}(\Delta),$
$\text{if } \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1) \to^* \mathtt{succ}(\langle u',\ e'\rangle)$
$\wedge\ \mathtt{slv}(\langle u',\ e'\rangle, \overline{d}, e_2) \to^* \mathtt{succ}(\langle u'',\ e''\rangle)$
$\wedge\ \Delta = \langle u'',\ e; e' \backslash e''\rangle$

(L2) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \mathtt{loc}\ e_1\ \mathtt{in}\ e_2) \quad \to \quad \mathtt{err}(er),$
$\text{if } \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1) \to^* \mathtt{succ}(\langle u',\ e'\rangle)$
$\wedge\ \mathtt{slv}(\langle u',\ e'\rangle, \overline{d}, e_2) \to^* \mathtt{err}(er)$

(L3) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \mathtt{loc}\ e_1\ \mathtt{in}\ e_2) \quad \to \quad \mathtt{err}(er),$
$\text{if } \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1) \to^* \mathtt{err}(er)$

During the solving of a local declaration, the environment e_1 must be solved first, and if that is solvable the environment e_2 is solved; the binders in e_1 are visible to the environment e_2 and then use environment difference to remove the binders in e_1 so they are not exported from the environment given after the local declarations have been solved.

Note that in rules (L2) and (L3), if solving of either of the environments contained in a **loc** environment leads to an error, then the constraint solver is terminated in the error state, with the parameter of the error that was raised during the recursion of the constraint solver.

For the new presentation of Skalpel, we must update the constraint solving rules.

To represent correctly the handling of environments, the *stackAction* must first be extended, and a new case for $\mathtt{isSucc}'$ must be added as shown below. With this extension to the stack machinery, when a local environment is encountered the new stack action (`loc`) is pushed on to the stack, as well as the environment e_2. Next e_1 is solved. After e_1 has been solved, e_2 is then solved, after which the environment

e_1 is discarded, so that its bindings are not exported. The `newPush` form is added to StackEv to allow us to determine when the environments on the stack need to be modified in order to control which environments to export correctly.

$$\begin{array}{lll} \mathsf{StackEv} & ::= & \ldots \mid \texttt{newPush} \\ \mathsf{StackAction} & ::= & \ldots \mid \texttt{loc} \end{array}$$

$$\begin{array}{lll} \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}@\langle\langle \texttt{newPush}, \overline{d}, \texttt{new}, x\rangle\rangle) & \rightarrow & \texttt{isSucc}'(\overrightarrow{e}@\langle\top\rangle, d, \overline{m}, \overrightarrow{st}, x) \\ \texttt{isSucc}'(\overrightarrow{e}@\langle e_1, e_2, e_3\rangle, d, \overline{m}, \overrightarrow{st}, \texttt{loc}) & \rightarrow & \texttt{isSucc}(\overrightarrow{e}@\langle e_1; e_3\rangle, \overline{m}, \overrightarrow{st}) \end{array}$$

The constraint solving additions which have been modified to work with the latest version of the Skalpel core can be seen in figure 6.4. It can be seen in this rule that the stack is modified to hold the new stack action, and the environment e_2, which will both be dealt with after the solving of e_1 is complete.

Figure 6.4 Constraint solving extension for local declarations (new core) : State $\rightarrow$ State

(L1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{slv}(\overrightarrow{e@\langle\top\rangle}, \overline{d}, \overline{m}, \overrightarrow{st}', e_1)$,
if $\overrightarrow{st}' = \overrightarrow{st}@\langle\langle \texttt{new}, \overline{d}, \texttt{new}, \texttt{loc}\rangle\rangle@\langle\langle \texttt{newPush}, \overline{d}, \texttt{new}, e_2\rangle\rangle$

6.1.5 Constraint Filtering (Minimisation and Enumeration)

Constraint filtering is extended as follows. This allows us to perform filtering in both of the environments contained in a **loc** environment.

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

Note that this extension is different from the semantics given in [Rah10] in order to fix a bug where e_2 was not filtered correctly.

6.1.6 Slicing

The tree syntax is first updated for programs below:

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

The toTree function must also be extended to handle local declarations. This extension can be seen below.

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

6.1.7 Minimality

It should be noted that the extra initially generated labelled environment variable in the constraint generation rule presented in this chapter is necessary. This labelled environment variable forces the binders of the locally defined environment to be dependent on the label of the local declaration as a whole . If this labelled environment variable was not used, typable slices could be produced as a result as the environment variable could not be sliced out.

6.2 Type declarations

6.2.1 External Syntax

This extension to the theory adds support for detecting errors involving type declarations in Standard ML, which take one type variable argument. Theory from the previous extension, local declarations, can be used in order to handle binder control for support of type declarations.

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\mathtt{loc}}$ with the additional extensions in this section. This set will be refereed to as $\mathsf{ExtLabSynt}_{\mathtt{TypDec}}$, which $\mathcal{P}^{L}{}_{\mathtt{TypDec}}$ ranges over.

The external syntax is extended as shown below.

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

Consider the program shown in figure 6.5, which shows an example of a program which has a type error where type functions are involved:

Figure 6.5 Erroneous SML program involving the use of type functions

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

This example is not typable as the datatype constructor `U` is defined to take an argument of `'a -> 'a -> 'a`, but it is applied to the identity function `'a -> 'a`.

6.2.2 Constraint Syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\mathtt{loc}}$ with the extensions listed in this section. This set is referred to as $\mathsf{IntLabSynt}_{\mathtt{TypDec}}$, which $\mathcal{C}^{L}{}_{\mathtt{TypDec}}$ ranges over.

The constraint syntax is extended below to represent what *pseudo type functions*, which are constraint terms of the form $\Lambda\alpha.\,\tau$. These pseudo type functions are considered normal type functions only when:

- the constraints on τ have been solved;

- τ is fully built, that is, $\mathcal{U}(\tau) = \tau$.

Internal type constructors are extended to contain pseudo type variables:

$$\mu \in \mathsf{ITyCon} ::= ... \mid \Lambda\alpha.\,\tau$$

In such a pseudo type function, the parameter α can be connected to τ via some constraints. Consider the initial constraints generated for `type 'a t = 'a` (some constraints have been omitted for clarity):

$$(\delta = \Lambda\alpha_1.\,\alpha_2); \texttt{loc}\,(\downarrow\texttt{'a}{=}\alpha_1)\,\texttt{in}\,(\uparrow\texttt{'a}{=}\alpha_2; \downarrow t{=}\delta)$$

It is important to note here that the internal type constructor $\Lambda\alpha_1.\,\alpha_2$ is only a type function via constraints. If no constraint is filtered out at constraint solving then the binder $\downarrow t{=}\forall\varnothing.\,\Lambda\alpha_1.\,\alpha_1$ will be generated, where $\Lambda\alpha_1.\,\alpha_1$ is a type function.

Quantification over internal type constructors is allowed with the following declaration:

$$\kappa \in \mathsf{TyConSem} ::= \mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa, \overline{d}\rangle$$

A modification is also made to type constructor binders as shown below.

$$\downarrow tc{=}\mu \xrightarrow{\mathsf{Bind}} \downarrow tc{=}\kappa$$

Here quantification over internal type constructors is allowed because internal type variables can now occur inside internal type constructors through the use of pseudo type functions. As an example, given the code fragment:

$$\texttt{type 'a t =}\ \langle..\rangle$$

the type function $\Lambda\alpha_1.\,\alpha_2$ is generated at constraint solving. The internal type variable α_2 is not bound, and so it needs to be quantified so that it will be renamed for accessors to `t`. Eventually the following binder is generated[1]:

$$\downarrow t{=}\forall\{\alpha_2\}.\,\Lambda\alpha_1.\,\alpha_2$$

[1]Dependencies are ignored here for readability purposes.

Figure 6.6 Constraint generation rules for type functions

Datatype names$(dn \triangleright \langle \delta, \alpha, e_1, e_2 \rangle)$

(G13) $\lceil tv\ \ tc \rceil^l \triangleright \langle \delta, \alpha, \downarrow tc \stackrel{l}{=} \delta, \downarrow tv \stackrel{l}{=} \alpha \rangle$

Declarations$(dec \triangleright e)$

(G18) $\texttt{datatype}\ dn \stackrel{l}{=} cb \triangleright (ev = ((\delta \stackrel{l}{=} \gamma);(\alpha_2 \stackrel{l}{=} \alpha_1\ \gamma);e_1;\texttt{loc}\ e_1'\ \texttt{in}\ \texttt{poly}(e_2)));ev^l$
$\Leftarrow dn \triangleright \langle \delta, \alpha_1, e_1, e_1' \rangle \wedge cb \triangleright \langle \alpha_2,\ e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \gamma, ev)$

(G30) $\texttt{type}\ dn \stackrel{l}{=} ty \triangleright (ev = ((\delta \stackrel{l}{=} \Lambda\alpha_1.\ \alpha_2);\texttt{loc}\ e_1'\ \texttt{in}\ (e_2;e_1)));ev^l$
$\Leftarrow dn \triangleright \langle \delta, \alpha_1, e_1, e_1' \rangle \wedge ty \triangleright \langle \alpha_2,\ e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

For example, without quantification of α_2, the following piece of code would generate an error, as that internal type variable would be bound to both the types int and bool.

```
type 'a t = ⟨..⟩
val x = 5 : int t
val y = true : bool t
```

By replacing the sliced out section of code with `'a`, this code becomes typable.

Some forms are also defined to state side conditions in the extension of the constraint solver presented later in this section. They are shown below:

$$\begin{array}{llll} tyf & \in & \mathsf{TyFun} & ::= \quad \delta \mid \Lambda\alpha.\ \tau \mid \langle tyf, \overline{d} \rangle \\ app & \in & \mathsf{App} & ::= \quad \tau\ tyf \end{array}$$

Finally, application of a substitution to a constraint term is extended:

$$(\Lambda\alpha.\ \tau)[sub] = \Lambda\alpha.\ \tau[\{\alpha\} \lhd sub], \text{if } \alpha \notin \mathsf{vars}(\{\alpha\} \lhd sub)$$

6.2.3 Constraint Generation

Changes are presented to the constraint generation algorithm in order to handle type functions in figure 6.6.

Note the order that environments are generated in rules (G18) and (G30). In the former, e_1 is generated *before* e_2 as datatype declarations can be recursive, whereas it is generated the other way round in the latter rule, as type declarations are not recursive.

The sets ShallowTyCon and LabCs are introduced which are defined as follows:

$$\begin{array}{lcll} stc & \in & \mathsf{ShallowTyCon} & ::= \quad \gamma \mid \Lambda\alpha.\,\alpha' \\ lc & \in & \mathsf{LabCs} & ::= \quad \cdots \mid \delta \stackrel{l}{=} stc \end{array}$$

To support the new version of the core, we must update the constraint generator as shown in figure 6.7.

Figure 6.7 Constraint generation rules for type functions (new core) : $\mathsf{ExtLabSynt}_{\mathtt{TypDec}} \times \mathbb{P}(\mathsf{Var}) \to \mathsf{Env}$

`Datatype names`

(G13) $[\![\lceil \texttt{tv}\ tc \rceil^l, \{\alpha, \delta\}]\!] = \langle \downarrow tc \stackrel{l}{=} \delta, \downarrow tv \stackrel{l}{=} \alpha \rangle$

`Declarations`

(G18) $[\![\texttt{datatype}\ dn \stackrel{l}{=} cb]\!] = \exists \langle \alpha_1, \alpha_2, \delta, \gamma \rangle . (ev = ((\delta \stackrel{l}{=} \gamma); (\alpha_2 \stackrel{l}{=} \alpha_1 \gamma); [\![dn, \{\alpha, \delta\}]\!](0);$
$\texttt{loc}\,[\![dn, \{\alpha, \delta\}]\!](1)\,\texttt{in}\,\texttt{poly}([\![cb, \alpha_2]\!]))); ev^l$

(G30) $[\![\texttt{type}\ dn \stackrel{l}{=} ty]\!] = \exists \langle \alpha_1, \alpha_2, \delta \rangle . (ev = ((\delta \stackrel{l}{=} \Lambda\alpha_1.\,\alpha_2);$
$\texttt{loc}\,[\![dn, \{\alpha, \delta\}]\!](1)\,\texttt{in}\,([\![ty, \alpha_2]\!]; [\![dn, \{\alpha, \delta\}]\!](0)))); ev^l$

6.2.4 Constraint Solving

The build function is updated to handle the new type function form, which is shown below.

$$\mathsf{build}(u, \Lambda\alpha.\,\tau) \quad = \quad \Lambda\alpha'.\,\mathsf{build}(u, \tau), \text{ if } \mathsf{build}(u, \alpha) = \alpha'$$

In the new presentation of the core, we do not need such an extension as our unifier set is defined differently.

The function which calculates the free variables in a term is updated to handle this new form, given below.

$$\begin{array}{lcl} \mathsf{freevars}(\alpha) & = & \{\alpha\} \setminus \mathsf{Dum} \\ \mathsf{freevars}(\tau_1 \to \tau_2) & = & \mathsf{freevars}(\tau_1) \cup \mathsf{freevars}(\tau_2) \\ \mathsf{freevars}(\tau\,\mu) & = & \mathsf{freevars}(\mu) \cup \mathsf{freevars}(\tau) \\ \mathsf{freevars}(\Lambda\alpha.\,\tau) & = & \mathsf{freevars}(\tau) \setminus \{\alpha\} \\ \mathsf{freevars}(x^{\overline{d}}) & = & \mathsf{freevars}(x) \\ \mathsf{freevars}(x) & = & \emptyset, \text{ if none of the above applies} \end{array}$$

Changes are presented to the constraint solving machinery in figure 6.8 in order to support type functions. This is updated to support the new version of the Skalpel core in figure 6.9.

Figure 6.8 Constraint solving rules for type functions

equality simplification

(S9) $\mathtt{slv}(\Delta, \overline{d}, \tau_2\,\mu = \tau) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, \tau'[\{\alpha \mapsto \tau_2\}] = \tau)$,
if $\mathsf{collapse}(\mu^{\emptyset}) = (\Lambda\alpha.\,\tau_1)^{\overline{d}'} \wedge \tau' = \mathsf{build}(\Delta, \tau_1^{\overline{d}'})$

(S10) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \tau_1\,\mu = \tau) \quad \rightarrow \quad \mathtt{succ}(\langle u,\ e\rangle)$,
if $\mathsf{collapse}(\mu^{\emptyset}) = \delta^{\overline{d}'} \wedge \delta \notin \mathsf{dom}(u)$

(S11) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \tau_1\,\mu = \tau) \quad \rightarrow \quad \mathtt{slv}(\langle u,\ e\rangle, \overline{d} \cup \overline{d}', \tau_1\,\mu' = \tau)$,
if $\mathsf{collapse}(\mu^{\emptyset}) = \delta^{\overline{d}'} \wedge u(\delta) = \mu'$

(S12) $\mathtt{slv}(\Delta, \overline{d}, \tau_1\,\mu_1 = \tau_2\,\mu_2) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}_1 \cup \overline{d}_2, \gamma_1 = \gamma_2; \tau_1 = \tau_2)$,
if $\mathsf{collapse}(\mu_1^{\overline{d}}) = \gamma_1^{\overline{d}_1} \wedge \mathsf{collapse}(\mu_2^{\emptyset}) = \gamma_2^{\overline{d}_2}$

(S13) $\mathtt{slv}(\Delta, \overline{d}, \tau_1 = \tau_2) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, \mu = \mathtt{arr})$,
if $\{\tau_1, \tau_2\} = \{\tau\,\mu, \tau_0 \to \tau_0'\} \wedge \mathsf{strip}(\mu) \in \mathsf{TyConName}$

equality constraint reversing

(R) $\mathtt{slv}(\Delta, \overline{d}, x = y) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, y = x)$,
if $s = \mathsf{Var} \cup \mathsf{Dependent} \cup \mathsf{App} \wedge y \in s \wedge x \notin s$,

binders

(B) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, {\downarrow}id{=}x) \quad \rightarrow \quad \mathtt{succ}(\langle u,\ e\rangle; ({\downarrow}id \stackrel{\overline{d}}{=} x))$, if $id \notin \mathsf{TyCon}$

(B6) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, {\downarrow}tc{=}\mu) \quad \rightarrow \quad \mathtt{succ}(\langle u,\ e\rangle; ({\downarrow}tc \stackrel{\overline{d}}{=} \forall\overline{\alpha}.\,\mu'))$,
if $\mu' = \mathsf{build}(u, \mu) \wedge \overline{\alpha} = \mathsf{freevars}(\mu')$

accessors

(A1) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', v = x[ren])$,
if $\Delta(id) = (\forall\overline{v}.\,x)^{\overline{d}'} \wedge \mathsf{dom}(ren) = \overline{v} \wedge \mathsf{dj}(\mathsf{vars}(\langle\Delta, v\rangle), \mathsf{ran}(ren))$

(A2) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, v = x)$,
if $\Delta(id) = x \wedge \mathsf{strip}(x)$ is not of the form $\forall\overline{v}.\,x$

The constraint solving rules are updated for accessors so that they handle universal quantification of internal type constructors and type schemes, and give a new rule which creates binders for universally quantified type constructors.

Before reducing applications of type functions to arguments, arguments on the body of the type function have been dealt with. Consider the two following environments, where $\gamma_1 \neq \gamma_2$:

$$\begin{array}{l}\text{Let}\ \ e_1\ \ \text{be}\ \ ((\alpha_1\,\gamma_1) = (\alpha_2\,\gamma_2)\,(\Lambda\alpha'.\,\alpha)); (\alpha = \alpha') \\ \text{Let}\ \ e_2\ \ \text{be}\ \ (\alpha = \alpha'); ((\alpha_1\,\gamma_1) = (\alpha_2\,\gamma_2)\,(\Lambda\alpha'.\,\alpha))\end{array}$$

When solving e_1, the first part of the composition environment is solved first and no error is generated for this. We throw away $\alpha_2\,\gamma_2$ and α' is constrained to be equal to $\alpha_1\,\gamma_1$. Then, $\alpha = \alpha'$ is solved, and α is then also made to be equal to $\alpha_1\,\gamma_1$, though again no error is generated. When dealing with the first constraint of e_1, it is not yet know if the constraint $\alpha = \alpha'$ and it is not yet know if more constraints on α' are present.

Figure 6.9 Constraint solving rules for type functions (new core) : State → State

equality simplification

(S9) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_2\ \mu = \tau) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau'[\{\alpha \mapsto \tau_2\}] = \tau)$
if $\mathsf{collapse}(\mu^{\varnothing}) = (\Lambda\alpha.\,\tau_1)^{\overline{d'}} \wedge\ \tau' = \mathcal{U}(\tau_1^{\overline{d'}})$

(S10) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1\ \mu = \tau) \rightarrow \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st})$
if $\mathsf{collapse}(\mu^{\varnothing}) = \delta^{\overline{d'}} \wedge\ \delta \notin \mathsf{dom}(\mathcal{U})$

(S11) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1\ \mu = \tau) \rightarrow \texttt{slv}(\overrightarrow{e}, d \cup \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1\mu' = \tau)$
if $\mathsf{collapse}(\mu^{\varnothing}) = \delta^{\overline{d'}} \wedge\ \mathcal{U}(\delta) = \mu'$

(S12) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1\ \mu_1 = \tau_2\ \mu_2) \rightarrow \texttt{slv}(\overrightarrow{e}, d \cup \overline{d}, \overline{m}, \overrightarrow{st}, \gamma_1 = \gamma_2; \tau_1 = \tau_2)$
if $\mathsf{collapse}(\mu_1^{\overline{d}}) = \gamma_1^{\overline{d}_1} \wedge\ \mathsf{collapse}(\mu_2^{\varnothing}) = \gamma_2^{\overline{d}_2}$

(S13) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1 = \tau_2) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \mu = \texttt{arr})$
if $\{\tau_1, \tau_2\} = \{\tau\ \mu, \tau_0 \rightarrow \tau_0'\} \wedge\ \mathsf{strip}(\mu) \in \mathsf{TyConName}$

equality constraint reversing

(R) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, x = y) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, y = x)$
if $s = \mathsf{Var} \cup \mathsf{Dependent} \cup \mathsf{App} \wedge\ y \in s\ \wedge\ x \notin s$

binders

(B1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}id{=}x) \rightarrow \texttt{isSucc}(\overrightarrow{e}; {\downarrow}id \overset{\overline{d}}{=} x, \overline{m}, \overrightarrow{st})$
if $id \notin \mathsf{TyCon}$

(B6) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}tc{=}\mu) \rightarrow \texttt{isSucc}(\overrightarrow{e}; ({\downarrow}tc \overset{\overline{d}}{=} \forall\overline{\alpha}.\,\mu'), \overline{m}, \overrightarrow{st})$
if $\mu' = \mathcal{U}(\mu)\ \wedge\ \overline{\alpha} = \mathsf{freevars}(\mu')$

accessors

(A1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\uparrow}id{=}v) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, v = x[ren])$
if $\mathcal{U}(id) = \forall\overline{v}.\,x^{d'} \wedge\ \mathsf{dom}(ren) = \overline{v} \wedge\ \mathsf{dj}(\mathsf{vars}(\langle\mathcal{U}, \overrightarrow{e}, v\rangle), \mathsf{ran}(ren))$

(A2) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\uparrow}id{=}v) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, v = x)$
if $\mathcal{U}(id) = x \wedge\ \mathsf{strip}(x)$ not of the form $\forall\overline{v}.\,x$

When solving e_2, an error is generated by the constraint solver. As the semantics of solving e_1 and e_2 should be the same, an attempt is made to eliminate environments of the form of e_1. This is done in the initial constraint generator by generating $(\alpha = \alpha')$ before $((\alpha_1\,\gamma_1) = (\alpha_2\,\gamma_2)\,(\Lambda\alpha.\,\alpha'))$.

Constraints of the form $\tau_1\,\delta = \tau$ where δ is unconstrained (see rule (S10)) are thrown away. This is done because δ could be of the form $\Lambda\alpha.\,\tau$ where the internal type variable does not occur in τ. Note that as such constraints are discarded, all the constraints on δ must be generated before $\tau_1\,\delta = \tau$ from the initial constraint generator and handled strictly before $\tau_1\,\delta = \tau$ during solving.

6.2.5 Slicing

First, the dot terms in DatName are updated:

$$\texttt{dot-e}(\overrightarrow{term}) \overset{\mathsf{DatName}}{\rightarrow} \texttt{dot-n}(\overrightarrow{term})$$

A new rule for terms of the form $\texttt{dot-n}(\overrightarrow{term})$ is given below.

$$(\mathsf{G31})\,\texttt{dot-n}(\langle \overrightarrow{term}_1, \ldots, \overrightarrow{term}_n \rangle) \triangleright \langle \delta, \alpha, \top, [e_1; \cdots; e_n] \rangle \Leftarrow \overrightarrow{term}_1 \triangleright e_1 \wedge \cdots \wedge \overrightarrow{term}_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \delta, \alpha)$$

The tree syntax for programs is also updated, which is given below.

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

The getDot function is updated as follows (the function now returns a `dotN` marker when applied to a `datname` node):

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

Finally, toTree is updated to handle type declarations:

$$\begin{array}{lcl} \mathsf{toTree}(\texttt{type}\ dn \stackrel{l}{=} ty) & = & \langle \langle \texttt{dec}, \texttt{decTyp} \rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(ty) \rangle \rangle \\ \mathsf{toTree}(\texttt{dot-n}(\overrightarrow{term})) & = & \langle \texttt{dotN}, \mathsf{toTree}(\overrightarrow{term}) \rangle \end{array}$$

6.3 Type annotations

6.3.1 External Syntax

In Standard ML, variables can be constrained by annotating them with a type. In this case, the necessary constraints should be generated so that such a variable is constrained to be of that type. The theory presented in this section achieves this.

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\texttt{TypDec}}$ with the additional extensions in this section. This set is referred to as $\mathsf{ExtLabSynt}_{\texttt{TypAnn}}$, which $\mathcal{P}^{L}{}_{\texttt{TypAnn}}$ ranges over.

The external syntax is extended as shown below. This handles annotations which can take place in expressions and in patterns.

$$\begin{array}{lcl} \mathsf{Exp} & ::= & \cdots \mid exp : {}^{l} ty \\ \mathsf{Pat} & ::= & \cdots \mid pat : {}^{l} ty \end{array}$$

An example of a program using type annotations is given below.

```
val rec g : unit -> unit = fn x => x
val u = g true
```

Without the type annotation on `g`, this code fragment would be typable. With this type annotation however, `g` is forced to take an argument of type `unit` and return an argument of type `unit`, which makes this program erroneous.

Two new sets are defined below, $\mathsf{LabTyVar}$ and $\mathsf{TyVarSeq}$.

$$\begin{array}{lcllcl} ltv & \in & \mathsf{LabTyVar} & ::= & tv_1^l \mid \texttt{dot-d}(\overrightarrow{term}) \\ tvseq & \in & \mathsf{TyVarSeq} & ::= & ltv \mid \epsilon_{\texttt{v}}^l \mid (ltv_1, \ldots, ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term}) \end{array}$$

In order to distinguish between explicit type variables in type variables sequences and occurrences in types, they are subscripted in explicit type variables (tv_1^l).

In order that type variable sequences can occur in value declarations, the definitions for these are updated below.

$$\begin{array}{l} \texttt{val}\ pat \stackrel{l}{=} exp \overset{\mathsf{Dec}}{\rightarrow} \texttt{val}\ tvseq\ pat \stackrel{l}{=} exp \\ \texttt{val rec}\ pat \stackrel{l}{=} exp \overset{\mathsf{Dec}}{\rightarrow} \texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp \end{array}$$

Consider the following untypable piece of code:

```
val rec 'a f = fn x =>
  let val rec g : 'a -> 'a = fn x => x
  in g true
  end
```

As `'a` is bound in the outer environment of the declaration of `g`, it is not generalised when generalising the type of `g`. When applying `g` to `true`, the non-generalised explicit type variable `'a` clashes with the type `bool`. Removing the binding of `'a` in the outer environment, or changing the type of `g` to be `'b -> 'b` will make the code typable.

6.3.2 Constraint Syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\texttt{TypDec}}$ with the extensions listed in this section. This set is referred to as $\mathsf{IntLabSynt}_{\texttt{TypAnn}}$, which ${\mathcal{C}^L}_{\texttt{TypAnn}}$ ranges over.

In the below extensions to sets representing binders and environments, the notion of *unconfirmed* binders is introduced.

$$\begin{array}{llll} \mathit{bind} & \in & \mathsf{Bind} & ::= \quad \cdots \mid {\downarrow}\mathit{tv}{=}\beta \\ e & \in & \mathsf{Env} & ::= \quad \cdots \mid \texttt{or}(e, \overline{d}) \end{array}$$

The symbol ${\downarrow}x$ is used to represent that x is an *unconfirmed* binder. Such binders are needed because for example in the last code example presented an unconfirmed binder is generated for `'a` in the declaration of `g`. If at constraint solving it is discovered that there exists a binder of the same name in the outer environment, then this binder is discarded.

The form $\texttt{or}(e, \overline{d})$ differs from $e^{\overline{d}}$; when using the form $\texttt{or}(e, \overline{d})$ only one of the dependencies in the set $\overline{d}$ needs to be satisfied for e to be satisfied. To abbreviate, $e^{\vee\overline{d}}$ is written for $\texttt{or}(e, \overline{d})$.

6.3.3 Constraint Generation

The set $\mathsf{labtyvars} : \mathsf{ExtLabSynt} \times \mathbb{P}(\mathsf{Var}) \to \mathbb{P}(\mathsf{ExtLabSynt})$ is defined which computes the set of labelled explicit type variables in an explicit type as follows:

$$
\begin{aligned}
&\mathsf{labtyvars}(vid^{l}_{\mathtt{e}}) && = \varnothing \\
&\mathsf{labtyvars}(\mathtt{let}^{l}\ dec\ \mathtt{in}\ exp\ \mathtt{end}) && = \mathsf{labtyvars}(exp) \\
&\mathsf{labtyvars}(\mathtt{fn}\ pat \stackrel{l}{=} exp) && = \mathsf{labtyvars}(pat) \cup \mathsf{labtyvars}(exp) \\
&\mathsf{labtyvars}(\lceil exp\ \ atexp \rceil^{l}) && = \mathsf{labtyvars}(exp) \cup \mathsf{labtyvars}(atexp) \\
&\mathsf{labtyvars}(exp : {}^{l} ty) && = \mathsf{labtyvars}(exp) \cup \mathsf{labtyvars}(ty) \\
&\mathsf{labtyvars}(vid^{l}_{\mathtt{p}}) && = \varnothing \\
&\mathsf{labtyvars}(vid^{l}\, atpat) && = \mathsf{labtyvars}(atpat) \\
&\mathsf{labtyvars}(pat : {}^{l} ty) && = \mathsf{labtyvars}(pat) \cup \mathsf{labtyvars}(ty) \\
&\mathsf{labtyvars}(tv^{l}) && = \{tv^{l}\} \\
&\mathsf{labtyvars}(ty_1 \stackrel{l}{\rightarrow} ty_2) && = \mathsf{labtyvars}(ty_1) \cup \mathsf{labtyvars}(ty_2) \\
&\mathsf{labtyvars}(\lceil ty\ \ ltc \rceil^{l}) && = \mathsf{labtyvars}(ty)
\end{aligned}
$$

This function does not extract *all* the explicit type variables occurring in an expression of a pattern, for example it does not extract the explicit type variables occurring in nested declarations (see case for let-expressions).

The function $\mathsf{labtyvarsdec} : \mathsf{TyVarSeq} \times \mathsf{Pat} \times \mathsf{Exp} \rightarrow \mathbb{P}(\mathsf{ExtLabSynt})$ is defined as shown below. Note that in this definition, the $tvseq, pat$ and exp are from those of recursive value declarations. This function gathers type variables in the pattern and expression arguments, not occurring in the type variable sequence, and annotates them with as many labels as they apply to.

$$\mathsf{labtyvarsdec}(tvseq, pat, exp) = \{tv^{\overline{l}} \mid f(tv) = \overline{l}\}, \text{ where}$$
$$f = \uplus\{tv \mapsto \{l\} \mid tv \text{ does not occur in } tvseq \wedge tv^{l} \in \mathsf{labtyvars}(pat) \cup \mathsf{labtyvars}(exp)\}.$$

In rules (G48)-(G50), explicit type variable binders are constructed. These are confirmed binders ($\downarrow tv{=}\beta$) and not unconfirmed binders ($\downarrow tv{=}\beta$) as type variable sequences are not context dependent. Unconfirmed binders are generated in rule (G17). These are generated after the confirmed binders, and this order is necessary. Note that the order the unconfirmed binders are generated in does not matter as the explicit type variables are all unique anyway.

In order to update the constraint generation for the new version of the Skalpel core, we present the new version of the constraint solving rules in figure 6.11.

6.3.4 Constraint Solving

The updated constraint solver is given in figure 6.12.

Rule (OR) only picks one dependency from the dependency set labelling an environment of the form $e^{\vee\overline{d}}$ because only one of them is needed for the constraint

Figure 6.10 Constraint generation rules for type annotations

Expressions$(exp \triangleright \langle \alpha,\ e \rangle)$

(G46) $exp : {}^{l}ty \triangleright \langle \alpha,\ e_1;e_2;(\alpha \stackrel{l}{=} \alpha_1);(\alpha \stackrel{l}{=} \alpha_2) \rangle \Leftarrow$
$exp \triangleright \langle \alpha_1,\ e_1 \rangle \wedge ty \triangleright \langle \alpha_2,\ e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

Patterns$(pat \triangleright \langle \alpha,\ e \rangle)$

(G47) $pat : {}^{l}ty \triangleright \langle \alpha,\ e_1;e_2;(\alpha \stackrel{l}{=} \alpha_1);(\alpha \stackrel{l}{=} \alpha_2) \rangle \Leftarrow$
$pat \triangleright \langle \alpha_1,\ e_1 \rangle \wedge ty \triangleright \langle \alpha_2,\ e_2 \rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

Labelled type variables$(ltv \triangleright e)$

(G48) $tv_1^l \triangleright \downarrow tv \stackrel{l}{=} \beta$

Type variable sequences$(tvseq \triangleright e)$

(G49) $\epsilon_{\mathsf{v}}^{l} \triangleright \top$

(G50) $(ltv_1, \ldots, ltv_n)^l \triangleright e_1;\cdots;e_n \Leftarrow ltv_1 \triangleright e_1 \wedge \cdots \wedge ltv_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n)$

Declarations$(dec \triangleright e)$

(G17) $\mathtt{val\ rec}\ tvseq\ pat \stackrel{l}{=} exp \triangleright (ev = e');ev^l$
$\Leftarrow tvseq \triangleright e_0 \wedge pat \triangleright \langle \alpha_1,\ e_1 \rangle \wedge exp \triangleright \langle \alpha_2,\ e_2 \rangle$
$\wedge e' = \mathtt{poly}(\mathtt{loc}\ e_0;e\ \mathtt{in}\ (e_1;e_2;(\alpha_1 \stackrel{l}{=} \alpha_2)))$
$\wedge \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n} \{tv_i^{\overline{l}_i}\}$
$\wedge e = ((\downarrow tv_1 \stackrel{l}{=} \beta_1)^{\vee \overline{l}_1};\cdots;(\downarrow tv_n \stackrel{l}{=} \beta_n)^{\vee \overline{l}_n})$
$\wedge \mathsf{dja}(e_0, e_1, e_2, ev, \beta_1, \ldots, \beta_n)$

(G45) $\mathtt{val}\ tvseq\ pat \stackrel{l}{=} exp \triangleright (ev = e');ev^l$
$\Leftarrow tvseq \triangleright e_0 \wedge pat \triangleright \langle \alpha_1,\ e_1 \rangle \wedge exp \triangleright \langle \alpha_2,\ e_2 \rangle$
$\wedge \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n} \{tv_i^{\overline{l}_i}\}$
$\wedge e = ((\downarrow tv_1 \stackrel{l}{=} \beta_1)^{\vee \overline{l}_1};\cdots;(\downarrow tv_n \stackrel{l}{=} \beta_n)^{\vee \overline{l}_n})$
$\wedge e' = \mathtt{expans}(\mathtt{loc}\ e_0;e\ \mathtt{in}\ (e_2;e_1;(\alpha_1 \stackrel{l}{=} \alpha_2)), \mathsf{expansive}(exp))$
$\wedge \mathsf{dja}(e_0, e_1, e_2, ev, \beta_1, \ldots, \beta_n)$

represented by the dependency set to be satisfied (this is listed in [Rah10] as a different rule but has been corrected in this thesis). Any dependency from $\overline{d}$ can be chosen.

Our version of this constraint solver which supports the latest version of the Skalpel core is given in figure 6.13.

6.3.5 Constraint Filtering (Minimisation and Enumeration)

The filt function is extended to support environments with dependencies where only one needs to be kept for the environment to stay alive during filtering below. This thesis presents a slight variation on this theory from that existing in [Rah10] to remove a case involving a feature which was defined later in that document, and so is not relevant for the presentation in this thesis.

Figure 6.11 Constraint generation rules for type annotations (new core) : $\mathsf{ExtLabSynt}_{\texttt{TypAnn}} \rightarrow \mathsf{Env}$

Expressions

(G46) $[\![exp : {}^{l} ty, \alpha]\!] = \exists\langle\alpha_1\alpha_2\rangle.[\![exp, \alpha_1]\!];[\![ty, \alpha_2]\!];(\alpha \stackrel{l}{=} \alpha_1);(\alpha \stackrel{l}{=} \alpha_2)$

Patterns

(G47) $[\![pat : {}^{l} ty, \alpha]\!] = \exists\langle\alpha_1\alpha_2\rangle.[\![pat, \alpha_1]\!];[\![ty, \alpha_2]\!];(\alpha \stackrel{l}{=} \alpha_1);(\alpha \stackrel{l}{=} \alpha_2)$

Labelled type variables

(G48) $[\![ty^{l}_{1}]\!] = \downarrow ty \stackrel{l}{=} \alpha$

Type variable sequences

(G49) $[\![\epsilon^{l}_{\mathtt{v}}]\!] = \top$

(G50) $[\![(ltv_1, ...ltv_n)^{l}]\!] = [\![ltv_1]\!];...;[\![ltv_n]\!]$

Declarations

(G17) $[\![\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp]\!] = \exists\langle\alpha_1, \alpha_2, ev\rangle.(ev =$
$\quad \texttt{poly}(\texttt{loc}\,[\![tvseq]\!];\, e\,\texttt{in}\,([\![pat, \alpha_1]\!];[\![exp, \alpha_2]\!];(\alpha_1 \stackrel{l}{=} \alpha_2))));\, ev^{l}$
$\quad \wedge\ \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n}\{tv_i^{\overline{l}_i}\}$
$\quad \wedge\ e = ((\downarrow tv_1 \stackrel{l}{=} \alpha_1)^{\vee\overline{l}_1};...;(\downarrow tv_n \stackrel{l}{=} \alpha_n)^{\vee\overline{l}_n})$
$\quad \wedge\ \mathsf{dja}(\alpha_1, ...\alpha_n)$

Figure 6.12 Constraint solving rules to handle type annotations

binders

(B9)	$\texttt{slv}(\Delta, \overline{d}, \downarrow tv{=}\beta)$	$\rightarrow$	$\texttt{succ}(\Delta;(\downarrow tv \stackrel{\overline{d}}{=} \beta))$,	if $\Delta(tv)$ is undefined
(B10)	$\texttt{slv}(\Delta, \overline{d}, \downarrow tv{=}\beta)$	$\rightarrow$	$\texttt{succ}(\Delta)$,	if $\Delta(tv)$ is defined

or environments

(OR)	$\texttt{slv}(\Delta, \overline{d}, e^{\vee\{d\}\cup\overline{d}'})$	$\rightarrow$	$\texttt{slv}(\Delta, \overline{d} \cup \{d\}, e)$	

The function which generates dummy variables is also updated to support the new notion of unconfirmed binders.

$$\mathsf{filt}(e^{\vee\overline{l}}, \overline{l}_1, \overline{l}_2) = \begin{cases} \mathsf{filt}(e, \overline{l}_1, \overline{l}_2)^{\vee\overline{l}'}, & \text{if } \overline{l}' = \overline{l} \cap (\overline{l}_1 \setminus \overline{l}_2) \neq \emptyset \\ \mathsf{dum}(\mathsf{strip}(e)), & \text{if } \mathsf{dj}(\overline{l}, \overline{l}_1 \setminus \overline{l}_2) \text{ and } \neg\mathsf{dj}(\overline{l}, \overline{l}_2) \\ \top, & \text{otherwise} \end{cases}$$

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

6.3.6 Slicing

As in the other extensions presented in this chapter, the tree syntax is updated to handle type annotations below:

$$\begin{array}{lll} \mathsf{Class} & ::= & \cdots \mid \texttt{labtyvar} \mid \texttt{tyvarseq} \\ \mathsf{Prod} & ::= & \cdots \mid \texttt{expTyp} \mid \texttt{patTyp} \mid \texttt{tyvarseqEm} \mid \texttt{tyvarseqSeq} \end{array}$$

The getDot function is then updated with type variable sequences:

Figure 6.13 Constraint solving rules for type annotations (new core) : State → State

binders

(B9) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \downarrow tv = \alpha) \rightarrow \texttt{isSucc}(\overrightarrow{e}; (\downarrow tv \stackrel{\overline{d}}{=} \alpha), \overline{m}, \overrightarrow{st})$
if $\overrightarrow{e}(tv)$ is undefined

(B10) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \downarrow tv = \alpha) \rightarrow \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st})$
if $\overrightarrow{e}(tv)$ defined

or environments

(OR) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e^{\vee\{d\}\cup\overline{d}'}) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d} \cup \{d\}, \overline{m}, \overrightarrow{st}, e)$

$$\begin{aligned} \mathsf{getDot}(\langle \texttt{labtyvar}, prod \rangle) &= \texttt{dotD} \\ \mathsf{getDot}(\langle \texttt{tyvarseq}, prod \rangle) &= \texttt{dotD} \end{aligned}$$

Figure 6.14 Extension of the conversion function from *terms* to *trees* to deal with type annotations and type variable sequences

Expressions

$\mathsf{toTree}(exp : {}^{l} ty) = \langle\langle \texttt{exp}, \texttt{expTyp} \rangle, l, \langle \mathsf{toTree}(exp), \mathsf{toTree}(ty) \rangle\rangle$

Patterns

$\mathsf{toTree}(pat : {}^{l} ty) = \langle\langle \texttt{pat}, \texttt{patTyp} \rangle, l, \langle \mathsf{toTree}(pat), \mathsf{toTree}(ty) \rangle\rangle$

Labelled type variables

$\mathsf{toTree}(tv_1^l) = \langle\langle \texttt{labtyvar}, \texttt{id} \rangle, l, \langle tv \rangle\rangle$

Type variable sequences

$\mathsf{toTree}(\epsilon_{\mathrm{v}}^l) = \langle\langle \texttt{tyvarseq}, \texttt{tyvarseqEm} \rangle, l, \langle\rangle\rangle$

$\mathsf{toTree}((ltv_1, \ldots, ltv_n)^l) =$
$\langle\langle \texttt{tyvarseq}, \texttt{tyvarseqSeq} \rangle, l, \mathsf{toTree}(\langle ltv_1, \ldots, ltv_n \rangle)\rangle$

The toTree function is extended in figure 6.14 to handle expressions and patterns which have type annotations.

6.4 Signatures

In this extension to the existing theory, Skalpel can be extended to support signatures. Note that this extension does not present any machinery to handle the following:

- Equality type specifications;
- Include specifications;
- Type sharing specifications.

The above extensions are presented later in sections 7.1, 7.4 and section 7.5 respectively.

There are some kinds of errors that are not handled in this section e.g. unmatched errors, where an identifier is specified in a signature but is not present in a structure which is constrained by that signature. There is an extension to handle this kind of error but it is not present in this document. Readers who wish to see how this is handled should see section 14.8 of [Rah10].

6.4.1 External Syntax

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\texttt{TypAnn}}$ with the additional extensions in this section. This set will be referred to as $\mathsf{ExtLabSynt}_{\texttt{Sig}}$, which ${\mathcal{P}^L}_{\texttt{Sig}}$ ranges over.

The external syntax is updated to include support for signatures as shown below.

sigid	$\in$	SigId		(signature identifiers)
sigdec	$\in$	SigDec	::=	`signature` *sigid* $\stackrel{l}{=}$ *sigexp*
			$\mid$	`dot-d`$(\overrightarrow{term})$
sigexp	$\in$	SigExp	::=	$sigid^l \mid$ `sig`^{l} $spec_1 \cdots spec_n$ `end`
			$\mid$	`dot-s`$(\overrightarrow{term})$
spec	$\in$	Spec	::=	`val` *vid* $:^l$ *ty*
			$\mid$	`type` dn^l
			$\mid$	`datatype` *dn* $\stackrel{l}{=}$ *cd*
			$\mid$	`structure` *strid* $:^l$ *sigexp*
			$\mid$	`dot-d`$(\overrightarrow{term})$
cd	$\in$	ConDesc	::=	$vid^l_{\mathsf{c}} \mid$ *vid* `of`^{l} *ty*
			$\mid$	`dot-e`$(\overrightarrow{term})$
id	$\in$	Id	::=	$\cdots \mid$ *sigid*
strexp	$\in$	StrExp	::=	$\cdots \mid$ *strexp* $:^l$ *sigexp* $\mid$ *strexp* $:>^l$ *sigexp*
topdec	$\in$	TopDec	::=	*strdec* $\mid$ *sigdec*
prog	$\in$	Program	::=	$topdec_1 ; \cdots ; topdec_n$

There are two types of signature constraint represented by this syntax.

1. Opaque signature constraints, written as *strexp* $:>^l$ *sigexp*
2. Translucent signature constraints, written as *strexp* $:^l$ *sigexp*

A structure which is constrained by a signature (either in an opaque or translucent manner) must implement all the specifications that are in the signature it is constrained by, and can define any additional identifiers in addition to this.

The distinction between a translucent and opaque signature constraint is that if a type constructor is defined in the signature of an opaque signature constraint, then the use of the type constructor in the structure constrained by that signature does not constrain the definition in the signature.

An example demonstrating the difference between opaque and translucent signatures can be seen in figure 6.15. In this figure, the difference between the structures `T1` and `T2` is that `T1` is constrained by the signature `s` in an opaque way, whereas `T2` is constrained by the same signature in a translucent way. There is an error in this example - the application `f true` in the declaration of `u1` is an error because the function `f` takes an argument of type `t` and *not* of type `bool`. In `u2` the same application is not part of any error as the function `f` used there takes a `bool` as an argument.

Figure 6.15 Example showing difference between opaque and translucent signatures

```
1  signature s =
2  sig
3    type t
4    val f : t -> t
5  end
6
7  structure S =
8  struct
9    type t = bool
10   val rec f = fn x => x
11 end
12
13 structure T1 = S :> s
14 structure T2 = S : s
15
16 val u1 = let open T1 in f true end
17 val u2 = let open T2 in f true end
```

6.4.2 Constraint Syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\mathtt{TypAnn}}$ with the extensions listed in this section. This set is referred to as $\mathsf{IntLabSynt}_{\mathtt{Sig}}$, which $\mathcal{C}^L{}_{\mathtt{Sig}}$ ranges over.

The constraint syntax is extended in figure 6.16.

Figure 6.16 Constraint syntax for signatures

β	$\in$	RigidTyVar		(set of rigid type variables)
svar	$\in$	SVar	$::=$	$v \mid \beta$
ρ	$\in$	FRTyVar	$::=$	$\alpha \mid \beta$
sig	$\in$	SigSem	$::=$	$e \mid \forall\overline{\delta}.\, e \mid \langle sig, \overline{d}\rangle$
bind	$\in$	Bind	$::=$	$\ldots \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	$::=$	$\ldots \mid \downarrow sigid{=}ev$
τ	$\in$	ITy	$::=$	$\ldots \mid \beta$
μ	$\in$	ITyCon	$::=$	$\ldots \mid \mathtt{tv}$
subty	$\in$	SubTy	$::=$	$\sigma \preceq_{vid} \sigma_2 \mid \kappa_1 \preceq_{tc} \kappa_2$
e	$\in$	Env	$::=$	$\ldots \mid e_1 : e_2 \mid \mathtt{ins}(e) \mid subty$

In this new constraint syntax rigid type variables are introduced, written β. These act as constant types but can be renamed and quantified over, so they are referred to as variables, but as they are considered as constant types they are not allowed to be equal to e.g. arrow types. As these rigid type variables are not allowed to be in the domain of the unifiers, they are not a member of the set SVar. It is for this reason that the set SVar is introduced (where "S" stands for "substitutable") as in the instantiation of type schemes β variables are allowed to be renamed. The pre-existing type variables of the form α are now referred to as the *flexible* type

variables, and the set FRTyVar contains both kinds of variable.

The definition of $\mathsf{atoms}(x)$ is extended to be the set of syntactic forms belonging to $\mathsf{SVar} \cup \mathsf{TyConName} \cup \mathsf{Dependency}$ and occurring in x whatever x is. Let $\mathsf{svars}(x)$ be $\mathsf{atoms}(x) \cap \mathsf{SVar}$. The forms of explicit type variable binders and type schemes are extended as follows:

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

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

$$\forall \overline{\alpha}.\, \tau \xrightarrow{\mathsf{Scheme}} \forall \overline{\rho}.\, \tau$$

In order to allow instantiation of different universally quantified forms, renamings are redefined:

$$\begin{aligned} ren \in \mathsf{Ren} = \{ren \quad & \mid ren = f_1 \cup f_2 \\ & \wedge\ f_1 \in \mathsf{FRTyVar} \to \mathsf{ITyVar} \\ & \wedge\ f_2 \in \mathsf{TyConVar} \to \mathsf{TyConVar} \\ & \wedge\ ren \text{ is injective} \\ & \wedge\ \mathsf{dj}(\mathsf{dom}(ren), \mathsf{dom}(ren))\} \end{aligned}$$

Note that flexible and rigid type variables are both renamed to flexible ones. So instantiating the type scheme $\forall\{\alpha\}.\, \alpha \to \alpha$ or the type scheme $\forall\{\beta\}.\, \beta \to \beta$ both result in a type of the form $\alpha' \to \alpha'$.

Substitutions are also redefined as shown below:

$$sub \in \mathsf{Sub} = \{sub \mid sub = \mathcal{U} \cup f \wedge\ \ f \in \mathsf{RigidTyVar} \to \mathsf{ITy}\}$$

Substitution on constraint terms is also updated as shown below.

$$svar[sub] = \begin{cases} x, & \text{if } sub(svar) = x \\ svar, & \text{otherwise} \end{cases}$$

The set *ins* is also defined to handle `ins` environments, as defined below.

$$ins \in \mathsf{Ins} = \{f \mid f \in \mathsf{TyConVar} \to \mathsf{TyConName} \wedge\ \ f \text{ is injective}\}$$

The new environment form defined in this extension $\texttt{ins}(e)$ is an instance of e where

internal type constructor variables are instantiated to internal type constructor names. The definition of *ins* defined above provides this instantiation.

The constraints of the form $\sigma \preceq_{vid} \sigma_2$ and $\kappa_1 \preceq_{tc} \kappa_2$ are subtyping constraints. In the definition of Standard ML [MTHM98], checking that one type is a subtype of another results in building a new type scheme built from σ_1 and σ_2. It is important to note therefore that $\sigma \preceq_{vid} \sigma_2$ can be considered as both a constraint and as an environment because it constrains σ_1 to be a subtype of σ_2 and can result in the generation of a binder of the form $\downarrow vid{=}\sigma$ at constraint solving, where σ is a new type scheme that has been generated. These are used when handling constraints of the form $e_1 : e_2$ and are used to check signature constraints on structures.

In Skalpel, $\forall\overline{\alpha_1}.\,\tau_1$ is a subtype of $\forall\overline{\alpha_2}.\,\tau_2$ iff $\tau_1[ren_1]$ can be made equal to $\tau_2[ren]$ for some ren_1 and ren_2, where ren_1 renames **both** the flexible and rigid type variables of τ_1 to fresh flexible type variables, and where ren_2 **only** renames the flexible type variables of τ_2 to fresh flexible variables. Note that ren_2 does not rename any rigid type variables because in the case of type schemes the rigid type variables are not allowed to be more specific whereas flexible type variables can be constrained further.

Allowing explicit type variables to be rigid terms helps to catch *too general* errors, where a structure defines a type which is less general in the structure than it is in the signature. While βs have not been renamed, they are considered as constant types with which is associated the name `tv`.

The role of rigid type variables are shown using the code presented in figure 6.17. With this code, the enumeration algorithm would find the error that `x` is defined in the signature `s` as a boolean, but inferred to be an integer in the structure definition. Given this, the minimisation algorithm will attempt to slice out the type `bool` out of the specification of `x`. This would mean that `x` has the form $\forall\{\alpha\}.\,\alpha$ in its specification and would lead to the *too general* signatures error to be generated. The issue is being able to tell whether this kind of error should be generated because the signature is really too general, or whether the type scheme is too general because information has been discarded. In order to avoid this, explicit type variables occurring in a signature are not bound to flexible type variables but to rigid type variables.

Figure 6.17 Example showing the use of the rigid type variables

```
1 signature s = sig val x : bool end
2 structure S = struct val x = 1 end
3 structure T = S :> s
```

Let $\mathsf{rigtyvars}(\mathcal{C}^L)$ be the set of rigid type variables occurring in $\mathcal{C}^L$, and let $\mathsf{tyconvars}(\mathcal{C}^L)$ be the set of internal type constructor variables occurring in $\mathcal{C}^L$. Let the function $\mathsf{tyvars} : \mathsf{ExtLabSynt}_{\mathtt{Sig}} \to \mathbb{P}(\mathsf{ExtLabSynt}_{\mathtt{Sig}})$ which computes the set of explicit type variables occurring in an explicit type, be defined as below.

$$\mathsf{tyvars}(ty) = \{ty \mid ty^l \in \mathsf{labtyvars}(ty)\}$$

The application of a substitution to a constraint term is extended below.

$$\begin{array}{lcl} x_1 \preceq_{id} x_2[sub] & = & x_1[sub] \preceq_{id} x_2[sub] \\ (e_1 : e_2)[sub] & = & e_1[sub] : e_2[sub] \\ \mathtt{ins}(e)[sub] & = & \mathtt{ins}(e[sub]) \end{array}$$

6.4.3 Constraint Generation

Extensions to the constraint generation algorithm is given in figure 6.18, which add support for signature related constraints.

The version of the constraint generator which supports for the new core is given in figure 6.19.

Figure 6.19 Constraint generation rules for signatures (new core - 1 of 2) : $\mathsf{ExtLabSynt}_{\mathtt{Sig}} \times \mathbb{P}(\mathsf{Var}) \to \mathsf{Env}$

Signature declarations

(G32) $[\![\mathtt{signature}\ sigid \stackrel{l}{=} sigexp]\!] = \exists\langle ev, ev'\rangle.ev' = ([\![sigexp, ev]\!]; \downarrow sigid \stackrel{l}{=} ev); ev'^l$

Signature expressions

(G33) $[\![sigid^l, ev]\!] = \uparrow sigid \stackrel{l}{=} ev$

(G34) $[\![\mathtt{sig}^l\ spec_1 \cdots spec_n\ \mathtt{end}, ev]\!] = \exists ev'.(ev \stackrel{l}{=} ev'); (ev' = ([\![spec_1]\!]; ...; [\![spec_n]\!]))$

Specifications

(G35) $[\![\mathtt{val}\ vid\ :\ ty]\!] = \exists\alpha.(ev = \mathtt{poly}(\mathtt{loc} \downarrow tv_1 \stackrel{l}{=} \beta_1 ... \downarrow tv_n \stackrel{l}{=} \beta_n\ \mathtt{in}$
$([\![ty, \alpha]\!]; \downarrow vid \stackrel{l}{=} \langle \alpha, \mathtt{v}\rangle))) \Leftarrow \mathsf{tyvars}(ty) = \{tv_1, ..., tv_n\}$

(G36) $[\![\mathtt{type}\ dn^l]\!] = \exists\langle\alpha, \delta\rangle.(ev = [\![dn, \{\alpha, \delta\}]\!](0)); ev^l$

(G37) $[\![\mathtt{structure}\ strid\ :^l\ sigexp]\!] = \exists\langle ev, ev'\rangle.(ev' = ([\![sigexp, ev]\!]; (\downarrow strid \stackrel{l}{=} ev))); ev'^l$

(G38) $[\![\mathtt{datatype}\ dn \stackrel{l}{=} cd]\!] = \exists\langle\delta, \alpha_1, \alpha_2, ev\rangle.(ev = ((\alpha_2 \stackrel{l}{=} \alpha_1\ \delta_1); e_1;$
$\mathtt{loc}\ e_1'\ \mathtt{in}\,\mathtt{poly}([\![cd, \alpha_2]\!]))); ev^l \wedge [\![dn, \{\delta, \alpha_1\}]\!] = \langle e_1, e_1'\rangle$

Structure expressions

(G39) $[\![strexp\ :^l\ sigexp, ev]\!] = \exists\langle ev_1, ev_2\rangle.[\![sigexp, ev_2]\!]; [\![strexp, ev_1]\!]; (ev \stackrel{l}{=} ev_1 : ev_2)$

(G40) $[\![strexp\ :>^l\ sigexp, ev]\!] = \exists\langle ev_1, ev_2, ev_3\rangle.[\![sigexp, ev_2]\!]; [\![strexp, ev_1]\!]; e; (ev \stackrel{l}{=}$
$\mathtt{ins}(ev_2)) \wedge e = (ev_3 \stackrel{l}{=} ev_1 : ev_2)$

Figure 6.18 Constraint generation rules for signatures

Signature declarations$(sigdec \triangleright e)$

(G32) $\texttt{signature}\ sigid \stackrel{l}{=} sigexp \triangleright ev' = (e;{\downarrow}sigid \stackrel{l}{=} ev);ev'^{l}$
$\Rightarrow\ sigexp \triangleright \langle ev,\ e\rangle \wedge \mathsf{dja}(e, ev')$

Signature expressions$(sigexp \triangleright \langle ev,\ e\rangle)$

(G33) $sigid^{l} \triangleright \langle ev, {\uparrow}sigid \stackrel{l}{=} ev\rangle$

(G34) $\texttt{sig}^{l}\ spec_1 \cdots spec_n\ \texttt{end} \triangleright \langle ev,\ (ev \stackrel{l}{=} ev');(ev' = (e_1;\cdots;e_n))\rangle$
$\Leftarrow\ spec_1 \triangleright e_1 \wedge \cdots \wedge spec_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev, ev')$

Specifications$(spec \triangleright e)$

(G35) $\texttt{val}\ vid\ :^{l}\ ty \triangleright (ev = e_2);ev^{l}$
$\Leftarrow\ ty \triangleright \langle \alpha,\ e\rangle \wedge \mathsf{tyvars}(ty) = \{tv_1, \ldots, tv_n\} \wedge \mathsf{dja}(e, ev, \beta_1, \ldots, \beta_n)$
where $e_2 = \texttt{poly}(\texttt{loc}\,{\downarrow}tv_1 \stackrel{l}{=} \beta_1;\cdots;{\downarrow}tv_n \stackrel{l}{=} \beta_n\ \texttt{in}\ (e;{\downarrow}vid \stackrel{l}{=} \langle \alpha, \mathtt{v}\rangle))$

(G36) $\texttt{type}\ dn^{l} \triangleright (ev = e);ev^{l} \Leftarrow dn \triangleright \langle \delta, \alpha, e, e'\rangle \wedge \mathsf{dja}(e, e', ev)$

(G37) $\texttt{structure}\ strid\ :^{l}\ sigexp \triangleright e_2 \Leftarrow sigexp \triangleright \langle ev,\ e\rangle \wedge \mathsf{dja}(e, ev')$
where $e_2 = (ev' = (e;({\downarrow}strid \stackrel{l}{=} ev)));ev'^{l}$

(G38) $\texttt{datatype}\ dn \stackrel{l}{=} cd \triangleright (ev = ((\alpha_2 \stackrel{l}{=} \alpha_1\ \delta_1);e_1;\texttt{loc}\ e_1'\ \texttt{in}\ \texttt{poly}(e_2)));ev^{l}$
$\Leftarrow\ dn \triangleright \langle \delta_1, \alpha_1, e_1, e_1'\rangle \wedge cd \triangleright \langle \alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \gamma, ev)$

Structure expressions$(strexp \triangleright \langle ev,\ e\rangle)$

(G39) $strexp\ :^{l}\ sigexp \triangleright \langle ev,\ e_2;e_1;(ev \stackrel{l}{=} ev_1{:}ev_2)\rangle$
$\Leftarrow\ strexp \triangleright \langle ev_1,\ e_1\rangle \wedge sigexp \triangleright \langle ev_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

(G40) $strexp\ :>^{l}\ sigexp \triangleright \langle ev,\ e_2;e_1;(ev_{\mathtt{dum}} \stackrel{l}{=} ev_1{:}ev_2);(ev \stackrel{l}{=} \texttt{ins}(ev_2))\rangle$
$\Leftarrow\ strexp \triangleright \langle ev_1,\ e_1\rangle \wedge sigexp \triangleright \langle ev_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, ev)$

Programs$(prog \triangleright e)$

(G41) $topdec_1\,;\cdots;\,topdec_n \triangleright e_1;\cdots;e_n$
$\Leftarrow\ topdec_1 \triangleright e_1 \cdots \wedge topdec_n \triangleright e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev)$

Figure 6.20 Constraint generation rules for signatures (new core - 2 of 2) : $\mathsf{ExtLabSynt}_{\mathtt{Sig}} \times \mathbb{P}(\mathsf{Var}) \to \mathsf{Env}$

Labelled type variables

(G48) $[\![ty_1^{l}]\!] = {\downarrow}ty \stackrel{l}{=} \beta$

Declarations

(G17) $[\![\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp]\!] = \exists\langle \alpha_1, \alpha_2, ev\rangle.(ev =$
$\texttt{poly}(\texttt{loc}\,[\![tvseq]\!];\, e\ \texttt{in}\,([\![pat, \alpha_1]\!];[\![exp, \alpha_2]\!];(\alpha_1 \stackrel{l}{=} \alpha_2))));\, ev^{l}$
$\wedge\ \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n}\{tv_i^{\bar{l}_i}\}$
$\wedge\ e = ((\lfloor tv_1 \stackrel{l}{=} \beta_1)^{\vee\bar{l}_1};\ldots;(\lfloor tv_n \stackrel{l}{=} \beta_n)^{\vee\bar{l}_n})$
$\wedge\ \mathsf{dja}(\beta_1, \ldots\beta_n)$

Programs

(G41) $[\![topdec_1, \ldots, topdec_n]\!] = [\![topdec_1]\!], \ldots, [\![topdec_n]\!]$

Type variable binders are edited below to bind to ρ variables instead of α variables.

$$ {\downarrow}tv \stackrel{l}{=} \alpha \overset{\mathsf{LabBind}}{\rightarrow} {\downarrow}tv \stackrel{l}{=} \rho $$

Rules (G35), (G36), (G37), and (G38) give constraint generation rules for specifications inside signatures. Some aspects of this, for example constraint generation for datatype declarations, can be seen to be very similar to the way that these are handled in the core presentation. As a result of this thesis, the number of supported specifications is extended when handling extensions presented later, such as include specifications, but note that the handling of other extensions with this old presentation of the theory is not shown, which can now be considered deprecated.

Constraint generation (G39) and (G40) show the contrast between constraints generated for programs where opaque signature constraints are used and where programs with translucent constraints are used. Note that in the latter rule, it can be seen that the `ins` form is called to handle opaque constraints, which was discussed in section 6.4.2.

6.4.4 Constraint Solving

The constraint solving states are extended to include a new kind of state, `match`, and add new error kinds to support errors that can occur using signatures.

$$\begin{array}{llll} \mathit{state} & \in \ \mathsf{State} & ::= & \cdots \mid \texttt{match}(\Delta, \overline{d}, e_1, e_2) \\ \mathit{ek} & \in \ \mathsf{ErrKind} & ::= & \cdots \mid \texttt{tyVarClash}(tv_1, tv_2) \mid \texttt{tooGeneral}(\mu_1, \mu_2) \end{array}$$

For the new version of the core, we must edit this definition (shown below) as the solver has changed:

$$\mathit{state} \ \in \ \mathsf{State} \ ::= \ \ldots \mid \texttt{match}(\overrightarrow{e}, \overline{d}, e_1, e_2)$$

The new error kinds are described below.

- `tooGeneral`(μ_1, μ_2) describes errors where there is a signature constraint on a structure where a signature specification is more general than that of the structure.
- `tyVarClash`(tv_1, tv_2) describes errors which exist when explicit type variable used in a signature specification do not directly correspond to that of the structure declaration.

The set of unifiers are extended as shown below (note that this extension also extends Sub):

Figure 6.21 Monomorphic to polymorphic environment function generalising flexible and rigid type variables

$$
\begin{array}{lcl}
\mathsf{toPoly}(\Delta, \downarrow vid{=}\tau) & = & \Delta;(\downarrow vid \overset{\overline{d}}{=} \forall \overline{\rho}.\, \tau'), \\
\quad \text{if} \left\{ \begin{array}{lcl} \tau' & = & \mathsf{build}(\Delta, \tau) \\ \overline{\rho} & = & (\mathsf{vars}(\tau') \cap \mathsf{FRTyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\Delta)) \cup \{\alpha_{\mathtt{dum}}\}) \\ \overline{d} & = & \{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\Delta) \wedge \alpha \in \mathsf{vars}(\tau') \setminus \overline{\rho}\} \end{array} \right. \\
\mathsf{toPoly}(\langle u,\ e\rangle, e_0^{\overline{d}}) & = & \langle u',\ e; e \backslash e'^{\overline{d}} \rangle, \text{ if } \mathsf{toPoly}(\langle u,\ e\rangle, e_0) = \langle u',\ e'\rangle \\
\mathsf{toPoly}(\Delta, e_1; e_2) & = & \mathsf{toPoly}(\Delta', e_2), \text{ if } \mathsf{toPoly}(\Delta, e_1) = \Delta' \\
\mathsf{toPoly}(\Delta, e) & = & \Delta; e, \text{ if none of the above applies}
\end{array}
$$

$$
\begin{array}{rcll}
u \in \mathsf{Unifier} = \{\bigcup_{i=1}^{4} f_i & \mid & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\
& \wedge & f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\
& \wedge & f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\
& \wedge & f_4 \in \mathsf{SigSemVar} \to \mathsf{SigSem}\}
\end{array}
$$

When generating type schemes, both the flexible (α) type variables and the rigid (β) type variables may be quantified over.

The toPoly function is extended in figure 6.21 with the addition that when calculating the type variable set to quantify over, rigid type variables can also be quantified over in addition to the flexible ones.

Now flexible and rigid type variables may be to be quantified over when generating type schemes, so the definitions are updated to achieve this. Let us also define the function $\mathsf{scheme} : \mathsf{Unifier} \times \mathbb{P}(\mathsf{SVar}) \times \mathsf{IntLabSynt}_{\mathtt{Sig}} \to \mathsf{Scheme}$:.

$$
\mathsf{scheme}(\mathcal{U}, \overline{svar}, \mathcal{C}^{L}{}_{\mathtt{Sig}}) = \forall \overline{svar} \cap \mathsf{svars}(\mathcal{C}^{L\prime}{}_{\mathtt{Sig}}).\, \mathcal{C}^{L\prime}{}_{\mathtt{Sig}}, \text{ if } \mathcal{C}^{L\prime}{}_{\mathtt{Sig}} = \mathcal{U}(\mathcal{C}^{L}{}_{\mathtt{Sig}})
$$

There is a need to be able to build up flexible and rigid type variables, and build up compositions of environments for solving rules involving constraints on signatures. This is done in the updated version of build shown below.

$$
\begin{array}{lcl}
\mathsf{build}(u, \downarrow id{=}x) & = & (\downarrow id{=}\mathsf{build}(u, x)) \\
\mathsf{build}(u, e_1; e_2) & = & \mathsf{build}(u, e_1); \mathsf{build}(u, e_2)
\end{array}
$$

The new constraint solver which has been updated to handle constraints generated for signatures is presented in figure 6.22 and figure 6.23. In figure 6.22 rules are shown which deal with the `slv` form and in figure 6.23 rules are shown which deal with the `match` form.

In rules (SM1)-(SM12), the environments generated for signatures and structures which are constrained by those signatures are compared, and check that the dec-

Figure 6.22 Constraint solving for signature related constraints (1)

equality simplification

(S14) $\mathtt{slv}(\Delta, \overline{d}, \tau_1 = \tau_2) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, \mu = \mathtt{tv})$,
if $\{\tau_1, \tau_2\} = \{\tau\,\mu, \beta\} \wedge \mathsf{strip}(\mu) \in \mathsf{TyConName}$

(S15) $\mathtt{slv}(\Delta, \overline{d}, \tau_1 = \tau_2) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d}, \mathtt{tv} = \mathtt{arr})$,
if $\{\tau_1, \tau_2\} = \{\tau_0 \rightarrow \tau_0', \beta\}$

(S16) $\mathtt{slv}(\Delta, \overline{d}, \beta_1 = \beta_2) \quad \rightarrow \quad \mathtt{err}(\langle \mathtt{tyVarClash}(tv_1, tv_2), \overline{d}\rangle)$,
if $\beta_1 \neq \beta_2$

(S17) $\mathtt{slv}(\Delta, \overline{d}, \mu_1 = \mu_2) \quad \rightarrow \quad \mathtt{err}(\langle \mathtt{tooGeneral}(\mu_1, \mu_2), \overline{d}\rangle)$,
if $\{\mu_1, \mu_2\} \in \{\{\mathtt{tv}, \mathtt{arr}\}, \{\mathtt{tv}, \gamma\}\}$

binders

(B) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \downarrow id{=}x) \quad \rightarrow \quad \mathtt{succ}(\langle u,\ e\rangle; (\downarrow id \stackrel{\overline{d}}{=} x))$,
if $id \notin \mathsf{SigId} \cup \mathsf{TyCon}$

(B7) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \downarrow sigid{=}e_1) \quad \rightarrow$
$\mathtt{succ}(\langle u,\ e\rangle; (\downarrow sigid \stackrel{\overline{d}}{=} \forall \mathsf{tyconvars}(e_2).\ e_2))$, if $e_2 = \mathsf{build}(u, e_1)$

instantiations

(I1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \mathtt{ins}(e_0)) \quad \rightarrow \quad \mathtt{succ}(\langle u,\ e; e_1[ins]\rangle)$,
if $\mathsf{build}(u, e_0) = e_1$
$\wedge\ \mathsf{dom}(ins) = \mathsf{tyconvars}(e_1) \wedge \mathsf{dj}(\mathsf{vars}(\langle u,\ e\rangle), \mathsf{ran}(ins))$

signature constraints

(SC1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1{:}e_2) \quad \rightarrow \quad \mathtt{match}(\langle u,\ e\rangle, \overline{d}, \mathsf{build}(u, e_1), \mathsf{build}(u, e_2))$

subtyping constraints

(SU1) $\mathtt{slv}(\Delta, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2) \quad \rightarrow$
$\mathtt{succ}(\langle u',\ e'; \downarrow vid \stackrel{\overline{d}}{=} \mathsf{scheme}(u', \overline{\rho}_1[ren_1] \cup \overline{\rho}_2[ren_2], \tau_2[ren_2])\rangle)$,
if $\forall i \in \{1,2\}.(\sigma_i = \forall \overline{\rho}_i.\ \tau_i \vee (\sigma_i = \tau_i \wedge \overline{\rho}_i = \emptyset \wedge \tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\}$
$\wedge\ \mathsf{dj}(\mathsf{vars}(\Delta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\Delta, \overline{d}, \tau_1[ren_1] = \tau_2[ren_2]) \rightarrow^* \mathtt{succ}(\langle u',\ e'\rangle)$

(SU2) $\mathtt{slv}(\Delta, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2) \quad \rightarrow \quad \mathtt{err}(er)$,
if $\forall i \in \{1,2\}.(\sigma_i = \forall \overline{\rho}_i.\ \tau_i \vee (\sigma_i = \tau_i \wedge \overline{\rho}_i = \emptyset \wedge \tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\}$
$\wedge\ \mathsf{dj}(\mathsf{vars}(\Delta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\Delta, \overline{d}, \tau_1[ren_1] = \tau_2[ren_2]) \rightarrow^* \mathtt{err}(er)$

(SU3) $\mathtt{slv}(\Delta, \overline{d}, \kappa_1 \preceq_{tc} \kappa_2) \quad \rightarrow$
$\mathtt{succ}(\langle u',\ e'; \downarrow tc \stackrel{\overline{d}}{=} \mathsf{scheme}(u', \overline{\alpha}_1[ren_1] \cup \overline{\alpha}_2[ren_2], \mu_2[ren_2])\rangle)$,
if $\forall i \in \{1,2\}.(\kappa_i = \forall \overline{\alpha}_i.\ \mu_i \wedge \mathsf{dom}(ren_i) = \overline{\alpha}_i) \wedge$
$\mathsf{dj}(\mathsf{vars}(\Delta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\Delta, \overline{d}, \mu_1[ren_1] = \mu_2[ren_2]) \rightarrow^* \mathtt{succ}(\langle u',\ e'\rangle)$

(SU4) $\mathtt{slv}(\Delta, \overline{d}, \kappa_1 \preceq_{tc} \kappa_2) \quad \rightarrow \quad \mathtt{err}(er)$,
if $\forall i \in \{1,2\}.(\kappa_i = \forall \overline{\alpha}_i.\ \mu_i \wedge \mathsf{dom}(ren_i) = \overline{\alpha}_i)$
$\wedge\ \mathsf{dj}(\mathsf{vars}(\Delta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\Delta, \overline{d}, \mu_1[ren_1] = \mu_2[ren_2]) \rightarrow^* \mathtt{err}(er)$

(SU5) $\mathtt{slv}(\Delta, \overline{d}, x_1 \preceq_{id} x_2) \quad \rightarrow \quad \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', y_1 \preceq_{id} y_2)$,
if (x_1 is of the form $y_1^{\overline{d}'} \wedge y_2 = x_2$) $\vee$ (x_2 is of the form $y_2^{\overline{d}'} \wedge y_1 = x_1$)

Figure 6.23 Constraint solving for signature related constraints (2)

structure/signature matching

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

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

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

(SM4) $\texttt{match}(\Delta, \overline{d}, e, \downarrow vid{=}\sigma_1) \rightarrow \texttt{slv}(\Delta, \overline{d}, \sigma_2 \preceq_{vid} \sigma_1)$,
if $e(vid) = \sigma_2$

(SM5) $\texttt{match}(\Delta, \overline{d}, e, \downarrow tc{=}\kappa_1) \rightarrow \texttt{slv}(\Delta, \overline{d}, \kappa_2 \preceq_{tc} \kappa_1)$,
if $e(tc) = \kappa_2$

(SM6) $\texttt{match}(\langle u_1,\ e_1\rangle, \overline{d}, e, \downarrow strid{=}e_0) \rightarrow \texttt{succ}(\langle u_2,\ e_1;e'^{\overline{d}}\rangle)$,
if $e(strid) = e'_0 \wedge \texttt{match}(\langle u_1,\ e_1\rangle, \overline{d}, e'_0, e_0) \rightarrow^* \texttt{succ}(\langle u_2,\ e_2\rangle)$
$\wedge\ e' = (\downarrow strid{=}e_1 \backslash e_2)$

(SM7) $\texttt{match}(\Delta, \overline{d}, e, \downarrow strid{=}e_0) \rightarrow \texttt{err}(er)$,
if $\texttt{match}(\Delta, \overline{d}, e(strid), e_0) \rightarrow^* \texttt{err}(er)$

(SM8) $\texttt{match}(\Delta, \overline{d}, e, \downarrow vid{=}is_1) \rightarrow \texttt{succ}(\Delta;(\downarrow vid{=}is))$,
if $e[vid] = is_2 \wedge (\mathsf{solvable}(is_1 \overset{\overline{d}}{=} is_2) \vee$
$\mathsf{strip}(is_1) = \mathtt{v}) \wedge is = \mathsf{ifNotDum}(is_1, is_2^{\overline{d}})$

(SM9) $\texttt{match}(\Delta, \overline{d}, e, \downarrow vid{=}is_1) \rightarrow \texttt{err}(er)$,
if $\mathsf{strip}(is_1) \neq \mathtt{v} \wedge \texttt{slv}(\Delta, \overline{d}, is_1 = e[vid]) \rightarrow^* \texttt{err}(er)$

(SM10) $\texttt{match}(\Delta, \overline{d}, e, \downarrow id{=}x) \rightarrow \texttt{succ}(\Delta;(\downarrow id{=}y))$,
if $e(id)$ is undefined $\wedge\ y = \mathsf{toDumVar}(x)$

(SM11) $\texttt{match}(\Delta, \overline{d}, e, ev) \rightarrow \texttt{succ}(\Delta;ev)$

(SM12) $\texttt{match}(\Delta, \overline{d}, e, e'^{\overline{d}'}) \rightarrow \texttt{match}(\Delta, \overline{d} \cup \overline{d}', e, e')$

larations in the signature are present in the structure definition. This is done irrespective of whether the signature constraint is opaque or translucent in nature.

In rules (SU1)-(SU5), subtyping constraints are handled. Note that in the updated presentation of this theory, in section 6.4, the rules for handling this are made simpler due to the way that error states are dealt with due to the new stack parameter of the constraint solver.

In the newest version of the constraint solver which supports the new code, we also make the following changes. A new *stackAction* value is defined for appending environments as shown below, used in rules (SU1) and (SU3) of the constraint solver:

$$\mathsf{StackAction} ::= \ldots \mid \texttt{append}(e)$$

$$\texttt{isSucc}'(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{append}(e')) \rightarrow \texttt{isSucc}(\overrightarrow{e}; e', \overline{m}, \overrightarrow{st})$$

Figures 6.24 and 6.25 define the constraint solving extensions to handle signatures. Rules (S14) to (S17) handle equality constraints with rigid type variables. Rules (SM1) to (SM12) handle matching structures with signatures, and rules (SU1) to (SU5) handle subtyping constraints.

Figure 6.24 Constraint solving rules to support signatures (new core - 1 of 2) : State → State

equality simplification

(S14) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1 = \tau_2) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \mu = \alpha)$
if $\{\tau_1, \tau_2\} = \{\tau\mu, \beta\} \wedge \mathsf{strip}(\mu) \in \mathsf{TyConName}$

(S15) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1 = \tau_2) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{tv} = \texttt{ar})$
if $\{\tau_1, \tau_2\} = \{\tau_0 \rightarrow \tau_0', \beta\}$

(S16) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \beta_1 = \beta 2) \rightarrow \mathsf{err}(\langle \texttt{tyVarClash}(,, \overline{d})\rangle)$
. if $\beta_1 \neq \beta_2$

(S17) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \mu_1 = \mu_2) \rightarrow \mathsf{err}(\langle \texttt{tooGeneral}(\mu_1, \mu_2), \overline{d}\rangle)$
if $\{\mu_1, \mu_2\} \in \{\{\texttt{tv}, \texttt{ar}\}, \{\texttt{tv}, \gamma\}\}$

binders

(B1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}id{=}x) \rightarrow \texttt{isSucc}(\overrightarrow{e}; ({\downarrow}id \stackrel{\overline{d}}{=} x), \overline{m}, \overrightarrow{st})$
if $id \notin \mathsf{SigId} \cup \mathsf{ITyCon}$

(B7) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}sigid{=}e_1) \rightarrow$
$\texttt{isSucc}(\overrightarrow{e}; ({\downarrow}sigid \stackrel{\overline{d}}{=} \forall \mathsf{tyconvars}(\mathcal{U}(e_1)).\mathcal{U}(e_1)), \overline{m}, \overrightarrow{st})$

(B9) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}tv = \beta) \rightarrow \texttt{isSucc}(\overrightarrow{e}; ({\downarrow}tv \stackrel{\overline{d}}{=} \beta), \overline{m}, \overrightarrow{st})$
if $\overrightarrow{e}(tv)$ is undefined

(B10) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}tv = \beta) \rightarrow \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st})$
if $\overrightarrow{e}(tv)$ defined

instantiations

$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{ins}(e_0)) \rightarrow \texttt{isSucc}(\overrightarrow{e}; e_1[ins], \overline{m}, \overrightarrow{st})$
if $\mathcal{U}(e_0) = e_1 \wedge \mathsf{dom}(ins) = \mathsf{tyconvars}(e_1)$
$\wedge\ \mathsf{dj}(\mathsf{vars}(\mathcal{U}) \cup \mathsf{vars}(\overrightarrow{e}), \mathsf{ran}(ins))$

signature constraints

$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e_1 : e_2) \rightarrow \texttt{match}(\overrightarrow{e}, \overline{d}, \mathcal{U}(e_1), \mathcal{U}(e_2))$

subtyping constraints

(SU1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \sigma_1 \preceq_{vid} \sigma_2) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}@\langle\langle \texttt{new}, \overline{d}, \texttt{new}, \texttt{append}(e_2)\rangle\rangle, e_1)$
if $\forall i \in \{1,2\}.\, (\sigma_i = \forall \overline{\rho}.\, \tau_i \vee (\sigma_i = \tau_i \wedge \rho_i = \varnothing \wedge \tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\}$
$\wedge\ \mathsf{dj}((\mathsf{vars}(\mathcal{U}) \cup \mathsf{vars}(\overrightarrow{e})), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ e_1 = \tau_1[ren] = \tau_2[ren]$
$\wedge\ e_2 = {\downarrow}vid \stackrel{l}{=} \mathsf{scheme}(\mathcal{U}, \overline{\rho}_1[ren_1] \cup \overline{\rho}_2[ren_2], \tau_2[ren_2])$

(SU3) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \kappa_1 \preceq_{tc} \kappa_2) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}@\langle\langle \texttt{new}, \overline{d}, \texttt{new}, \texttt{append}(e_2)\rangle\rangle, e_1)$
if $\forall i \in \{1,2\}.\, (\kappa_i = \forall \overline{\alpha}.\, \mu_i \wedge \mathsf{dom}(ren_i) = \overline{\alpha}_i)$
$\wedge\ \mathsf{dj}((\mathsf{vars}(\mathcal{U}) \cup \mathsf{vars}(\overrightarrow{e})), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ e_1 = \mu_1[ren] = \mu_2[ren]$
$\wedge\ e_2 = {\downarrow}tc \stackrel{l}{=} \mathsf{scheme}(\mathcal{U}, \overline{\alpha}_1[ren_1] \cup \overline{\alpha}_2[ren_2], \mu_2[ren_2])$

(SU5) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, x_1 \preceq_{id} x_2) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d} \cup \overline{d}', \overline{m}, \overrightarrow{st}, y_1 \preceq_{id} y_2)$
if (x_1 is of the form $y_1^{\overline{d}'} \wedge y_2 = x_2$) $\vee$ (x_2 is of the form $y_2^{\overline{d}'} \wedge y_1 = x_1$)

Figure 6.25 Constraint solving rules to support signatures (new core - 2 of 2) : State → State

polymorphic environments

(P1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{poly}(\downarrow vid \stackrel{\overline{d}'}{=} \alpha)) \quad \rightarrow \quad \texttt{isSucc}(\overrightarrow{e}; \sigma, \overline{m}, \overrightarrow{st})$
if $\overline{\rho} = \mathsf{frtyvars}(\mathcal{U}(\alpha)) \backslash \bigcup\{\mathsf{frtyvars}(\mathcal{U}(x)) \mid x \in m\}$
$\wedge\ \overline{d}'' = \overline{d}' \cup \mathsf{deps}(\mathsf{vars}(\mathcal{U}(\alpha)) \triangleleft \{\mathcal{U}(x) \mid x \in \overline{m}\})$
$\wedge\ \sigma = \downarrow vid{=}\langle \forall \overline{\rho}.\mathcal{U}(\alpha), \overline{d}'' \rangle$

structure/signature matching

(SM1) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \top) \quad \rightarrow \quad \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st})$

(SM2) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e_1; e_2) \quad \rightarrow \quad \texttt{match}(\overrightarrow{e}'', \overline{d}, \overline{m}, \overrightarrow{st}, e, e_2)$
if $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e_1) \rightarrow \texttt{succ}(\overrightarrow{e}'')$

(SM3) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e_1; e_2) \quad \rightarrow \quad \mathsf{err}(er)$
if $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e_1) \rightarrow \mathsf{err}(er)$

(SM4) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \downarrow vid{=}\sigma_1) \quad \rightarrow \quad \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \sigma_2 \preceq_{vid} \sigma_1)$
, if $e'(vid) = \sigma_2$

(SM5) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \downarrow tc{=}\kappa_1) \quad \rightarrow \quad \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \kappa_2 \preceq_{vid} \kappa_1)$
, if $e'(tc) = \kappa_2$

(SM6) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \downarrow strid{=}e_0) \quad \rightarrow \quad \texttt{isSucc}(\overrightarrow{e_b}; y^{\overline{d}}, \overline{m}, \overrightarrow{st})$
if $e'(strid) = e_0'$
$\wedge\ \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e_0', e_0) \rightarrow \texttt{succ}(\overrightarrow{e''})$
$\wedge\ \overrightarrow{e''} = \overrightarrow{e_b}@\langle y \rangle$

(SM7) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \downarrow strid{=}e_0) \quad \rightarrow \quad \mathsf{err}(er)$
if $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e'(strid), e_0) \rightarrow \mathsf{err}(er)$

(SM10) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \downarrow id{=}x) \quad \rightarrow \quad \texttt{isSucc}(\overrightarrow{e}; y, \overline{m}, \overrightarrow{st})$
if $\overrightarrow{e}(id)$ is undefined $\wedge y = \mathsf{dum}(\downarrow id{=}x)$

(SM11) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', ev) \quad \rightarrow \quad \texttt{isSucc}(\overrightarrow{e}; ev, \overline{m}, \overrightarrow{st})$

(SM12) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e_1^{\overline{d}'}) \quad \rightarrow \quad \texttt{match}(\overrightarrow{e}, \overline{d} \cup \overline{d}', \overline{m}, \overrightarrow{st}, e', e_1)$

(SM13) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', ev = e'') \quad \rightarrow \quad \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e'')$

(SM14) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \exists a.e'') \quad \rightarrow \quad \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', e''[\{a \rightarrow a'\}])$,
if $a' \notin \mathsf{atoms}(\langle \mathcal{U}, e'' \rangle)$

Let us consider the piece of code in figure 6.26.

The code in this figure is not typable because `c` is declared in the signature to be of type `bool`, whereas in the structure `S` which is constrained by this signature, this value is applied to `()`. In this example, if at attempt is made to try to slice out the specification of `c`, a dummy binding would exist for `c` in the environment of the signature. Consider the case that τ_1 is used instead of τ_2 in rule (SU1) to build the binder for `c` in the environment for the structure `T`. In this case the binder $\downarrow$`c`$=\forall\varnothing.$`bool` is generated, which would clash with the arrow type generated for the application `c ()`, and the following slice would be produced as a result:

Figure 6.26 Clash between `bool` and unit

```
1  signature s =
2  sig
3    val c = bool
4  end
5
6  structure S =
7  struct
8    val c = true
9  end
10
11 structure T = S : s
12 val x = let
13           open T
14         in
15           c ()
16         end
```

$\langle$..`structure S = struct val c = true end`
..`structure T = S :` $\langle$..$\rangle$
..$\langle$..`open T..c ()`..$\rangle$..$\rangle$

It is important to note that this is not a complete type error slice - in fact it is typable! If `S` was constrained by a signature which did not specify `c`, then the last occurrence of `c` would be free and there would therefore not be an error. A minimal slice would be the following:

$\langle$..`signature s = sig val c :` $\langle$..$\rangle$ `end`
..`structure S = struct val c = true end`
..`structure T = S : c`
..$\langle$..`open T..c ()`..$\rangle$..$\rangle$

In rule (S1), from a subtyping constraint of the form $\sigma_1 \preceq_{vid} \sigma_2$ a new type scheme σ is generated and used to create a binder $\downarrow vid{=}\sigma$.

Let us look more closely at the way these new type schemes are generated. Let σ be of the form $\forall\overline{\rho_1}.\tau_1$ and σ_2 be of the form $\forall\overline{\rho_2}.\tau_2$. Fresh instances of τ are generated terms giving τ_1' and τ_2'. The type τ_1' is obtained from τ_1 by renaming the flexible and rigid type variables in $\overline{\rho_1}$. As τ_2 should be not more general than τ_1, **only** the flexible type variables are renamed in ρ_2. It is also verified that τ_1' can be made equal to τ_2'. Finally, σ is generated by building up τ_2' to obtain τ and then by renaming the flexible and rigid type variables in $\overline{\rho_1} \cup \overline{\rho_2}$ and quantifying over those variables in τ.

6.4.5 Constraint Filtering (Minimisation and Enumeration)

The filtering functions are modified in order that it can filter new signature-related environments. There is a need to filter unlabelled environment variables (which is handled with the v case).

$$\begin{array}{lcl} \mathsf{filt}(e_1{:}e_2, \overline{l}_1, \overline{l}_2) & = & \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2){:}\mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2) \\ \mathsf{filt}(\mathtt{ins}(e), \overline{l}_1, \overline{l}_2) & = & \mathtt{ins}(\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)) \\ \mathsf{filt}(v, \overline{l}_1, \overline{l}_2) & = & v \end{array}$$

The definition of toDumVar is edited in order to generate dummy variables for signature environments:

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

6.4.6 Slicing

The syntax for programs is extended below to include signatures and their specifications:

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

The getDot function is also extended to support signature declarations, signature expressions, and specifications:

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

The toTree function is extended in figure 6.27 for signature declarations, both signature and structure expressions, specifications in signatures and for programs.

Figure 6.27 Extension of toTree to deal 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

$\mathsf{toTree}(sigid^l) = \langle\langle \texttt{sigexp}, \texttt{id}\rangle, l, \langle sigid\rangle\rangle$
$\mathsf{toTree}(\texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end}) =$
$\langle\langle \texttt{sigexp}, \texttt{sigexpSig}\rangle, l, \langle \mathsf{toTree}(spec_1), \ldots, \mathsf{toTree}(spec_n)\rangle\rangle$

Specifications

$\mathsf{toTree}(\texttt{val}\ vid\ :^l\ ty) = \langle\langle \texttt{spec}, \texttt{specVal}\rangle, l, \langle vid, \mathsf{toTree}(ty)\rangle\rangle$
$\mathsf{toTree}(\texttt{type}\ dn^l) = \langle\langle \texttt{spec}, \texttt{specTyp}\rangle, l, \langle \mathsf{toTree}(dn)\rangle\rangle$
$\mathsf{toTree}(\texttt{datatype}\ dn \stackrel{l}{=} cd) =$
$\langle\langle \texttt{spec}, \texttt{specDat}\rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(cd)\rangle\rangle$
$\mathsf{toTree}(\texttt{structure}\ strid\ :^l\ sigexp) =$
$\langle\langle \texttt{spec}, \texttt{specStr}\rangle, l, \langle strid, \mathsf{toTree}(sigexp)\rangle\rangle$

Structure expressions

$\mathsf{toTree}(strexp\ :^l\ sigexp) =$
$\langle\langle \texttt{strexp}, \texttt{strexpTr}\rangle, l, \langle \mathsf{toTree}(strexp), \mathsf{toTree}(sigexp)\rangle\rangle$
$\mathsf{toTree}(strexp\ :>^l\ sigexp) =$
$\langle\langle \texttt{strexp}, \texttt{strexpOp}\rangle, l, \langle \mathsf{toTree}(strexp), \mathsf{toTree}(sigexp)\rangle\rangle$

Programs

$\mathsf{toTree}(topdec_1\,\texttt{;}\cdots\texttt{;}\,topdec_n) =$
$\langle \texttt{dotD}, \langle \mathsf{toTree}(topdec_1), \ldots, \mathsf{toTree}(topdec_n)\rangle\rangle$

This chapter has shown how the pre-existing extensions have been modified in order to support the new version of the Skalpel core. In the next chapter, we will show how the new extensions build on top of the work presented both in this chapter and in chapter 3.

Chapter 7

Extensions to Skalpel Core

In chapter 4, the Skalpel core was presented which dealt with a subset of the Standard ML language. There complicated extensions were avoided, which are presented here. The extensions provided in this chapter are listed below.

- Equality types;
- Abstract Type Declarations;
- Duplicate Identifiers in Specifications;
- Include Specifications;
- Type Sharing;
- Infixity.

The first four of these extensions were already attempted in [Rah10] however since there now exists a newer, more polished version of the Skalpel core, these extensions have been updated to be correct for this new version, fixing any bugs where they exist.

7.1 Equality types

This section presents an extension to Skalpel which adds support for equality type errors in Standard ML.

Only equality types in Standard ML may be checked for equality. Examples of types that can are strings[1] (eg `"a"="b"` is false), integers (eg `2=2` is true), and the unit type (eg `()=()` is true).

In Standard ML, an equality type can be defined in signatures by using the `eqtype` keyword, or by using the `type` keyword under certain conditions. Using type annotations with a special kind of type variable will also set an equality type status.

7.1.1 External Syntax

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\mathtt{Sig}}$ with the additional extensions in this section. We will refer to this set as $\mathsf{ExtLabSynt}_{\mathtt{Eqtype}}$, which $\mathcal{P}^{L}{}_{\mathtt{Eqtype}}$ ranges over.

The external syntax is extended to include a new set containing equality type variables, and the `eqtype` keyword in signatures (defined in section 6.4) as follows:

$$\begin{array}{lcll} \mathit{eqtv} & \in & \mathsf{EqTypeVar} & \text{(Equality type variables)} \\ \mathit{spec} & \in & \mathsf{Spec} & ::= \ldots \mid \mathtt{eqtype}\ \mathit{dn}^{l} \end{array}$$

Note that $\mathsf{TyVar} \cap \mathsf{EqTypeVar} = \varnothing$ to avoid ambiguity.

An example untypable program with an equality type error is shown below.

```
structure S : sig eqtype x end =
 struct
    type 'a x = 'a -> 'a
 end;
```

This program is not typable because the type `x` in the structure is declared to be a function type, which is not an equality type, but in the signature the type `x` is constrained to be an equality type.

[1] This assumes the user has not rebound the `string` or `int` types from the default declaration in the basis provided with the compiler in use or the one provided as part of Skalpel.

7.1.2 Constraint Syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\mathtt{Sig}}$ with the modifications to the constraint syntax listed in this section. We refer to this set as $\mathsf{IntLabSynt}_{\mathtt{Eqtype}}$, which ${\mathcal{C}^L}_{\mathtt{Eqtype}}$ ranges over.

We introduce into the constraint syntax equality type variables, and a notion of an equality type status. The set of internal types are also redefined to annotate each internal type variable with an internal equality type variable. Furthermore, we annotate rigid type variables with an equality type variable.

Figure 7.1 Extensions to Syntax of Constraint Terms

θ	$\in$	IEqTypeVar (equality type variables)		
Σ	$\in$	EqTypeStatus	$::=$	EQ_TYPE | NEQ_TYPE | SIG_TYPE
τ	$\in$	ITy	$::=$	$\langle\alpha,\theta\rangle \mid \langle\beta,\theta\rangle \mid \tau\ \mu \mid \tau_1 \rightarrow \tau_2 \mid \langle\tau,\overline{d}\rangle$
μ	$\in$	ITyCon	$::=$	$\ldots \mid \langle\delta,\theta\rangle$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle$
v	$\in$	Var	$::=$	$\ldots \mid \theta$

The symbol Σ representing equality type status values is used to attribute information to expressions which indicate whether they can be checked for equality or not. We need to generate constraints with equality type status information attached so that we are able to detect equality type errors (for example, when an expression is constrained to be both NEQ_TYPE and EQ_TYPE simultaneously there is an equality type error). The SIG_TYPE status is a special status used when handling type declarations in signatures, which is discussed later in this section.

The constraint syntax for pseudo type functions is redefined so that α variables are paired with an equality type variable:

$$tyf \in \mathsf{TyFun} ::= \delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$$

as are substitutions to constraint terms involving pseudo type functions:

$$(\Lambda\langle\alpha,\theta\rangle.\,\tau)[sub] = \Lambda\langle\alpha,\theta\rangle.\,\tau[\{\alpha,\theta\} \lhd sub], \text{if } \{\alpha,\theta\} \not\subseteq \mathsf{vars}(\{\alpha,\theta\} \lhd sub)$$

7.1.3 Constraint Generation

The constraint generation rules can be seen in figures 7.2 and 7.3.

Figure 7.2 Extension of constraint generation rules for equality types (1 of 2) : $\mathsf{ExtLabSynt} \times \mathbb{P}(\mathsf{Var}) \to \mathsf{Env}$

Expressions

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

(G2) $[\![\mathtt{let}^l\ dec\ \mathtt{in}\ exp\ \mathtt{end}, \{\alpha, \theta\}]\!] =$
$[\exists \langle \alpha_2, \theta_2 \rangle . [\![dec]\!]; [\![exp, \langle \alpha_2, \theta_2 \rangle]\!]; (\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_2, \theta_2 \rangle)]$

(G3) $[\![\lceil exp\ \ atexp \rceil^l, \{\alpha, \theta\}]\!] = \exists \langle \alpha_1, \theta_1, \alpha_2, \theta_2 \rangle . [\![exp, \langle \alpha_1, \theta_1 \rangle]\!]; [\![atexp, \langle \alpha_2, \theta_2 \rangle]\!];$
$(\langle \alpha_1, \theta_1 \rangle \stackrel{l}{=} \langle \alpha_2, \theta_2 \rangle \to \langle \alpha, \theta \rangle)$

(G4) $[\![\mathtt{fn}\ pat \stackrel{l}{\Rightarrow} exp, \{\alpha, \theta\}]\!] = [\exists \langle \alpha_1, \theta_1, \alpha_2, \theta_2, ev \rangle . (ev = [\![pat, \langle \alpha_1, \theta_1 \rangle]\!]); ev^l;$
$[\![exp, \langle \alpha_2, \theta_2 \rangle]\!]; (\theta \stackrel{l}{=} \mathsf{NEQ\_TYPE}); (\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle \to \langle \alpha_2, \theta_2 \rangle)]$

(G46) $[\![exp : {}^l ty, \{\alpha, \theta\}]\!] = \exists \langle \alpha_1, \alpha_2, \theta_1, \theta_2 \rangle . [\![exp, \langle \alpha_1, \theta_1 \rangle]\!]; [\![ty, \langle \alpha_2, \theta_2 \rangle]\!];$
$(\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle); (\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_2, \theta_2 \rangle); (\theta_1 \stackrel{l}{=} \theta_2)$

Patterns

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

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

(G8) $[\![\lceil ldcon\ \ atpat \rceil^l, \{\alpha, \theta\}]\!] = \exists \langle \alpha_1, \theta_1, \alpha_2, \theta_2 \rangle . [\![ldcon, \langle \alpha_1, \theta_1 \rangle]\!]; [\![atpat, \langle \alpha_2, \theta_2 \rangle]\!];$
$(\langle \alpha_1, \theta_1 \rangle \stackrel{l}{=} \langle \alpha_2, \theta_2 \rangle \to \langle \alpha, \theta \rangle)$

(G47) $[\![pat : {}^l ty, \langle \alpha, \theta \rangle]\!] = \exists \langle \alpha_1, \alpha_2, \theta_1, \theta_2 \rangle . [\![pat, \langle \alpha_1, \theta_1 \rangle]\!]; [\![ty, \langle \alpha_2, \theta_2 \rangle]\!];$
$(\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle); (\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_2, \theta_2 \rangle); (\theta_1 \stackrel{l}{=} \theta_2)$

Labelled type constructors

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

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

Types

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

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

(G12) $[\![ty_1 \stackrel{l}{\to} ty_2, \{\alpha, \theta\}]\!] = \exists \langle \alpha_1, \theta_1, \alpha_2, \theta_2 \rangle . [\![ty_1, \langle \alpha_1, \theta_1 \rangle]\!]; [\![ty_2, \langle \alpha_2, \theta_2 \rangle]\!];$
$(\theta \stackrel{l}{=} \mathsf{NEQ\_TYPE}); (\langle \alpha, \theta \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle \to \langle \alpha_2, \theta_2 \rangle)$

Datatype names

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

(G59) $[\![\lceil eqtv\ \ tc \rceil^l, \{\alpha, \theta\}]\!] = \langle \exists \delta . \downarrow tc \stackrel{l}{=} \langle \delta, \theta \rangle, (\theta \stackrel{l}{=} \mathsf{EQ\_TYPE}); \downarrow tv \stackrel{l}{=} \langle \alpha, \theta \rangle \rangle$

Figure 7.3 Extension of constraint generation rules for equality types (2 of 2) : $\mathsf{ExtLabSynt} \times \mathbb{P}(\mathsf{Var}) \rightarrow \mathsf{Env}$

Constructor bindings

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

(G16) $[\![dcon\ \texttt{of}^{\,l}\ ty, \{\alpha, \theta\}]\!] = \exists \langle \alpha', \theta', \alpha_1, \theta_1 \rangle . [\![ty, \langle \alpha_1, \theta_1 \rangle]\!](\langle \alpha', \theta' \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle \rightarrow \langle \alpha, \theta \rangle);$
$(\downarrow dcon \stackrel{l}{=} \langle \alpha', \theta' \rangle)$

Declarations

(G17) $[\![\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp]\!] = \exists \langle \alpha_1, \alpha_2, \theta_1, \theta_2, ev \rangle . (ev = \texttt{poly}(\texttt{loc}\,[\![tvseq]\!]; e\ \texttt{in}$
$([\![pat, \langle \alpha_1, \theta_1 \rangle]\!]; [\![exp, \langle \alpha_2, \theta_2 \rangle]\!]; (\langle \alpha_1, \theta_1 \rangle \stackrel{l}{=} \langle \alpha_2, \theta_2 \rangle))))); ev^l$
$\wedge\ \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n} \{tv_i^{\bar{l}_i}\}$
$\wedge\ e = ((\downarrow tv_1 \stackrel{l}{=} \langle \beta_1, \theta_1 \rangle)^{\vee \bar{l}_1}; ...; (\downarrow tv_n \stackrel{l}{=} \langle \beta_n, \theta_n \rangle)^{\vee \bar{l}_n})$
$\wedge\ \mathsf{dja}(\beta_1, ...\beta_n, \theta_1, ...\theta_n)$

(G18) $[\![\texttt{datatype}\ dn \stackrel{l}{=} cb]\!] = \exists \langle \alpha_1, \alpha_2, \alpha_3, \theta_1, \theta_2, \theta_3, \delta, \gamma, ev \rangle . (ev = ((\delta \stackrel{l}{=} \gamma);$
$(\langle \alpha_2, \theta_2 \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle \gamma); [\![dn, \langle \alpha_3, \theta_3 \rangle]\!](0);$
$\texttt{loc}\,[\![dn, \langle \alpha_3, \theta_3 \rangle]\!](1)\,\texttt{in}\,\texttt{poly}([\![cb, \langle \alpha_2, \theta_2 \rangle]\!]))); ev^l$

(G30) $[\![\texttt{type}\ dn \stackrel{l}{=} ty]\!] = \exists \langle \alpha_1, \alpha_2, \alpha_3, \theta_1, \theta_2, \theta_3, \delta, ev \rangle . (ev = ((\delta \stackrel{l}{=} \Lambda \langle \alpha_1, \theta_1 \rangle . \langle \alpha_2, \theta_2 \rangle);$
$\texttt{loc}\,[\![dn, \langle \alpha_3, \theta_3 \rangle]\!](1)\,\texttt{in}\,((\theta_2 \stackrel{l}{=} \theta_3); [\![ty, \langle \alpha_2, \theta_2 \rangle]\!];$
$[\![dn, \langle \alpha_3, \theta_3 \rangle]\!](0)))); ev^l$

Specifications

(G35) $[\![\texttt{val}\ vid\ :\ ty]\!] = \exists \langle \alpha, \theta \rangle . (ev = \texttt{poly}(\texttt{loc} \downarrow tv_1 \stackrel{l}{=} \langle \beta_1, \theta_1 \rangle ... \downarrow tv_n \stackrel{l}{=} \langle \beta_n, \theta_n \rangle\ \texttt{in}$
$([\![ty, \langle \alpha, \theta \rangle]\!]; \downarrow vid \stackrel{l}{=} \langle \alpha, \theta \rangle))) \Leftarrow \mathsf{tyvars}(ty) = \{tv_1, ..., tv_n\}$

(G36) $[\![\texttt{type}\ dn^l]\!] = \exists \{\alpha, \theta, \delta\} . (ev = (\theta \stackrel{l}{=} \mathsf{SIG\_TYPE}); \downarrow y \stackrel{l_2}{=} \langle \delta, \theta \rangle); ev^l$
$\Leftarrow [\![dn, \{\alpha, \theta, \delta\}]\!](0) = \downarrow y \stackrel{l_2}{=} \delta$

(G60) $[\![\texttt{eqtype}\ dn^l]\!] = \exists \langle \alpha, \theta, \delta, ev \rangle . (ev = (\theta \stackrel{l}{=} \mathsf{EQ\_TYPE}); [\![dn, \{\alpha, \theta, \delta\}]\!](0)); ev^l$

(G38) $[\![\texttt{datatype}\ dn \stackrel{l}{=} cd]\!] = \exists \langle \delta, \alpha_1, \alpha_2, \alpha_3, \theta_1, \theta_2, \theta_3, ev \rangle . (ev =$
$((\langle \alpha_2, \theta_2 \rangle \stackrel{l}{=} \langle \alpha_1, \theta_1 \rangle\ \delta_1); [\![dn, \{\alpha_3, \theta_3, \delta\}]\!](0);$
$\texttt{loc}\,[\![dn, \{\alpha_3, \theta_3, \delta\}]\!](1)\,\texttt{in}\,\texttt{poly}([\![cd, \langle \alpha_2, \theta_2 \rangle]\!]))); ev^l$

A number of these constraint generation rules have been updated to annotate internal type variables with an equality type variable, θ. Some have also been updated in order to enforce certain equality type status constraints on the θ variable. For example, in rule (G4), the θ variable in constrained to be of status NEQ_TYPE. The reason for this is that functions cannot be checked for equality. Similarly there are cases where an EQ_TYPE status is enforced. This happens for example in rule (G59), where explicit equality type variables are used, and in rule (G60), where a type is defined in in a signature using the `eqtype` keyword.

The ShallowTyCon set is redefined to pair equality type variables with internal type variables:

$$stc \in \mathsf{ShallowTyCon} ::= \gamma \mid \Lambda\langle\alpha, \theta\rangle.\, \langle\alpha', \theta'\rangle$$

7.1.4 Constraint Solving

The definition of `Unifier` is redefined to handle the unification of equality type variables and status values as follows:

$$\begin{aligned} u \in \text{Unifier} = \{\textstyle\bigcup_{i=1}^{4} f_i \quad \mid \quad & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\ & \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\ & \wedge f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\ & \wedge f_4 \in \mathsf{IEqTypeVar} \to (\mathsf{EqTypeStatus} \cup \mathsf{IEqTypeVar})\} \end{aligned}$$

We also must extend error kinds to represent equality type errors, as can be seen by the extension to the set holding error kinds below.

$$ek \in \mathsf{ErrKind} ::= ... \mid \texttt{equalityTypeErr}(\Sigma_1, \Sigma_2)$$

The set VarE is introduced to ease the definition of the constraint solving rules which carries the same definition as the set Var but with the change that internal type variables are annotated with a θ variable:

$$ve \in \mathsf{VarE} ::= \langle\alpha, \theta\rangle \mid \delta \mid ev$$

In order to allow renaming of θ variables when making binders polymorphic, we must extend the substitution semantics:

$$\langle\tau, \theta\rangle[sub] = \langle\tau[sub], \theta[sub]\rangle$$

The function $\texttt{opaqueEq} : \mathsf{Env} \to \mathsf{Env}$ is used to change the equality type status of SIG_TYPE which is initially assigned to type declarations in signatures after a signature has been checked against its structure. We do not generate a NEQ_TYPE status straight away as if a type were declared as an equality type is a structure, but using the `type` keyword in a structure, we would erroneously generate an equality type error when these declarations are compared.

$$
\begin{array}{lcl}
\texttt{opaqueEq}(\exists x.y) & = & \exists x.\texttt{opaqueEq}(y) \\
\texttt{opaqueEq}(ev = x) & = & ev = \texttt{opaqueEq}(x) \\
\texttt{opaqueEq}(ev) & = & ev \\
\texttt{opaqueEq}(e_1; e_2) & = & \texttt{opaqueEq}(e_1); \texttt{opaqueEq}(e_2) \\
\texttt{opaqueEq}(bind) & = & bind \\
\texttt{opaqueEq}(acc) & = & acc \\
\texttt{opaqueEq}(c) & = & c, \text{ if } c \neq (\theta \stackrel{l}{=} \mathsf{SIG\_TYPE}) \\
\texttt{opaqueEq}(\theta \stackrel{l}{=} \mathsf{SIG\_TYPE}) & = & \theta \stackrel{l}{=} \mathsf{NEQ\_TYPE} \\
\texttt{opaqueEq}(\texttt{poly}(e)) & = & \texttt{poly}(e) \\
\texttt{opaqueEq}(\texttt{loc}\ e_1\ \texttt{in}\ e_2) & = & \texttt{loc}\ e_1\ \texttt{in}\ e_2
\end{array}
$$

We define a new set ITyEqVar as ITyVar ∪ IEqTypeVar.

Figures 7.4 and 7.6 show how the constraint solving rules are extended to support equality types.

Figure 7.4 Extension of constraint solving rules for equality types (1 of 3) : State → State

`equality constraint reversing`

(R) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ct = ct') \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ct' = ct)$,
if $s = \mathsf{VarE} \cup \mathsf{Dependent} \cup \mathsf{App} \wedge ct' \in s \wedge ct \notin s$

`equality simplification`

(S4) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \langle \tau_1 \to \tau_2, \theta_1 \rangle = \langle \tau_3 \to \tau_4, \theta_2 \rangle) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, (\theta_1 = \theta_2); (\tau_1 = \tau_3); (\tau_2 = \tau_4))$

(S9) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_2\ \mu = \tau) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau'[\{\langle \alpha, \theta \rangle \mapsto \tau_2\}] = \tau)$, if $\mathsf{collapse}(\mu^{\varnothing}) = (\Lambda \langle \alpha, \theta \rangle . \tau_1)^{\overline{d}'}$

(S14) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1 = \tau_2) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \mu = \langle \alpha, \theta \rangle)$
if $\{\tau_1, \tau_2\} = \{\tau\mu, \langle \beta, \theta \rangle\} \wedge\ \mathsf{strip}(\mu) \in \mathsf{TyConName}$

(S15) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \tau_1 = \tau_2) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{tv} = \texttt{ar})$, if $\{\tau_1, \tau_2\} = \{\tau_0 \to \tau_0', \langle \beta, \theta \rangle\}$

(S16) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \langle \beta_1, \theta_1 \rangle = \langle \beta_2, \theta_2 \rangle) \rightarrow$
$\mathsf{err}(\langle \texttt{tyVarClash}(,, \overline{d}) \rangle)$, if $\langle \beta_1, \theta_1 \rangle \neq \langle \beta_2, \theta_2 \rangle$

(S28) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \langle \delta_1, \theta_1 \rangle = \langle \delta_2, \theta_2 \rangle) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, (\theta_1 = \theta_2); (\delta_1 = \delta_2))$

Figure 7.5 Extension of constraint solving rules for equality types (1 of 3) : State → State

`binders/empty/dependent/variables`

(B) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\downarrow}id{=}\langle\alpha,\theta\rangle) \quad \rightarrow$
$\texttt{isSucc}(\overrightarrow{e}; {\downarrow}id \stackrel{\overline{d}}{=} \langle\alpha,\theta\rangle, \overline{m} \cup \{\alpha^{\overline{d}}\}, \overrightarrow{st}),$ if $id \notin \mathsf{TyCon}$

(B2) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, bind) \quad \rightarrow$
$\texttt{isSucc}(\overrightarrow{e}; bind^{\overline{d}}, \overline{m}, \overrightarrow{st}),$ if $bind \neq {\downarrow}id{=}\langle\alpha,\theta\rangle$

`accessors`

(A1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\uparrow}id{=}ve) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ve = x[ren])$
if $\mathcal{U}(id) = \forall\overline{v}.\, x^{d'} \wedge\ \mathsf{dom}(ren) = \overline{v} \wedge\ \mathsf{dj}(\mathsf{vars}(\langle\mathcal{U}, \overrightarrow{e}, ve\rangle), \mathsf{ran}(ren))$

(A2) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, {\uparrow}id{=}ve) \rightarrow \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ve = x)$
if $\mathcal{U}(id) = x \wedge\ \mathsf{strip}(x)$ not of the form $\forall\overline{v}.\, x$

`instantiations`

(I1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{ins}(e_0)) \rightarrow \texttt{isSucc}(\overrightarrow{e}; e_2[ins], \overline{m}, \overrightarrow{st})$
if $\mathcal{U}(e_0) = e_1 \wedge\ e_2 = \mathsf{opaqueEq}(e_1) \wedge\ \mathsf{dom}(ins) = \mathsf{tyconvars}(e_2) \wedge$
$\mathsf{dj}(\mathsf{vars}(\mathcal{U}) \cup \mathsf{vars}(\overrightarrow{e}), \mathsf{ran}(ins))$

`structure/signature matching`

(SM5) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', {\downarrow}tc{=}\kappa_1) \rightarrow \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \kappa_2 \preceq_{vid} \kappa_1),$
if $e'(tc) = \kappa_2 \wedge (\kappa_1 \neq \langle\delta,\theta\rangle \vee \kappa_2 \neq \langle\delta',\theta'\rangle$

(SM5) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', {\downarrow}tc{=}\kappa_1) \rightarrow \texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}', e', \kappa_2 \preceq_{vid} \kappa_1),$
if $e'(tc) = \kappa_2 \wedge \kappa_1 = \langle\delta,\theta\rangle \wedge \kappa_2 = \langle\delta',\theta'\rangle \wedge \overrightarrow{st}' = \overrightarrow{st}@\langle\langle\texttt{new}, \overline{d}, \texttt{new}, \theta = \theta'\rangle\rangle$

Figure 7.6 Extension of constraint solving rules for equality types (2 of 3) : State → State

`polymorphic environments`

(P1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{poly}({\downarrow}vid \stackrel{\overline{d}'}{=} \langle\alpha,\theta\rangle)) \rightarrow \texttt{isSucc}(\overrightarrow{e}; \sigma, \overline{m}, \overrightarrow{st}),$
if $\bar{y} = \mathsf{ityeqvars}(\mathcal{U}(\alpha)) \setminus \bigcup\{\mathsf{ityeqvars}(\mathcal{U}(x)) \mid x \in m\}$
$\wedge\ \overline{d}'' = \overline{d}' \cup \mathsf{deps}(\mathsf{vars}(\mathcal{U}(\alpha)) \triangleleft \{\mathcal{U}(x) \mid x \in \overline{m}\})$
$\wedge\ \sigma = {\downarrow}vid{=}\langle\forall(\bar{y} \cup \{\theta\}).\, \langle\mathcal{U}(\alpha),\theta\rangle, \overline{d}''\rangle$

`equality types`

(E1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \langle\alpha,\theta\rangle = \langle\tau,\theta'\rangle) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}@\langle\texttt{new}, \overline{d}, \texttt{new}, \alpha = \tau\rangle, \theta = \theta')$

(E2) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \langle\alpha,\theta\rangle = \langle\tau_1,\theta_1\rangle \rightarrow \langle\tau_2,\theta_2\rangle) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}@\langle\texttt{new}, \overline{d}, \texttt{new}, \alpha = \tau_1 \rightarrow \tau_2\rangle, \theta = \theta_1)$

(E3) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \langle\alpha,\theta\rangle = \langle\tau,\theta'\rangle\gamma) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \alpha = \tau\gamma)$

(E4) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \Sigma_1 = \Sigma_2) \rightarrow$
$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}),$ if $\Sigma_1 = \Sigma_2 \vee \{\mathsf{SIG\_TYPE}\} \subseteq \{\Sigma_1, \Sigma_2\}$

(E5) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \Sigma_1 = \Sigma_2) \rightarrow$
$\texttt{equalityTypeErr}(\Sigma_1, \Sigma_2),$ if $\Sigma_1 \neq \Sigma_2 \wedge \{\mathsf{SIG\_TYPE}\} \not\subseteq \{\Sigma_1, \Sigma_2\}$

Rule (E4) and (E5) handle the comparison of equality type statuses. In the case

of a clash of statuses between EQ_TYPE and NEQ_TYPE we generate an error, if the status values are the same or one of the status values is SIG_TYPE then we succeed. Rule (I1) deals with turning equality constraints where equality type variables are assigned to the SIG_TYPE status to an equality constraint where NEQ_TYPE is the status mapped to instead. It is necessary to wait until after the structure and signature have been compared to generate this status or types declared as equality in the structure would clash with the signature, which would be erroneous.

7.1.5 Slicing

The addition in figure 7.7 is made to the specifications section of toTree to handle the `eqType` keyword in signatures.

Figure 7.7 Extensions to toTree

Specifications

$\text{toTree}(\texttt{eqtype}\ dn^{l}) = \langle\langle \texttt{spec, specEqtype}\rangle, l, \langle \text{toTree}(cd)\rangle\rangle$

7.1.6 Worked Example

Let us consider the program, written with labels included, shown in figure 7.8.

Figure 7.8 Ill typed code containing an equality type error

```
1  let^l1
2    datatype ⌈''a mydt⌉^l3 =^l2 A of ^l4''a^l9
3  in
4    ⌈A^l10(fn x^l7 =>^l6 x^l8)⌉^l5
5  end
```

This is a basic example showing the presence of an equality type error using only the external language syntax features presented in the Skalpel core. The constraints that are generated for this example are given in figure 7.9.[2]

Using the new constraint solving rules defined in this section, we now solve the following constraints to check whether this program is typable. Our set of unifiers $\mathcal{U}$ is initialised to $\varnothing$, and we begin the solving algorithm:

$\texttt{slv}(\langle\rangle, \varnothing, \varnothing, \langle\rangle, e)$, where e is

[2]If the reader wishes to refresh their memory on how to run a constraint generator defined in this document on a labelled program, it is recommend to review section 4.3 and example one in chapter 5, which go into more detail on how to apply the constraint generation rules.

Figure 7.9 Constraints generated for program in figure 7.8

$[\exists\langle\alpha_2,\theta_2\rangle.e_1; e_2; (\langle\alpha,\theta\rangle \stackrel{l_1}{=} \langle\alpha_2,\theta_2\rangle)]$

where e_1 is:
$\exists\langle\alpha_1,\alpha_2,\alpha_3,\theta_1,\theta_2,\theta_3,\delta,\gamma,ev\rangle.ev \stackrel{l_2}{=} ((\delta \stackrel{l_2}{=} \gamma); (\langle\alpha_2,\theta_2\rangle \stackrel{l_2}{=} \langle\alpha_1,\theta_1\rangle\gamma);$
$\downarrow\texttt{mydt} \stackrel{l_3}{=} \langle\delta,\theta_3\rangle; \texttt{loc}\,(\theta_3 \stackrel{l_3}{=} \mathsf{EQ\_TYPE}); \downarrow\texttt{''a} \stackrel{l_3}{=} \langle\alpha_3,\theta_3\rangle\ \texttt{in poly}(\exists\langle\alpha',\theta',\alpha'',\theta''\rangle.$
$\uparrow\texttt{''a} \stackrel{l_9}{=} \langle\alpha'',\theta''\rangle; (\langle\alpha',\theta'\rangle \stackrel{l_4}{=} \langle\alpha'',\theta''\rangle \to \langle\alpha_2,\theta_2\rangle); (\downarrow\texttt{A} \stackrel{l_4}{=} \langle\alpha',\theta'\rangle))); ev^{l_2}$

and e_2 is:
$\exists\langle\alpha_3,\theta_3,\alpha_4,\theta_4\rangle.\uparrow\texttt{A} \stackrel{l_{10}}{=} \langle\alpha_3,\theta_3\rangle; [\exists\langle\alpha_5,\theta_5,\alpha_6,\theta_6,ev\rangle.(ev = \downarrow\texttt{x} \stackrel{l_7}{=} \langle\alpha_5,\theta_5\rangle; ev^{l_6}; \uparrow\texttt{x} \stackrel{l_9}{=} \langle\alpha_6,\theta_6\rangle;$
$\theta_4 \stackrel{l_6}{=} \mathsf{NEQ\_TYPE}); (\langle\alpha_4,\theta_4\rangle \stackrel{l_6}{=} \langle\alpha_5,\theta_5\rangle \to \langle\alpha_6,\theta_6\rangle)]; (\langle\alpha_3,\theta_3\rangle \stackrel{l_5}{=} \langle\alpha_4,\theta_4\rangle \to \langle\alpha_2,\theta_2\rangle)$

$[\exists\langle\alpha_2,\theta_2\rangle.\exists\langle\alpha_1,\alpha_2,\alpha_3,\theta_1,\theta_2,\theta_3,\delta,\gamma,ev\rangle.ev \stackrel{l_2}{=} ((\delta \stackrel{l_2}{=} \gamma); (\langle\alpha_2,\theta_2\rangle \stackrel{l_2}{=} \langle\alpha_1,\theta_1\rangle\gamma);$
$\downarrow\texttt{mydt} \stackrel{l_3}{=} \langle\delta,\theta_3\rangle; \texttt{loc}\,(\theta_3 \stackrel{l_3}{=} \mathsf{EQ\_TYPE}); \downarrow\texttt{''a} \stackrel{l_3}{=} \langle\alpha_3,\theta_3\rangle\ \texttt{in poly}(\exists\langle\alpha',\theta',\alpha'',\theta''\rangle.$
$\uparrow\texttt{''a} \stackrel{l_9}{=} \langle\alpha'',\theta''\rangle; (\langle\alpha',\theta'\rangle \stackrel{l_4}{=} \langle\alpha'',\theta''\rangle \to \langle\alpha_2,\theta_2\rangle); (\downarrow\texttt{A} \stackrel{l_4}{=} \langle\alpha',\theta'\rangle))); ev^{l_2};$
$\exists\langle\alpha_3,\theta_3,\alpha_4,\theta_4\rangle.\uparrow\texttt{A} \stackrel{l_{10}}{=} \langle\alpha_3,\theta_3\rangle; [\exists\langle\alpha_5,\theta_5,\alpha_6,\theta_6,ev\rangle.(ev = \downarrow\texttt{x} \stackrel{l_7}{=} \langle\alpha_5,\theta_5\rangle; ev^{l_6}; \uparrow\texttt{x} \stackrel{l_9}{=} \langle\alpha_6,\theta_6\rangle; \theta_4 \stackrel{l_6}{=} \mathsf{NEQ\_TYPE}); (\langle\alpha_4,\theta_4\rangle \stackrel{l_6}{=} \langle\alpha_5,\theta_5\rangle \to \langle\alpha_6,\theta_6\rangle)]; (\langle\alpha_3,\theta_3\rangle \stackrel{l_5}{=} \langle\alpha_4,\theta_4\rangle \to \langle\alpha_2,\theta_2\rangle); (\langle\alpha,\theta\rangle \stackrel{l_1}{=} \langle\alpha_2,\theta_2\rangle)]$

We apply rules (U4), (X) (renaming α_2 and θ_2 to α_1 and θ_1 respectively), (C1) (moving the rest of the constraints from the let expression into the stack $\overrightarrow{st}$), (X) (renaming $\alpha_1,\theta_1,\alpha_2,\theta_2,\alpha_3,\theta_3,ev,\delta,\gamma$ to $\alpha_3,\theta_3,\alpha_4,\theta_4,\alpha_9,\theta_9,ev_1,\delta_9,\gamma_9$ respectively) and (C1), where we then call:

$$\begin{aligned}\texttt{slv}(\langle\rangle,\varnothing,\varnothing,\overrightarrow{st},\quad & (ev_1 = ((\delta_9 \stackrel{l_2}{=} \gamma_9); (\langle\alpha_4,\theta_4\rangle \stackrel{l_2}{=} \langle\alpha_9,\theta_9\rangle\gamma_9); \downarrow\texttt{mydt} \stackrel{l_3}{=} \langle\delta_9,\theta_9\rangle;\\ & \texttt{loc}\,(\theta_9 \stackrel{l_3}{=} \mathsf{EQ\_TYPE}); \downarrow\texttt{''a} \stackrel{l_3}{=} \langle\alpha_9,\theta_9\rangle\ \texttt{in}\\ & \texttt{poly}(\exists\langle\alpha',\theta',\alpha'',\theta''\rangle.\uparrow\texttt{''a} \stackrel{l_9}{=} \langle\alpha'',\theta''\rangle;\\ & (\langle\alpha',\theta'\rangle \stackrel{l_4}{=} \langle\alpha'',\theta''\rangle \to \langle\alpha_4,\theta_4\rangle); (\downarrow\texttt{A} \stackrel{l_4}{=} \langle\alpha',\theta'\rangle))))\end{aligned}$$

Rules (U4), (C1), (D), (U3), (C1), (D), (E3), (U3), (C1), (D), and (B2) are then applied.

$$\begin{aligned}\texttt{slv}(\langle\downarrow\texttt{mydt} \stackrel{\{l_2,l_3\}}{=} \gamma_1\rangle,\{l_2,l_3\},\varnothing,\overrightarrow{st},\quad & \downarrow\texttt{''a} \stackrel{l_3}{=} \langle\alpha_1,\theta_1\rangle; \theta_1 \stackrel{l_3}{=} \mathsf{EQ\_TYPE};\\ & \texttt{poly}(\exists\langle\alpha',\theta',\alpha_3,\theta_3\rangle.\uparrow\texttt{''a} \stackrel{l_9}{=} \langle\alpha_3,\theta_3\rangle;\\ & (\langle\alpha',\theta'\rangle \stackrel{l_4}{=} \langle\alpha_3,\theta_3\rangle \to \langle\alpha_4,\theta_4\rangle);\\ & \downarrow\texttt{A} \stackrel{l_4}{=} \langle\alpha',\theta'\rangle))\end{aligned}$$

where the unifier $\mathcal{U}$ is $\{\delta_9 \mapsto \gamma_9, \alpha_4 \mapsto \alpha_9\gamma_9\}$.

Rules (C1), (D), (B1), (C1), (D), (U3) are applied to leave the `poly` expression:

$$\begin{aligned}&\texttt{slv}(\langle\downarrow\texttt{mydt} \stackrel{\{l_2,l_3\}}{=} \gamma_1; \downarrow\texttt{''a} \stackrel{\{l_2,l_3\}}{=} \langle\alpha_1,\theta_1\rangle\rangle,\{l_2,l_3\},\{\alpha_1^{\{l_2,l_3\}}\},\overrightarrow{st},\\ &\quad \texttt{poly}(\exists\langle\alpha',\theta',\alpha_3,\theta_3\rangle.\uparrow\texttt{''a} \stackrel{l_9}{=} \langle\alpha_3,\theta_3\rangle; (\langle\alpha',\theta'\rangle \stackrel{l_4}{=} \langle\alpha_3,\theta_3\rangle \to \langle\alpha_4,\theta_4\rangle); \downarrow\texttt{A} \stackrel{l_4}{=} \langle\alpha',\theta'\rangle))\end{aligned}$$

where the unifier $\mathcal{U}$ is now $\{\delta_9 \mapsto \gamma_9, \alpha_4 \mapsto \alpha_9\gamma_9, \theta_1 \mapsto \mathsf{EQ\_TYPE}\}$.

To solve this polymorphic call we apply rules (P6) (where $\alpha', \theta', \alpha_3, \theta_3$ are renamed to $\alpha_5, \theta_5, \alpha_6, \theta_6$), (P4), (P5), (D), (A1), (E1), (U3), (U3), (P4), (P5), (D), (E2), (U3), (U3), (P1). After these applications, we then call[3]:

$$\begin{array}{l}\mathtt{slv}(\langle\downarrow\mathtt{mydt} \overset{\{l_2,l_3\}}{=} \gamma_1; \downarrow\text{''}\mathtt{a} \overset{\{l_2,l_3\}}{=} \langle\alpha_1,\theta_1\rangle; \downarrow\mathtt{A} \overset{l_4}{=} \forall\{\alpha_6,\alpha_4\}.\langle\alpha_6 \to \alpha_4, \theta_5\rangle\rangle,\\ \quad \{l_2,l_3,l_9,l_4\}, \{\alpha_1^{\{l_2,l_3\}}\}, \overrightarrow{st},\\ \quad ev^{l_2}; \exists\langle\alpha_3,\theta_3,\alpha_4,\theta_4\rangle.\uparrow\mathtt{A} \overset{l_{10}}{=} \langle\alpha_3,\theta_3\rangle; [\exists\langle\alpha_5,\theta_5,\alpha_6,\theta_6,ev\rangle.(ev = \downarrow\mathtt{x} \overset{l_7}{=} \langle\alpha_5,\theta_5\rangle);\\ \quad ev^{l_6}; \uparrow\mathtt{x} \overset{l_9}{=} \langle\alpha_6,\theta_6\rangle; (\theta_4 \overset{l_6}{=} \mathsf{NEQ\_TYPE}); (\langle\alpha_4,\theta_4\rangle \overset{l_6}{=} \langle\alpha_5,\theta_5\rangle \to \langle\alpha_6,\theta_6\rangle)];\\ \quad (\langle\alpha_3,\theta_3\rangle \overset{l_5}{=} \langle\alpha_4,\theta_4\rangle \to \langle\alpha_2,\theta_2\rangle); (\langle\alpha,\theta\rangle \overset{l_1}{=} \langle\alpha_1,\theta_1\rangle))\end{array}$$

where the unifier $\mathcal{U}$ is now:

$$\{\delta_9 \mapsto \gamma_9, \alpha_4 \mapsto \alpha_9\gamma_9, \theta_1 \mapsto \mathsf{EQ\_TYPE}, \theta_6 \mapsto \theta_1, \alpha_6 \mapsto \alpha_1, \theta_5 \mapsto \theta_6, \alpha_5 \mapsto (\alpha_6 \to \alpha_4)\}$$

We apply rules (C1), (D), (V), (X), (C1), (D), (A1), (E1), (U3), (U3), (C1), (U4), (X), (C1), (U4), (D), (B), (C1), (D), and (V) to yield:

$$\begin{array}{l}\mathtt{slv}(\langle\downarrow\mathtt{mydt} \overset{\{l_2,l_3\}}{=} \gamma_1; \downarrow\text{''}\mathtt{a} \overset{\{l_2,l_3\}}{=} \langle\alpha_1,\theta_1\rangle; \downarrow\mathtt{A} \overset{l_4}{=} \forall\{\alpha_6,\alpha_4\}.\langle\alpha_6 \to \alpha_4, \theta_5\rangle; ev^{\{l_2,l_3,l_9,l_4\}};\\ \quad ev_2^{\{l_2,l_3,l_9,l_4,l_{10},l_7\}}\rangle, \{l_2,l_3,l_9,l_4,l_{10},l_7\}, \{\alpha_1^{\{l_2,l_3\}}\}, \overrightarrow{st},\\ \quad \uparrow\mathtt{x} \overset{l_9}{=} \langle\alpha_{10},\theta_{10}\rangle; (\theta_8 \overset{l_6}{=} \mathsf{NEQ\_TYPE}); (\langle\alpha_8,\theta_8\rangle \overset{l_6}{=} \langle\alpha_9,\theta_9\rangle \to \langle\alpha_{10},\theta_{10}\rangle);\\ \quad (\langle\alpha_7,\theta_7\rangle \overset{l_5}{=} \langle\alpha_8,\theta_8\rangle \to \langle\alpha_2,\theta_2\rangle); (\langle\alpha,\theta\rangle \overset{l_1}{=} \langle\alpha_1,\theta_1\rangle))\end{array}$$

where the unifier $\mathcal{U}$ is now:

$$\{\delta_9 \mapsto \gamma_9, \alpha_4 \mapsto \alpha_9\gamma_9, \theta_1 \mapsto \mathsf{EQ\_TYPE}, \theta_6 \mapsto \theta_1, \alpha_6 \mapsto \alpha_1, \theta_5 \mapsto \theta_6,$$

$$\alpha_5 \mapsto (\alpha_6 \to \alpha_4), \theta_7 \mapsto \theta_5, \alpha_7 \mapsto \alpha_9 \to \alpha_{10}, ev_2 \mapsto \downarrow\mathtt{x}{=}\langle\alpha_9,\theta_9\rangle\}$$

Applying rules (C1), (D), (A1), (E1), (U3), (U3), (C1), (D), (U3), (C1), (D), (E2), (U6), (R), (U3), (U3) leaves the following to be solved:

$$\begin{array}{l}\mathtt{slv}(\langle\downarrow\mathtt{mydt} \overset{\{l_2,l_3\}}{=} \gamma_1; \downarrow\text{''}\mathtt{a} \overset{\{l_2,l_3\}}{=} \langle\alpha_1,\theta_1\rangle; \downarrow\mathtt{A} \overset{l_4}{=} \forall\{\alpha_6,\alpha_4\}.\langle\alpha_6 \to \alpha_4, \theta_5\rangle; ev^{\{l_2,l_3,l_9,l_4\}};\\ \quad ev_2^{\{l_2,l_3,l_9,l_4,l_{10},l_7\}}\rangle, \{l_2,l_3,l_9,l_4,l_{10},l_7,l_6\}, \{\alpha_1^{\{l_2,l_3\}}\}, \overrightarrow{st},\\ \quad (\langle\alpha_7,\theta_7\rangle \overset{l_5}{=} \langle\alpha_8,\theta_8\rangle \to \langle\alpha_2,\theta_2\rangle); (\langle\alpha,\theta\rangle \overset{l_1}{=} \langle\alpha_1,\theta_1\rangle))\end{array}$$

where the unifier $\mathcal{U}$ is now:

$$\{\delta_9 \mapsto \gamma_9, \alpha_4 \mapsto \alpha_9\gamma_9, \theta_1 \mapsto \mathsf{EQ\_TYPE}, \theta_6 \mapsto \theta_1, \alpha_6 \mapsto \alpha_1, \theta_5 \mapsto \theta_6,$$

$$\alpha_5 \mapsto (\alpha_6 \to \alpha_4), \theta_7 \mapsto \theta_5, \alpha_7 \mapsto \alpha_9 \to \alpha_{10}, ev_2 \mapsto \downarrow\mathtt{x}{=}\langle\alpha_9,\theta_9\rangle, \theta_{10} \mapsto \theta_9,$$

[3]The constraints in the stack have now been moved into the last argument of `slv` by `isSucc`.

$$\alpha_{10} \mapsto \alpha_9, \theta_8 \mapsto \mathsf{NEQ\_TYPE}, \theta_9 \mapsto \mathsf{NEQ\_TYPE}, \alpha_8 \mapsto \alpha_9 \rightarrow \alpha_{10}\}$$

Finally, rules (C1), (D), (E2), (U6), (R), (U6) are used, after which rule (E5) generates an equality type error. After minimisation, the final slice presented is:

$$\langle .. \texttt{datatype ''a mydt = A of ''a} .. \texttt{A (fn} \langle .. \rangle \texttt{)} .. \rangle$$

7.1.7 Tuples/records, and datatypes with more than one constructor

Let us look at another example involving equality type errors:

```
datatype 'a mydt = firstCons of 'a | secondCons of 'a;

fun getReal () = 5.0;

val x = firstCons (1, getReal (), 5);
val y = firstCons (2, getReal (), 6);

x = y
```

This is also not typable. The expression `x = y` results in constraints that `x` and `y` should both be equality types, but the middle component of both the `x` and the `y` tuple is the number of type real returned by `getReal`, which makes both `x` and `y` not equality types, which causes an equality type error. The highlighting produced by the Skalpel analysis engine is shown below.

```
datatype 'a mydt = firstCons of 'a | secondCons of 'a;

fun getReal () = 5.0;

val x = firstCons (1, getReal (), 5);
val y = firstCons (2, getReal (), 6);

x = y
```

The machinery for handling tuples, records, and datatypes with more than one constructor is not presented in this document, but is considered in the implementation. The implementation has been extended to detect equality type errors which might arise using such features too. When a tuple is used, the equality type status of the tuple as a whole depends on the equality type status of each of the

components of a tuple - if any of the individual components have an NEQ_TYPE status then the tuple as a whole cannot be checked for equality. Similarly, when a datatype is defined *all* of the constructors of that datatype must admit equality, otherwise comparison between any two datatype constructors of that datatype results in an equality type error.

We have demonstrated how equality types are supported in Skalpel, and edited the majority of the existing theory to achieve this. We have also shown a full worked example to demonstrate how this theory operates and locates equality type errors in Standard ML code.

7.2 Abstract type declarations

In this section we present an extension to the existing theory to add support for abstract type declarations.

In order to do this, extensions need to be made to the external syntax and to the constraint generation rules, but by making use of the extension to the Skalpel core which allows for greater flexibly in deciding which binders to export in a term defined in 6.1, no extensions are needed to the constraint solving algorithm.

7.2.1 External syntax

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\mathtt{Sig}}$ with the additional extensions in this section. We will refer to this set as $\mathsf{ExtLabSynt}_{\mathtt{Abstype}}$, which $\mathcal{P}^{L}{}_{\mathtt{Abstype}}$ ranges over.

We introduce the `abstype` keyword to the external syntax as follows:

$$dec ::= \ldots \mid \texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}$$

An example SML program is shown in figure 7.10 where the `abstype` keyword is used and part of an error.

Figure 7.10 Example SML program containing the `abstype` keyword

```
1 abstype 'a mydt = MyConstructor of int
2  with val rec f = fn x => MyConstructor x end;
3 f true;
```

This program is erroneous as the constructor `MyConstructor` is defined to take an integer argument, but in the second line a boolean is passed to the function `f`, which applies its argument to the constructor `MyConstructor`.

7.2.2 Constraint generation

We use the **loc** keyword from the extension to the core presented in 6.1 in the new constraint generation rule which is shown below.

$$\begin{aligned}(\mathsf{G61})\ [\![\texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}]\!] \quad &= \exists\langle\alpha_1, \alpha_2, ev\rangle.(ev = ((\alpha_1 \stackrel{l}{=} \alpha_2); \\ &\qquad [\![dn, \alpha_1]\!]; \texttt{loc}\ \texttt{poly}([\![cb, \alpha_2)\ \texttt{in}\ [\![dec]\!])); ev^l\end{aligned}$$

The use of the **loc** environment will ensure that binders of datatype constructors are visible to the declaration *dec* that is specified but *not* accessible outside of the `abstype` definition. This constraint generation rule is a modification of the one presented in the Skalpel core and other any errors that can occur (that is, errors not a cause of the now restricted scope of abstract type constructor binders) have already been reported on in chapter 4.

7.2.3 Slicing

We must first update the labelled abstract syntax tree forms to include a slice containing the `abstype` keyword:

$$prod \in \mathsf{Prod} ::= \ldots \mid \texttt{abstype}$$

Finally, we update toTree to include slices involving abstract type declarations, as shown below.

$$\mathsf{toTree}(\texttt{abstype}\ dn \overset{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}) = \\ \langle\langle \texttt{dec}, \texttt{decAbstype}\rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(cb), \mathsf{toTree}(dec)\rangle\rangle$$

We have added support for abstract declarations in this extension. The extension also shows how a new language feature can be supported without necessarily extending every stage of the engine and re-using previous components.

7.3 Duplicate Identifiers in Specifications

Presented here is an extension to create errors for code where a typename is defined more than once in specifications. For example, the following code is untypable:

```
signature S =
sig
  type x = int
  type y = bool
  type x = string
end
```

In the above example, the typename `x` is defined twice. The theory presented here is also extended in section 7.4 in order to handle cases where typenames can be defined in signature which are included with the `include` keyword.

No alterations need to be made to the external syntax, as the syntax for type and datatype declarations is given in section 6.4, with the syntax for the `eqtype` keyword given in section 7.1.

7.3.1 Constraint Syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\mathtt{Sig}}$ with the extensions listed in this section. We refer to this set as $\mathsf{IntLabSynt}_{\mathtt{DuplicateId}}$, which $\mathcal{C}^{L}{}_{\mathtt{DuplicateId}}$ ranges over.

A new environment form `duplicates` is defined in figure 7.11.

Figure 7.11 Extensions to Syntax of Constraint Terms

e	$\in$	Env	$::=$	... $\mid$ `duplicates`(e)

We create this environment during constraint generation to indicate that we need to check the environment specified in the argument for bindings which contain the same typename. We only use this environment during constraint generation of signatures[4].

[4]It is not necessary to check for duplicate identifier declarations inside of structure or functor definitions, as these are perfectly legitimate, and identifiers are simply rebound in this case.

7.3.2 Constraint Generation

Figure 7.12 Extension of constraint generation rules for detection of duplicate binding of typenames in specifications

Signature expressions

(G34) $[\![\texttt{sig}^{l}\ \ spec_1 \cdots spec_n\ \ \texttt{end}, ev]\!] = \exists ev'.(ev \stackrel{l}{=} ev'); (ev' = ([\![spec_1]\!]; ...; [\![spec_n]\!])); \texttt{duplicates}(ev')$

The only change to the rule specified originally in figure 6.19 is the addition of environment composition with the new `duplicates` environment. We will act upon this in constraint solving and attempt to detect errors there[5].

7.3.3 Constraint Solving

First, we extend *state* with a new form as shown below.

$$state \in \mathsf{State} ::= ... \mid \texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \mathbb{P}(id \times l), \overrightarrow{e'})$$

We create a new set which holds value identifiers or type constructors, as either can be part of a duplicate specification error:

$$vidTc \in \mathsf{VidTyCon} ::= vid \mid tc$$

In order to represent errors that can be generated when the same identifier is located more than once in the same signature, we need to extend the kinds of error that we can represent as follows:

$$ek \in \mathsf{ErrKind} ::= ... \mid \texttt{duplicateIdErr}(\langle vidTc, l\rangle, \langle vidTc, l'\rangle)$$

First we define the function `sameId`, which takes an environment as an argument and checks for duplicate type or constructor names in bindings. In the event that duplicates are found, an error is generated and we present the labels attributed to the identifiers that we have found to be the same. Otherwise, we continue to solve whatever else is left in the stack.

[5]While it would perhaps be simpler to just report errors at this phase (constraint generation), by looking at the environments that have been generated, we choose not to do this in order to keep our constraint generation and solving phases strictly separate.

The extensions to the constraint solving rules for `sameId` are given in figures 7.13 and 7.14.

Figure 7.13 Extension of constraint solving rules for duplicate identifier checks in signatures (1 of 2)

$$
\begin{array}{ll}
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@ev) & \rightarrow \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\mathcal{U}(ev)) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@e_1; e_2) & \rightarrow \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@e_2@e_1) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\exists a.e') & \rightarrow \\
\texttt{sameId}(\overrightarrow{e}, \overline{d} \cup \overline{d}', \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}@e'[\{a \mapsto a'\}]), & \\
\text{if } a' \notin \mathsf{atoms}(\langle \mathcal{U}, e' \rangle) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\downarrow vid \stackrel{l}{=} ts) & \rightarrow \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel \cup \{(vid, l)\}, \overrightarrow{st}') & \\
\text{if } vid \notin \mathsf{dom}(idLabel) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\downarrow vid \stackrel{l}{=} ts) & \rightarrow \\
\texttt{duplicateIdErr}(\langle vid, l \rangle, \langle vid, idLabel(vid) \rangle) & \\
\text{if } vid \in \mathsf{dom}(idLabel) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\downarrow tc \stackrel{l}{=} tcs) & \rightarrow \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel \cup \{(tc, l)\}, \overrightarrow{st}') & \\
\text{if } vid \notin \mathsf{dom}(idLabel) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\downarrow tc \stackrel{l}{=} tcs) & \rightarrow \\
\texttt{duplicateIdErr}(\langle tc, l \rangle, \langle tc, idLabel(vid) \rangle) & \\
\text{if } vid \in \mathsf{dom}(idLabel) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \langle\rangle) & \rightarrow \\
\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}) & \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}@e) & \rightarrow \\
\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}) & \\
\quad \text{if } e \notin \mathsf{Bind} \cup \mathsf{Var} \wedge e \neq \langle\rangle \wedge e \neq \exists a.e' &
\end{array}
$$

Figure 7.14 Extension of constraint solving rules for duplicate identifier checks in signatures (2 of 2)

`instantiations`

$$
\begin{array}{lll}
\text{(I1)} \quad \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{ins}(e_0)) & \rightarrow & \texttt{isSucc}(\overrightarrow{e}; e_2[ins], \overline{m}, \overrightarrow{st}) \\
\quad \text{if } \mathcal{U}(e_0) = e_1 & & \\
\quad \wedge\ e_2 = \mathsf{opaqueEq}(e_1) & & \\
\quad \wedge\ \mathsf{dom}(ins) = \mathsf{tyconvars}(e_2) & & \\
\quad \wedge\ \mathsf{dj}(\mathsf{vars}(\mathcal{U}) \cup \mathsf{vars}(\overrightarrow{e}), \mathsf{ran}(ins)) & & \\
\text{(I2)} \quad \texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{duplicates}(e_0)) & \rightarrow & \texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \{\}, \langle e_0 \rangle)
\end{array}
$$

7.3.4 Slicing

No changes are required to the slicing functions, the extensions reported in section 6.4.6 are sufficient. We do not need to update our set that holds external syntax keywords as the theory for duplicate identifiers does not require introduction of any new external syntax.

In this extension we have demonstrated how support for errors involving duplicate identifiers which have been declared in the same specification can be detected. This extension, like the others, is tested by our analysis engine test database, discussed further in appendix C.

7.4 Include specifications

This section adds support for include specifications to the existing theory. There was some pre-existing support for this feature in the Skalpel analysis engine but nothing had been written formally to support include specifications. The initial implementation support included the handling of errors where a signature included a type which was involved in a type constructor clash, but not for example the case where a signature was included which defined a type which had already been defined in that signature.

Let us look at an example where a program has an error as a specification has been included multiple times.

```
signature S = sig
  type 'a a = 'a -> 'a
end
signature S' = sig
  include S
  type 'b b = 'b -> 'b
  include S
end
```

The error occurs in the signature `S'` in this case, as they type `a` has been included in the signature twice. A type can only be defined in a signature at most once.

7.4.1 External syntax

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\texttt{Abstype}}$ with the additional extensions in this section. We will refer to this set as $\mathsf{ExtLabSynt}_{\texttt{Include}}$, which ${\mathcal{P}^L}_{\texttt{Include}}$ ranges over.

The external syntax has to be extended to also contain the `include` keyword:

Figure 7.15 External syntax for include specifications

spec	$\in$	Spec	$::=$	`include` $sigexp^l$

7.4.2 Constraint syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\texttt{DuplicateId}}$. No changes are required to the constraint syntax, the constraint syntax extended in section 6.4 and section 7.3 is sufficient to represent the necessary constraints.

7.4.3 Constraint generation

An extension to the initial constraint generation rules is made to extend Skalpel with support for the `include` keyword in figure 7.16.

Figure 7.16 Constraint generation rules for include specification : $\mathsf{ExtLabSynt}_{\texttt{Include}} \times \mathbb{P}(\mathsf{Var}) \rightarrow \mathsf{Env}$

(G62) $[\![\texttt{include}\ sigexp^{l}]\!] = \exists ev_1.[\![sigexp, ev_1]\!]$

7.4.4 Constraint solving

In order for us to locate type names inside signature expressions where the expression is a *sigid*, we must extend `sameId` to handle accessors as shown below.

Figure 7.17 Extension of constraint solving rules for duplicate identifier checks in include specifications

$\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@{\uparrow}id{=}v) \rightarrow$
$\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@\overrightarrow{e}(id))$, if $\overrightarrow{e}(id)$ defined

$\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}'@{\uparrow}id{=}v) \rightarrow$
$\texttt{sameId}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, idLabel, \overrightarrow{st}')$, if $\overrightarrow{e}(id)$ undefined

7.4.5 Slicing

Finally, we must extend slicing to support the `include` keyword by extending our syntax for programs and toTree as follows.

$$\mathsf{Prod} ::= \ldots \mid \texttt{specInclude}$$

$$\mathsf{toTree}(\texttt{include}\ sigexp^{l}) = \langle\langle \texttt{spec}, \texttt{specInclude}\rangle, l, \langle \mathsf{toTree}(sigexp)\rangle\rangle$$

We have demonstrated how Skalpel handles include specifications. Next, we will describe how errors involving type sharing specifications are detected and presented in Skalpel.

7.5 Type Sharing

In Standard ML, types can be 'shared', where two types are equal to the same type. Additional constraints need to be generated to achieve this, and how we handle these kinds of errors is described below.

7.5.1 External syntax

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\texttt{Include}}$ with the additional extensions in this section. We will refer to this set as $\mathsf{ExtLabSynt}_{\texttt{Sharing}}$, which ${\mathcal{P}^L}_{\texttt{Sharing}}$ ranges over.

This extension adds support for type sharing, and this is achieved through the use of the `sharing` keyword in Standard ML. We extend our external syntax to include this new keyword as seen in figure 7.18.

Figure 7.18 Extension to external syntax for type sharing

$spec \in \mathsf{Spec} ::= \ldots \mid \texttt{sharing type}\ vid_1 = vid_2$

There are some notable differences in this presentation to that as defined in the definition of Standard ML [MTHM98]. They are:

- Only two value identifiers can be present in a sharing specification in this presentation. In the definition, there is no limit to how many types can be specified to be shared. We support this in our implementation.
- A sharing specification can only be defined after some other specification has already been defined. We do not carry such restrictions here, as we will generate errors when a sharing specification is the first specification in a signature anyway (as the types which are shared will have not been defined yet in that signature, which is erroneous).

7.5.2 Constraint syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\texttt{DuplicateId}}$ with the extensions listed in this section. We refer to this set as $\mathsf{IntLabSynt}_{\texttt{Sharing}}$, which ${\mathcal{C}^L}_{\texttt{Sharing}}$ ranges over.

We extend the constraint syntax with two forms:

1. A form which allows us to represent two types which are connected with a sharing constraint;

2. A form which represents the need to check that the types which are to be shared are defined in the signature in which the sharing specification is present *before* the sharing specification is given. (The reason why this cannot be done by generating accessors is given in section 7.5.4.)

Figure 7.19 Extensions to constraint syntax to support type sharing

$e \in \mathsf{Env} ::= \ldots \mid \texttt{sharingSig}(ev) \mid \texttt{sharingType}(e_1, e_2)$

7.5.3 Constraint generation

Two rules are added to the constraint generation machinery. Rule (G34) wraps the environment variable containing a `sig`-declaration in a call to the `sharingSig` function, defined in section 7.5.4. This function will check that types which are to be shared occur in the signature declaration before the point of the sharing specification.

Rule (G35) is an entirely new rule, which takes the two value identifiers which are to be shared and puts them in the internal constraint syntax environment `sharingType` to represent this.

Figure 7.20 Changes to existing constraint generation rules : $\mathsf{ExtLabSynt}_{\texttt{Sharing}} \times \mathbb{P}(\mathsf{Var}) \rightarrow \mathsf{Env}$

(G34) $[\![\texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end}, ev]\!] = \exists ev'.(ev \stackrel{l}{=} ev'); (ev' = ([\![spec_1]\!]; \ldots; [\![spec_n]\!])); \texttt{sharingSig}(ev')$

Figure 7.21 New constraint generation rules : $\mathsf{ExtLabSynt}_{\texttt{Sharing}} \times \mathbb{P}(\mathsf{Var}) \rightarrow \mathsf{Env}$

(G35) $[\![\lceil \texttt{sharing type}\ exp_1 = exp_2 \rceil^l]\!] =$
$\exists ev.ev = \texttt{sharingType}(\exists \alpha_1.[\![exp_1, \alpha_1]\!], \exists \alpha_2.[\![exp_2, \alpha_2]\!]); ev^l$

7.5.4 Constraint solving

The constraint solver is extended to support the new constraint syntax form `sharingSig`. This extension can be seen in figures 7.23 and 7.24.

We extend *state* with a new form as follows:

$$state \in \mathsf{State} ::= \ldots \mid \texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, \overline{vid}, e')$$

The bulk of the machinery for handling errors involving undefined value identifiers in type sharing specifications is handled in figure 7.24. We cannot check for

these errors merely with accessors, as if the value identifier is defined before a signature specification, but not defined in that signature which in turn contains a specification sharing that type with another type, then this is illegal in ML. Note however that it would technically be possible to do this by checking that the label of binder connected to the connected accessor, and check that it is less than the number of the label of the `sig`-expression, but while this would save some complexity in the definitions here this would create a dependency on the way that we label programs, which is undesirable.

We need to extend the stack mechanism for the new `vidSharingCheck` function for operations such as environment composition and stack checking. When we check the stack for success in this function, we want to keep track of the value identifiers that we have detected so far in the signature, so that we can determine whether any are designated to be shared which are not bound. We start by extending *stackEv* with a new form:

$$stackEv \in \mathsf{StackEv} = ... \mid \texttt{sha}(e)$$

We extend this form as when checking for value identifiers in `vidSharingCheck`, we never change the environment in the constraint solving rules $\overrightarrow{e}$. As this is the case, this slot to designate if the initial environment that was present at the time this stack item was pushed to the stack or if the most recent environment present at the time the `isSucc` function was called is not needed. We recycle this slot in this extension to contain an environment we still need to check for value identifiers. We do not put this environment in the fourth slot of the stack (the slot which can already hold an environment for solving), as such environments get passed to the `slv` function.

We introduce the `isSuccVSC` function, which operates similarly to `isSucc` but carries with it the current set of value identifiers that we have so far discovered when analyzing the environment given to `vidSharingCheck`. The rules for this are shown in figure 7.22.

In order to represent errors that can occur when using type sharing, we extend the set of errors with a new form `sharingErr` as shown below:

$$ek \in \mathsf{ErrKind} ::= ... \mid \texttt{sharingErr}$$

In constraint solving rules (VSC4), (VSC5), (VSC6) and (VSC7) we handle the case where we detect a binder in the environment. We check these binders against

Figure 7.22 Rules for `isSuccVSC`

$\texttt{isSuccVSC}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}'@\langle\langle \texttt{sha}(e'), \overline{d}, \texttt{new}, \top\rangle\rangle, found) \rightarrow$
$\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, e')$
$\texttt{isSuccVSC}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}'@next, found) \rightarrow \texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}@next)$,
where $next \neq \langle\langle \texttt{sha}(e'), \overline{d}, \texttt{new}, \top\rangle\rangle$

Figure 7.23 New constraint solving rules for handling type sharing : State → State

`type sharing`

(SH1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{sharingSig}(ev)) \rightarrow \texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, \{\}, ev)$

(SM15) $\texttt{match}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, e', \texttt{sharingType}(\exists\alpha_1.{\uparrow}vid_1 \overset{l_1}{=} \alpha_1, {\uparrow}vid_2 \overset{l_2}{=} \alpha_2)) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}', \overline{m}, \overrightarrow{st}, e'(vid_1) = e'(vid_2))$, where $\overline{d}' = d \cup \{l_1, l_2\}$

the set of binders we have been gathering ($found$), and if we detect a duplicate then we raise an error.

7.5.5 Slicing

We must update our slicing mechanism involving signatures to handle the new sharing keyword. First, our syntax of programs is updated as follows with a form which will allow us tto represent type sharing:

$$\text{Prod} ::= \ldots \mid \texttt{specSharing}$$

Finally, toTree must be updated to handle the `sharing type` external syntax as shown in figure 7.25.

Figure 7.25 Extension to toTree

$$\text{toTree}(\lceil \texttt{sharing type}\ exp_1 = exp_2 \rceil^l) =$$

$$\langle\langle \texttt{sigexp}, \texttt{specSharing}\rangle, l, \langle \text{toTree}(exp_1), \text{toTree}(exp_2)\rangle\rangle$$

We have shown how Skalpel handles errors involving type sharing specifications, by extending our support for signatures. In the next and final extension to the theory, we will demonstrate how errors involving infix operators and handled.

Figure 7.24 Declaration of `vidSharingCheck` : tuple(Env) × $\mathbb{P}$(Monomorphic) × tuple(tuple(StackEv) × $\mathbb{P}$(Dependency) × $\mathbb{P}$(StackMono) × tuple(StackAction)) × $\mathbb{P}$(Dependency) × $\mathbb{P}$(VId) × Env

(VSC1) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, ev) \rightarrow$
$\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, \mathcal{U}(ev))$

(VSC2) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, \downarrow vid{=}x) \rightarrow$
$\texttt{isSuccVSC}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, found \cup \{vid\})$

(VSC3) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, e^{\overline{d}'}) \rightarrow$
$\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d} \cup \overline{d}', found, e)$

(VSC4) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found,$
$\texttt{sharingType}(\exists \alpha_1.{\uparrow}vid_1 \overset{l_1}{=} \alpha_1, \exists \alpha_2.{\uparrow}vid_2 \overset{l_2}{=} \alpha_2))$
$\rightarrow \texttt{err}(\langle \texttt{sharingErr}, \overline{d} \cup \{l_1, l_2\}),$ if $\{vid_1, vid_2\} \cap found = \varnothing$

(VSC5) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found,$
$\texttt{sharingType}(\exists \alpha_1.{\uparrow}vid_1 \overset{l_1}{=} \alpha_1, \exists \alpha_2.{\uparrow}vid_2 \overset{l_2}{=} \alpha_2))$
$\rightarrow \texttt{err}(\langle \texttt{sharingErr}, \overline{d} \cup \{l_2\}),$ if $vid_1 \in found \wedge vid_2 \notin found$

(VSC6) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found,$
$\texttt{sharingType}(\exists \alpha_1.{\uparrow}vid_1 \overset{l_1}{=} \alpha_1, \exists \alpha_2.{\uparrow}vid_2 \overset{l_2}{=} \alpha_2))$
$\rightarrow \texttt{err}(\langle \texttt{sharingErr}, \overline{d} \cup \{l_1\}),$ if $vid_2 \in found \wedge vid_1 \notin found$

(VSC7) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found,$
$\texttt{sharingType}(\exists \alpha_1.{\uparrow}vid_1 \overset{l_1}{=} \alpha_1, \exists \alpha_2.{\uparrow}vid_2 \overset{l_2}{=} \alpha_2))$
$\rightarrow \texttt{isSuccVSC}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, found),$ if $\{vid_1, vid_2\} \subseteq found$

(VSC8) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, e_1; e_2) \rightarrow$
$\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}', \overline{d}, found, e_1),$
where $\overrightarrow{st}' = \overrightarrow{st} @ \langle\langle \texttt{sha}(e_2), \overline{d}, \texttt{new}, \top \rangle\rangle$

(VSC9) $\texttt{vidSharingCheck}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, \overline{d}, found, e') \rightarrow$
$\texttt{isSuccVSC}(\overrightarrow{e}, \overline{m}, \overrightarrow{st}, found),$
where $e' \neq e^{\overline{d}} \wedge e' \neq bind \wedge e' \neq ev \wedge e' \neq \texttt{sharingType}(e_1, e_2) \wedge$
$e' \neq e_1; e_2$

7.6 Operator Infixity

This section presents an extension to Skalpel so that it can detect errors in programs involving infix operators.

```
fun l1l (a,b) = "(" ^ a ^ "," ^ b ^ ")";
fun l2l (a,b) = "(" ^ a ^ "," ^ b ^ ")";
fun l3l (a,b) = "(" ^ a ^ "," ^ b ^ ")";
fun r1r (a,b) = "(" ^ a ^ "," ^ b ^ ")";
fun r2r (a,b) = "(" ^ a ^ "," ^ b ^ ")";
fun r3r (a,b) = "(" ^ a ^ "," ^ b ^ ")";

infix 1 l1l;
infix 2 l2l;
infix 3 l3l;
infixr 1 r1r;
infixr 2 r2r;
infixr 3 r3r;
```

It is important to note that one option of implementing infix operator error detection would be to edit the internal representation of the user code during parsing, with the aim that we would be spared complexity in our solver. We have chosen not to do this, primarily for the reason that internal modification of user programs in this way is partly responsible for the poor errors that are produced by the available compilers in the first place, and we wish to ensure that we *never* present an internally modified version of submitted code to any programmer.

7.6.1 External syntax

The external syntax used by this extension to the existing theory is $\mathsf{ExtLabSynt}_{\texttt{Sharing}}$ with the additional extensions in this section. We will refer to this set as $\mathsf{ExtLabSynt}_{\texttt{Infix}}$, which $\mathcal{P}^L{}_{\texttt{Infix}}$ ranges over.

There are four new keywords that exist in the external (Standard ML) syntax. These are listed below.

- `infix`. Specifies an operator is infix, which associates to the *left*. Takes an argument specifying how high the precedence is.
- `infixr` Specifies an operator is infix, which associates to the *right*. an argument specifying how high the precedence is.

- `nonfix`. Takes infixity status away from an operator.
- `op`. Allows an operator to be used in a nonfix manner, even though it is still defined as infix.

We also must have some notion of expressing digits in the external syntax for the purpose of specification of operator infixity precedence. The changes to the external syntax are presented in figure 7.26.

Figure 7.26 External syntax changes to handle infixity

dec	$\in$	Dec	$::=$ ... | `infix` *digit vid* | `infixr` *digit vid* | `nonfix` *vid*
vid	$\in$	VId	$::=$ ... | `op` *vvar* | `op` *dcon*
digit	$\in$	Digit	$::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

7.6.2 Constraint syntax

The constraint syntax used in this extension is $\mathsf{IntLabSynt}_{\texttt{Sharing}}$ with the extensions listed in this section. We refer to this set as $\mathsf{IntLabSynt}_{\texttt{Infix}}$, which $\mathcal{C}^{L}{}_{\texttt{Infix}}$ ranges over.

We present the constraint syntax updated to handle an internal notion of operator infixity in figure 7.27.

Figure 7.27 Edits to constraint syntax for infix operators

bind	$\in$	Bind	$::=$	... $\mid \downarrow vid \stackrel{l}{=} \langle \alpha, dir, digit \rangle$
dir	$\in$	Direction	$::=$	$L \mid R \mid NONE$
e	$\in$	Env	$::=$	`infix`$(dir, digit, vid, l)$
				| `infixCheck`$(\overrightarrow{\alpha}, digit, digit', dir, \overrightarrow{\alpha'}, \overrightarrow{l_c}, \overrightarrow{l})$

We extend our internal representation of binders to have a form where we can bind to an internal type variable, a direction indicating in which direction the operator associates, and a digit expressing its precedence.

We create a new set *dir* which indicates the direction in which the infix operator associates. This can be either L, R or $NONE$.

We introduce a new environment `infix`, which is used to represent internally an infix operator. We also create a new environment, `infixCheck`, which is used to detect errors involving infixity. This is discussed further in the section on constraint solving.

7.6.3 Constraint generation

Now that we have to consider the notion of infix identifiers, we must edit the rule for application. We do this by editing rule (G3) (which handles application) in the constraint generator, as shown in figure 7.28.

We also create some new rule rules, (G63), (G64) and (G65), which handle constraint generation for infixity specifications (eg. left infixity, no infixity).

Figure 7.28 Modifications to constraint generation rules : $\mathsf{IntLabSynt}_{\mathtt{Infix}} \times \mathbb{P}(\mathsf{Var}) \to \mathsf{Env}$

(G3)	$[\![exp_1^{l_1}\, atexp_2^{l_2} ... atexp_n^{l_n}, \alpha]\!] = \exists\langle\alpha_1, ..., \alpha_n\rangle.[\![exp_1, \alpha_1]\!]; [\![atexp_2, \alpha_2]\!]; ...; [\![atexp_n, \alpha_n]\!]; \mathtt{infixCheck}(\langle\rangle, 9, L, \langle\alpha_1, \alpha_2, ..., \alpha_n, \alpha\rangle, \langle l_1, ..., l_n\rangle)$, where exp is not of the form $exp\, atexp$.
(G63)	$[\![\mathtt{infix}^l\ \ digit\ \ vid]\!] = \mathtt{infix}(L, digit, vid, l)$
(G64)	$[\![\mathtt{infixr}^l\ \ digit\ \ vid]\!] = \mathtt{infix}(R, digit, vid, l)$
(G65)	$[\![\mathtt{nonfix}^l\ \ vid]\!] = \mathtt{infix}(NONE, 0, vid, l)$
(G66)	$[\![\mathtt{op}^l\ \ vid\ \ exp_1\ \ ...\ \ exp_n]\!] = \mathtt{loc}\ \mathtt{infix}(NONE, 0, vid, l)\ \mathtt{in}\ [\![vid\ \ exp_1\ \ ...\ \ exp_n]\!]$

One of the challenges that come with an extension such as this is ensuring that the constraint generator is kept simple. We do not interleave constraint generation and solving. With this change to the application rule we retain the (relative) simplicity of the constraint generation algorithm.

7.6.4 Constraint solving

To represent errors that can arise when using operator infixity, we extend the set of errors we can represent with two new forms. The first is `infixAsNofix`, which allows us to represent an infix operator being used in a nonfix way without the `op` keyword, and `illegalInfixity` for errors when two infix operators with the same precedence but in a different direction are placed around a term. The extensions to the set ErrKind are shown below.

$$ek \in \mathsf{ErrKind} ::= ... \mid \mathtt{infixAsNofix} \mid \mathtt{illegalInfixity}$$

We extend the set of unifiers to allow mappings from internal type variables to a direction, digit, and value identifier:

$$
\begin{aligned}
u \in \text{Unifier} = \{\textstyle\bigcup_{i=1}^{5} f_i \quad | \quad & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\
& \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\
& \wedge f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\
& \wedge f_4 \in \mathsf{IEqTypeVar} \to (\mathsf{EqTypeStatus} \cup \mathsf{IEqTypeVar}) \\
& \wedge f_5 \in \mathsf{TyConVar} \to \mathsf{TyConVar} \times \mathsf{Direction} \times \mathsf{Digit}\}
\end{aligned}
$$

The new constraint solving rules which handle operator infixity are given in figures 7.29 and 7.30.

Figure 7.29 Constraint solving rules for operator infixity (1 of 2) : State → State

`operator infixity`

(IN1) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'}, 9, digit, dir, \langle\alpha_1, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, ..., l_n\rangle)) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, (\alpha_1 \overset{\{l_1,l_2\}}{=} \alpha_2 \rightarrow \alpha'_2);$
$\texttt{infixCheck}(\overrightarrow{\alpha'}, 9, L, 9, L, \langle\alpha'_2, \alpha_3, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, l_3, ..., l_n\rangle)),$
if $\mathcal{U}(\alpha_1) = \alpha'_1 \rightarrow \alpha'_2$

(IN2) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st},$
$\texttt{infixCheck}(\langle\rangle, prevDigit, digit, dir, \langle\alpha_1, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, ..., l_n\rangle)) \rightarrow$
$\texttt{err}(\langle \texttt{infixAsNofix}, \overline{d} \cup \{l_1\}\rangle),$
if $\mathcal{U}(\alpha) = \langle\alpha', L, digit\rangle \vee \mathcal{U}(\alpha) = \langle\alpha', R, digit\rangle$

(IN3) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'}, prevDigit, prevDir, 0, R, \overrightarrow{\alpha}, \overrightarrow{l_c}, \langle\rangle)) \rightarrow$
$\texttt{isSucc}(\overrightarrow{e}, \overline{m}, \overrightarrow{st})$

(IN4) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st},$
$\texttt{infixCheck}(\langle\alpha', ..., \alpha^{n'}\rangle,$
$prevDigit, prevDir, digit, dir, \langle\alpha_1, ..., \alpha_n\rangle, \langle l_{c1}, ..., l_{cn}\rangle, \langle l_1, ..., l_n\rangle)) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, ct;$
$\texttt{infixCheck}(\langle\alpha', ..., \alpha^{(n-1)'}, \alpha'''_i)\rangle,$
$digit, dir, digit, dir, \langle\alpha_3, ..., \alpha_n\rangle, \langle l_{c1}, ..., l_{c(n-1)}, l_1\rangle, \langle l_3, ..., l_n\rangle),$
if $\mathcal{U}(\alpha_1) = \langle\alpha_i, dir', digit'\rangle \wedge dir = dir' \wedge digit = digit'$
$\wedge\ ct = (\alpha_1 \overset{\{l_{cn},l_1,l_2\}}{=} \alpha^{n'} \rightarrow a_2 \rightarrow a'''_i) \wedge \mathcal{U}(\alpha_i) = \alpha'_i \rightarrow \alpha''_i \rightarrow \alpha'''_i$

(IN5) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st},$
$\texttt{infixCheck}(\langle\alpha', ..., \alpha^{n'}\rangle,$
$prevDigit, prevDir, digit, dir, \langle\alpha_1, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, ..., l_n\rangle)) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, (\alpha_2 \overset{\{l_2,l_3\}}{=} \alpha_3 \rightarrow \alpha'_2)$
$\texttt{infixCheck}(\langle\alpha', ..., \alpha^{n'}\rangle,$
$prevDigit, prevDir, digit, dir, \langle\alpha_1, \alpha'_2, \alpha_4, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, l_4, ..., l_n\rangle)),$
if $\mathcal{U}(\alpha_1) = \langle\alpha', dir', digit'\rangle \wedge dir = dir' \wedge digit = digit'$
$\wedge\ dir' \neq NONE \wedge \mathcal{U}(\alpha_2) = \alpha'_1 \rightarrow \alpha'_2$

(IN6) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'},$
$prevDigit, prevDir, digit, L, \langle\alpha_1, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, ..., l_n\rangle)) \rightarrow$
$\texttt{err}(\langle \texttt{illegalInfixity}, \overline{d} \cup \{l_1\}\rangle),$
if $\mathcal{U}(\alpha_1) = \langle\alpha', dir', digit'\rangle \wedge\ \ prevDigit = digit' \wedge prevDir \neq dir'$
$\wedge dir' \neq NONE$

Figure 7.30 Constraint solving rules for operator infixity (2 of 2) : State → State

(IN7) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'}, prevDigit, prevDir, digit, dir, \langle\alpha_1, ..., \alpha_n\rangle, \overrightarrow{l_c}, \langle l_1, ..., l_n\rangle)) \rightarrow$
$\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'}@\alpha_1, prevDigit, prevDir, digit, dir, \langle\alpha_2, ..., \alpha_n\rangle, \overrightarrow{l_c}@l_1, \langle l_2..., l_n\rangle))),$
if $\mathcal{U}(\alpha_1) \neq \langle\alpha', dir', digit'\rangle \wedge \mathcal{U}(\alpha_1) \neq \tau_1 \rightarrow \tau_2$

(IN8) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'}, prevDigit, prevDir, digit, L, \langle\rangle, \overrightarrow{l_c}, \langle\rangle)) \rightarrow$
$\texttt{infixCheck}(\langle\rangle, digit, R, digit, R, \mathsf{rev}(\overrightarrow{\alpha'}), \langle\rangle, \mathsf{rev}(\overrightarrow{l_c}))$

(IN9) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infixCheck}(\overrightarrow{\alpha'}, prevDigit, prevDir, digit, R, \langle\rangle, \overrightarrow{l_c}, \langle\rangle)) \rightarrow$
$\texttt{infixCheck}(\langle\rangle, digit - 1, L, digit - 1, L, \mathsf{rev}(\overrightarrow{\alpha'}), \langle\rangle, \mathsf{rev}(\overrightarrow{l_c})),$
if $digit \neq 0$

(IN10) $\texttt{slv}(\overrightarrow{e}, \overline{d}, \overline{m}, \overrightarrow{st}, \texttt{infix}(dir, digit, vid, l)) \rightarrow$
$\texttt{isSucc}(\overrightarrow{e}; \downarrow vid \overset{\overline{d}\cup\{l\}}{=} \langle\alpha, dir, digit\rangle, \overline{m}, \overrightarrow{st})$
if $\overrightarrow{e}(vid) = \downarrow vid \overset{\overline{d}}{=} \alpha$

We solve these constraints by handling the top precedence operators which associate to the *left* as described in the rules below. When we reach the end of the applications, we *reverse* the order of the internal type variables, labels etc. and then do the same thing again, this time checking for *right* associativity. By reversing the order of the internal type variables, we can reuse the mechanism for left associativity, rather that implementing machinery to handle right associativity directly. After we have finished this, we drop down to the next infixity level (if it exists, succeed otherwise), reversing the set of internal type variables again and handling *left* infix operators, and continue until we succeed.

Our environment `infixCheck` has seven arguments. The first is the tuple of internal type variables which we have already processed for the current infixity precedence and direction we are currently searching for. The second argument indicates the previous digit of precedence for the last infix operator that we detected. The third argument is the current digit of operator precedence we are currently handling. The fourth argument is the direction of associativity which we are currently dealing with, the fifth argument is the tuple of internal type variables still to be processed. The sixth argument is the labels for the internal type variables that we have already processed and the final argument is the label tuple for the internal type variables which we have yet to process.

We discuss each constraint solving rule below.

- We generate errors involving infixity in rules (IN6), which handles the case where we see that some value is surrounded by two infix operators of the

same precedence but operating in a different direction, and in rule (IN2), which detects that a function which is declared as an infix operator is being used in a manner as if it were not infix.

- In rule (IN3), we handle the case where we are handling infix operators which associate to the right with precedence 0 and we have no more internal type variables yet to handle. This means that we have finished doing applications, and we generate success. Similarly:
  - Rule (IN8) handles the case where we have run out of internal type variables to handle are we are checking for left-associativity, and so it reverses the internal type variables we have found so far and we will then go on to look for operators which associate to the right.
  - Rule (IN9) is the case where we again have run out of internal type variables to look for but we have just handled all operators which associate to the right with some precedence greater than 0. In this case we decrease the precedence number that we are looking for, reverse the internal type variable and label sets, and start looking for operators which associate to the left with this new precedence.
- In rule (IN7), we deal with the case where we are looking at an internal type variable and we notice that it is not an infix operator, nor is it a function. In this case, we do nothing, and move on to process the next internal type variable.
- We handle the case of non-infix function application in rule (IN1). If we see that the internal type variable that we are currently processing is an arrow type, then we apply an argument to it.
- In rules (IN4) and (IN5), we handle the case where we are dealing with an infix operator which matches the precedence and the direction that we are looking for. If we see that the internal type variable to the right of this is mapped to in the set of unifiers as an arrow type, then we apply an argument to it in rule (IN5). Otherwise, we use rule (IN4) and apply arguments to the infix operator.
- Finally, in rule (IN10), we handle the case where we are dealing with an environment which specifies that a new infix status has been defined for some value identifier. In this case we generate a new binder for that value identifier so that we can keep track of its status correctly.

7.6.5 Slicing

We must update our slicing mechanism involving signatures to handle the new keywords that were introduced. First, our syntax of programs is updated as follows:

$$\mathsf{Prod} ::= \ldots \mid \texttt{decInfix} \mid \texttt{decInfixr} \mid \texttt{decNonfix} \mid \texttt{decOp}$$

We also update toTree to handle new keywords presented in this section.

$$\mathsf{toTree}(\lceil \texttt{decInfix}\ \mathit{digit}\ \mathit{vid} \rceil^l) = \langle \langle \texttt{dec}, \texttt{decInfix} \rangle, l, \langle \mathit{digit}, \mathsf{toTree}(\mathit{vid}) \rangle \rangle$$

$$\mathsf{toTree}(\lceil \texttt{decInfixr}\ \mathit{digit}\ \mathit{vid} \rceil^l) = \langle \langle \texttt{dec}, \texttt{decInfixr} \rangle, l, \langle \mathit{digit}, \mathsf{toTree}(\mathit{vid}) \rangle \rangle$$

$$\mathsf{toTree}(\lceil \texttt{decNonfix}\ \mathit{vid} \rceil^l) = \langle \langle \texttt{dec}, \texttt{decNonfix} \rangle, l, \mathsf{toTree}(\mathit{vid}) \rangle$$

$$\mathsf{toTree}(\lceil \texttt{decOp}\ \mathit{vid} \rceil^l) = \langle \langle \texttt{dec}, \texttt{decOp} \rangle, l, \mathsf{toTree}(\mathit{vid}) \rangle$$

In this chapter we have extended the theory defined in chapter 4 with several new extensions all of which contribute to supporting the Standard ML language. Our analysis engine has been enhanced to support all of the extensions shown here and more, and handles cases which we do not present here so that the theory is not vastly complicated, such as tuples and the equality types support for them, datatypes with multiple arguments, mutual recursion etc. Note that e.g. equality types are supported for all Standard ML features, and not just for the subset presented in the theory. We believe that additional extensions can be written in a similar way to that presented here, by defining rules which govern how each Skalpel component operates, in order to target new languages.

7.7 Summary

This chapter has updated some previously-existing extensions to operate with the latest version of thee Skalpel core described in this thesis. We have also demonstrated a number of entirely new key improvements to Skalpel in order to support more features that are present in the Standard ML programming language such as equality types and type sharing. In the next chapter we shall given an overview of the properties of Skalpel, and show that the extensions given in this chapter to not break e.g. termination.

Chapter 8

Properties of Skalpel

In this chapter, we discuss some of the properties of Skalpel. Section 8.1 gives some discussion on the properties of the Skalpel core, section 8.2 gives discussion on the properties of the extensions and section 2.3 discusses how Skalpel matches the criteria which are laid out in [YMTW00] which describe how error reporting should be approached.

8.1 Properties of the Skalpel Core and Extensions

8.1.1 Skalpel Core: Constraint Generator

Lemma 8.1.1 (Constraint generator compositionality)**.**
The constraint generator shown in figure 4.17 is compositional.

Proof. The algorithm given in figure 4.17 is compositionally built on the syntax of figure 4.2. □

Next, we show that constraint generation is linear in size.

Lemma 8.1.2 (Size of Constraint Generation)**.**
The constraint generator shown in figure 4.17 is linear in the program's size.

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

Lemma 8.1.3 (Termination of Constraint Generation Algorithm)**.** *The constraint generator shown in figure 4.17 terminates.*

Proof. Let us define an *atomic constraint generation rule* as constraint generation rule which does not have a recursive call inside. For example, the atomic constraint generation rules in figure 4.17 are (G1), (G5), (G6), (G7) (G9), (G10), (G13), (G14), (G19) and (G21). For a constraint generation run

$$\mathsf{cstgen}'(\mathcal{P}^L, \overline{v})$$

either $\mathcal{P}^L$ will be atomic in nature or it will not. If it is not, we recurse with $\mathsf{cstgen}'(\mathcal{P}^{L\prime}, \overline{v}')$, on some $\mathcal{P}^{L\prime}$ inside $\mathcal{P}^L$, such that $\mathcal{P}^{L\prime}$ is strictly smaller than $\mathcal{P}^L$. Rules which recurse with strictly smaller parts of external syntax are rules (G2) (`let` syntax removed in recursive call), (G3) (application syntax removed), (G4) (`fn` syntax removed), (G8) (application removed), (G11) (application removed), (G12) (arrow removed), (G16) (`of` syntax removed), (G17) (`val rec` removed), (G18) (`datatype` syntax removed), (G20) (`structure` syntax removed), and (G22) (`struct` syntax removed). When we inevitably reach an atomic $\mathcal{P}^L$, we halt and return the generated environment e. □

8.1.2 Skalpel Core: Constraint Solver

Our constraint solver for the Skalpel core will always terminate.

Lemma 8.1.4 (Constraint Solving Terminates)**.**
The constraint solver given in figure 4.19 terminates.

Proof. By inspection of the rules:

- (R) Flips constraints. Constraints which are flipped can never be re-flipped, as it forces the variable on to the left hand side which will result in completely finishing the environment we are handling under rules (U4) or (U6), otherwise rules (U1), (U2), and (U3) can be used which all result in completely resolving this environment by way of either success or error.
- (S1) Throws away argument/adds to environment or unifier, and checks for success.
- (S2) Same as rule (D) but for equality constraints.
- (S3) Breaks two application into two equality constraint terms. We never build new applications of constraints, so we cannot return to this point.
- (S4) Breaks two arrow types into two equality constraints. We never build new arrow types and so cannot return to this point.

- (S5) Breaks an equality constraint of an application and an arrow type down into a new equality constraint containing no applications or arrow types, with the right hand side set as an 'arr' form. The left hand side of the equality constraint will always be a variable, an 'arr' form, or a dependent. In the case of a dependent form, we use rule (D) and return to either a variable form or an 'arr' form. In the case of an 'arr' form, we will completely solve this environment under rule (S1). In the case of a variable, we rely on rules (U1-U6), none of which build equality constraints except in the case of (U6), which will immediately take the result of the application of the variable on the left hand side to the unifier function and map that to the 'arr' form under rule (U3).
- (S6) Terminates immediately with an error
- (U1) Terminates immediately with an error.
- (U2) Throws away argument/adds to environment or unifier, and checks for success.
- (U3) Throws away argument/adds to environment or unifier, and checks for success.
- (U4) Breaks apart an equality constraint between a variable and some other constraint, and assigns the variable in the unifier function to the result of solving the constraint. The new environment to solve is strictly smaller.
- (U6) Applies the unifier function to a variable, and creates a new equality constraint with a new left hand side. One of the other (U1)-(U4) rules will be called, and as rules (U1)-(U3) all completely solve the environment we are handling now, this rule hinges on (U4) to terminates.
- (B) Throws away argument/adds to environment or unifier, and checks for success.
- (B2) Throws away argument/adds to environment or unifier, and checks for success.
- (X) Simply removes existential quantification. We do not ever re-existentially quantify a term.
- (E) Throws away argument/adds to environment or unifier, and checks for success.
- (D) We remove labels in this rule, and do not put them on in any other rule (except P1, but the resulting term is added to the environment in the first argument, which we never subtract from).

- (V) Throws away argument/adds to environment or unifier, and checks for success.
- (C1) Breaks a composition environment down to solve the first part (e_1), then the second (e_2). As e_1 is never the same as $e_1; e_2$ (unless e2 is top, in which case after solving e_1 we terminate under rule (E)), it can be considered strictly 'smaller', and we do not ever build environments which are 'larger' than the initial environment we are solving (where a 'large' environment is deemed as having more composition operators than a 'smaller' environment, excluding the additions of the top symbol, which we have mentioned already).
- (A1) Throws away an accessor to create an equality constraint. That equality constraint will always terminate under one of the unifier access rules, and we never build any accessors during solving.
- (A3) Throws away argument/adds to environment or unifier, and checks for success.
- (P1) Throws away argument/adds to environment or unifier, and checks for success.
- (P2) Unwraps a polymorphic environment, but care must be taken as we build another environment which is a subset of the one we are handling, and put it on the stack. As the new stack element is a binder wrapped in a polymorphic environment, which will strictly always result in rule (P1) being called, which is already terminating, rules (P1) and (P5) make this rule terminating.
- (P3) Same as rule (D) but inside polymorphic environments.
- (P4) Same as rule (C1) but inside polymorphic environments.
- (P5) Unwraps a polymorphic environment. The environment unrwapped can in no case be re-wrapped and so solving on recursion is strictly smaller.
- (P6) Same as rule (X) but inside polymorphic environments.

□

8.1.3 Minimiser

The next lemma shows that no label which contributes to an error is removed during minimisation and no remaining labels after minimisation are extraneous.

Lemma 8.1.5. *After a run* $\langle e, \langle ek, \bar{l}\rangle\rangle \overset{\mathsf{min}}{\rightarrow} \langle ek', \bar{l}'\rangle$ *of the minimisation algorithm shown in figure 3.19, the following hold:*

1. $\forall l \in \bar{l}'$, *it holds* $\mathtt{slv}(\top, \varnothing, \varnothing, \varnothing, e') \rightarrow^* \mathtt{succ}$,
   where $e' = \mathsf{filt}(e, \mathsf{labs}(e) \backslash \{l\}, \bar{l}' \backslash \{l\})$
2. $\mathtt{slv}(\top, \varnothing, \varnothing, \varnothing, e'') \rightarrow^* \mathtt{err}(er)$,
   where $e'' = \mathsf{filt}(e, \mathsf{labs}(e), \bar{l}')$.

Proof. In order for a label which contributes to an error to be removed erroneously, we must remove a label in $\overline{l_1} \cup \overline{l_2} \cup \{l\}$ of a minimisation step. In the case of $\overline{l_1}$ and $\overline{l_2}$, we only take out elements which are not in the label set that was reported by an additional run of the constraint solving algorithm (rule (MIN1)). Any labels we take out from these sets are extraneous, as the constraint solver therefore managed to find the same error without these elements. As $\{l\} \subset \bar{l}$, and we have established we do not erroneously throw any labels away from $\bar{l}$, $\{l\}$ is therefore not thrown away erroneously. Therefore no labels are erroneously thrown away in $\overline{l_1} \cup \overline{l_2} \cup \{l\}$, and so the minimisation algorithm only generates a strictly smaller error. Furthermore, as we test all labels during minimisation, no label is untested, and so all labels remaining after minimisation are part of the minimised error. □

8.1.4 Enumeration

After the enumeration algorithm has stopped, the errors that have been found are all the minimal type errors in the analyzed piece of code. The next lemma shows that any errors found during enumeration and presented to the user will always be minimal.

Lemma 8.1.6. *Given an output* $\mathtt{errors}(\overline{er})$ *of the enumeration algorithm (figure 3.19), for all errors err in* $\overline{er}$ *it holds that err is a minimal error.*

Proof. Any errors which are found by the enumeration algorithm (rule `(ENUM4)` handles the case where an error is found) will always be minimal as errors in this rule, the sole rule for dealing with errors, are passed to the minimisation algorithm. Lemma 8.1.5 shows that this error will therefore be minimal. We never add any extra labels to errors during minimisation, and so we retain the property that these errors are minimal. □

8.1.5 Slicing

We now show that in figure 3.23, only those parts of the program which they wrote and contribute to the error are shown to the user (that is, syntax is not

shown which was not written by the user and which does not contribute to the error).

Lemma 8.1.7. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, it is the case that for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \overline{l}, id\rangle$ *or* $\langle dot, id\rangle$ *or* id*, then* id *is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. By inspection of the definition of toTree in figure 3.23. For any rule matching labelled program syntax $\mathcal{P}^L$ shown in figure 3.23, it holds that if $\mathsf{toTree}(\mathcal{P}^L)$ is of the form $\langle node, \overline{l}, id\rangle$ or $\langle dot, id\rangle$ or id, then $id = \mathcal{P}^L$, or $\mathcal{P}^L$ is a piece of syntax containing id. As we only generate program labels (l) for parts of the user program and nothing else (figure 4.1), and the input of toTree is syntax that is represented by a program label l, it is the case that $\mathcal{P}^L$ is a piece of labelled syntax written by the user. We have established the output of toTree is equal to or is part of $\mathcal{P}^L$, and we have established that $\mathcal{P}^L$ is syntax written by the user, so it follows that the output of toTree is a piece of syntax that was written by the user and sent as input to Skalpel. □

8.1.6 Overall system

Lemma 8.1.8. *For an erroneous program taken as input, Skalpel produces minimal type error slices of all errors present in the user program, which contain all and only parts of the user program responsible for the errors.*

Proof. Given a user program P, we produce a labelled program $\mathcal{P}^L$ and start the constraint generation algorithm with $\mathsf{cstgen}(\mathcal{P}^L)$, producing an environment e. We start enumeration with $\mathsf{enum}(e)$ and generate a filter set $\overline{l}$, and if $\mathsf{filt}(e, \mathsf{labs}(e), \overline{l}) \xrightarrow{\mathsf{isErr}} er$ we generate a minimal error with $\langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \overline{l}'\rangle$, and collect all such errors that arise using different filters in $\overline{er}$. We then perform slicing on all of the minimal errors in $\overline{er}$ and present them to the user. The combination of lemmas 8.1.5, 8.1.6, and 8.1.7 show that this lemma holds. □

8.2 Extensions to the Core

The extensions to the Skalpel core given in chapter 7 do not break any of the properties given above. We give below lemmas and proofs which demonstrate this to be the case.

There are some lemmas and proofs which are not affected and so we do not discuss those. These are the lemmas concerning compositionality of the constraint

generator, constraints being linear in the program size, and the minimization, enumeration and final lemmas which all operate at a higher level and so are therefore unaffected by such extensions. Termination properties however are affected, and so we inspect each extension and verify no properties are broken.

8.2.1 Local declarations

Lemma 8.2.1. *The extension to constraint generation given in section 6.1 does not break the termination property of the constraint generator.*

Proof. Rule (G29) is not atomic in nature, and recurses only on two $\mathcal{P}^L$ values, both of which are smaller than some $\mathcal{P}^{L\prime}$ taken as input, as we remove the `loc` syntax. Inevitably we will recurse with $\mathsf{cstgen}'(\mathcal{P}^L{}_1, v_1)$ and $\mathsf{cstgen}'(\mathcal{P}^L{}_2, v_2)$, where $\mathcal{P}^L{}_1$ and $\mathcal{P}^L{}_2$ are atomic pieces of syntax by lemma 8.1.3 and we will terminate. □

Lemma 8.2.2. *The extension to constraint solver given in section 6.1 does not break the termination property of the constraint solver.*

Proof. Rule (L1) recurses the constraint solver on two environments, e_1 and e_2, both of which are smaller in size than some e taken as input as the **loc** environment is removed from e. This rule solves these environments consecutively. As solving all other environments has been shown to terminate (lemma 8.1.4), this property holds. □

Lemma 8.2.3. *Given an output of* toTree *(figure 3.23), $tree_{out}$, it is the case that for any $tree_{inside}$ inside $tree_{out}$, if $tree_{out}$ or $tree_{inside}$ is of the form $\langle node, \overline{l}, id\rangle$ or $\langle dot, id\rangle$ or id, then id is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.2 Type declarations

Lemma 8.2.4. *The extension to constraint generation given in section 6.2 does not break the termination property of the constraint generator.*

Proof. Rule (G13) is an atomic constraint generation rule and so terminates. Rule (G18) recurses on a datatype name, and composes the resulting environment with a `loc` expression. As both have been shown to terminate in lemmas 8.2.2 and

8.1.4, this rule terminates. Rule (G30) recurses on a datatype name, an internal type, and a `loc` environment, which have been shown to terminate in lemmas 8.2.2 and 8.1.4, and so the termination property holds. □

Lemma 8.2.5. *The extension to the constraint solver given in section 6.2 does not break the termination property of the constraint solver.*

Proof. Rule (R) flips constraints. Once flipped, constraints are never re-flipped, so this rule is terminating. Rules (S10), (B1) and (B6) check for success and so terminate. Rule (S9) performs substitution on internal type constructors and builds an internal type. Once built, this type constructor can never again be decomposed back to the initial type constructor, and so this rule terminates. Similarly for rules (S11) and (S13). Rule (S12) breaks applications for internal type constructors into two smaller constraints and recurses. As these are never recomposed, this is terminating. Rules (A1) and (A2) deal with accessors and create new constraints to be solved - as we never rebuild accessors and all other rules are shown to terminate, termination holds. □

Lemma 8.2.6. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, it is the case that for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \overline{l}, id\rangle$ *or* $\langle dot, id\rangle$ *or* id*, then* id *is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.3 Type annotations

Lemma 8.2.7. *The extension to the constraint generator given in section 6.3 does not break the termination property of the constraint generator.*

Proof. Rules (G46) and (G47) recurse on expressions and internal types, both of which have been shown to terminate in lemma 8.1.3. Rules (G48) and (G49) are atomic in nature and so terminate. Rule (G50) recurses on type variable sequences, calling rule (G48) which is atomic so terminates. Rule (G17) creates a `poly` form, which has already been shown to terminate, and all recursive calls inside have already been shown also to terminate. □

Lemma 8.2.8. *The extension to the constraint solver given in section 6.3 does not break the termination property of the constraint solver.*

Proof. Rules (B9) and (B10) deal with their unconfirmed binder argument by either adding them to the environment or throwing them away, and so terminate. Rule (OR) deals with the dependencies annotated on an environment and union them with the dependency set, then recursing on the environment argument. As we never add dependencies back on to an environment given as an argument, termination holds. □

Lemma 8.2.9. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, it is the case that for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \bar{l}, id \rangle$ *or* $\langle dot, id \rangle$ *or* id*, then* id *is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.4 Signatures

Lemma 8.2.10. *The extension to the constraint generator given in section 6.4 does not break the termination property of the constraint generator.*

Proof. Rules (G33) and (G48) are both atomic constraint generation rules and so terminate. Rules (G36), (G38), (G35) and (G17) have no new function calls and so their termination property is unaffected. Rule (G37) strips and analyzes each `sigexp` form using rules (G39) and (G40), as in ML there can only be a finite number of these this rule terminates provided the other rules also terminate. Rule (G34) decomposes each `spec` form and recurses, so this terminates providing the other rules terminate. Rule (G32) has only one recursive call which has already been shown to terminate. Rules (G39) and (G40) recurse on signature and (strictly smaller) structure expressions and so will inevitably reach an atomic form and terminate. Rule (G41) handles each *topdec*, of which there can only be a finite number and each form has been shown to terminate therefore termination holds. □

Lemma 8.2.11. *The extension to the constraint solver given in section 6.4 does not break the termination property of the constraint solver.*

Proof. Rules (S16), (S17), (B1), (B7), (B9), (B10), (I1), (P1), (SM1), (SM3), (SM6), (SM7), (SM10) and (SM11) all either raise an error or check for success, and so terminate. Rule (S14) decomposes an internal type into an application of an internal type constructor to an internal type, and such constraints and never recomposed, so this rule terminates. Similarly with rule (S15). Rule (SU5)

strips dependencies off generality constraints and recurses, and such dependencies are never re-applied, and so this rule is terminating. Rule (SM2) breaks apart two environments, and solves each individually, rules (SM4), (SM5) and (SM6) deal with binders for value identifiers and type constructors, creating generality constraints between what exists in the set of unifiers. As we do this at most once for that binder, these rules are terminating. Rule (SM12) deals with dependencies on environments, which are stripped and never re-applied, rule (SM13) throws away an equality constraint and deals only with the environment, and rule (SM14) removes existential constraints which are not re-applied so these rules terminate. Rules (SU1) and (SU3) build environments from generality constraints, this is done strictly once as we never rebuild generality constraints in any other rules, and so termination holds. □

Lemma 8.2.12. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \overline{l}, id\rangle$ *or* $\langle dot, id\rangle$ *or* id*, then* id *is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.5 Equality types

Lemma 8.2.13. *The extension to the constraint generator given in section 7.1 does not break the termination property of the constraint generator.*

Proof. All constraint generation rules that have been extended are only extended with atomic forms which require no additional recursion, and so the termination property is not broken. □

Lemma 8.2.14. *The extension to the constraint solver given in section 7.1 does not break the termination property of the constraint solver.*

Proof. In all cases, where pre-existing rules have been modified such that variables have been annotated with equality types, it is the case that the rules have been changed to create new constraints between equality types, so termination is deferred to rules which solve those constraints. We shall inspect each solving rule for equality types in turn. Rule (E1) breaks its constraint to be solved into two constraints to be solved individually. The constraint pushed onto the stack has already been shown to terminate, so this rule terminates depending on termination for constraints of equality between θ variables. The same applies to rules (E2) and (E3). Rule (E4) checks for success and throws away its constraint so terminates, and (E5) terminates immediately with error so termination holds. □

Lemma 8.2.15. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, it is the case that for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \overline{l}, id\rangle$ *or* $\langle dot, id\rangle$ *or* id*, then* id *is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.6 Abstract type declarations

In this section no proof is needed that the constraint solver terminates, as it is unchanged as a result of this extension. The same applies for the lemmas on producing slices that are strictly only part of the user syntax.

Lemma 8.2.16. *The extension to the constraint generator given in section 7.2 does not break the termination property of the constraint generator.*

Proof. Rule (G61) recurses on the constraint generator with pieces of constraint syntax *dn*, *cb* and *dec*. As all recursions of this form have already been shown to terminate in lemma 8.1.3, so this property holds. □

8.2.7 Duplicate Identifiers in Specifications

Lemma 8.2.17. *The extension to the constraint generator given in section 7.3 does not break the termination property of the constraint generator.*

Proof. The constraint generation rule (G34) is extended to be composed with another environment which is not recursed upon, so the termination property holds. □

Lemma 8.2.18. *The extension to the constraint solver given in section 7.3 does not break the termination property of the constraint solver.*

Proof. Rule (I1) checks for success and so terminates. Rule (I2) calls the `sameId` function which handles environments in the same way as they are in the constraint solver, and so the termination property holds. □

Lemma 8.2.19. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, it is the case that for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \overline{l}, id\rangle$ *or* $\langle dot, id\rangle$ *or* id*, then* id *is part of the program syntax the user wrote.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.8 Include Specifications

In this section no proof is needed that the constraint solver terminates, as it is unchanged as a result of this extension.

Lemma 8.2.20. *The extension to the constraint generator given in section 7.4 does not break the termination property of the constraint generator.*

Proof. Rule (G62) simply existentially quantifies the result of a recursion on the constraint generator with the *sigexp* it took as an argument. As $[\![sigexp, ev]\!]$ has already been shown to terminate in lemma 8.1.3, this property holds. □

Lemma 8.2.21. *Given an output of* toTree *(figure 3.23),* $tree_{out}$*, it is the case that for any* $tree_{inside}$ *inside* $tree_{out}$*, if* $tree_{out}$ *or* $tree_{inside}$ *is of the form* $\langle node, \overline{l}, id\rangle$ *or* $\langle dot, id\rangle$ *or* id*, then* id *is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.9 Type Sharing

Lemma 8.2.22. *The extension to the constraint generator given in section 7.5 does not break the termination property of the constraint generator.*

Proof. Rule (G34) does not add any recursion since the rule was last examined in lemma 8.1.3 and so terminates. Rule (G35) recurses on expressions, which have been shown to terminate in lemma 8.1.3, so the termination property holds. □

Lemma 8.2.23. *The extension to the constraint solver given in section 7.5 does not break the termination property of the constraint solver.*

Proof. Rule (SM15) strips existential quantifiers from a constraint. As existential quantifiers once stripped are never applied again in any other rule, this constraint generation call is terminating. Rule (SH1) calls the `vidSharingCheck` function and so termination depends on the termination of `vidSharingCheck`.

Rule (VSC1) looks up the environment variable given in the argument in the unifier map. As this then calls one of the following rules, and they are shown to terminate, this rule terminates. Rule (VSC2) checks for success immediately, and so terminates. Rule (VSC3) strips dependencies off an environment and recurses,

and as they are never added on to an environment again, this rule terminates. Rules (VSC4), (VSC5) and (VSC6) raise an error and so terminate. Rule (VSC7) checks for success and so terminates. Rule (VSC8) splits a composition environment into two parts and solves each individually - as they are never recomposed this rule terminates. Rule (VSC9) checks for success and so terminates. □

Lemma 8.2.24. *Given an output of* toTree *(figure 3.23), $tree_{out}$, it is the case that for any $tree_{inside}$ inside $tree_{out}$, if $tree_{out}$ or $tree_{inside}$ is of the form $\langle node, \overline{l}, id\rangle$ or $\langle dot, id\rangle$ or id, then id is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

8.2.10 Operator Infixity

Lemma 8.2.25. *The extension to the constraint generator given in section 7.6 does not break the termination property of the constraint generator.*

Proof. Rules (G63), (G64) and (G65) are all atomic constraint generation rules and so terminate. Rules (G66) and (G3) have been extended only with non-recursive forms, and all recursive calls in these rules have already been shown to terminate, so the termination property holds. □

Lemma 8.2.26. *The extension to the constraint solver given in section 7.6 does not break the termination property of the constraint solver.*

Proof. Rules (IN2), (IN3), (IN6) and (IN10) either raise an error or check for success, and so terminate. Rule (IN9) changes the direction of analyzing the infixity constraints at a lower precedence in a different direction. As we only decrement the infixity precedence which we are analyzing, and rule (IN3) terminates at the lowest precedence, rule (IN9) is terminating. Rule (IN8) flips the direction of precedence that we are analyzing without decrementing the infixity precedence. As we never re-flip direction of analysis without first decrementing the infixity precedence (rule (IN9)), this rule is terminating. Rules (IN1), (IN4), (IN5) and (IN7) all process the type variable they are analyzing and move forward to the next variable to process, which is terminating due to rules (IN8) and (IN9) dealing with reaching the $\langle\rangle$ case of type variables to handle. □

Lemma 8.2.27. *If the output of* toTree *(figure 3.23) or any sub-tree of that tree is of the form $\langle node, \overline{l}, id\rangle$ or $\langle dot, id\rangle$ or id, then id is part of the program syntax the user wrote and was sent as input to Skalpel.*

Proof. We extend toTree to handle forms which represent part of the user syntax, and so this lemma holds due to lemma 8.1.7. □

Chapter 9

Conclusions and Future Work

9.1 Conclusion

In this thesis, we have given a critique of and enhanced the initial Skalpel presentation in chapters 3 and 4 respectively. We have also updated the previously existing extensions to be correctly defined on the latest version of the Skalpel core in chapter 6, and created several new extensions to this theory in chapter 7. We have also looked at the properties of this theory, such as termination of algorithms, which is present in chapter 8. In addition, the efficiency of the implementation has been inspected, including a full profile of the Skalpel analysis engine which can be found in section 4.7. Furthermore, we have also made a number of practical contributions, such as developing a new method of documenting Standard ML code in appendix B, and developing frameworks for debugging and testing designed to aid in future research in appendix C.

9.2 Abstracting parts of the implementation to support other languages

It is hoped that in the future Skalpel will support other languages than just Standard ML. We believe the type error slicing techniques discussed in this thesis make finding and fixing errors easier, and so use with other languages will allow this work to benefit a greater number of users. Furthermore, with access to this wider user base, we will be able to test the effectiveness of Skalpel more easily.

We now discuss options for future work. In order to get closer to this goal, one approach could be to attempt to abstract away parts of the current implementa-

tion, such as the parser for the language that we wish to handle, the constraint generation rules, and so on. Ideally, we would take for example as arguments to the analysis engine the parser, abstract syntax tree, constraint generation rules, constraint solving rules etc. along with a program and from this produce slices of the untypable parts of the supplied program if they exist.

It is believed that setting up a framework such as this would not cause a significant performance hit to the implementation, as the time spent interpreting the various new files submitted by the user (e.g. constraint generator) would likely be insignificant compared to the amount of time we spent solving constraints that have been generated for the user program.

Some investigation would need to be carried out first to determine what would be the best way that e.g. constraints could be represented. Ideally, the procedure of constructing such files to allow Skalpel to support another language should not be so arduous that it would have to be done by a Skalpel developer, and end users would be able to build such files to support their own languages without a great amount of effort.

9.3 Making the Skalpel implementation more efficient

In its current state the Skalpel implementation can take an arbitrarily long time to generate and solve constraints for programs with a large size.

Were the implementation to be made more efficient, we could be able to reach the point where Skalpel could be run automatically whenever, for example, the user saves their file of source code.

One route to this goal would be to enhance Skalpel so that it can use the results of previous runs via some kind of cache (for example using hash consing). For example, were a user to change part of a program and run Skalpel again, Skalpel could filter out any constraints which were solved which have labels attached which correspond to the program points which the user changed. It is quite possible that we will not be able to have Skalpel run automatically when the user saves their file without some kind of recycling of previous computations.

In addition, it would be interesting to see how different features of an SML program have an impact on performance and what relationship that has to the number of labels that were generated for that program, and how efficiency changes affect each language construct.

9.3.1 Booleans in hash table mapping

We introduced in chapter 4 an approach where maps from integers to booleans are used to represent program points. Booleans have been chosen as the range of this map with the goal that minimisation could be done in a cleaner way, by toggling the boolean flag which represents whether this label is to be sliced out. While yet untested as issues involving memory cost are currently more important, it is very likely this will create a saving in the time taken to do minimisation, in addition to the space saving that will be made. This approach is promising, and further investigation is recommended, building on the work written in the source code repository.

9.3.2 Different phases of slice specificity

The master branch of the source repository contains a system where the constraint generator is executed once where all constraints are labelled with maximum precision. This allows us to be very specific in the error reports as we have a label for every possible part of an error we wish to express.

Outlined here is an alternative approach, where multiple constraint generation runs are completed to build constraints with increasing specificity with respect to program points (labels).

The reason for doing this is that if there are less program labels for a given program, the time taken to do all operations related to program labelling (such as union operations) could be reduced significantly. However, it is not acceptable for us to provide poor quality error reports to the user, so we must also generate high quality error reports that adhere to the properties described discussed in section 2.3.

Consider the case where a user has defined 1000 complicated functions, and an error exists between a small number of them. With the current approach, the time spent unioning the labels while dealing with the constraints of these functions could be arbitrarily large and Skalpel could fail badly e.g. run out of memory. We could either automatically, or at the users discretion, employ a tactic whereby we only generate one label for a function and all the constraints in that function are labelled with the same label. During unification, we would then discover the same error as we would previously (as the same constraints exist), but we would only be able to identify the e.g. 3 functions that are involved in the generated error. We would then need to relabel the program again for those 3 functions to some degree off specificity. If we assume that we then label

the functions involved again with maximum precision, we would then perform unification again and get a precise error which can then be presented to the user.

It is quite conceivable that the time spent doing unification twice (or more if desired) would outweigh the benefit that may come from having far smaller label sets, but it is possible such an approach may have some merit when configured correctly (e.g. doing this for `let`-expressions at the user's discretion). Most importantly, this would allow us to be able to report errors faster at the cost of lesser precision, where we would then present the user with a more precise error when it is available. In the cases of complicated programs where Skalpel may run out of memory, we would at least be able to produce some kind of error report, even if it is not absolutely precise.

Experimenting with this approach to program labelling is left for future work.

9.4 Improve the communication mechanism between Emacs and Skalpel

The Emacs front-end currently uses a file-based approach to communicate with Skalpel, and in addition, this file-based approach creates files which are executed by Emacs. Developing a solution to communicate with Emacs where we do not execute files (so for example communication is all done via sockets), or better yet do not use files at all, is desirable.

9.5 Extend the test framework

There are many useful extensions that can be made to the test framework so that the implementation can become more robust than it currently is. Such extensions include verifying that Emacs can actually display slices to the user, that SML/NJ can build the Skalpel project, and daily profiling of the analysis engine binary.

In general, any addition to the test framework which tests some component of the Skalpel project which was previously untested (such as which Emacs versions we support, whether builds for supported operating systems can be constructed, etc.) is desirable.

Appendix A

List of Symbols, Sets, and Other Abbreviations

This appendix gives a list of the various symbols, sets, functions, and other abbreviations that have been used in this thesis.

Appendices A.1 and A.2 list symbols used, the set they range over (written N/A in section A.1 if this is not applicable), a description and point of definition (section A.1 sorts this information by symbol, A.2 sorts this information by set). Sections A.3 and A.4 give brief descriptions and points of definition for function definitions and other abbreviations used, respectively. Sections are denoted § and figures are denoted $\mathcal{F}$, with the section or figure in parenthesis denoting the definition location in [Rah10], if applicable.

A.1 Symbols

Symbol Set	Brief Description	Definition
$\bullet$ N/A	When used as $fct \bullet e$, it means that a functor is applied for an environment.	(§14.9.2)
$\rightsquigarrow$ N/A	Works like the arrow type ($\rightarrow$), but works on environment variables to work on functors.	(§14.9.2)
$\diamond$ N/A	Augments a type scheme with a set of pairs, allowing further constraining of τ.	(§14.9.2)
$\forall\overline{\rho}.\tau$ N/A	Shorthand notation for $\forall\overline{\rho}.\varnothing \diamond \tau$.	(§14.9.2)
$\preceq$ N/A	$\sigma_1 \preceq \sigma_2$ means that σ_1 is at least as general as σ_2.	$\mathcal{F}$6.16 (§14.7.2)
$\downarrow$ N/A	An unconfirmed type variable binder which can be discarded when doing constraint solving.	$\mathcal{F}$6.3.2 (§14.6.2)
$\cap$ N/A	Used to represent an intersection type scheme.	(§14.9.2)
:> N/A	An opaque signature constraint.	$\mathcal{F}$6.16 (§14.7.1)
: N/A	A translucent signature constraint.	$\mathcal{F}$6.16 (§14.7.1)
Λ N/A	Denotes a pseudo type function.	§6.2.2 (§14.3.2)
$\twoheadrightarrow$ N/A	When x $\twoheadrightarrow$ y in a set S, syntactic forms of the form x are replaced by y.	(§14)
$\updownarrow$ N/A	Used before a name of a binder to indicate that it is an unconfirmed binder (can turn into a $\downarrow$ or a $\uparrow$).	(§14.1.2)

$\stackrel{y}{=}$ N/A	If y is a d or $\bar{d}$, $\downarrow vid \stackrel{y}{=} \langle \sigma, is \rangle$ abbreviates $\downarrow vid \stackrel{y}{=} \sigma; \downarrow vid \stackrel{y}{=} is$.	§4.2 (§14.1.2)
$\uplus$ N/A	An infix operator which unions two sets if they are disjoint as per the definition of dj.	§3.2.1 (§1)
$\sqsubseteq_{\bar{l}}$ N/A	The subslice relation, can be used to determine the minimality of a type error slice of a structure declaration.	(§11.9)
$\hookrightarrow$ N/A	The smallest possible relation that in $u, e \triangleright e_1 \hookrightarrow e_2$ it is the case that u, e, e_1 and e_2 are satisfied.	(§11.9)
θ IEqTypeVar	Internal equality type variables.	§7.1
Σ EqTypeStatus	Contains the values that an equality type status can be set to.	§7.1
$\oplus$ N/A	Allows for the generation of unifiers which do not contain dummy variables.	§3.2.1 (§11.6.5)
$\rightarrow$ N/A	Indicates a constraint solving step.	§4.4 (§11.6.1)
$\rightarrow^*$ N/A	Reflexive and transitive closure of $\rightarrow$.	§4.4 (§11.6.1)
Ψ SemanticJudgementDep	Same as Φ, but including dependencies.	(§11.4.3)
$\langle x, d \rangle$ N/A	Syntactic sugar for $\langle x, \{d\} \rangle$.	§4.2 (§11.3.4)
x^y N/A	Syntactic sugar for $\langle x, y \rangle$ provided y is a d or $\bar{d}$.	§4.2 (§11.3.4)
$x_1 \stackrel{y}{=} x_2$ N/A	Syntactic sugar for $\langle x_1 = x_2, y \rangle$.	§4.2.3 (§11.3.4)
$\downarrow x_1 \stackrel{y}{=} x_2 =$	Syntactic sugar for $\langle \downarrow x_1 = x_2, y \rangle$.	§4.2.3 (§11.3.4)

N/A		
Δ Context	Known as a constraint solving context, equivalent to $\langle u, e \rangle$.	($\mathcal{F}$11.4)
$\uparrow x_1 \overset{y}{=} x_2 =$ N/A	Syntactic sugar for $\langle \uparrow x_1{=}x_2, y \rangle$.	§4.2 (§11.3.4)
$[e]$ N/A	Syntactic sugar for $(ev_{\texttt{dum}} = e)$.	§4.2.3 (§11.3.4)
Φ SemanticJudgement	Holds the form $u, e \triangleright e_1 \hookrightarrow e_2$.	(§11.4.3)
$\triangleright$ N/A	The smallest relation satisfying Skalpel's constraint generation rules.	(§11.5.1)
; N/A	Used to compose environments and considered to be associative.	§4.2.3 (§11.3.1)
$\mathcal{F}$ N/A	Denotes that the number following this symbol refers to a latex figure number.	§A
§ N/A	Denotes that the number following this symbol refers to a latex section number.	§A
$\top$ N/A	An empty environment and satisfied constraint.	§4.2.3 (§11.3.1)
$\odot$ N/A	Arises from the need to slice away binders but take care of accessors that might become captured.	(§14.8.1)
$\lceil \rceil$ N/A	Placed around some terms in order to make label placement clear, and so as to not confuse the reader with ().	§4.1 (§11.2)
δ TyConVar	Type constructor variables.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
γ TyConName	Type constructor names.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)

α ITyVar	Internal type variables.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
η IdStatusVar	Used disambiguate value variables from datatype constructors.	(§14.1.2)
β RigidTyVar	Rigid type variables. Constant types that can renamed/quantified.	$\mathcal{F}$6.16 (§14.7.2)
ϕ FuncVar	Functor variables.	(§14.9.2)
ξ ITyVarSeqVar	Type variable sequence variables.	(§14.10.2)
ω ITySeqVar	Type sequence variables.	(§14.10.2)
ρ FRTyVar	Consists of Flexible and Rigid type variables.	$\mathcal{F}$6.16 (§14.7.2)
κ TyConSem	Allows internal type constructors to be universally quantified, or annotated with dependencies.	§6.2.2 (§14.3.2)
μ LabName	Internal type constructor, this can appear anywhere inside a constraint.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
τ LabTy	An internal type, this can appear anywhere inside a constraint.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
σ Scheme	A type scheme, subject to alpha conversion. This can be anywhere inside a constraint.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
a Atom	Atomic value.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
acc Accessor	An accessor, which may or may not be associated with a binder.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)

app App	Allows for application of type functions to internal types.	§6.2.2 (§14.3.2)
`arr` *N/A*	Binary arrow constructor.The exists to allow constraints between $\rightarrow$ and any unary type constructor.	§4.1 (§11.3.1)
atexp AtExp	Atomic expression, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
atpat AtPat	Atomic pattern, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
bind Bind	Represents the bindings of values in programs.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
c *N/A*	A raw status variable representing unary datatype constructors.	(§14.1.2)
c EqCs	Equality constraints between different two terms of the same type.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
cap LazyCapture	Used to further constrain internal types in constraints involving functors.	(§14.9.2)
cb ConBind	Datatype constructor binding, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
cd ConDesc	External syntax for value identifiers.	§6.4.1 (§14.7.1)
cg InitGen	Initial constraint generator. Returns form of either an (e), or something of the form $\langle v, e\rangle$ where e constrains v.	(§11.5.1)
class Class	A set defines the different classes that can be assigned to nodes.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
conexp ConExp	*conexp* must be either a datatype constructor (that is not equal to `ref`) or an exception constructor.	(§14.5)
ct	Constraint terms.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)

CsTerm		
d N/A	A raw status variable representing nullary datatype constructors.	(§14.1.2)
d Dependency	Dependancy. $\langle term, \bar{d} \rangle$ means 'term' depends on program nodes which have labels in $\bar{d}$.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
dcon DatCon	Datatype constructors, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
de DepEnv	Holds relations from Dependency to DepStatus values.	(§11.4.3)
dec Dec	A declaration in the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
dep Dependent	Can be replaced by terms which can be annotated with dependencies.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
dir Direction	A set containing directions of infix precedence.	§7.6.2
dn DatName	Typename of a datatype, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
dot Dot	Holds dot markers, e.g. `dotE`.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
`drop` N/A	Given dependency environment de, if $de(d) =$ `drop` then dependency d is unsatisfied.	(§11.4.3)
ds DepStatus	Holds different dependency statuses (namely `keep`, `drop`, and `keep-only-binders`).	(§11.4.3)
e Env	A form which can act as both a constraint and an environment.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
ek ErrKind	Holds all sorts of different kinds of errors that the user can have made (e.g. type constructor clashes).	$\mathcal{F}$4.18 ($\mathcal{F}$11.8)

eqtv EqTypeVar	Equality type variables part of the external syntax.	§7.1.1
er Error	Represents errors that have been detected.	$\mathcal{F}$4.18 ($\mathcal{F}$11.8)
es EnvScheme	An environment scheme, where environments are quantified.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
ev EnvVar	Environment variables, can be mapped to environments in the set of unifiers.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
exp Exp	An expression, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
fct Func	Allows for functor constraints to be constructed.	(§14.9.2)
fctsem FuncSem	Allows quantification of *fct* forms with any member of the set v.	(§14.9.2)
fundec FunDec	External syntax for functor declarations.	(§14.9.1)
funid FunId	Functor identifiers.	(§14.9.1)
ge GenEnv	A restriction of e.	(§11.5.3)
id Id	An identifier, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
ins Ins	A set holding instantiations. Note that Ins $\subseteq$ Sub.	§6.4.2 (§14.7.2)
ipe InPolyEnv	A restriction of e.	(§11.5.3)

is IdStatus	An identifier status, can either be a status variable or a raw id status *ris*.	(§14.1.2)
k AppKind	Holds different kinds of application, either `T` (for 'Type') or `S` (for 'Status').	(§14.1.4)
`keep` *N/A*	Given dependency environment *de*, if $de(d) = \texttt{drop}$ then dependency *d* is unsatisfied.	(§11.4.3)
`keep-only-binders` *N/A*	A status, if an *e* has a dependancy *d* with this status, *e*'s binders and *e* variables and turned into dummies.	(§11.4.3)
l Label	Labels, these are program locations which we use to track blame for errors.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
lacc LabAcc	Labelled accessors; a restriction of *e*.	(§11.5.3)
`lazy` *N/A*	Denotes a parameter of a functor, used to control which type schemes become lazy type schemes.	(§14.9.2)
lbind LabBind	A restriction of *e*.	(§11.5.2)
lc LabCs	A restriction of *e* containing only labelled equality constraints.	(§11.5.3)
ldcon LabDatCon	Labelled datatype constructor, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
lev LabEnvVar	Labelled environment variable, a restriction of *e*.	(§11.5.3)
ltc LabTyCon	Type constructor annotated with a label, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
ltv LabTyVar	Holds labelled type variables to differentiate between occurrences in types and in type variable sequences.	§6.3.1 (§14.6.1)
lvid	Contains labelled value identifiers.	(§14.1.1)

LabId		
node Node	A node of the labelled abstract syntax tree used when generating program slices.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
nonexp NonExp	Holds non-expansive expressions.	(§14.5)
$\overline{m}$ Monomorphic	Internal type variables annotated with dependencies describing why it is monomorphic.	§4.4
p N/A	A raw status variable representing unresolvable statuses.	(§14.1.2)
pat Pat	A pattern in the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
pe PolyEnv	A restriction of e.	(§11.5.3)
prod Prod	Holds different productions that can occur in a slice.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
prog Program	External syntax for a sequence of *topdec*s.	§6.4.1 (§14.7.1)
ren Ren	Used to instantiate type schemes. A strict subset of Unifier. Stands for 'renaming'.	$\mathcal{F}$4.4 ($\mathcal{F}$11.4)
ris RawIdStatus	Contains different types of statuses for variables to disambiguate value variables and datatype constructors.	(§14.1.2)
sbind SolvBind	A subset of Env containing binder expressions.	(§11.6.6)
se SolvEnv	A solved environment.	(§11.6.6)
serhs SolvEnvRHS	Env + composing environments and dependency annotation.	(§11.6.6)

sfct SolvFunc	Can be replaced by functor variables or solved environments of the form $e_1 \rightsquigarrow e_2$.	(§14.9.4)
sfctsem SolvFuncSem	Allows for universal quantification over *sfct* and annotation with dependencies.	(§14.9.4)
sig SigSem	Allows universal quantification of environments with δ variables.	$\mathcal{F}$6.16 (§14.7.2)
sigdec SigDec	External syntax for the declaration of a signature.	§6.4.1 (§14.7.1)
sigexp SigExp	External syntax for signature expressions.	§6.4.1 (§14.7.1)
sigid SigId	Contains signature identifiers.	§6.4.1 (§14.7.1)
sit ShallowITy	A restriction of the set τ.	(§11.5.2)
spec Spec	External syntax for the declarations that can occur inside a signature.	§6.4.1 (§14.7.1)
sq ITySeq	Sequences of internal types.	(§14.10.2)
stackAction StackAction	Contains forms which can be in the last element of tuples in the stack argument during constraint solving.	$\mathcal{F}$4.22
stackEv StackEv	Contains forms which can be in the first element of tuples in the stack argument during constraint solving.	$\mathcal{F}$4.22
stackMono StackMono	Contains forms which can be in the third element of tuples in the stack argument during constraint solving.	$\mathcal{F}$4.22
state State	Contains the various states that the constraint solver can be in.	$\mathcal{F}$4.18 ($\mathcal{F}$11.8)

stc ShallowTyCon	Can be a type constructor name or $\Lambda\alpha.\alpha'$.	§7.1.3 (§14.3.2)
strdec StrDec	A structure declaration, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
strexp StrExp	Structure expression (the body of a structure), part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
strid StrId	Structure identifiers, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
sub Sub	Similar to the set Unifier (A.2), but allows substitution of more syntactic forms e.g. rigid type variables.	$\mathcal{F}$4.4 ($\mathcal{F}$11.4)
subty SubTy	Used to represent subtyping constraints.	$\mathcal{F}$6.16 (§14.7.2)
sv SchemeVar	Set of scheme variables.	(§14.9.2)
svar SVar	Substitutable variable. Same as the set Var plus the rigid type variables.	$\mathcal{F}$6.16 (§14.7.2)
tc TyCon	Type constructors, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
tcs ITyConScheme	An internal type constructor scheme, where internal type constructors are quantified.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
term Term	A individual term, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
tfi TypFunIns	Declared but thought to be now unused.	(§14.3.2)
topdec TopDec	External syntax for top level declarations. A *strdec* or *strexp*.	§6.4.1 (§14.7.1)
tree	A labelled abstract syntax tree, used during slicing.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)

Tree		
ts ITyScheme	An internal type scheme, where an internal type is universally quantified.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
tv TyVar	External type variables, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
tvdeps ITyVarToDeps	A function from internal type variables to dependency sets.	(§11.6.7)
tvseq TyVarSeq	Holds sequences of labelled type variables.	§6.3.1 (§14.6.1)
ty Ty	An internal type.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
tyf TyFun	A set holding representations for type functions.	§6.2.2 (§14.3.2)
u *N/A*	A raw status variable representing unconfirmed context dependent statutes.	(§14.1.2)
u Unifier	The set of unifiers generated by Skalpel's constraint solver.	($\mathcal{F}$11.4)
v *N/A*	A raw status variable representing value variables.	(§14.1.2)
v Var	An extra metavariable definition.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
ve VarE	Contains the set of variables like Var, but α variables are annotated with a Σ.	§7.1.4
vid VId	Value identifier. Can be either a value variable or a datatype constructor. Part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
vidTc VidTyCon	A set containing value identifiers and type constructors.	§7.3.3

vsq ITyVarSeq	Sequences of internal type variables.	(§14.10.2)
vvar ValVar	Value variables, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)

A.2 Sets

Set Symbol	Brief Description	Definition
Accessor *acc*	An accessor, which may or may not be associated with a binder.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
App *app*	Allows for application of type functions to internal types.	§6.2.2 (§14.3.2)
AppKind *k*	Holds different kinds of application, either `T` (for 'Type') or `S` (for 'Status').	(§14.1.4)
AtExp *atexp*	Atomic expression, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
AtPat *atpat*	Atomic pattern, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
Atom *a*	Atomic value.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
Bind *bind*	Represents the bindings of values in programs.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
Class *class*	A set defines the different classes that can be assigned to nodes.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
ConBind *cb*	Datatype constructor binding, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
ConDesc	External syntax for value identifiers.	§6.4.1 (§14.7.1)

cd		
ConExp *conexp*	*conexp* must be either a datatype constructor (that is not equal to `ref`) or an exception constructor.	(§14.5)
Context Δ	Known as a constraint solving context, equivalent to $\langle u, e \rangle$.	($\mathcal{F}$11.4)
CsTerm *ct*	Constraint terms.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
DatCon *dcon*	Datatype constructors, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
DatName *dn*	Typename of a datatype, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
Dec *dec*	A declaration in the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
DepEnv *de*	Holds relations from Dependency to DepStatus values.	(§11.4.3)
DepStatus *ds*	Holds different dependency statuses (namely `keep`, `drop`, and `keep-only-binders`).	(§11.4.3)
Dependency *d*	Dependancy. $\langle term, \bar{d} \rangle$ means 'term' depends on program nodes which have labels in $\bar{d}$.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
Dependent *dep*	Can be replaced by terms which can be annotated with dependencies.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
Direction *dir*	A set containing directions of infix precedence.	§7.6.2
Dot *dot*	Holds dot markers, e.g. `dotE`.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
Dum *N/A*	Contains dummy variables, eacrh of which acts like a fresh variable.	(§11.3.3)

EnumState N/A	The set of states that the enumeration algorithm can be in.	§3.8.5 (§11.7.5)
Env e	A form which can act as both a constraint and an environment.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
EnvScheme es	An environment scheme, where environments are quantified.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
EnvVar ev	Environment variables, can be mapped to environments in the set of unifiers.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
EqCs c	Equality constraints between different two terms of the same type.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
EqTypeStatus Σ	Contains the values that an equality type status can be set to.	§7.1
EqTypeVar $eqtv$	Equality type variables part of the external syntax.	§7.1.1
ErrKind ek	Holds all sorts of different kinds of errors that the user can have made (e.g. type constructor clashes).	$\mathcal{F}$4.18 ($\mathcal{F}$11.8)
Error er	Represents errors that have been detected.	$\mathcal{F}$4.18 ($\mathcal{F}$11.8)
Exp exp	An expression, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
FRTyVar ρ	Consists of Flexible and Rigid type variables.	$\mathcal{F}$6.16 (§14.7.2)
FunDec $fundec$	External syntax for functor declarations.	(§14.9.1)
FunId $funid$	Functor identifiers.	(§14.9.1)

Func *fct*	Allows for functor constraints to be constructed.	(§14.9.2)
FuncSem *fctsem*	Allows quantification of *fct* forms with any member of the set v.	(§14.9.2)
FuncVar ϕ	Functor variables.	(§14.9.2)
GenEnv *ge*	A restriction of e.	(§11.5.3)
IEqTypeVar θ	Internal equality type variables.	§7.1
ITyConScheme *tcs*	An internal type constructor scheme, where internal type constructors are quantified.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
ITyScheme *ts*	An internal type scheme, where an internal type is universally quantified.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
ITySeq *sq*	Sequences of internal types.	(§14.10.2)
ITySeqVar ω	Type sequence variables.	(§14.10.2)
ITyVar α	Internal type variables.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
ITyVarSeq *vsq*	Sequences of internal type variables.	(§14.10.2)
ITyVarSeqVar ξ	Type variable sequence variables.	(§14.10.2)
ITyVarToDeps *tvdeps*	A function from internal type variables to dependency sets.	(§11.6.7)
Id	An identifier, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)

id		
IdStatus *is*	An identifier status, can either be a status variable or a raw id status *ris*.	(§14.1.2)
IdStatusVar η	Used disambiguate value variables from datatype constructors.	(§14.1.2)
InPolyEnv *ipe*	A restriction of e.	(§11.5.3)
InitGen *cg*	Initial constraint generator. Returns form of either an (e), or something of the form $\langle v, e\rangle$ where e constrains v.	(§11.5.1)
Ins *ins*	A set holding instantiations. Note that Ins $\subseteq$ Sub.	§6.4.2 (§14.7.2)
LabAcc *lacc*	Labelled accessors; a restriction of e.	(§11.5.3)
LabBind *lbind*	A restriction of e.	(§11.5.2)
LabCs *lc*	A restriction of e containing only labelled equality constraints.	(§11.5.3)
LabDatCon *ldcon*	Labelled datatype constructor, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
LabEnvVar *lev*	Labelled environment variable, a restriction of e.	(§11.5.3)
LabId *lvid*	Contains labelled value identifiers.	(§14.1.1)
LabName μ	Internal type constructor, this can appear anywhere inside a constraint.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
LabTy τ	An internal type, this can appear anywhere inside a constraint.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)

LabTyCon *ltc*	Type constructor annotated with a label, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
LabTyVar *ltv*	Holds labelled type variables to differentiate between occurrences in types and in type variable sequences.	§6.3.1 (§14.6.1)
Label *l*	Labels, these are program locations which we use to track blame for errors.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
LazyCapture *cap*	Used to further constrain internal types in constraints involving functors.	(§14.9.2)
Monomorphic $\overline{m}$	Internal type variables annotated with dependencies describing why it is monomorphic.	§4.4
Node *node*	A node of the labelled abstract syntax tree used when generating program slices.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
NonExp *nonexp*	Holds non-expansive expressions.	(§14.5)
Pat *pat*	A pattern in the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
PolyEnv *pe*	A restriction of e.	(§11.5.3)
Prod *prod*	Holds different productions that can occur in a slice.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
Program *prog*	External syntax for a sequence of *topdec*s.	§6.4.1 (§14.7.1)
RawIdStatus *ris*	Contains different types of statuses for variables to disambiguate value variables and datatype constructors.	(§14.1.2)
Ren *ren*	Used to instantiate type schemes. A strict subset of Unifier. Stands for 'renaming'.	$\mathcal{F}$4.4 ($\mathcal{F}$11.4)

RigidTyVar β	Rigid type variables. Constant types that can renamed/quantified.	$\mathcal{F}$6.16 (§14.7.2)
SVar *svar*	Substitutable variable. Same as the set Var plus the rigid type variables.	$\mathcal{F}$6.16 (§14.7.2)
Scheme σ	A type scheme, subject to alpha conversion. This can be anywhere inside a constraint.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
SchemeVar *sv*	Set of scheme variables.	(§14.9.2)
SemanticJudgement Φ	Holds the form $u, e \triangleright e_1 \hookrightarrow e_2$.	(§11.4.3)
SemanticJudgementDep Ψ	Same as Φ, but including dependencies.	(§11.4.3)
ShallowITy *sit*	A restriction of the set τ.	(§11.5.2)
ShallowTyCon *stc*	Can be a type constructor name or $\Lambda\alpha.\alpha'$.	§7.1.3 (§14.3.2)
SigDec *sigdec*	External syntax for the declaration of a signature.	§6.4.1 (§14.7.1)
SigExp *sigexp*	External syntax for signature expressions.	§6.4.1 (§14.7.1)
SigId *sigid*	Contains signature identifiers.	§6.4.1 (§14.7.1)
SigSem *sig*	Allows universal quantification of environments with δ variables.	$\mathcal{F}$6.16 (§14.7.2)
SolvBind *sbind*	A subset of Env containing binder expressions.	(§11.6.6)
SolvEnv	A solved environment.	(§11.6.6)

se		
SolvEnvRHS *serhs*	Env + composing environments and dependency annotation.	(§11.6.6)
SolvFunc *sfct*	Can be replaced by functor variables or solved environments of the form $e_1 \rightsquigarrow e_2$.	(§14.9.4)
SolvFuncSem *sfctsem*	Allows for universal quantification over *sfct* and annotation with dependencies.	(§14.9.4)
Spec *spec*	External syntax for the declarations that can occur inside a signature.	§6.4.1 (§14.7.1)
StackAction *stackAction*	Contains forms which can be in the last element of tuples in the stack argument during constraint solving.	$\mathcal{F}$4.22
StackEv *stackEv*	Contains forms which can be in the first element of tuples in the stack argument during constraint solving.	$\mathcal{F}$4.22
StackMono *stackMono*	Contains forms which can be in the third element of tuples in the stack argument during constraint solving.	$\mathcal{F}$4.22
State *state*	Contains the various states that the constraint solver can be in.	$\mathcal{F}$4.18 ($\mathcal{F}$11.8)
StrDec *strdec*	A structure declaration, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
StrExp *strexp*	Structure expression (the body of a structure), part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
StrId *strid*	Structure identifiers, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
Sub *sub*	Similar to the set Unifier (A.2), but allows substitution of more syntactic forms e.g. rigid type variables.	$\mathcal{F}$4.4 ($\mathcal{F}$11.4)
SubTy *subty*	Used to represent subtyping constraints.	$\mathcal{F}$6.16 (§14.7.2)

Term *term*	A individual term, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
TopDec *topdec*	External syntax for top level declarations. A *strdec* or *strexp*.	§6.4.1 (§14.7.1)
Tree *tree*	A labelled abstract syntax tree, used during slicing.	$\mathcal{F}$3.22 ($\mathcal{F}$11.15)
Ty *ty*	An internal type.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
TyCon *tc*	Type constructors, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
TyConName γ	Type constructor names.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
TyConSem κ	Allows internal type constructors to be universally quantified, or annotated with dependencies.	§6.2.2 (§14.3.2)
TyConVar δ	Type constructor variables.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
TyFun *tyf*	A set holding representations for type functions.	§6.2.2 (§14.3.2)
TyVar *tv*	External type variables, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
TyVarSeq *tvseq*	Holds sequences of labelled type variables.	§6.3.1 (§14.6.1)
TypFunIns *tfi*	Declared but thought to be now unused.	(§14.3.2)
Unifier *u*	The set of unifiers generated by Skalpel's constraint solver.	($\mathcal{F}$11.4)

VId *vid*	Value identifier. Can be either a value variable or a datatype constructor. Part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
ValVar *vvar*	Value variables, part of the external labelled syntax.	$\mathcal{F}$4.1 ($\mathcal{F}$11.2)
Var *v*	An extra metavariable definition.	$\mathcal{F}$4.2 ($\mathcal{F}$11.3)
VarE *ve*	Contains the set of variables like Var, but α variables are annotated with a Σ.	§7.1.4
VidTyCon *vidTc*	A set containing value identifiers and type constructors.	§7.3.3

A.3 Mathematical functions

Function	Short Description (if applicable)	Definition
abstract	Used to rebuild the environment associated with the parameter of a functor and to abstract the functor over the intersection types and type constructor names defined in its paramter.	(§14.9.4)
app	Given a Δ, an id and an element of AppKind named k, will look through Δ for a $\downarrow id = x$ expression, returning x which has kind type k.	(§14.1.4)
atoms	Given an argument x, a syntactic form belonging to set Var $\cup$ TyConName $\cup$ Dependency, and occurring in 'x'.	§4.2.4 (§11.3.2)
bindings	Extracts the bindings between binders and accessors in a given environment.	(§11.9)
build	A substitution operation similar that is recursively called in variable case, undefined on $\forall$ schemes and environments, and it collapses dependencies.	(§11.6.2)

collapse	Combines nested levels of dependencies. E.g. collapse($\langle\langle x, \bar{d}_1\rangle, \bar{d}_2\rangle$) returns $\langle x, \bar{d}_1 \cup \bar{d}_2\rangle$.	§4.2 (§11.3.1)
compatible	Checks two value identifier status values are compatible with each other.	(§14.1.4)
complete	Returns whether the environment in its argument is fully defined or not (i.e. not composed by an environment variable, filtered binder or dummy variable).	(§14.8.2)
declares	True if the class at the root of the tree is some kind of declaration.	§3.9.2 (§11.8.4)
deps	Given an argument x, defined to be atoms(x) $\cap$ Dependency.	§4.2.4 (§11.3.2)
diff	Allows for getting back a solved version of an environment, after all the constants have been dealt with.	(§11.6.3)
dj	Check sets are disjoint. dj($s_1, ..., s_n$) holds iff $\forall i, j \in \{1, ..., n\}$, if $i \neq j$ then $s_i \cap s_j = \emptyset$.	§3.2.1 (§1)
dja	Freshens generated variables and type constructor names.	(§11.3.3)
dom	Domain of a set. Given a set of pairs setOfPairs, dom(setOfPairs) $= \{x \mid (x, y) \in$ setOfPairs$\}$.	§3.2.1 (§1)
dot-d	A function which takes as arguments *term* values and outputs something of the form $[e_1; ...; e_n]$.	§3.9.1 ($\mathcal{F}$11.14)
dot-e	A function which takes as arguments *term* values and outputs something of the form $\langle\alpha, [e_1; ...; e_n]\rangle$.	§3.9.1 ($\mathcal{F}$11.14)
dot-i	A function which takes *term*s as arguments and outputs something of the form $\langle\alpha, \eta, [e_1, ..., e_n]\rangle$.	§3.9.1 (§14.1.6)
dot-n	A function which takes as arguments *term* values and outputs something of the form $\langle\delta, \alpha, \top, [e_1; ...; e_n]\rangle$.	§3.9.1 ($\mathcal{F}$11.14)
dot-p	A function which takes as arguments *pat* values and outputs something of the form $\langle\alpha, e_1; ...; e_n\rangle$.	§3.9.1 ($\mathcal{F}$11.14)

dot-s	A function which takes as arguments *term* values and outputs something of the form $\langle ev, [e_1; ...; e_n] \rangle$.	§3.9.1 ($\mathcal{F}$11.14)
dum	Deals with the conversion of environments into dummy environments.	§3.8.2 ($\mathcal{F}$11.11)
duplicate	Used to duplicate intersection types when instantiating a type scheme that captures intersection types (of the form $\forall \overline{\rho}.\overline{cap} \diamond \tau$ where $\overline{cap} \neq \varnothing$).	(§14.9.4)
duplicates	A form which holds an environment which has to be checked for duplicate value identifiers.	§7.3.1
e	The relation for the enumeration algorithm.	§4.2.7 (§11.7.5)
enum	Enumerates the errors in a given environment.	§3.8.5 ($\mathcal{F}$11.12)
err	Error state of the constraint solver.	§4.18 ($\mathcal{F}$11.8)
errors	A state of the enumeration algorithm representing found errors.	§3.8.5 (§11.7.5)
expans	Given an environment and a set of sets of dependencies, reports e as monomorphic in one or the set in the set of sets of dependencies is satisfied, and is then a dependent poly(e) environment.	(§14.5.2)
expansive	Extracts the dependencies for an expression to be expansive.	(§14.5.1)
expansiveCon	Helper function for expansive.	(§14.5.1)
filt	Used to check solvability of constraints in which some constraints are discarded.	§3.8.2 ($\mathcal{F}$11.11)
flat	Compresses sequences of terms of the form $\langle ..x..\langle ..y..\rangle ..\rangle$ become $\langle ..x..y..\rangle$. Note that this cannot be done in such a way that variables accidentally captured by binders.	§3.9.2 (§11.8.4)
freevars	Calculates the free internal type variables of an internal type of an internal type constructor.	§6.2.4 (§14.3.4)

genLazy	Given an application of a functor, computes type schemes from those generated for the functor's body, the head of which is $\forall\overline{\rho}.\overline{cap} \diamond \tau$ depending on the types in $\overline{cap}$.	(§11.9.4)
getBinders	Gets binders in a given environment.	(§14.8.2)
getDeps	Given arguments $(\alpha, \tau, \bar{d})$, returns dependency set occurring in τ on the path from its root node to any occurrence of α.	(§11.6.7)
getDot	Generates terms in *dot* from nodes.	$\mathcal{F}$3.24 (§11.8.3)
head	Extracts the head from intersection type schemes (e.g. $\mathsf{head}(\sigma_1 \cap \sigma_2) = \mathsf{head}(\sigma_1)$.	(§14.9.2)
ifNotDum	Ensures dummy status binder cannot bind anything other than a dummy status.	(§14.1.4)
`infix`	A form which represents that an operator has some new infix/nonfix status.	§7.6.2
`infixCheck`	A state of constraint solving which checks for errors involving infixity precedence/direction of value identifiers.	§7.6.2
instance	Allows generation of type schemes.	§4.2.7 (§11.4.3)
inters	Extracts the types (and their tails) from the intersection types from a given constraint solving context.	(§14.9.4)
isClass	Used to check the class at the root of the tree.	§3.9.2 (§11.8.4)
isErr	Checks whether an environment contains an error.	§4.4 (§11.6.5)
`isSucc`	Checks whether the stack parameter of the constraint solver has any unsolved elements, and takes appropriate action.	$\mathcal{F}$4.23
`isSucc'`	A helper function called by `isSucc` which takes action on the stack action parameter.	$\mathcal{F}$4.23
lBinds	Extracts the labels labelling binders in a given environment.	§3.8.1 (§11.7.1)

labs	Given an argument x, defined to be $\mathsf{atoms}(x) \cap \mathsf{Label}$.	§4.2 (§11.3.2)
labtyvars	Computes set of labelled explicit type variables in an explicit type.	§6.3.3 (§14.6.3)
labtyvarsdec	Given a sequence of type variables, a pattern and an expression will return the labelled type variables occurring in the pattern, or the expession, but not the intersection.	§6.3.3 (§14.6.3)
min	The relation for the minimisation algorithm.	§3.19 (§11.7.4)
monos	Returns a set containing the dependent monomorphic type variables occurring in an environment with respect to a unifier.	(§11.6.4)
monos′	Similar to monos but gathers labels more effectively using the getDeps function.	(§11.6.7)
nonDums	Given an argument x, gets the set of variables in x after first extracting all the dummy variables.	(§11.3.3)
`opaqueEq`	Changes part of an equality constraint from the temporary SIG_TYPE status to a NEQ_TYPE status.	§7.1.4
`or`	An environment of the form $\texttt{or}(e, \bar{d})$ is kept alive at constraint filtering if at least one of the dependencies in $\bar{d}$ is satisfied. Can be written $e^{\vee \bar{d}}$.	§6.3.2 (§14.6.2)
pattern	True if the class at the root of the tree is some kind of pattern.	§3.9.2 (§11.8.4)
`poly`	Promotes bindings in argument to be polymorphic.	§4.2 (§11.3.1)
putDeps	Given arguments (τ, tvdeps), constrains every $\alpha \in \mathsf{dom}(\mathit{tvdeps})$ of the occurrences of α in τ with dependence set $\mathit{tvdeps}(\alpha)$.	(§11.6.7)
ran	Range of a set. Given a set of pairs `setOfPairs`, $\texttt{ran}(\texttt{setOfPairs}) = \{y \mid (x, y) \in \texttt{setOfPairs}\}$.	§3.2.1 (§1)

rebuild	Builds up the type uses gathered (in intersection type schemes) while solving the constraints generated for functors.	(§14.9.4)
`sameId`	A state of the constraint solver which detects duplicate identifiers in signatures.	§7.3.3
scheme	Computes a $\forall$ quantified form from a variable set, a variable and a constant term.	§6.4.4 (§14.7.4)
`sha`	A form used with the environment stack $\overrightarrow{st}$, used in `vidSharingCheck`.	§7.5.4
shadowsAll	If given an argument e, then some of the binders in e may be shadowed, note that also in the case $(e_2; e_1)$ e_1 shadows the whole of the environment e_2.	§4.2.6 (§11.4.2)
`sharingSig`	Takes an environment variable which is used to create sharing equality constraints.	§7.3.3
`sharingType`	Holds two types which are to be constrained to be equal.	§7.5.2
sl	Constructs minimal type error slices from errors found by the enumeration algorithm. Has helper function sl_1 and sl_2.	§?? ($\mathcal{F}$11.17)
`slv`	A state the constraint solver can be in.	§4.6.3 ($\mathcal{F}$11.8)
solvable	Determines whether an environment is solvable.	§4.4 (§11.6.5)
`statusClash`	An error kind, which takes two status as arguments and represents a clash between these statuses.	(§14.1.4)
strip	Takes outer dependencies away from a term. E.g. `strip`$(\langle x, \bar{d} \rangle)$ gives back x.	$\mathcal{F}$4.2 (§11.3.1)
`succ`	Success state of the constraint solver.	§4.18 ($\mathcal{F}$11.8)
tail	Extracts the tail from intersection type schemes (e.g. $\mathsf{tail}(\sigma_1 \cap \sigma_2, u) = \mathsf{tail}(\sigma_2, u)$.	(§14.9.2)

test	Tests whether a label can be removed from a slice.	§3.8.4 (§11.7.4)
tidy	Cleans up slice representation of sequences of declarations in structure expressions.	§3.9.2 (§11.8.4)
toLazy	Transforms type schemes into lazy types schemes.	(§14.9.2)
toPoly	Given a Δ and a dependent monomorphic value identifier binder, generates a polymorphic binder by quantifying type variables not occurring in types of the monomorphic binders of Δ.	($\mathcal{F}$11.9)
toTree	Associates a *tree* with each *term*.	§3.23 ($\mathcal{F}$11.16)
toV	Takes an environment and turns the unconfirmed binders $\updownarrow vid = \alpha$ inside that environment into confirmed binders with a status variable $\downarrow vid = \langle \alpha, \mathtt{v} \rangle$.	(§14.1.3)
tyvars	Computes the set of explicit type variables occurring in an explicit type.	§6.4.2 (§14.7.2)
vars	Given an argument x, defined to be $\mathsf{atoms}(x) \cap \mathsf{Var}$.	§4.2 (§11.3.2)
`vidSharingCheck`	A state of constraint solving, used to detect if a type is to be shared that does not exist.	§7.5.4

A.4 Other Abbreviations

Abbreviation	Short Description	Definition
Core-TES	Core Type Error Slicer, refers to a subset of the design of Skalpel which handles datatypes, structures, and pattern matching matching among a select few other features.	(§10.3)
HW-TES	The Type Error Slicer outlined by Haack and Wells.	(§10.2)

SML	Standard MetaLanguage, the programming language that Skalpel takes as input which may or may not be typable, and outputs type error slices for.	§1 (§2.4.3)
SML/NJ	Stands for Standard ML of New Jersey. A compiler for Standard ML.	§1.3 (§2.4.3)
TES	Type Error Slicer, a tool which will take as input type errors and output slices as defined in [HW03].	(§2.4.3)
ULTRA	The research group at Heriot-Watt university which develops Skalpel.	§1.3 (§2.4.3)

Appendix B

Creating An Alternative Method For ML Code Documentation

We believe that due to the size and complexity of the Skalpel analysis engine, a powerful documentation tool is needed to adequately explain the structures, signatures, functions, types, and other implementation information not only to new users to the project but also to active developers who will over time lose familiarity with features which have already been implemented. Over time and with each new extension to the core it became increasingly important to have a tool which had the capacity to generate documentation for the analysis engine, which is written in Standard ML, and this appendix describes a new way of documenting Standard ML code.

This appendix is structured as follows. We will begin with a review of two existing ML code documentation tools, and explain why those tools were unsuitable for our purposes. Following this there will then be a description of why we did not choose a language-independent option, and discuss the alternative that was created.

B.1 SMLDoc [SMLb]

SMLDoc [SMLb] is a tool which was distributed as part of the SML# [SMLa] compiler written by Kiyoshi Yamatodani, but now is available on the Github hosting site[1]. It is based closely on a version the Javadoc documentation system and so the documentation produced closely resembles that.

Given a list of source files, SMLDoc will generate documentation for all structures,

[1] https://github.com/jhckragh/SMLDoc. Last accessed July 2013

signatures and functors in the source code and allow browsing the documentation by the name of the structure/signature/functor.

This documentation includes a full list of the declarations that are made (such as value declarations, datatype declarations, exceptions, etc.) and creates sections giving documentation extracted from special comments[2] placed in the source code in text format. Hyperlinks are created in the documentation whenever a type is defined, so when for example it is used in another structure, the definition can be reached easily.

Some special tags can be used in the special comments in the source code, figure B.1 gives a full list of the tags which have been implemented (extracted from the SMLDoc manual page).

Figure B.1 SMLDoc special tags

@author author of the entity
@copyright copyright
@contributor contributor
@exception indicates an exception the entity (a function) may raises
@params gives names to formal parameters of functions and value constructors
@param a description of a formal parameter
@return a description about return value of the function
@see related items (specified text is not analyzed in the current version)
@throws same as the @exception tag
@version a text which indicates the version of the entity

The documentation features available in SMLDoc seem stable, and the primary reason we did not choose to use this documentation tool was because of its limited feature set. In the approach discussed in section B.4 we have access to many additional features, some of which are particularly useful because it allows us to connect the implementation and the theory more easily. For example, some of the additional features we gain in B.4 include:

- Ability to insert LaTeXinto comments and have them rendered in the documentation. This is particularly useful as we can connect the formal presentation with our implementation.
- Generation of diagrams (class diagrams etc).
- Source code browsing.
- Generation of LaTeXdocumentation and man pages in addition to HTML output, so we can include that in source releases.

[2] The format used is an extra star after a comment starting block, e.g. (** special comment here *).

- A complete list of all type declarations in the project.
- Hyperlinks to the source code.
- Ability to include images.

Furthermore, the size of the user base for SMLDoc is unknown, and the last update to the online repository was two years ago at this time of writing, which is another reason why we chose to create our own solution. Where possible we wish to use a method which has a large user base and so is constantly under development. This concern is eliminated in Skalpel's approach, as discussed in section B.4.

B.2 ML-Doc [MLD]

ML-Doc [MLD] is described in its manual page as a system for producing reference manuals for Standard ML libraries. It is different from systems such as Javadoc (and by extension, SMLDoc), because it is designed such that the documentation is what the authors refer to as "the primary artifact", where source code files are created from the documentation. This software is most useful for creating documents like the SML Basis Library reference manual, but the ability to document a project such as Skalpel is not its primary goal.

B.3 Code independent documentation tools

The reason we did not choose to use a code independent documentation tool, where for example the syntax of comments is given and then all of the documentation exists within those comments without understanding the program structure, is that the amount of information that would need to be contained inside comments would be of a high volume, and it would be necessary to document every single variable in order for it to show in the documentation. For example, it would not be possible to tell whether a declaration is missing from the documentation that has been produced because a comment did not exist for it. It is for these reasons that we did not adopt such an approach.

B.4 Extending Doxygen [DOX] to create a new alternative for documenting ML code

Doxygen [DOX] is an active project with a large user base which allows users to document their code in a feature-rich, flexible way. Doxygen already has built-in support for numerous languages including C, C#, PHP, Java, and Python among others. It has the ability to generate HTML and LaTeX documentation, as well as support for RTF, PostScript, hyperlinked PDF, compressed HTML and Unix man page support. For these reasons it was decided that extending Doxygen to support SML would be the best approach. In this section, we call Doxygen extended with support for SML `DoxygenSML`. Note that although this work was designed for Skalpel, it can be used to document other SML programs as we demonstrate.

Doxygen operates by gathering all of the source code files it is to document in a given directory, searching sub-directories where necessary, and parsing each source code file. As Doxygen already understands the implementation language used, it creates a skeletal outline of documentation for variables, classes etc. used in the source code, and the comments placed before these are parsed and information inside is then used to fill out the documentation.

The implementation of Doxygen is constructed such that extending it to support a new language is done in a modular fashion. Adding the ability to support SML included writing a lexer and parser for the SML language which is then processed using the `flex` and `YACC` tools.

The entire analysis engine has been documented with this new tool by taking pre-existing documentation and changing its format to suit Doxygen, or by the creation of new documentation.

The special comment syntax that has been used is a left parenthesis followed by two stars. There are too many special tags to list here[3], but some interesting ones are given below in order to convey the idea of how tags might be used and what they might be used for.

- \f$. Marks the start and end of a LaTeX expression. These will be rendered and placed in the documentation.

- \image. Inserts an image into the documentation.

[3] A full list can be found at http://www.stack.nl/~dimitri/doxygen/manual/commands.html. Last accessed July 2013.

- \exception. Starts an exception description.

- \defgroup. Allows definition of a group name, which entities can be added to in order to build a new hierarchical structure.
- \brief. Starts a paragraph that is a brief description, used in for example lists at the start of the documentation page.

Doxygen can be executed on the configuration files by running `/path/to/doxygen config-file-name` in a terminal.

B.4.1 Brief discussion of patch

A crucial element of patching a system in development such as Doxygen is that there should be as little change to the existing code as possible. This is the case with the Standard ML patch that has been written. This can be seen from the changelog of the existing files below, which we include to demonstrate that the work here is not tightly bound to the currently existing implementation and so is poorly engineered. Note that the file names beginning with the prefix "sml" are entirely new contributions which result from this thesis, and so the large additions to them shown at the bottom does not affect the pre-existing code base.

```
Doxyfile                | 2 +
src/Makefile.in         | 5 +-
src/config.l            | 2 +
src/docsets.cpp         | 1 +
src/doxygen.cpp         | 3 +
src/libdoxygen.pro.in   | 4 +
src/libdoxygen.t.in     | 6 +
src/types.h             | 3 +-
src/util.cpp            | 6 +-
src/smlcode.h           | 45 ++
src/smlcode.l           | 1462 ++++++++++++++++++++++++++++++++++++++++++++++++++++
src/smlscanner.h        | 67 +++
src/smlscanner.l        | 1513 ++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Knowledge of C++ was required to create this patch, and Flex and Bison are used in order to parse the Standard ML language and determine what documentation

should be generated. That said, some parts of this patch needed to be written in a specific way to handle ML features, such as supporting the `and` keyword in Standard ML, which can be used to bind e.g. functions and variables, but determining precisely what is bound by the `and` keyword requires some care when handling some expressions.

This can be demonstrated with `let` bindings. The way that `let` bindings are handled is by the use of a stack. For example, consider the following Standard ML code fragment:

```
fun myFunction1 arg1 arg2 = arg1
and myFunction2 arg1 arg2 = let
                                val myVal1 = 1
                                and myVal2 = 2
                            in
                                myVal2 - myVal1
                            end
and myValFunction3 () = ()
```

In this fragment it should be noted that inside the declarations of the `let` expression, the `and` keyword is binding value declarations, whereas directly after it is binding function declarations. Keeping a stack of keywords which are being bound handles this problem. The same is true of local and abstract type declarations.

Figure B.2 shows an example of a basic SML program which has been annotated with Doxygen comments. Part of the Doxygen output for this example program is shown in figure B.3. Function bodies are not shown in this screenshot but are available via a hyperlink in the implementation.

B.4.2 Examples of generated output

An example of the HTML output that is generated by Doxygen is shown in figure B.2. Man page output is shown in figure B.4, and an example of the LaTeX output can be seen in appendix D. The full LaTeX documentation is not appended to this thesis as at this time of writing it approaches 1000 pages in length.

It should be noted that only in the HTML version of the documentation is source browsing available as bundling the entire analysis engine source code in Man or PDF format would be completely unmanageable. HTML browsing of the implementation is possible by the hyperlinks generated across the documentation to the points in the source files at which they were defined, which can be useful to see additional implementation detail about a definition.

Figure B.2 An example of an ML program documented with Doxygen

```
1  (** A structure to handle feature x.
2   * Perhaps structure Y should be integrated into this? *)
3  structure MyStructure =
4  struct
5     (** A datatype to handle representation of names and ages. *)
6     datatype mydt = INT of int | STRING of string
7     (** The identity function.
8      * \deprecated Not used since feature B came out. *)
9     fun id x = x
10    (** Multiplies the first argument with the integer of
11     * mydt in the second argument.
12     * \arg a An integer.
13     * \arg b Something of type #mydt, which is used in structure
14     *        Z, represented as \f$\alpha_1\f$ in the theory. *)
15    fun mulEx' a (INT(5)) = a * 5
16      | mulEx' _ _ = raise Fail("mulEx' called where second
17                      argument is not of the form INT(_).")
18 end
```

B.4.3 Doxygen configuration file and layout for Standard ML

There are two files which sit in the Standard ML analysis engine directory which configure options to Doxygen. The first is `doxygen-config-file` and the second is `DoxygenLayout.xml`. Both of these files are discussed below.

Doxygen Configuration file

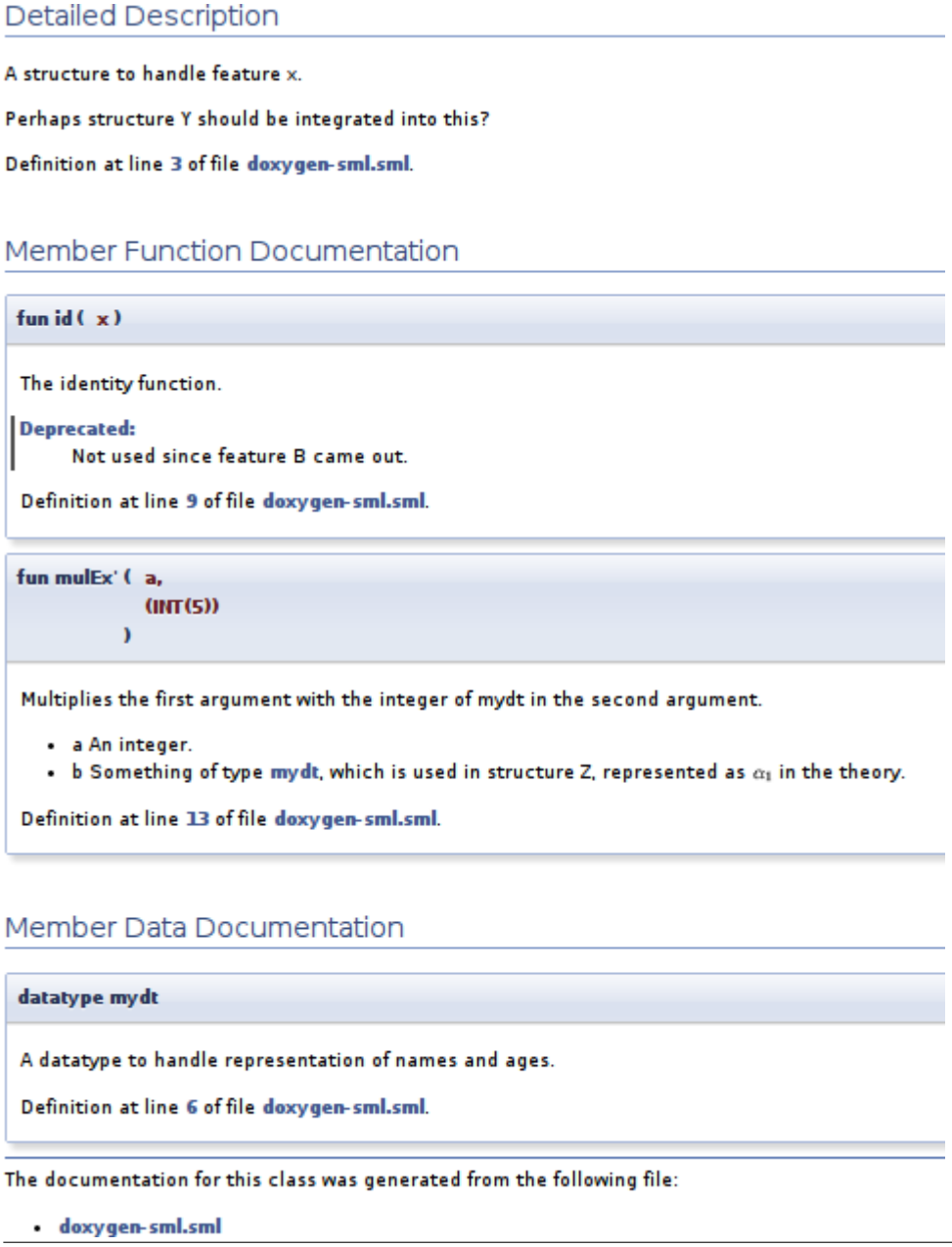
This file configures Doxygen options such as the project name, its brief description, output directory, what formats of documentation to generate (e.g. HTML/-Man) etc.

Other noteworthy features of this configuration file include a generated bug list (triggered with the use of the \bug command in the parsed code), a TODO list (\todo command), a deprecated list), warnings for undocumented parts of the analysis engine, addition of a source code browser with hyperlink navigation, and the linking to a LaTeXstyle file for generating images for the LaTeXembedded into the comments of the code.

Doxygen Layout File

Doxygen allows special configuration in this file to allow the documentation generated to have the correct terminology for a specific language. For example instead

Figure B.3 Part of Doxygen output for figure B.2.



Detailed Description

A structure to handle feature x.

Perhaps structure Y should be integrated into this?

Definition at line 3 of file doxygen-sml.sml.

Member Function Documentation

fun id (x)

The identity function.

Deprecated:
Not used since feature B came out.

Definition at line 9 of file doxygen-sml.sml.

fun mulEx' (a,
(INT(5))
)

Multiplies the first argument with the integer of mydt in the second argument.

- a An integer.
- b Something of type mydt, which is used in structure Z, represented as α_1 in the theory.

Definition at line 13 of file doxygen-sml.sml.

Member Data Documentation

datatype mydt

A datatype to handle representation of names and ages.

Definition at line 6 of file doxygen-sml.sml.

The documentation for this class was generated from the following file:

- doxygen-sml.sml

of having classes and header files as exists in C++, we have structures and signatures as somewhat approximate equivalents. The configuration that takes place in this file allows us to create the correct terminology in the documentation. A brief extract of this file is shown below:

```
<tab type="classlist" [...] title="Module list" intro="All structures, signatures and
                                    [...]
                        <tab type="classindex" [...] title="Module index"/>
<tab type="hierarchy" [...] title="Module hierarchy" intro="This list of structures [...]
```

In the first line of the above fragment, the “Class list” terminology that is generated by default becomes “Module list”, the second line changes “Class Index”

Figure B.4 Man Page Output from `DoxygenSML`

```
MyStructure(3)              Skalpel Analysis Engine Code Documentation              MyStructure(3)

NAME
       MyStructure -

       A structure to handle feature x.

SYNOPSIS
   Public Member Functions
       fun id
           The identity function.
       fun mulEx'
           Multiplies the first argument with the integer of mydt in the second argument.

   Static Protected Attributes
       datatype mydt
           A datatype to handle representation of names and ages.

Detailed Description
       A structure to handle feature x.

       Perhaps structure Y should be integrated into this?

       Definition at line 3 of file doxygen-sml.sml.

Member Function Documentation
   fun id (x)
       The identity function.

       Deprecated
           Not used since feature B came out.

       Definition at line 9 of file doxygen-sml.sml.

   fun mulEx' (a, (INT(5)))
       Multiplies the first argument with the integer of mydt in the second argument.

       · a An integer.
       · b Something of type mydt, which is used in structure Z, represented as $lpha_1$ in the theory.
       Definition at line 13 of file doxygen-sml.sml.
Member Data Documentation
   datatype mydt [static],  [protected]
       A datatype to handle representation of names and ages.
       Definition at line 6 of file doxygen-sml.sml.

Author
       Generated automatically by Doxygen for Skalpel Analysis Engine Code Documentation from the
       source code.

Version Git SHA-1: 516581ab8a940506eef3459abSun2May81832014                          MyStructure(3)
 Manual page doxygen-sml_MyStructure.3 line 1/55 (END) (press h for help or q to quit)
```

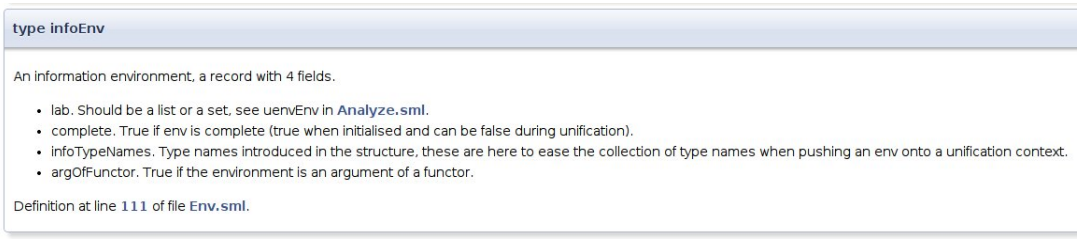
to “Module Index” and the third line changes “Heirarchy” to “Module Heirarchy”. The description for each of these is changed from its default definition to something which is sensible for the Standard ML language. The rest of the XML tags in this file are defined in a similar manner.

B.4.4 Type/Datatype declarations

A design decision was taken with type and datatype declarations that constructors for type declarations would not be shown in the brief description of the documentation at the top of the source files. The reason for this is that some of the datatypes declarations in the analysis engine are quite sizable in their declaration, and so documentation of these constructors should appear only in the detailed documentation. An example of this is shown in figure B.5.

This decision was made to allow for cleaner descriptions of datatype constructors without getting bogged down in technical details.

Figure B.5 Figure showing a type declaration in Doxygen



type infoEnv

An information environment, a record with 4 fields.

- lab. Should be a list or a set, see uenvEnv in Analyze.sml.
- complete. True if env is complete (true when initialised and can be false during unification).
- infoTypeNames. Type names introduced in the structure, these are here to ease the collection of type names when pushing an env onto a unification context.
- argOfFunctor. True if the environment is an argument of a functor.

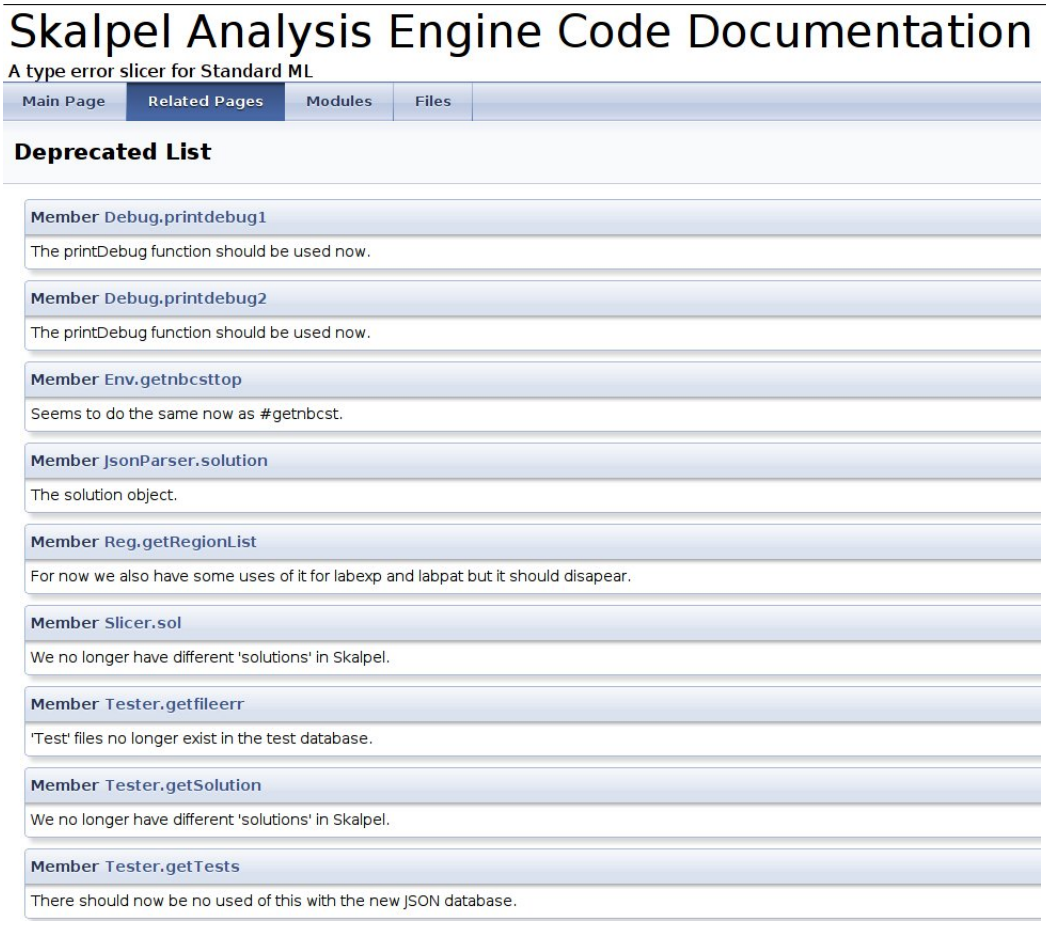
Definition at line 111 of file Env.sml.

B.4.5 Source Code Viewing and Deprecated list

Doxygen has been configured for source code viewing, and additional items as mentioned previously such as a list of deprecated functions.

The current deprecated list for Skalpel is shown in figure B.6. This is a basic example of how functions in Skalpel can be marked for future users not to use, which we currently have no mechanism for. A description is given for each deprecated feature. Currently deprecated functions in Skalpel are deprecated because of either a new test database format, a new debugging mechanism, or the removal of the concept of having different approaches of some algorithms (known as 'solutions'), where one solution is now deemed better and support for the others has been dropped.

Figure B.6 Doxygen Deprecated List



Skalpel Analysis Engine Code Documentation

A type error slicer for Standard ML

Main Page | Related Pages | Modules | Files

Deprecated List

Member Debug.printdebug1
The printDebug function should be used now.

Member Debug.printdebug2
The printDebug function should be used now.

Member Env.getnbcsttop
Seems to do the same now as #getnbcst.

Member JsonParser.solution
The solution object.

Member Reg.getRegionList
For now we also have some uses of it for labexp and labpat but it should disapear.

Member Slicer.sol
We no longer have different 'solutions' in Skalpel.

Member Tester.getfileerr
'Test' files no longer exist in the test database.

Member Tester.getSolution
We no longer have different 'solutions' in Skalpel.

Member Tester.getTests
There should now be no used of this with the new JSON database.

B.5 Uses of this tool outside of Skalpel

It is hoped that this extension of a documentation tool to support Standard ML will allow its use not only in the Skalpel implementation but also in other SML projects whose authors wish for generated code documentation such as this. Any developers that wish to do this will be able to access the patch to the Doxygen source code in the publicly available Skalpel repository, and modify it to suit their own needs if necessary.

Appendix C

Skalpel implementation

This appendix discusses aspects which relate to the implementation of various parts of Skalpel as a whole. Section C.1 gives an overview of which languages and tools are used in this project. Section C.2 discusses changes to the command-line interface, section C.3 describes changes made to the way we interact with compilers in order to build the Skalpel analysis engine, and section C.4 describes changes made to the way that we represent our test framework. Section C.5 looks at the implementation of Doxygen, section C.6 looks at the implementation of the debug framework, and section C.7 looks at the implementation that was associated with our test framework for the Skalpel project. Section C.8 discusses implementation changes made to the way we handle errors that can occur during parsing. Finally, section C.9 describes some of the implementation edits that occurred while looking at how Skalpel could be evaluated through the means of an experiment.

C.1 Languages and Tools used in the Skalpel project

We use a number of different languages and assorted technologies in the Skalpel project. We list some of these below:

- Standard ML. The majority of this project is implemented in the Standard ML language, as the analysis engine is itself implemented in Standard ML.
- Emacs Lisp. In order to to use Emacs as a front-end to Skalpel, there exists an implementation in Emacs Lisp which is loaded by Emacs.
- Git. All of the work on Skalpel is stored in a Git repository to allow sharing of the implementation easily with other developers, to allow version

comparison, have automatic backups, maintain separate versions (branching), allow the importing of changes suggested by other developers (merge request), tag changes with descriptions, and other vital functions.

- Vim script. This is used for the experimental Vim interface. This can currently display a slice but future work is required to turn this into a fully usable interface.

- C++. In order to extend Doxygen to support Standard ML, C++ was used. Implementation for this can be seen inside the patch which should be applied to the Doxygen source code.

- BASH Scripting. BASH is used to automate various tasks, such as building packages for distributions, running test frameworks, etc.

- Autoconf and Make. The GNU build tools are used inside this project. Wherever any implementation has to be built in some way there should be a Makefile there to aid the user in building the implementation. Autoconf is used to generate configure scripts, which in turn can be used to build any Makefile files.

- NSIS. The NSIS tool is used to build the Windows installer which can install Skalpel on a Windows computer.

- JSON. This is used to represent the results for our analysis engine test suite. It is expected that in the future more of this will be used, for example in our communication with the Emacs user interface.

- Perl. Initially, Perl was used to create a command-line interface for the user. This has now been replaced so support is embedded inside the analysis engine directly (and so is now implemented in Standard ML).

- XML. This is used to specify some settings for how the Doxygen documentation should be generated for the analysis engine.

- LaTeX. Much of the documentation surrounding Skalpel, including the user guide, is written in LaTeX.

- `rpmbuild`. This is used to build RPMs, which are packages for Linux systems such as Red Hat.

- Doxygen. A patch on the Doxygen tool is used to document our analysis engine. More information on this can be found in C.5.

- MLton. The MLton compiler builds the analysis engine. The SML/NJ and PolyML compilers are also supported for the same purpose, but these

compilers are less efficient than the MLton build and so these compilers are not typically used to build Skalpel. Support for these compilers is however present as they can sometimes be useful during the development of the analysis engine due to their top-level read-eval print loop they provide and their shorter build time.

C.2 Command-line interface

The command-line interface has been completely re-written since the start of this project. In order to show there was a justification for doing this we examine the original system below.

Initially, the Skalpel analysis engine binary was an entity which took 9 arguments to run. This was not done by the way of command line options, and all 9 arguments needed to be present for Skalpel to run and this was documented nowhere. There was no help available from the binary, and it was not intended that the user would ever execute the binary directly. Instead they could use two methods of interaction:

1. The Emacs user interface, which as it called the Skalpel analysis engine binary protected the user from ever having to interact with it.
2. The command-line interface, a collection of shell scripts which did take command-line options in a usable manner and allowed the user to see program slices in the terminal.

The primary problem was that the analysis engine did not support exporting program slices in a format that could be understood by the Bash [BAS] shell, and the command-line interface added this support by calling the analysis engine to output slices formatted in Perl, and then converted them to Bash scripts. There were numerous problems with this approach:

- Were the user to attempt to interact with the Skalpel binary directly, they would receive no help and would certainly not be able to guess the arguments required for the analysis engine to operate. This was deemed to be a likely outcome as the analysis engine binary was placed in standard $PATH directories on Linux systems;
- The command-line interface would need to check the file-system periodically for any Perl slices which had been output by the Skalpel binary;

- If the Skalpel binary were to fail silently for any reason, the command-line interface would not recognize this and hang;
- If the disk was full, strange errors would occur, and the command-line interface would wait eternally for the analysis engine to output errors;
- There was the potential issue of security problems. After one of the collection of Bash scripts that made up the command-line interface had converted the Perl output from the analysis engine to a Bash script, it would then execute all the Bash scripts on the system matching a certain file name format. Were this procedure to be hijacked, a script with any malicious command could be executed;
- The command-line interface contained bugs, and when it crashed half way through execution, left Perl and Bash output from the analysis engine and the command-line interface itself on the file system;
- Packaging was made complicated as we needed to supply three scripts and three manual pages to explain each one, and explain to the user why they should not execute the Skalpel binary directly.

Firstly, the analysis engine was extended to understand the command-line interface options in a similar format to the approach that was provided by the shell script, then an extension was made so that program slices in Bash format could be exported directly from the analysis engine. This solved all of the issues listed above. In addition, this meant that the analysis engine would not need to write to the file system at all as we could simply write the program slices to standard output (`stdout`), eliminating any security concerns which existed.

C.3 Removal of `cmtomlb` dependency for pure 64-bit support

It was previously the case that the analysis engine had a dependency that users building Skalpel with a compiler other than SML/NJ needed to have both the sources of the MLton compiler and the SML/NJ compiler installed. As the SML/NJ compiler does not officially support 64-bit systems, this complicated the process of building the Skalpel binary from the source code repository.

It is now possible to do this more easily, as PolyML, MLton, or SML/NJ users can simply use whichever compiler they have installed, without needing the `cmtomlb` utility from MLton and the SML/NJ compiler sources installed. We now generate

the build files for each of the compilers from the `.tes` files that Skalpel uses to operate on larger SML code bases. Converting the `.tes` files to `.cm` files (SML/NJ), `.mlb` files (MLton), or `.poly` files (Poly/ML) requires no additional tools other than those distributed with Skalpel. This will make the project more accessible to developers using certain compilers, as they will not need such additional tools from other compilers, and also solves such headaches for Skalpel developers during the development of the Skalpel tool itself.

C.4 Test database format

The test database was initially written in SML, and as a result SML/NJ had to be used in interactive mode in order to run tests on the Skalpel implementation. This complicated automated daily testing of Skalpel and excluded users of other compilers and so this approach was changed so that the entire test database was converted to JSON [JSO] format. The biggest advantage to this is now tests on the analysis engine can be checked easily from the command-line by users with any of the SML compilers.

In addition to this a control file was created (the `test-control` file in Standard ML analysis engine test directory of the Skalpel repository). This allows easy configuration of tests which are to be run, in order that we can run a subset of all of the tests we have easily. The format of this file is itself JSON, and contains a list of tests to be checked. By simply creating a new test control file and using the appropriate Skalpel command-line arguments, different configurations can be run.

It is believed that by adding support for JSON to the analysis engine there is scope to improve some of the other file communication that we do by also using the JSON format for it. An example of this would be our communication mechanism with the Emacs user interface. At this current time, Skalpel outputs *executable* Emacs lisp code to a file, and then Emacs will execute this file. It is suggested that a better solution would be to output the correct information in JSON format, which would remove any security concerns, and would be easy to parse given the very small size of the JSON specification. It is highly recommended at this point that any future front-ends to Skalpel parse the output JSON files in order to convey errors to users (that is, if a file-based communication method must be used).

C.5 Doxygen documentation

The analysis engine is documented with comments understood by extension to the Doxygen [DOX] system described in appendix B. All of this documentation exists in the structure declarations, and not in the signature declarations (the patch has been written so that declarations in signatures are linked to their structure counterparts so information is still easy to find).

C.6 Debug Framework

A system has been put in place inside the analysis engine to allow a greater degree of control while debugging. This was implemented as it became necessary to be able to toggle the debugging of specific features on and off as opposed to, for example, comment in and out potentially vast numbers of print statements. This could be particularly true when debugging the abstract syntax tree or the constraint generator, where there are many paths of execution.

Figure C.1 Path through the constraint generator for: `fun x (y:real) z = y = y`

```
% skalpel -d CONSTRAINT_PATH test.sml
[Skalpel: parsing...]
[Skalpel: constraint generation...]
(CONSTRAINT_PATH) Analyze.sml: A.DecFVal
(CONSTRAINT_PATH) Analyze.sml: ├─A.FValBind
(CONSTRAINT_PATH) Analyze.sml: ├─A.FValBindOne
(CONSTRAINT_PATH) Analyze.sml: │   └─A.FValBindCore
(CONSTRAINT_PATH) Analyze.sml: │       ├─A.LabFMatch
(CONSTRAINT_PATH) Analyze.sml: │       │   └─A.FMatchApp
(CONSTRAINT_PATH) Analyze.sml: │       │       ├─A.FMatchApp
(CONSTRAINT_PATH) Analyze.sml: │       │       │   ├─A.FMatchId
(CONSTRAINT_PATH) Analyze.sml: │       │       │   │   └─A.Ident (str="x", lab = 40769)
(CONSTRAINT_PATH) Analyze.sml: │       │       │   └─A.LabAtPat
(CONSTRAINT_PATH) Analyze.sml: │       │       │       └─A.AtPatParen
(CONSTRAINT_PATH) Analyze.sml: │       │       │           └─A.LabPat (lab = 40772)
(CONSTRAINT_PATH) Analyze.sml: │       │       │               └─A.PatTyped (lab = 40773)
(CONSTRAINT_PATH) Analyze.sml: │       │       │                   ├─A.LabPat (lab = 40774)
(CONSTRAINT_PATH) Analyze.sml: │       │       │                   │   └─A.Ident (str="y", lab = 40775)
(CONSTRAINT_PATH) Analyze.sml: │       │       │                   └─A.LabType
(CONSTRAINT_PATH) Analyze.sml: │       │       │                       └─A.TypeTyCon
(CONSTRAINT_PATH) Analyze.sml: │       │       │                           ├─A.TypeRowEm
(CONSTRAINT_PATH) Analyze.sml: │       │       │                           └─f_longtycon function
(CONSTRAINT_PATH) Analyze.sml: │       │       └─A.LabAtPat
(CONSTRAINT_PATH) Analyze.sml: │       │           └─A.Ident (str="z", lab = 40781)
(CONSTRAINT_PATH) Analyze.sml: │       └─A.LabExp (lab = 40782)
(CONSTRAINT_PATH) Analyze.sml: │           └─A.ExpOp (st = "=", lab=40783)
(CONSTRAINT_PATH) Analyze.sml: │               ├─A.LabExp (lab = 40784)
(CONSTRAINT_PATH) Analyze.sml: │               │   └─ A.ExpAtExp (no constraints generated, calls f_atexp)
(CONSTRAINT_PATH) Analyze.sml: │               │       └─A.AtExpId (calling f_longidexp with parameter id)
(CONSTRAINT_PATH) Analyze.sml: │               │           └─ f_longidexp function (longid=(y,40785))
(CONSTRAINT_PATH) Analyze.sml: │               └─A.LabExp (lab = 40786)
(CONSTRAINT_PATH) Analyze.sml: │                   └─ A.ExpAtExp (no constraints generated, calls f_atexp)
(CONSTRAINT_PATH) Analyze.sml: │                       └─A.AtExpId (calling f_longidexp with parameter id)
(CONSTRAINT_PATH) Analyze.sml: │                           └─ f_longidexp function (longid=(y,40787))
(CONSTRAINT_PATH) Analyze.sml: └─f_typevarvallist function (empty case)
[Skalpel: enumeration...]
[Skalpel: unification...]
Equality type required

Slice in context:

/home/jpirie/repos/skalpel/lib/basis.sml:
1-45:            ...
46:            structure Basis :> sig
47-107:          ...
108:           type real
109-142:             ...
143:           end = _structBasis
144:           ...
145:           open Basis
/tmp/test.sml:
1:             fun x (y:real) z = y = y
```

Developers can use the `-d` command-line parameter to control what features they wish to toggle. For example, by passing `-d CONSTRAINT_PATH` to the analysis engine, a full path through the constraint generator can be seen in figure C.1 for the program:

```
fun x (y:real) z = y = y
```

It is important to note that expensive debugging statements (such as when `-d STATE` is passed to the analysis engine) are wrapped in anonymous functions and then applied to `unit` if that debugging feature is enabled. Were that argument to be used without wrapping the debug statement in a function, it would always take an exceptionally long time (hours) to run Skalpel just once as it would convert the entire internal state to a string every time the minimisation algorithm was executed, irrespective of whether the debugging feature was enabled on the command-line or not. Such functions are applied to unit if the debugging feature associated with such a debug statement is enabled.

C.7 Test Framework

In order that we can have faith that Skalpel works correctly, it needs to be tested regularly. This is done daily by means of a cron job which executes a script that has been written which assesses various Skalpel attributes including:

- Cloning the repository;
- Building the Skalpel binary using the Poly/ML and MLton compilers;
- Running the test suite for the Skalpel analysis engine;
- Scanning the website for dead links;

Many extensions could and should be made to this test framework in the future. Examples of future useful extensions include:

- Verifying that the Emacs interface works with Skalpel;
- Profiling the binary as discussed in section 4.7.1 and auto-generating profile information;
- Testing the implementation for an experiment to evaluate Skalpel;

- Testing that Doxygen can build documentation present inside the Skalpel analysis engine.

C.7.1 Description of analysis engine portion of test framework

As discussed in section C.4, the way that we store tests and check they are correct has been updated. We have already discussed in this appendix the benefits to using our new representation over our old one, so here we take a more in-depth look at how tests in Skalpel are represented.

In the root of the Skalpel repository, there is a `testing/` folder which has the following structure:

1. `analysis-engine-tests/`
2. `master-test-files/`
3. `scripts`
4. `skalpel-evaluation`

We do not discuss the contents of the `skalpel-evaluation` implementation here and this is left to section C.9.

Inside the `analysis-engine-tests/standard-ml` folder (we only currently support Standard ML, there are no other directories in `analysis-engine-tests/`), tests are broken down into what feature they are testing, and each directory represents a feature being tested. For example, the `equality-types` sub-directory contains a range of tests which test the implementation of the equality types feature. For every test, we also have a solution file for that test, which is in JSON format. Due to the somewhat verbose nature of the solution files, there is not one included here verbatim, but they can be found by browsing our online repository[1].

In order to discuss the composition of the solution JSON files, we must first define the notation used by JSON.

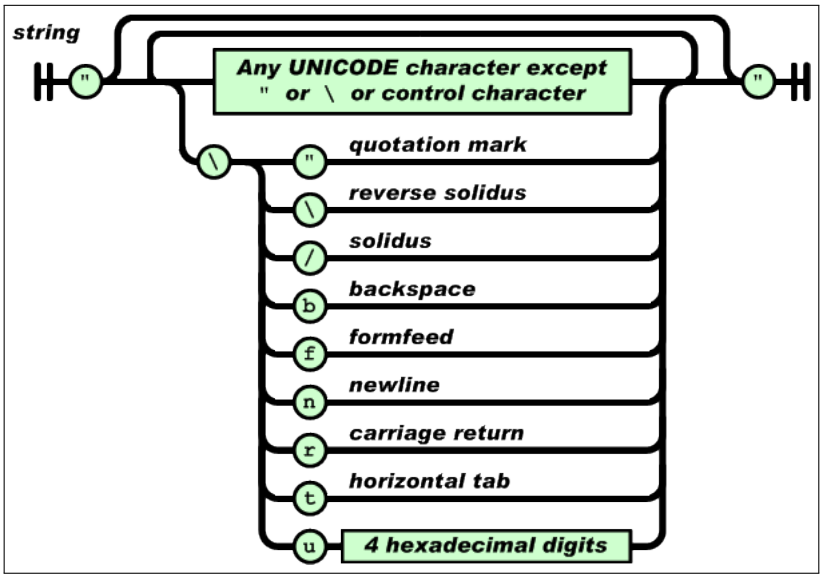
[1]The following URL provides a direct link to browsing the master branch of our source code repository: https://gitorious.org/skalpel/skalpel/source/master. Last accessed May 2014.

JSON notation

The notation for JSON is not complicated in nature, which allows for an elegant representation of test files. We provide here the same syntax diagrams that are provided by those who developed JSON [JSO].

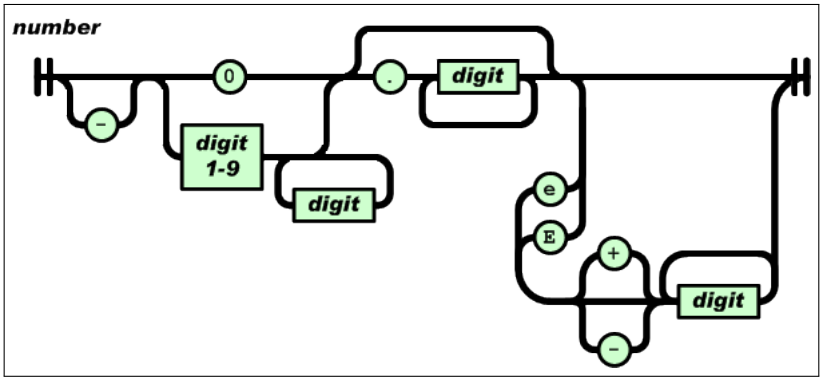
- **Strings**. The strings supported are very similar to C-strings. The syntax diagram for strings is shown in figure C.2.

Figure C.2 Syntax diagram for a JSON string [JSO]



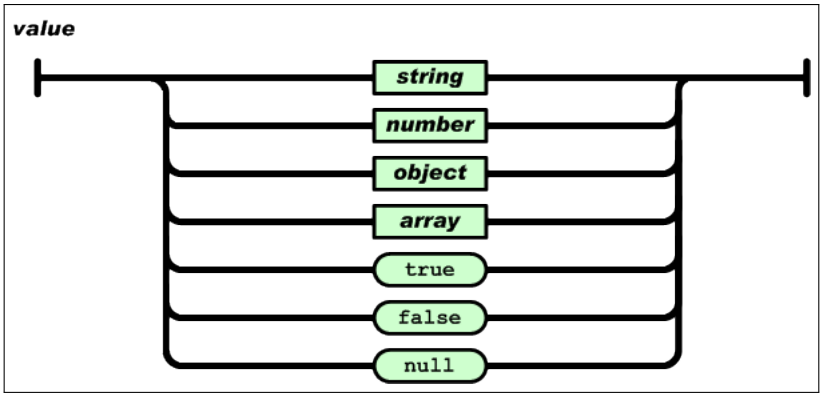
- **Numbers**. The syntax diagram for strings is shown in figure C.3. We can represent numbers in JSON using scientific notation.

Figure C.3 Syntax diagram for a JSON number [JSO]



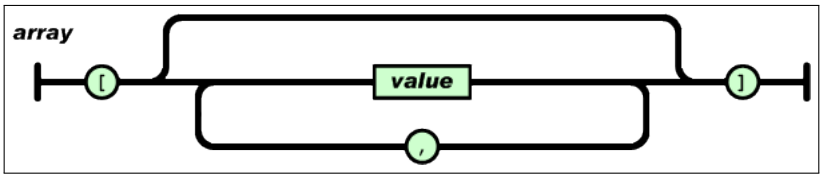
- **Value**. A value can either be a string, a number, an object (defined below), an array (defined below), `true`, `false` or `null`. We show the syntax diagrams for values in figure C.4

Figure C.4 Syntax diagram for a JSON value [JSO]



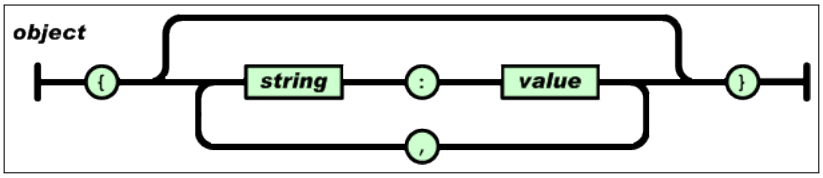
- **Array**. Arrays are an ordered lists of values. Arrays start with the character "[" and end with the character "]". We show the syntax diagrams for arrays in figure C.5.

Figure C.5 Syntax diagram for a JSON array [JSO]



- **Object**. An object is a set of string and value pairs (we will call the string portion of these the *name*), each separated by commas. Objects begin with the "{" character and end with the "}" character. We show the syntax diagrams for objects in figure C.6.

Figure C.6 Syntax diagram for a JSON object [JSO]



Structure of solution file

We have stated that for each test that is to be run, there is a solution file. At the top level, we have an object with the following strings which have some value associated with them: **errors**, **time**, **tyvar**, **ident**, **constraint**, **labels**, **minimisation**, **solution**, **basis**, **timelimit**, **labelling**, **final**, **name**.

Each of these is now discussed in turn.

The "errors" top-level object name

The errors name has an array of objects as its value. Each object in the array represents an error that was found, and has the following structure:

- **identifier**. The unique identifier associated with the error, represented as a number.
- **labels**. An object with the names *count* which is the number of labels that are part of the error (an integer) and *labelNumbers* which is an array of integers which are the labels that contributed to the error.
- **kind**. An object with the fields *errorKindName* which contains the error kind as a string, and *errorKindInfo*, which may or may not contain further names which describe end point labels (integers) depending on the kind of error.
- **time**. The time at which the this error was detected (an integer).
- **slice**. A string representing the program slice that was produced for this error. Note that the unicode characters ⟨ and ⟩ appear in this string.
- **assumptions**. Holds an array of integers which holds labels representing context dependent information.
- **regions**. Contains the regions of the source code which are part of the error. Represented as an array of objects, each of which represents regions for a given file of source code. Each object has the names *fileName*, which is a string containing the name of the file containing the error regions, and *regionList*, which is itself an array of objects. Each object in the *regionList* array has the fields *nodeType*, a string representing the kind of node, *fromLine* and *fromColumn*, representing the start point of the region in the source file, *toLine* and *toColumn*, which represents the end point of the region, *color* which represents the colour the region should be

highlighted in and *weight* which is deprecated and now always set to the integer 1.

The "time" top-level object name

The time name has an object as its value. This object is of the following structure:

- **analysis**. Has the value of a number indicating the time at which we finished constraint generation.
- **enumeration**. Has the value of a number indicating the time at which we finished enumeration.
- **minimisation**. Has the value of a number indicating the time at which we finished minimisation.
- **slicing**. Has the value of a number indicating the time at which we finished generating program slices.
- **html**. Has the value of a number indicating the time at which we finished generating HTML output if applicable.

The "tyvar" top-level object name

This object has two fields, *tyvar*, which contains the number of internal type variables that were generated for the source file containing the error, and *assoc*, which is an array of objects each with two fields *id* and *str*, which contain an integer representing an identifier and a string representing an explicit type variable, structure or type name respectively.

The "ident" top-level object name

A deprecated field, containing the same information as *tyvar* but does not contain the initial field holding the number of internal type variables that were generated.

The "constraint" top-level object name

A record with three names *total*, *top*, and *syntactic*, which are all integers representing how many total, type, and syntactic constraints were generated during execution respectively.

The "labels" top-level object name

An integer representing how many labels were generated during Skalpel's execution.

The "minimisation" top-level object name

A boolean representing whether the error is minimised or not.

The "solution" top-level object name

Deprecated. Contains an integer which is a number that was used during the project of [Rah10] which was used to differentiate between different development approaches used by Skalpel. Now a constant set to the integer 9 when solutions are generated.

The "basis" top-level object name

An integer representing the configuration of the basis that was used when Skalpel was executed.

The "timelimit" top-level object name

An integer representing the time limit that Skalpel was given to find errors.

The "labelling" top-level object name

A string representing internal dependency information that is generated during Skalpel's execution.

The "final" top-level object name

Represents whether this error is final as a boolean. Can be set to true if we are reporting initial errors to the user, which at this time is always the case for end users.

The "name" top-level object name

A string representing the name that can be given to the test file. This is often set to "dummy" as a placeholder value if we do not want to give a test a specific name.

With this new JSON representation in place, we reduced our dependence on specific compilers, which allowed us to be more supportive of users using particular compilers, reduced our dependency on a 32-bit architecture (due to the nature of the SML/NJ source implementation), and made the project less dependent as a whole on Standard ML, which is what must be done before the Skalpel analysis engine can be extended to target any other languages. It allowed also easier specification of tests that we choose to run by editing a master file which controls which tests to run, which is helpful while developing the Skalpel tool itself.

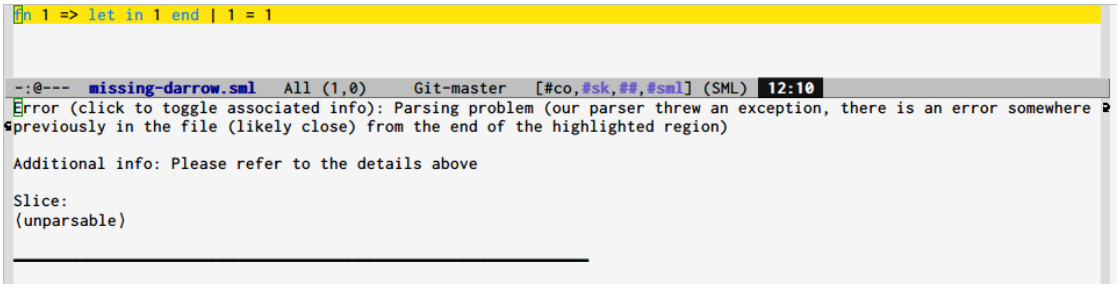
C.8 Parser improvements

Skalpel has been hindered in the past by a poor quality parser. Given that it is our belief that Skalpel produces better quality type errors than are given by the available compilers, it was unfortunate that the case was previously that Skalpel had significantly worse syntax error reporting. For example, consider the below code fragment.

```
fn 1 => let in 1 end | 1 = 1
```

It was previously the case that Skalpel produce the error shown in figure C.7. It can be seen from this figure that Skalpel would highlight the entire program, which is an extremely poor error report.

Figure C.7 Skalpel's previous poor parsing error



It is clear that this syntax error is of a lesser quality than the one reported by the Poly/ML compiler as follows [2]:

```
> use "missing-darrow.sml";
Error- in 'missing-darrow.sml', line 1.
=> expected but = was found
Exception- Fail "Static Errors" raised
```

While the Skalpel project is focused on high quality *type* errors, and not high quality *syntax* errors, this was nevertheless a problem which had existed for years and became imperative to fix.

The new error that Skalpel reports is shown in figure C.8. Now Skalpel reports an accurate error, and the user will be aided significantly more in fixing the problem.

Figure C.8 Skalpel's improved parsing error

Updates have been made to the parser so that these problems are now reported correctly. The solution to this issue has fixed a number of parsing problems that existed in Skalpel - this was done by editing the parsing mechanism in order to allow it to attempt to replace what it suspects are erroneous program tokens with other program tokens in order to pinpoint the source of the error.

In addition, changes were made to the way we parse our "magic comments", which are comments formed with a special syntax (a left parenthesis followed by two stars) that can be used to specify additional options to Skalpel (such as other files to be checked, though this can also be achieved by other means) inside a source code file. This means slices for some information contained inside a magic comment which is erroneous can be shown. Further information on magic comments can be found in our user guide.

[2] This example shows the output from the Poly/ML compiler, but other compilers e.g. MLton will give the same result.

C.9 Evaluation of Effectiveness of Skalpel

Some implementation was necessary to design an experiment to evaluate Skalpel in order to check the results that users submit are correct, and report an appropriate error otherwise.

In order to keep the implementation cost to an absolute minimum (as time spent designing an elaborate framework was deemed to be the incorrect way to approach this problem), functions were written somewhere inside the SML test files that the user had to write[3] which tested that the user had implemented the correct semantics, and a bash script was written to compile the Standard ML code file to a binary and execute it (reporting errors with Skalpel or MLton as appropriate). The script is written in under 110 lines of code, and it is hoped that at such a small size, maintenance of this implementation for any future experiments will be trivial.

In this appendix we have given an overview of the work that has been done on the Skalpel implementation. All of this work can be seen by inspecting the Skalpel source code repository[4], and by looking at the previous versions of the Skalpel implementation available in Git. Furthermore, given the extensive work that has been done in this area, including developing an entirely new method of documenting ML code, we believe future researchers using this implementation have been greatly aided in navigating and understanding the implementation across the project as a whole as easily as can be made possible.

[3] These functions did not necessarily appear at the end of the file, as this could have introduced a bias in the experiment due to the nature of the W algorithm.

[4] Our master branch of our source code repository can be found at the follows URL: https://gitorious.org/skalpel/skalpel/source/master.

Appendix D

Doxygen documentation extract in LaTeXformat

Below we show an example of the documentation that has been generated by Doxygen for a small component of the Skalpel implementation in LaTeX format. We do not give the full documentation of the Skalpel implementation in PDF format here, as at this current time of writing it approaches 800 pages in length. A full version of this documentation (in other formats including HTML and man pages) can be built from the Skalpel source code repository using the instructions provided in the `documentation/skalpel-developer-info` folder which exists at the root of the Skalpel source repository.

5.93.1 Detailed Description

Definition at line 63 of file RunSlicer.sml.

Public Member Functions

- fun setWebDemo

  Takes a boolean value b, if true then we are generating a binary for the web demo.
- fun finishedLispM

  Enhance these messages with the messages from EH.DeadBranch (failure of the slicer) and add a message saying if type errors have been discovered (success of the slicer)
- fun finishedLispMessage1

  A message for lisp indicatng the slicer terminated successfully (but applogises for it...?).
- fun finishedLispMessage2

  A message for lisp indicatng the slicer terminated with an error.
- fun finishedPerlM

  Builds the finished perl message.
- fun finishedPerlMessage1

  A message for perl indicatng the slicer terminated with success (but appologises for it..?).
- fun finishedPerlMessage2

  A message for perl indicatng the slicer terminated with an error.
- fun getFileBasAndNum

  Takes the basis flag and a file to be used as the basis.
- fun getDebugProblem

  Called by preslicer' if Tester.slicergen returns NONE.
- fun f

  Takes unit as an argument, and returns a string indicating a problem occurred.
- fun getBoolFile

  Returns a two tuple of a boolean (whether suff is a suffix of file), and the file name without the suffix (the empty string if suff is not a suffix.
- fun genOutputFile

  If suff is a suffix of ffile, then an output file will be created from the file name, concatenated with '-', the counter, and the suffix, which contains the result of fdebug applied to str.
- fun genFinished

  Generates the -finished file after the -counter files (see genOutputFile) have been generated.
- fun printIdError

  Prints the integer value of iderror.
- fun printErrors

  Prints the items in the errors list.
- fun printFound

  Prints out the error it found, and how long it took to find the error.
- fun export

  Calls the necessary functions to write the .html, .xml, .sml etc to the system.
- fun commslicerp'

  The primary function in this file which calls the functions necessary to run the slicer, including the main Testing.slicing function.
- fun run

  Calls the slicing function in Tester.sml.
- fun slicerCheckDevMode

  calls commslicerp', if we are not developing (not dev) then the error is handled, otherwise we leave the error so we can debug.

- fun commslicerp

  Called by the emacs interface; sets no tab and uses the solution from sol in utils/Solution.sml.
- fun slicerFull

  The full version of the slicer function with all arguments.
- fun printLegend

  Prints the legend that is used when we output slices into the terminal.
- fun printReset

  A function which prints the sting argument and appnds the string #Debug.textReset.
- fun smlTesStrArgs

  A function which has been created so that the slicer can be used in various ways without the need to program a new function every time.
- fun printHelp

  Prints the help text for the user (gives command-line options).
- fun checkFileSuffix

  Checks the file suffixes of the files the user has specified (.html for HTML output, etc.).
- fun parse

  Parses the arguments specified on the command-line.
- fun slicerGen

  Used by the MLton and Poly/ML entry points.
- fun smlnjEntryPoint

  Entry point for the SML/NJ compiler.
- fun mltonEntryPoint

  Entry point for the MLton compiler.
- fun polymlEntryPoint

  Entry point for the Poly/ML compiler.

Value declarations

- val webdemo

  Set to true if we are building a binary for the webdemo.
- val error

  Set to be the same as the error as defined in Tester (#Tester.error).
- val myfilehtml

  Set to be the same as the myfilehtml as defined in Tester (#Tester.myfilehtml).
- val terminalSlices

  Setting for showing slices in the terminal; we set the default setting to NO_DISPLAY.
- val SKALPEL_VERSION

  A value which should not be manually edited, the git hash of the repository is automatically inserted here during compilation.
- val file

  File to be used as output.
- val file'

  File to be used as output with a .tmp extension.
- val stout

  Output stream.
- val fin

  Filename to generate the finished file, indicating Skalpel has finished.
- val fin'

  Filename to generate the finished file, indicating Skalpel has finished, with a .tmp extension.
- val dbghtml

Appendix E

Progression of external and constraint syntax from initial definition

This appendix shows the additions that are made to the external syntax, and to the constraint syntax, since its original definition in chapter 4. The portions highlighted in red are the extensions which are new to that figure and were not present in the previous associated external syntax / constraint syntax figure.

E.1 Core definition

Figure E.1 External labelled syntax

external syntax (what the programmer sees, plus labels)

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

extra metavariables

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

Figure E.2 Syntax of constraint terms

$$
\begin{array}{rcll}
ev & \in & \mathsf{EnvVar} & \text{(environment variables)} \\
\delta & \in & \mathsf{TyConVar} & \text{(type constructor variables)} \\
\gamma & \in & \mathsf{TyConName} & \text{(type constructor names)} \\
\alpha & \in & \mathsf{ITyVar} & \text{(internal type variables)} \\
d & \in & \mathsf{Dependency} & ::= \quad l \\
\mu & \in & \mathsf{ITyCon} & ::= \quad \delta \mid \gamma \mid \mathtt{arr} \mid \langle \mu, \overline{d} \rangle \\
\tau & \in & \mathsf{ITy} & ::= \quad \alpha \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle \tau, \overline{d} \rangle \\
ts & \in & \mathsf{ITyScheme} & ::= \quad \forall \overline{v}.\, \tau \\
tcs & \in & \mathsf{ITyConScheme} & ::= \quad \forall \overline{v}.\, \mu \\
es & \in & \mathsf{EnvScheme} & ::= \quad \forall \overline{v}.\, e \\
c & \in & \mathsf{EqCs} & ::= \quad \mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2 \\
bind & \in & \mathsf{Bind} & ::= \quad \downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \\
acc & \in & \mathsf{Accessor} & ::= \quad \uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha \\
e & \in & \mathsf{Env} & ::= \quad \top \mid ev \mid bind \mid acc \mid c \mid \mathtt{poly}(e) \mid \exists a.e \mid e_2;e_1 \mid \langle e, \overline{d} \rangle
\end{array}
$$

extra metavariables

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

E.2 Extension for local declarations

Figure E.3 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
vid	$\in$	VId	$::=$	$vvar \mid dcon$
ltc	$\in$	LabTyCon	$::=$	tc^l
$ldcon$	$\in$	LabDatCon	$::=$	$dcon^l$
ty	$\in$	Ty	$::=$	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	$::=$	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	$::=$	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	$::=$	$\texttt{val rec}\ pat \stackrel{l}{=} exp \mid \texttt{open}^l\ strid \mid \texttt{datatype}\ dn \stackrel{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
$atexp$	$\in$	AtExp	$::=$	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	$::=$	$atexp \mid \texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l$
$atpat$	$\in$	AtPat	$::=$	vid^l_{p}
pat	$\in$	Pat	$::=$	$atpat \mid \lceil ldcon\ atpat \rceil^l$
$strdec$	$\in$	StrDec	$::=$	$dec \mid \texttt{structure}\ strid \stackrel{l}{=} strexp$
$strexp$	$\in$	StrExp	$::=$	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$

extra metavariables

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

Figure E.4 Syntax of constraint terms

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

extra metavariables

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

E.3 Extension for type declarations

Figure E.5 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
vid	$\in$	VId	$::=$	$vvar \mid dcon$
ltc	$\in$	LabTyCon	$::=$	tc^l
$ldcon$	$\in$	LabDatCon	$::=$	$dcon^l$
ty	$\in$	Ty	$::=$	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	$::=$	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	$::=$	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	$::=$	$\texttt{val rec}\ pat \overset{l}{=} exp \mid \texttt{open}^l\ strid \mid \texttt{datatype}\ dn \overset{l}{=} cb \mid$ $\texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \mid \texttt{type}\ dn \overset{l}{=} ty$
$atexp$	$\in$	AtExp	$::=$	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	$::=$	$atexp \mid \texttt{fn}\ pat \overset{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l$
$atpat$	$\in$	AtPat	$::=$	vid^l_{p}
pat	$\in$	Pat	$::=$	$atpat \mid \lceil ldcon\ atpat \rceil^l$
$strdec$	$\in$	StrDec	$::=$	$dec \mid \texttt{structure}\ strid \overset{l}{=} strexp$
$strexp$	$\in$	StrExp	$::=$	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$

extra metavariables

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

Figure E.6 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
d	$\in$	Dependency	$::=$	l
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \langle \mu, \overline{d} \rangle$
τ	$\in$	ITy	$::=$	$\alpha \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle \tau, \overline{d} \rangle$
ts	$\in$	ITyScheme	$::=$	$\forall \overline{v}.\, \tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall \overline{v}.\, \mu$
es	$\in$	EnvScheme	$::=$	$\forall \overline{v}.\, e$
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall \overline{\alpha}.\, \mu \mid \langle \kappa, \overline{d} \rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\alpha.\,\tau \mid \langle tyf, \overline{d} \rangle$
app	$\in$	App	$::=$	$\tau\ tyf$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1$ $\mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \langle e, \overline{d} \rangle$

extra metavariables

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

E.4 Extension for type annotations

Figure E.7 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
vid	$\in$	VId	$::=$	$vvar \mid dcon$
ltc	$\in$	LabTyCon	$::=$	tc^l
$ldcon$	$\in$	LabDatCon	$::=$	$dcon^l$
ty	$\in$	Ty	$::=$	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	$::=$	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	$::=$	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	$::=$	$\texttt{val rec}\ tvseq\ pat \overset{l}{=} exp \mid \texttt{open}^l\ strid$
				$\mid \texttt{datatype}\ dn \overset{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
				$\mid \texttt{type}\ dn \overset{l}{=} ty$
$atexp$	$\in$	AtExp	$::=$	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	$::=$	$atexp \mid \texttt{fn}\ pat \overset{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp : {}^l ty$
ltv	$\in$	LabTyVar	$::=$	$ty^l{}_{\mathtt{l}} \mid \texttt{dot-d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	$::=$	$ltv \mid \epsilon^l{}_{\mathtt{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	$::=$	vid^l_{p}
pat	$\in$	Pat	$::=$	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat : {}^l ty$
$strdec$	$\in$	StrDec	$::=$	$dec \mid \texttt{structure}\ strid \overset{l}{=} strexp$
$strexp$	$\in$	StrExp	$::=$	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$

extra metavariables

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

Figure E.8 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
d	$\in$	Dependency	$::=$	l
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \langle \mu, \overline{d} \rangle$
τ	$\in$	ITy	$::=$	$\alpha \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle \tau, \overline{d} \rangle$
ts	$\in$	ITyScheme	$::=$	$\forall \overline{v}.\, \tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall \overline{v}.\, \mu$
es	$\in$	EnvScheme	$::=$	$\forall \overline{v}.\, e$
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall \overline{\alpha}.\, \mu \mid \langle \kappa, \overline{d} \rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\alpha.\,\tau \mid \langle tyf, \overline{d} \rangle$
app	$\in$	App	$::=$	$\tau\ tyf$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1$
				$\mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e, \overline{d}) \mid \langle e, \overline{d} \rangle$

extra metavariables

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

E.5 Extension for signatures

Figure E.9 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
$sigid$	$\in$	SigId		(signature identifiers)
vid	$\in$	VId	::=	$vvar \mid dcon$
ltc	$\in$	LabTyCon	::=	tc^l
$ldcon$	$\in$	LabDatCon	::=	$dcon^l$
ty	$\in$	Ty	::=	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	::=	$dcon^l_{\mathtt{c}} \mid dcon\ \mathtt{of}^{\,l}\ ty$
dn	$\in$	DatName	::=	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	::=	$\mathtt{val\ rec}\ tvseq\ pat \stackrel{l}{=} exp \mid \mathtt{open}^l\ strid$
				$\mid \mathtt{datatype}\ dn \stackrel{l}{=} cb \mid \mathtt{local}^l\ dec_1\ \mathtt{in}\ dec_2\ \mathtt{end}$
				$\mid \mathtt{type}\ dn \stackrel{l}{=} ty$
$atexp$	$\in$	AtExp	::=	$vid^l_{\mathtt{e}} \mid \mathtt{let}^l\ dec\ \mathtt{in}\ exp\ \mathtt{end}$
exp	$\in$	Exp	::=	$atexp \mid \mathtt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$
ltv	$\in$	LabTyVar	::=	$ty^l{}_{\mathtt{l}} \mid \mathtt{dot\text{-}d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	::=	$ltv \mid \epsilon^l{}_{\mathtt{v}} \mid (ltv, ..., ltv_n)^l \mid \mathtt{dot\text{-}d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	::=	$vid^l_{\mathtt{p}}$
pat	$\in$	Pat	::=	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$
$sigdec$	$\in$	SigDec	::=	$\mathtt{signature}\ sigid \stackrel{l}{=} exp \mid \mathtt{dot\text{-}d}(\overrightarrow{term})$
$sigexp$	$\in$	SigExp	::=	$sigid^l \mid \mathtt{sig}^l\ spec_1 \cdots spec_n\ \mathtt{end} \mid \mathtt{dot\text{-}s}(\overrightarrow{term})$
$spec$	$\in$	Spec	::=	$\mathtt{val}\ vid\ :^l\ ty \mid \mathtt{type}\ dn^l \mid$
				$\mathtt{datatype}\ dn \stackrel{l}{=} cd \mid \mathtt{structure}\ strid\ :^l\ sigexp$
				$\mid \mathtt{dot\text{-}d}(\overrightarrow{term})$
cd	$\in$	ConDesc	::=	$vid^l_{\mathtt{c}} \mid vid\ \mathtt{of}^{\,l}\ ty \mid \mathtt{dot\text{-}e}(\overrightarrow{term})$
$topdec$	$\in$	TopDec	::=	$strdec \mid sigdec$
$prog$	$\in$	Program	::=	$topdec_1; ...topdec_n$
$strdec$	$\in$	StrDec	::=	$dec \mid \mathtt{structure}\ strid \stackrel{l}{=} strexp$
$strexp$	$\in$	StrExp	::=	$strid^l \mid \mathtt{struct}^l\ strdec_1 \cdots strdec_n\ \mathtt{end}$
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$

extra metavariables

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

Figure E.10 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
d	$\in$	Dependency	$::=$	l
ts	$\in$	ITyScheme	$::=$	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	$::=$	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\mu, \overline{d}\rangle$
τ	$\in$	ITy	$::=$	$\alpha \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \beta \mid \langle\tau, \overline{d}\rangle$
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa, \overline{d}\rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\alpha.\,\tau \mid \langle tyf, \overline{d}\rangle$
app	$\in$	App	$::=$	$\tau\ \ tyf$
$svar$	$\in$	SVar	$::=$	$v \mid \beta$
ρ	$\in$	FRTyVar	$::=$	$\alpha \mid \beta$
sig	$\in$	SigSem	$::=$	$e \mid \forall\overline{\delta}.\,e \mid \langle sig, \overline{d}\rangle$
$subty$	$\in$	SubTy	$::=$	$\sigma \preceq_{vid}\ \sigma_2 \mid \kappa_1 \preceq_{tc}\ \kappa_2$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha \mid \downarrow sigid{=}ev$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1$ $\mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e, \overline{d}) \mid \langle e, \overline{d}\rangle$ $\mid e_1 : e_2 \mid \texttt{ins}(e) \mid subty$

extra metavariables

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

E.6 Extension for equality types

Figure E.11 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
$sigid$	$\in$	SigId		(signature identifiers)
$eqtv$	$\in$	EqTypeVar		(Equality type variables)
vid	$\in$	VId	::=	$vvar \mid dcon$
ltc	$\in$	LabTyCon	::=	tc^l
$ldcon$	$\in$	LabDatCon	::=	$dcon^l$
ty	$\in$	Ty	::=	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	::=	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	::=	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	::=	$\texttt{val rec}\ tvseq\ pat \overset{l}{=} exp \mid \texttt{open}^l\ strid$
				$\mid \texttt{datatype}\ dn \overset{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
				$\mid \texttt{type}\ dn \overset{l}{=} ty$
$atexp$	$\in$	AtExp	::=	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	::=	$atexp \mid \texttt{fn}\ pat \overset{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$
ltv	$\in$	LabTyVar	::=	$ty^l{}_{\mathtt{l}} \mid \texttt{dot-d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	::=	$ltv \mid \epsilon^l{}_{\mathtt{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	::=	vid^l_{p}
pat	$\in$	Pat	::=	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$
$sigdec$	$\in$	SigDec	::=	$\texttt{signature}\ sigid \overset{l}{=} exp \mid \texttt{dot-d}(\overrightarrow{term})$
$sigexp$	$\in$	SigExp	::=	$sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \mid \texttt{dot-s}(\overrightarrow{term})$
$spec$	$\in$	Spec	::=	$\texttt{val}\ vid :^l ty \mid \texttt{type}\ dn^l \mid$
				$\texttt{datatype}\ dn \overset{l}{=} cd \mid \texttt{structure}\ strid :^l sigexp$
				$\mid \texttt{eqtype}\ dn^l \mid \texttt{dot-d}(\overrightarrow{term})$
cd	$\in$	ConDesc	::=	$vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^{\,l}\ ty \mid \texttt{dot-e}(\overrightarrow{term})$
$topdec$	$\in$	TopDec	::=	$strdec \mid sigdec$
$prog$	$\in$	Program	::=	$topdec_1; ...topdec_n$
$strdec$	$\in$	StrDec	::=	$dec \mid \texttt{structure}\ strid \overset{l}{=} strexp$
$strexp$	$\in$	StrExp	::=	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$

extra metavariables

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

Figure E.12 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
θ	$\in$	IEqTypeVar		(equality type variables)
d	$\in$	Dependency	$::=$	l
ts	$\in$	ITyScheme	$::=$	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	$::=$	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\delta,\theta\rangle \mid \langle\mu,\overline{d}\rangle$
τ	$\in$	ITy	$::=$	$\langle\alpha,\theta\rangle \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle\beta,\theta\rangle \mid \langle\tau,\overline{d}\rangle$
Σ	$\in$	EqTypeStatus	$::=$	EQ_TYPE | NEQ_TYPE | SIG_TYPE
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa,\overline{d}\rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$
app	$\in$	App	$::=$	$\tau\ tyf$
$svar$	$\in$	SVar	$::=$	$v \mid \beta$
ρ	$\in$	FRTyVar	$::=$	$\alpha \mid \beta$
sig	$\in$	SigSem	$::=$	$e \mid \forall\overline{\delta}e \mid \langle sig,\overline{d}\rangle.$
$subty$	$\in$	SubTy	$::=$	$\sigma \preceq_{vid} \sigma_2 \mid \kappa_1 \preceq_{tc} \kappa_2$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle \mid \downarrow sigid{=}ev$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1$ $\mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e,\overline{d}) \mid \langle e,\overline{d}\rangle$ $\mid e_1 : e_2 \mid \texttt{ins}(e \mid subty)$

extra metavariables

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

E.7 Extension for abstract type declarations

Figure E.13 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
$sigid$	$\in$	SigId		(signature identifiers)
$eqtv$	$\in$	EqTypeVar		(Equality type variables)
vid	$\in$	VId	::=	$vvar \mid dcon$
ltc	$\in$	LabTyCon	::=	tc^l
$ldcon$	$\in$	LabDatCon	::=	$dcon^l$
ty	$\in$	Ty	::=	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	::=	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	::=	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	::=	$\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp \mid \texttt{open}^l\ strid$
				$\mid \texttt{datatype}\ dn \stackrel{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
				$\mid \texttt{type}\ dn \stackrel{l}{=} ty \mid \texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}$
$atexp$	$\in$	AtExp	::=	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	::=	$atexp \mid \texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$
ltv	$\in$	LabTyVar	::=	$ty^l{}_{\mathtt{l}} \mid \texttt{dot-d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	::=	$ltv \mid \epsilon^l{}_{\mathtt{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	::=	vid^l_{p}
pat	$\in$	Pat	::=	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$
$sigdec$	$\in$	SigDec	::=	$\texttt{signature}\ sigid \stackrel{l}{=} exp \mid \texttt{dot-d}(\overrightarrow{term})$
$sigexp$	$\in$	SigExp	::=	$sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \mid \texttt{dot-s}(\overrightarrow{term})$
$spec$	$\in$	Spec	::=	$\texttt{val}\ vid :^l ty \mid \texttt{type}\ dn^l \mid \texttt{datatype}\ dn \stackrel{l}{=} cd \mid$
				$\texttt{structure}\ strid :^l sigexp$
				$\mid \texttt{eqtype}\ dn^l \mid \texttt{dot-d}(\overrightarrow{term})$
cd	$\in$	ConDesc	::=	$vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^{\,l}\ ty \mid \texttt{dot-e}(\overrightarrow{term})$
$topdec$	$\in$	TopDec	::=	$strdec \mid sigdec$
$prog$	$\in$	Program	::=	$topdec_1; ...topdec_n$
$strdec$	$\in$	StrDec	::=	$dec \mid \texttt{structure}\ strid \stackrel{l}{=} strexp$
$strexp$	$\in$	StrExp	::=	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$

extra metavariables

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

Figure E.14 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
θ	$\in$	IEqTypeVar		(equality type variables)
d	$\in$	Dependency	::=	l
ts	$\in$	ITyScheme	::=	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	::=	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	::=	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	::=	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\delta,\theta\rangle \mid \langle\mu,\overline{d}\rangle$
τ	$\in$	ITy	::=	$\langle\alpha,\theta\rangle \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle\beta,\theta\rangle \mid \langle\tau,\overline{d}\rangle$
Σ	$\in$	EqTypeStatus	::=	EQ_TYPE | NEQ_TYPE | SIG_TYPE
tfi	$\in$	TypFunIns	::=	$\tau_1.\tau_2$
κ	$\in$	TyConSem	::=	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa,\overline{d}\rangle$
tyf	$\in$	TyFun	::=	$\delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$
app	$\in$	App	::=	$\tau\ tyf$
$svar$	$\in$	SVar	::=	$v \mid \beta$
ρ	$\in$	FRTyVar	::=	$\alpha \mid \beta$
sig	$\in$	SigSem	::=	$e \mid \forall\overline{\delta}e \mid \langle sig,\overline{d}\rangle.$
$subty$	$\in$	SubTy	::=	$\sigma \preceq_{vid} \sigma_2 \mid \kappa_1 \preceq_{tc} \kappa_2$
c	$\in$	EqCs	::=	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	::=	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	::=	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle \mid \downarrow sigid{=}ev$
e	$\in$	Env	::=	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1 \mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e,\overline{d}) \mid \langle e,\overline{d}\rangle \mid e_1 : e_2 \mid \texttt{ins}(e \mid subty)$

extra metavariables

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

E.8 Extension for duplicate identifiers in specifications

Figure E.15 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
$sigid$	$\in$	SigId		(signature identifiers)
$eqtv$	$\in$	EqTypeVar		(Equality type variables)
vid	$\in$	VId	::=	$vvar \mid dcon$
ltc	$\in$	LabTyCon	::=	tc^l
$ldcon$	$\in$	LabDatCon	::=	$dcon^l$
ty	$\in$	Ty	::=	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	::=	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	::=	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	::=	$\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp \mid \texttt{open}^l\ strid$
				$\mid \texttt{datatype}\ dn \stackrel{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
				$\mid \texttt{type}\ dn \stackrel{l}{=} ty \mid \texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}$
$atexp$	$\in$	AtExp	::=	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	::=	$atexp \mid \texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$
ltv	$\in$	LabTyVar	::=	$ty^l{}_1 \mid \texttt{dot-d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	::=	$ltv \mid \epsilon^l{}_{\mathsf{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	::=	vid^l_{p}
pat	$\in$	Pat	::=	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$
$sigdec$	$\in$	SigDec	::=	$\texttt{signature}\ sigid \stackrel{l}{=} exp \mid \texttt{dot-d}(\overrightarrow{term})$
$sigexp$	$\in$	SigExp	::=	$sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \mid \texttt{dot-s}(\overrightarrow{term})$
$spec$	$\in$	Spec	::=	$\texttt{val}\ vid :^l ty \mid \texttt{type}\ dn^l \mid$
				$\texttt{datatype}\ dn \stackrel{l}{=} cd \mid \texttt{structure}\ strid :^l sigexp$
				$\mid \texttt{eqtype}\ dn^l \mid \texttt{dot-d}(\overrightarrow{term})$
cd	$\in$	ConDesc	::=	$vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^{\,l}\ ty \mid \texttt{dot-e}(\overrightarrow{term})$
$topdec$	$\in$	TopDec	::=	$strdec \mid sigdec$
$prog$	$\in$	Program	::=	$topdec_1; ...topdec_n$
$strdec$	$\in$	StrDec	::=	$dec \mid \texttt{structure}\ strid \stackrel{l}{=} strexp$
$strexp$	$\in$	StrExp	::=	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$

extra metavariables

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

Figure E.16 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
θ	$\in$	IEqTypeVar		(equality type variables)
d	$\in$	Dependency	$::=$	l
ts	$\in$	ITyScheme	$::=$	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	$::=$	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\delta,\theta\rangle \mid \langle\mu,\overline{d}\rangle$
τ	$\in$	ITy	$::=$	$\langle\alpha,\theta\rangle \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle\beta,\theta\rangle \mid \langle\tau,\overline{d}\rangle$
Σ	$\in$	EqTypeStatus	$::=$	EQ_TYPE | NEQ_TYPE | SIG_TYPE
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa,\overline{d}\rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$
app	$\in$	App	$::=$	$\tau\ \ tyf$
$svar$	$\in$	SVar	$::=$	$v \mid \beta$
ρ	$\in$	FRTyVar	$::=$	$\alpha \mid \beta$
sig	$\in$	SigSem	$::=$	$e \mid \forall\overline{\delta}e \mid \langle sig,\overline{d}\rangle.$
$subty$	$\in$	SubTy	$::=$	$\sigma \preceq_{vid} \sigma_2 \mid \kappa_1 \preceq_{tc} \kappa_2$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle \mid \downarrow sigid{=}ev$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1$ $\mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e,\overline{d}) \mid \langle e,\overline{d}\rangle$ $\mid e_1 : e_2 \mid \texttt{ins}(e \mid subty) \mid \texttt{duplicates}(e)$

extra metavariables

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

E.9 Extension for include specifications

Figure E.17 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
$sigid$	$\in$	SigId		(signature identifiers)
$eqtv$	$\in$	EqTypeVar		(Equality type variables)
vid	$\in$	VId	::=	$vvar \mid dcon$
ltc	$\in$	LabTyCon	::=	tc^l
$ldcon$	$\in$	LabDatCon	::=	$dcon^l$
ty	$\in$	Ty	::=	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	::=	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	::=	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	::=	$\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp \mid \texttt{open}^l\ strid$
				$\mid \texttt{datatype}\ dn \stackrel{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
				$\mid \texttt{type}\ dn \stackrel{l}{=} ty \mid \texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}$
$atexp$	$\in$	AtExp	::=	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	::=	$atexp \mid \texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$
ltv	$\in$	LabTyVar	::=	$ty^l{}_{\mathtt{l}} \mid \texttt{dot-d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	::=	$ltv \mid \epsilon^l{}_{\mathtt{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	::=	vid^l_{p}
pat	$\in$	Pat	::=	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$
$sigdec$	$\in$	SigDec	::=	$\texttt{signature}\ sigid \stackrel{l}{=} exp \mid \texttt{dot-d}(\overrightarrow{term})$
$sigexp$	$\in$	SigExp	::=	$sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \mid \texttt{dot-s}(\overrightarrow{term})$
$spec$	$\in$	Spec	::=	$\texttt{val}\ vid :^l ty \mid \texttt{type}\ dn^l \mid \texttt{datatype}\ dn \stackrel{l}{=} cd \mid$
				$\texttt{structure}\ strid :^l sigexp \mid \texttt{include}\ sigexp^l$
				$\mid \texttt{eqtype}\ dn^l \mid \texttt{dot-d}(\overrightarrow{term})$
cd	$\in$	ConDesc	::=	$vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^{\,l}\ ty \mid \texttt{dot-e}(\overrightarrow{term})$
$topdec$	$\in$	TopDec	::=	$strdec \mid sigdec$
$prog$	$\in$	Program	::=	$topdec_1; ...topdec_n$
$strdec$	$\in$	StrDec	::=	$dec \mid \texttt{structure}\ strid \stackrel{l}{=} strexp$
$strexp$	$\in$	StrExp	::=	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$

extra metavariables

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

Figure E.18 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
θ	$\in$	IEqTypeVar		(equality type variables)
d	$\in$	Dependency	::=	l
ts	$\in$	ITyScheme	::=	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	::=	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	::=	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	::=	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\delta,\theta\rangle \mid \langle\mu,\overline{d}\rangle$
τ	$\in$	ITy	::=	$\langle\alpha,\theta\rangle \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle\beta,\theta\rangle \mid \langle\tau,\overline{d}\rangle$
Σ	$\in$	EqTypeStatus	::=	EQ_TYPE | NEQ_TYPE | SIG_TYPE
tfi	$\in$	TypFunIns	::=	$\tau_1.\tau_2$
κ	$\in$	TyConSem	::=	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa,\overline{d}\rangle$
tyf	$\in$	TyFun	::=	$\delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$
app	$\in$	App	::=	$\tau\ tyf$
$svar$	$\in$	SVar	::=	$v \mid \beta$
ρ	$\in$	FRTyVar	::=	$\alpha \mid \beta$
sig	$\in$	SigSem	::=	$e \mid \forall\overline{\delta}e \mid \langle sig,\overline{d}\rangle.$
$subty$	$\in$	SubTy	::=	$\sigma \preceq_{vid} \sigma_2 \mid \kappa_1 \preceq_{tc} \kappa_2$
c	$\in$	EqCs	::=	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	::=	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	::=	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle \mid \downarrow sigid{=}ev$
e	$\in$	Env	::=	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1 \mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e,\overline{d}) \mid \langle e,\overline{d}\rangle \mid e_1 : e_2 \mid \texttt{ins}(e \mid subty) \mid \texttt{duplicates}(e)$

extra metavariables

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

E.10 Extension for type sharing

Figure E.19 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label		(labels)
tv	$\in$	TyVar		(type variables)
tc	$\in$	TyCon		(type constructors)
$strid$	$\in$	StrId		(structure identifiers)
$vvar$	$\in$	ValVar		(value variables)
$dcon$	$\in$	DatCon		(datatype constructors)
$sigid$	$\in$	SigId		(signature identifiers)
$eqtv$	$\in$	EqTypeVar		(Equality type variables)
vid	$\in$	VId	::=	$vvar \mid dcon$
ltc	$\in$	LabTyCon	::=	tc^l
$ldcon$	$\in$	LabDatCon	::=	$dcon^l$
ty	$\in$	Ty	::=	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$
cb	$\in$	ConBind	::=	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^{\,l}\ ty$
dn	$\in$	DatName	::=	$\lceil tv\ tc \rceil^l$
dec	$\in$	Dec	::=	$\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp \mid \texttt{open}^l\ strid$
				$\mid \texttt{datatype}\ dn \stackrel{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$
				$\mid \texttt{type}\ dn \stackrel{l}{=} ty \mid \texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}$
$atexp$	$\in$	AtExp	::=	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$
exp	$\in$	Exp	::=	$atexp \mid \texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$
ltv	$\in$	LabTyVar	::=	$ty^l{}_{\mathtt{l}} \mid \texttt{dot-d}(\overrightarrow{term})$
$tvseq$	$\in$	TyVarSeq	::=	$ltv \mid \epsilon^l{}_{\mathtt{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$
$atpat$	$\in$	AtPat	::=	vid^l_{p}
pat	$\in$	Pat	::=	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$
$sigdec$	$\in$	SigDec	::=	$\texttt{signature}\ sigid \stackrel{l}{=} exp \mid \texttt{dot-d}(\overrightarrow{term})$
$sigexp$	$\in$	SigExp	::=	$sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \mid \texttt{dot-s}(\overrightarrow{term})$
$spec$	$\in$	Spec	::=	$\texttt{val}\ vid :^l ty \mid \texttt{type}\ dn^l \mid \texttt{datatype}\ dn \stackrel{l}{=} cd \mid$
				$\texttt{structure}\ strid :^l sigexp \mid \texttt{include}\ sigexp^l$
				$\mid \texttt{eqtype}\ dn^l \mid \texttt{dot-d}(\overrightarrow{term})$
				$\mid \texttt{sharing type}\ vid_1 = vid_2$
cd	$\in$	ConDesc	::=	$vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^{\,l}\ ty \mid \texttt{dot-e}(\overrightarrow{term})$
$topdec$	$\in$	TopDec	::=	$strdec \mid sigdec$
$prog$	$\in$	Program	::=	$topdec_1; ...topdec_n$
$strdec$	$\in$	StrDec	::=	$dec \mid \texttt{structure}\ strid \stackrel{l}{=} strexp$
$strexp$	$\in$	StrExp	::=	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$

extra metavariables

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

Figure E.20 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
θ	$\in$	IEqTypeVar		(equality type variables)
d	$\in$	Dependency	$::=$	l
ts	$\in$	ITyScheme	$::=$	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	$::=$	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\delta,\theta\rangle \mid \langle\mu,\overline{d}\rangle$
τ	$\in$	ITy	$::=$	$\langle\alpha,\theta\rangle \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle\beta,\theta\rangle \mid \langle\tau,\overline{d}\rangle$
Σ	$\in$	EqTypeStatus	$::=$	EQ_TYPE | NEQ_TYPE | SIG_TYPE
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa,\overline{d}\rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$
app	$\in$	App	$::=$	$\tau\ \ tyf$
$svar$	$\in$	SVar	$::=$	$v \mid \beta$
ρ	$\in$	FRTyVar	$::=$	$\alpha \mid \beta$
sig	$\in$	SigSem	$::=$	$e \mid \forall\overline{\delta}e \mid \langle sig,\overline{d}\rangle.$
$subty$	$\in$	SubTy	$::=$	$\sigma \preceq_{vid}\ \sigma_2 \mid \kappa_1 \preceq_{tc}\ \kappa_2$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle \mid \downarrow sigid{=}ev$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1$ $\mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e,\overline{d}) \mid \langle e,\overline{d}\rangle$ $\mid e_1 : e_2 \mid \texttt{ins}(e \mid subty) \mid \texttt{duplicates}(e)$ $\mid \texttt{sharingSig}(ev) \mid \texttt{sharingType}(e_1,e_2)$

extra metavariables

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

E.11 Extension for operator infixity

Figure E.21 External labelled syntax

external syntax (what the programmer sees, plus labels)

l	$\in$	Label	(labels)	$tv \in$ TyVar	(type variables)
tc	$\in$	TyCon	(type constructors)		
$strid$	$\in$	StrId	(structure identifiers)		
$vvar$	$\in$	ValVar	(value variables)		
$dcon$	$\in$	DatCon	(datatype constructors)		
$sigid$	$\in$	SigId	(signature identifiers)		
$eqtv$	$\in$	EqTypeVar	(Equality type variables)		
vid	$\in$	VId	$::=$	$vvar \mid dcon \mid \texttt{op}\ vvar \mid \texttt{op}\ dcon$	
ltc	$\in$	LabTyCon	$::=$	tc^l	
$ldcon$	$\in$	LabDatCon	$::=$	$dcon^l$	
ty	$\in$	Ty	$::=$	$tv^l \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^l$	
cb	$\in$	ConBind	$::=$	$dcon^l_{\mathsf{c}} \mid dcon\ \texttt{of}^l\ ty$	
dn	$\in$	DatName	$::=$	$\lceil tv\ tc \rceil^l$	
$digit$	$\in$	Digit	$::=$	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	
dec	$\in$	Dec	$::=$	$\texttt{val rec}\ tvseq\ pat \stackrel{l}{=} exp \mid \texttt{open}^l\ strid$	
				$\mid \texttt{datatype}\ dn \stackrel{l}{=} cb \mid \texttt{local}^l\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}$	
				$\mid \texttt{type}\ dn \stackrel{l}{=} ty \mid \texttt{abstype}\ dn \stackrel{l}{=} cb\ \texttt{with}\ dec\ \texttt{end}$	
				$\mid \texttt{infix}\ digit\ vid \mid \texttt{infixr}\ digit\ vid \mid \texttt{nonfix}\ vid$	
$atexp$	$\in$	AtExp	$::=$	$vid^l_{\mathsf{e}} \mid \texttt{let}^l\ dec\ \texttt{in}\ exp\ \texttt{end}$	
exp	$\in$	Exp	$::=$	$atexp \mid \texttt{fn}\ pat \stackrel{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^l \mid exp :^l ty$	
ltv	$\in$	LabTyVar	$::=$	$ty^l{}_1 \mid \texttt{dot-d}(\overrightarrow{term})$	
$tvseq$	$\in$	TyVarSeq	$::=$	$ltv \mid \epsilon^l{}_{\mathsf{v}} \mid (ltv, ..., ltv_n)^l \mid \texttt{dot-d}(\overrightarrow{term})$	
$atpat$	$\in$	AtPat	$::=$	vid^l_{p}	
pat	$\in$	Pat	$::=$	$atpat \mid \lceil ldcon\ atpat \rceil^l \mid pat :^l ty$	
$sigdec$	$\in$	SigDec	$::=$	$\texttt{signature}\ sigid \stackrel{l}{=} exp \mid \texttt{dot-d}(\overrightarrow{term})$	
$sigexp$	$\in$	SigExp	$::=$	$sigid^l \mid \texttt{sig}^l\ spec_1 \cdots spec_n\ \texttt{end} \mid \texttt{dot-s}(\overrightarrow{term})$	
$spec$	$\in$	Spec	$::=$	$\texttt{val}\ vid :^l ty \mid \texttt{type}\ dn^l \mid \texttt{datatype}\ dn \stackrel{l}{=} cd \mid$	
				$\texttt{structure}\ strid :^l sigexp \mid \texttt{include}\ sigexp^l$	
				$\mid \texttt{eqtype}\ dn^l \mid \texttt{dot-d}(\overrightarrow{term}) \mid \texttt{sharing type}\ vid_1 = vid_2$	
cd	$\in$	ConDesc	$::=$	$vid^l_{\mathsf{c}} \mid vid\ \texttt{of}^l\ ty \mid \texttt{dot-e}(\overrightarrow{term})$	
$topdec$	$\in$	TopDec	$::=$	$strdec \mid sigdec$	
$prog$	$\in$	Program	$::=$	$topdec_1; ...topdec_n$	
$strdec$	$\in$	StrDec	$::=$	$dec \mid \texttt{structure}\ strid \stackrel{l}{=} strexp$	
$strexp$	$\in$	StrExp	$::=$	$strid^l \mid \texttt{struct}^l\ strdec_1 \cdots strdec_n\ \texttt{end}$	
				$\mid strexp :^l sigexp \mid strexp :>^l sigexp$	

extra metavariables

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

Figure E.22 Syntax of constraint terms

ev	$\in$	EnvVar		(environment variables)
δ	$\in$	TyConVar		(type constructor variables)
γ	$\in$	TyConName		(type constructor names)
α	$\in$	ITyVar		(internal type variables)
β	$\in$	RigidTyVar		(set of rigid type variables)
θ	$\in$	IEqTypeVar		(equality type variables)
$\bowtie$	$\in$	InfixVar		(infixity variables)
d	$\in$	Dependency	$::=$	l
ts	$\in$	ITyScheme	$::=$	$\forall\overline{v}.\,\tau$
tcs	$\in$	ITyConScheme	$::=$	$\forall\overline{v}.\,\mu$
es	$\in$	EnvScheme	$::=$	$\forall\overline{v}.\,e$
μ	$\in$	ITyCon	$::=$	$\delta \mid \gamma \mid \texttt{arr} \mid \Lambda\alpha.\tau \mid \texttt{tv} \mid \langle\delta,\theta\rangle \mid \langle\mu,\overline{d}\rangle$
τ	$\in$	ITy	$::=$	$\langle\alpha,\theta\rangle \mid \tau\,\mu \mid \tau_1 \to \tau_2 \mid \langle\beta,\theta\rangle \mid \langle\tau,\overline{d}\rangle$
dir	$\in$	Direction	$::=$	$L \mid R \mid NONE$
Σ	$\in$	EqTypeStatus	$::=$	EQ_TYPE | NEQ_TYPE | SIG_TYPE
tfi	$\in$	TypFunIns	$::=$	$\tau_1.\tau_2$
κ	$\in$	TyConSem	$::=$	$\mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa,\overline{d}\rangle$
tyf	$\in$	TyFun	$::=$	$\delta \mid \Lambda\langle\alpha,\theta\rangle.\,\tau \mid \langle tyf,\overline{d}\rangle$
app	$\in$	App	$::=$	$\tau\ tyf$
$svar$	$\in$	SVar	$::=$	$v \mid \beta$
ρ	$\in$	FRTyVar	$::=$	$\alpha \mid \beta$
sig	$\in$	SigSem	$::=$	$e \mid \forall\overline{\delta}e \mid \langle sig,\overline{d}\rangle.$
$subty$	$\in$	SubTy	$::=$	$\sigma \preceq_{vid}\ \sigma_2 \mid \kappa_1 \preceq_{tc}\ \kappa_2$
c	$\in$	EqCs	$::=$	$\mu_1 = \mu_2 \mid e_1 = e_2 \mid \tau_1 = \tau_2$
$bind$	$\in$	Bind	$::=$	$\downarrow tc{=}tcs \mid \downarrow strid{=}es \mid \downarrow tv{=}ts \mid \downarrow vid{=}ts \mid \downarrow tv = \beta \mid \downarrow sigid{=}sig \mid \downarrow vid \overset{l}{=} \bowtie$
acc	$\in$	Accessor	$::=$	$\uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\langle\alpha,\theta\rangle \mid \uparrow vid{=}\langle\alpha,\theta\rangle \mid \downarrow sigid{=}ev$
e	$\in$	Env	$::=$	$\top \mid ev \mid bind \mid acc \mid c \mid \texttt{poly}(e) \mid \exists a.e \mid e_2;e_1 \mid \texttt{loc}\ e_1\ \texttt{in}\ e_2 \mid \texttt{or}(e,\overline{d}) \mid \langle e,\overline{d}\rangle \mid e_1 : e_2 \mid \texttt{ins}(e \mid subty) \mid \texttt{duplicates}(e) \mid \texttt{sharingSig}(ev) \mid \texttt{sharingType}(e_1,e_2)$

extra metavariables

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

Bibliography

[BAS] Bash - bourne-again shell. http://tiswww.case.edu/php/chet/bash/bashtop.html. Last accessed 13th September 2013.

[Bra04] Bernd Braßel. Typehope: There is hope for your type errors. In *In 16th International Workshop on Implementation and Application of Functional Languages (IFL'04)*, Lübeck, Germany, September 8-10 2004. University of Kiel. Report 0408., 2004.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[Dow01] Gilles Dowek. Handbook of automated reasoning. chapter Higher-order unification and matching, pages 1009–1062. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[DOX] Doxygen homepage. http://www.stack.nl/~dimitri/doxygen/. Last accessed July 2013.

[DPR05] Roberto Di Cosmo, Francois Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. In *Seventh International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 179–193, Nara, Japan, April 2005. Springer.

[EAS] EasyOcaml homepage. http://easyocaml.forge.ocamlcore.org/. Last accessed 1st May 2013.

[EMA] Emacs. http://www.gnu.org/software/emacs. Last accessed 13th July 2012.

[Gor00] Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[GS12] Takumi Goto and Isao Sasano. An approach to completing variable names for implicitly typed functional languages. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 131–140, New York, NY, USA, 2012. ACM.

[Han10] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[HH09] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electron. Notes Theor. Comput. Sci.*, 236:163–183, April 2009.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, pages 29–60, 1969.

[HW03] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European conference on Programming*, ESOP'03, pages 284–301, Berlin, Heidelberg, 2003. Springer-Verlag.

[JSO] JSON specfication. http://www.json.org/. Last accessed 13th July 2012.

[LG06] Benjamin Lerner and Grossman. Seminal: searching for ML type-error messages. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 63–73, New York, NY, USA, 2006. ACM.

[LY00] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

[Mac84] David MacQueen. Modules for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, New York, NY, USA, 1984. ACM.

[Mca98] Bruce J. Mcadam. On the unification of substitutions in type inference. In *Implementation of Functional Languages (IFL '98)*, pages 139–154. Springer-Verlag, 1998.

[McA02] B. J. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, University of Edinburgh, 2002.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[MLD] Mldoc homepage. http://people.cs.uchicago.edu/~jhr/tools/ml-doc.html. Last accessed July 2013.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4:258–282, 1982.

[MTHM98] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (Revised)*. The MIT Press, 55 Hayward Street, Cambridge, MA 02142-1493, USA, 1998.

[OL98] K. Yi O. Lee. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20:707–723, 1998.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

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

[PRO] *Prin Of Programming Languages*. McGraw-Hill Education (India) Pvt Limited.

[Pur91] Paul W Purdom. A practical unification algorithm. *Information Sciences*, 55(1):123–127, 1991.

[Rah10] Vincent Rahli. Investigations in intersection types: Confluence and semantics of expansion in the lamba-calculus, and a type error slicing method. http://www.macs.hw.ac.uk/~rahli/articles/thesis.pdf, 2010. Ph.D. Thesis. Last accessed Monday 16th July 2012.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[RWK10] Vincent Rahli, Joe Wells, and Fairouz Kamareddine. A constraint system for a SML type error slicer. http://www.macs.hw.ac.uk/~rahli/articles/type-error-slicing-2010-08-19-TR-0079-BW.pdf, 2010. Last accessed Monday 16th July 2012.

[SBG09a] Tom Schrijvers, Maurice Bruynooghe, and John P Gallagher. From monomorphic to polymorphic well-typings and beyond. In *Logic-Based Program Synthesis and Transformation*, pages 152–167. Springer, 2009.

[SBG09b] Tom Schrijvers, Maurice Bruynooghe, and JohnP. Gallagher. From monomorphic to polymorphic well-typings and beyond. In Michael Hanus, editor, *Logic-Based Program Synthesis and Transformation*, volume 5438 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2009.

[SMLa] Sml# homepage. http://www.pllab.riec.tohoku.ac.jp/smlsharp/. Last accessed July 2013.

[SMLb] Smldoc homepage. http://www.pllab.riec.tohoku.ac.jp/smlsharp/?SMLDoc. Last accessed July 2013.

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

[Tip95] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[Yan00] Jun Yang. Explaining type errors by finding the source of a type conflict. In *In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.

[Yan01] Jun Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, 2001.

[YMTW00] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. Improved type error reporting. In *In Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 71–86, 2000.

Index