A static semantics for Haskell

Simon L Peyton Jones
Philip Wadler
Department of Computing Science, University of Glasgow, G12 8QQ
(email: {simonpj, wadler}@cs.glasgow.ac.uk)

February 19, 1992

Abstract

This paper gives a static semantics for a large subset of Haskell, including giving a translation into a language without overloading.

It is our intention to cover the complete language in due course.

One innovative aspect is the use of ideas from the second-order lambda calculus to record type information in the program.

Contents

List of Figures

1 Introduction

The purpose of this paper is to give a complete static semantics for Haskell (Hudak et al. [1990]).

One approach to giving a static semantics is to give a translation from the full language into a core, which is then given a static semantics. This is economical, but since the translations are sometimes elaborate, it can lead to incomprehensible error messages from the type checker. Instead we have chosen to give a static semantics for the whole language directly.

Wadler & Blott [1989] give a formal set of type rules to describe the form of overloading in Haskell. While compact and elegant, the rules are quite hard to understand, especially since they do not take explicit account of Haskell class declarations. In effect they make each class have a single overloaded operation and identify the class with this operation. This is adequate, because the operation could be a dictionary, but it does not mesh very well with Haskell, and performing the mental translation between the two is quite tricky.

In this paper we provide a set of type rules which avoids these difficulties, and covers the complete language rather than a small subset.

1.1 Overview

We have modelled this paper on "The definition of Standard ML" (Harper, Milner & Tofte [1989]), which provided a very useful source of ideas and a high standard to work to.

In addition to performing type inference, our type system gives a *translation* of the original source program into one in which two implied aspects are made explicit:

- Polymorphism is made explicit, using type abstraction and application exactly as in the second-order lambda calculus.

  It is often necessary in later stages of the compiler to know the types of parts of the program. For example, a strictness analyser needs to know this when when looking for fixed points, and a code generator may be able to generate better code when it knows that only certain types can occur.

  It would be possible to include type information by annotating the parse tree of the program, but it is very hard to maintain this information when the program is transformed by subsequent compiler passes. Using the machinery of the second-order lambda calculus is (so far as we know) an original way to ensure that type information is maintained consistently in the face of such transformations. It is rather nice to see a practical application of this very beautiful calculus.

- Overloading is made explicit, by passing extra dictionary parameters to overloaded functions; these dictionaries include the methods to implement the operations of the class.

  Exactly how this is done is quite subtle, and is explained informally in Section 2.

Section 3 with a summary of the notation we use throughout the paper.

Section 4 then gives the abstract syntax of both the source language and the target translated language. This is followed by the type inference rules themselves, beginning with the rules for whole programs in Section 5 and working down to smaller constructs (Sections 6 through 12). A non-linear order of reading is more likely to be productive for these sections than a linear pass.

A careful correspondence is maintained between the source syntax and the inference rules:

- for every syntactic nonterminal there is a judgement form, and
- for every syntactic production of a nonterminal there is an inference rule for the judgement form corresponding to that nonterminal.

In addition, there are a few further inference rules which are not syntax driven. Examples are the rules SPEC and GEN for introducing and eliminating $\forall$ in polymorphic types.

1.2 Shortcomings

The paper is still incomplete. The aspects of Haskell that are not covered (and will be) are, in rough order of priority:

- Imports.
- Ambiguity.
- `deriving` clauses in `data` declarations.
- Default declarations in `class` declarations.
- Constant and `n+k` patterns. These require something substantially more sophisticated than the $\overset{\text{pat}}{\vdash}$ judgement given later. The reason for this is that constants involve some translation due to overloading, and `n+k` patterns require both this and some further translation to subtract the `k`.

In addition we hope to add some further sections giving advice about implementation.

Finally, here is list of unresolved issues:

- Prove substitution lemma.
- How do we prove a principal type theorem?
- Do all derivations give the same translation? Can we proved different ones equivalent?
- There is an interaction between monomorphic pattern-bindings and ambiguity. Consider the expression `x where y=1`. Making `y` monomorphic will result in a free dictionary, which will at some stage be resolved by the ambiguity mechanism. In other (non-numeric) cases this might actually cause an error, because the ambiguity-resolution

mechanism is restricted to numeric classes. All in all it seems better to exclude such unused definitions, and we assume this is done by the initial dependency analysis. But, we can't simply remove unused defns in case they contain a type error! Also at top level, we must be careful to eliminate only thosre which are not exported.

2 A sketch of how overloading is resolved

The distinctive feature of the Haskell type system is the provision of systematic overloading. In this section we sketch informally how overloading is implemented; we assume familiarity with the programmers-eye-view of Haskell.

The overloading is resolved by translating the program into an equivalent non-overloaded one, in which overloaded functions take extra parameters. For example:

```
f :: (Ord a, Num a) => a -> a -> a
f x y = if (x>y) then x+y else x
```

`x` and `y` must belong to a type which is an instance of both `Ord` (for the `>` operation) and `Num` (for the `+` operation). The phrase `(Ord a, Num a)` is called a *context*, and consists of an ordered sequence of *class assertions*.

The translation of `f` looks like this:

```
f dord dnum x y = if ((>) dord x y) then ((+) dnum x y) else x
```

`f` takes two extra *dictionary arguments* `dord` and `dnum`, one for each class assertion in the context in the type of `f`, and in the same order[1].

In the translated program, the functions `(>)` and `(+)` are *selectors* which extract the appropriate method from the dictionary to which they are applied; the method thus extracted is what does the actual comparison or addition.

Now consider a call of `f`:

```
g :: Int -> Int
g x = f x x
```

Here we have deliberately given `g` a less general type signature than it could have. Here is the translation of `g`:

```
g x = f ordInt numIntD x x
```

Two extra parameters are supplied to `f`, namely the `Ord` dictionary and the `Num` dictionary for the `Int` type.

The formal type rules deliberately abstract away from whether the dictionary arguments are passed in a curried fashion (as in this example), or in an uncurried fashion as a tuple of dictionaries. In our examples we will stick to the curried form.

[1] Strictly speaking, translated programs have types attached to each variable in a pattern, but we omit them here for clarity.

2.1 Dictionaries

What does a dictionary contain? Its format is set by a class declaration; here, for example, is one taken from the `Prelude`:

```
class (Real a, Fractional a) => RealFloat a where
        properFraction :: a -> (Integer, a)
        approxRational :: a -> a -> Rational
```

The dictionary for a class has two parts, either of which may be empty:

- The first part consists of the dictionaries for the superclasses, in this case `Real` and `Fractional`.
- The second part consists of the methods for the class itself, in this case methods for `properFraction` and `approxRational`.

The dictionary will usually be represented by a tuple of all these components, though the actual type rules abstract away from the details of the representation.

2.2 Instances

An instance declaration may have a context; for example

```
instance (Eq a, Eq b) => Eq (a,b) where
        (==) (x1,y1) (x2,y2) = (x1 == x2) & (y1 == y2)
```

From this instance declaration we produce a *dictionary function* which, given dictionaries for equality on things of type `a` and `b`, returns a dictionary for equality on things of type `(a,b)`. Suppose this function is called `eqPairFun`.

Now, given the function definition

```
f :: (Eq a) => a -> a -> Bool
f x y = (x,y) == (y,x)
```

we translate to the following

```
f eqD x y = (==) (eqPairFun eqD eqD) (x,y) (y,x)
```

3 Notation

This section summarises our notational conventions.

3.1 Sequences

The notation

$$\langle x_1, \ldots, x_n \rangle$$

stands an *ordered* sequence of the identifiers $x_1, \ldots, x_n$. Sequences can be concatenated with $+\!\!+$.

3.2 Judgements

There is one *judgement form* for each nonterminal in the abstract syntax, and one *rule* for that judgement form for each production of that nonterminal.

In total therefore, there is a large number of judgements, but they all take one of two forms. The first is quite conventional:

$$\mathit{inherited} \overset{\text{nonterminal}}{\vdash} \mathit{construct} : \mathit{synthesised}$$

which can be read: "given the information *inherited* we can derive the information *synthesised* from the construct *construct*", where *construct* is the right-hand side of one of the productions of *nonterminal*. The most typical form of this general scheme is to deduce types, thus:

$$\mathit{environment} \overset{\text{nonterminal}}{\vdash} \mathit{construct} : \mathit{type}$$

but in general *synthesised* may not be a type; it might be a set of value-type bindings, or an overloaded context, for example.

The rules are designed to make the derivation of a type-inference algorithm as easy as possible. In particular, the *inherited* parts will generally be arguments to the type-inference function, and the *synthesised* parts will generally be its result.

The annotation *nonterminal* on the turnstile serves to identify the judgement form. At the beginning of the rules for each judgement form we always give the general form of the judgement, which makes explicit the form of the judgement.

Many of the rules also express the *translation* of a language construct, in which case the second form of judgement is used:

$$\mathit{inherited} \overset{\text{nonterminal}}{\vdash} \mathit{construct} \leadsto \mathbf{construct} : \mathit{synthesised}$$

which can be read: "given the information *inherited* we can derive the information *synthesised* from the construct *construct*, which translates to **construct**".

In general, we use italic for source language nonterminals (*nonterminal*), and bold font for target language nonterminals (**nonterminal**).

3.3 Types

We distinguish throughout between *syntactic types* written by a Haskell programmer, which have the abstract syntax given in Section 4, and *semantic types* manipulated by the type inference rules. The former are denoted by words, such as *monotype*, while the latter are denoted by symbols, such as τ. The correspondence between the two is given by this table:

Monotypes	τ	$\rightarrow$	α		*Typevariables*
		$\mid$	$\chi\ \tau_1 \ldots \tau$	$(k \geq 0)$	*Structuredtype*
		$\mid$	$\tau_1 \rightarrow \tau_2$		*Functiontype*
		$\mid$	$[\tau]$		*Listtype*
		$\mid$	$(\tau_1, \ldots, \tau_k)$	$(k = 0 \text{ or } k \geq 2$	*Tupletype*
		$\mid$	`Bool`		*Booleantype*
Polytypes	σ	$\rightarrow$	τ		
		$\mid$	$\forall \alpha_1 \ldots \alpha_k .\ \sigma$		*Universalquantification*
		$\mid$	$\theta \Rightarrow \sigma$		*Predication*
Contexts	θ	$\rightarrow$	$\langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n \rangle$		

Figure 1: Syntax of semantic types

	Syntactic form	Semantic form
Type variable	*tyvar*	α
Monotype	*monotype*	τ
Overloaded type	(none)	ρ
Polytype	*polytype*	σ
Type constructor	*tycon*	χ
Class	*class*	κ
Context	*context*	θ

This distinction leads to a number of rather trivial-looking rules which translate syntactic types into semantic types. They are not as trivial as they seem, since they perform three useful functions:

- Checking that only in-scope type variables, constructors and class names are used.
- Checking that type constructors are only used with the correct arity, and expanding type synonyms.
- Since the syntactic names of constructors and classes each appear as the domain of an environments (see Section 3.4), the environment combiners will check that a type constructor or class is not declared more than once.

Almost all of these goals could also be achieved by merging the syntactic and semantic forms, and giving rather simpler validation rules instead. The exception is the expansion of type synonyms, which is only conveniently done by translation.

Figure 1 gives the syntax of semantic types. The unit type, () is treated as a case of the tuple type with zero components. There is no 1-tuple type, however.

The syntax of monotypes, τ, includes cases for functions, lists, tuples and so on. All of these can be regarded as special cases of the second production, with particular type constructors χ.

Environment	Notation	Type
Total environment	E	(TCE, TVE, VE, CE, IE)
Value environment	VE	(CVE, GVE, LVE)
Constructor value environment	CVE	$con \mapsto \sigma$
Global value environment	GVE	$var \mapsto \sigma$
Local value environment	LVE	$var \mapsto \tau$
Type-constructor environment	TCE	$tycon \mapsto \Lambda\alpha_1 \ldots \alpha_n.\ \tau$
Type-variable environment	TVE	$tyvar \mapsto \alpha$
Class environment	CE	$class \mapsto (\kappa, \forall\alpha(\theta, methods : GVE))$
Instance environment	IE	(GIE, LIE)
Global instance environment	GIE	$\mathbf{dfun} \mapsto \forall\alpha_1 \ldots \alpha_n.\ \theta \Rightarrow \kappa\ \tau$
Local instance environment	LIE	$\mathbf{dvar} \mapsto \kappa\ \tau$

Figure 2: Environments

The first two productions for polytypes, σ are unsurprising. The third, $\theta \Rightarrow \sigma$, is a predicated type, where the context, θ, is an ordered list of predicates of form $\kappa\ \tau$ enclosed in angle brackets $\langle\rangle$.

Notice that the syntax permits some rather peculiar types, such as:

$$\forall\alpha.\ \kappa_1\ \alpha \Rightarrow \forall\beta.\ \kappa_2\ \beta \Rightarrow \alpha \rightarrow \beta \rightarrow \alpha$$

These types only arise (only) from `class` declarations. For example, the declaration

```
class Foo x where
        op :: Bar y => x -> y -> x
```

attributes just such a type to `op`. The operational reading is that `op` is a function taking a dictionary for `Foo`, from which it extracts the method for `op`. This method is a function which takes a dictionary for `Bar` in order to do its work.

The function $fv()$ applies to polytypes (and hence overloaded types and monotypes) and returns the free type variables of the type. It extends in the natural way to sets of types.

3.4 Environments

An *environment* is a finite function whose domain is some name space. Sometimes we package environments up into tuples, and still call the resulting tuple an environment.

There are a number of different sorts of environments, which are given in Figure 2.

There are a number of generic operations which can be performed on environments, which are described next. After this we discuss each individual environment separately.

3.4.1 Generic operations on environments

Environments may be written explicitly using curly brackets thus: $GVE_1 = \{var_1 : \sigma_1, var_2 : \sigma_2\}$. An empty environment is denoted $\{\}$.

The domain and range of an environment may be extracted using the functions $dom()$ and $ran()$ respectively. Thus $dom(GVE_1) = \{var_1, var_2\}$, and $ran(GVE_1) = \{\sigma_1, \sigma_2\}$.

An environment is applied to a member of its domain to return the corresponding element of the range. For example, $GVE_1\ var_2 = \sigma_2$.

The notation $TE\ of\ E$ extracts the type environment TE from the total environment E, and similarly for all components of a tupled environment.

Every local value environment is also a global value environment, since every monotype is also a polytype. The environments $\forall\alpha.\ GVE$ and $\theta \Rightarrow GVE$ are defined thus:

$$\begin{array}{rcl}(\forall\alpha.\ GVE)\ var & = & \forall\alpha.\ (GVE\ var)\\ (\theta \Rightarrow GVE)\ var & = & \theta \Rightarrow (GVE\ var)\end{array}$$

The operations $\oplus$ and $\vec{\oplus}$ combine environments. The former checks that the domains of its arguments are distinct, while the latter "shadows" its left argument with its right. For example:

$$\begin{array}{rcl}(GVE_1 \oplus GVE_2)\ var & = & \begin{cases} GVE_1\ var & \text{if } var \in dom(GVE_1) \text{ and } var \notin dom(GVE_2)\\ GVE_2\ var & \text{if } var \in dom(GVE_2) \text{ and } var \notin dom(GVE_1)\end{cases}\\ (GVE_1 \vec{\oplus} GVE_2)\ var & = & \begin{cases} GVE_1\ var & \text{if } var \in dom(GVE_1) \text{ and } var \notin dom(GVE_2)\\ GVE_2\ var & \text{if } var \in dom(GVE_2)\end{cases}\end{array}$$

and similarly for all other environments. For brevity, we write $E_1 \oplus E_2$ instead of a tuple of the sums of the components of E_1 and E_2; and we write $E \oplus CE$ to combine CE into the appropriate component of E (and similarly for other components of tupled environments).

Implicit side conditions There are two implicit side conditions associated with environments, as follows:

- If $E_1 \oplus E_2$ appears in a rule, then the side condition $dom(E_1) \cap dom(E_2) = \emptyset$ is implied.
- Similarly, if $E\ var$ appears in a rule, then the side condition $var \in dom(E)$ is implied.

3.4.2 Value environments

Value environments are quite conventional, mapping variable and constructor names to types.

We distinguish the *local value envionment*, whose range is monotypes, from the *global value environment* whose range is polytypes.

3.4.3 Type constructor environment

The type constructor environment, TCE, binds *all* type constructors, both those introduced by `type` and those introduced by `data`. It maps the name of a type constructor onto a function which maps types onto types. The arguments supplied to this function will be the arguments to the type constructor.

In the case of type synonyms, introduced with `type`, the application implements the expansion of the synonym.

In the case of a type constructor *tycon* introduced with `data`, the function will always be of the form

$$\Lambda\alpha_1 \ldots \alpha_n.\chi\ \tau_1 \ldots \tau_k$$

where χ is the semantic type constructor corresponding to *tycon*. The application has achieved no more than to check that the constructor is used with the correct arity.

3.4.4 Type-variable environment

A type variable environment just maps type variable syntactic names to their semantic counterparts. Its main use is local, in inference rules which validate type signatures written explicitly in Haskell, and checks that only in-scope type variables are mentioned.

However, it is part of the total environment, because it also binds the type variables mentioned in the headers of `class` and `instance` declarations.

3.4.5 Class environment

The class environment gives information about the semantic name and signature of a class. Members of its domain take the form

$$(\kappa, \forall\alpha(\theta, \mathit{methods} : \mathit{GVE})$$

Here κ is the semantic name of the class and α is the type variable with respect to which overloading takes place. θ specifies its superclasses, in the order they must appear in a dictionary for the class. Finally, GVE gives the type signature of the class, while *methods*, a sequence of the variable names in $\mathit{dom}(\mathit{GVE})$, establishes the order in which the methods must appear in a dictionary for the class.

3.4.6 Global instance environment

The global instance environment associates a dictionary function name **dfun** with a type of form

$$\forall\alpha_1 \ldots \alpha_n.\ \theta \Rightarrow \kappa\ (\chi\ \beta_1 \ldots \beta_m)$$

This says that **dfun** is a (polymorphic) function which will take dictionaries corresponding to θ, and map them into a dictionary giving methods for the $(\chi\ \beta_1 \ldots \beta_m)$ instance of class κ. Notice that because instance declarations are syntactically restricted to declaring an instance of a class over a type of form $(\chi\ \beta_1 \ldots \beta_m)$, the types in the global instance environment also have this form.

It is important that each type constructor χ is an instance of a given class κ at most one way. This is ensured by adding the following extra implicit side condition to the operation

$GIE_1 \oplus GIE_2$: no two types, from the range of GIE_1 and GIE_2 respectively, should have both the same class κ and same type constructor χ. In addition $\oplus$ should ensure that the dictionary function names are disjoint as usual, but since they are introduced by the type checker itself, this is just a way of saying that it should introduce a distinct name each time.

In a practical implementation the global instance environment is indexed by (κ, χ) pairs rather than the dictionary function names.

An important invariant is that these global instance types have no free variables; the $\forall.\ \alpha_1 \ldots \alpha_n$ binds all the type variables $\beta_1 \ldots \beta_m$ and those free in θ.

3.4.7 Local instance environment

The local instance environment maps a dictionary variable **dvar** to an expression of form $\kappa\ \tau$. This asserts that **dvar** is a dictionary corresponding to the τ instance of class κ. In contrast to the global instance environment, the type τ may have free type variables.

The function *zip* constructs a local instance environment from a sequence of dictionary names *dicts* and a context θ, thus

$$\{d_1 : \kappa_1\ \tau_1, \ldots, d_n : \kappa_n\ \tau_n\} = zip\ \langle \mathbf{dvar}_1, \ldots, \mathbf{dvar}_n \rangle\ \langle \kappa_1\ \tau_1, \ldots, \kappa_n\ \tau_n \rangle$$

4 Abstract syntax

In this section we give the abstract syntax of the source and translated language.

4.1 Source syntax

The abstract syntax of the source language is given in Figures 3, 4 and 5.

The abstract syntax expresses the *parse tree* of the program. The punctuation symbols of the concrete syntax, such as keywords and parentheses, which guide the parse, are therefore redundant and can be omitted. In particular, we omit from the abstract syntax all parentheses used for grouping, and all mention of operator fixity and precedence. To preserve legibility of the abstract syntax, however, we retain most of the remaining punctuation. An implementation would not do so.

The abstract syntax also differs from the *concrete* syntax of Haskell in a number of other important respects:

Grouping of declarations. The various sorts of declarations (type, class, instance and value) are assumed to be sorted into groups, as is shown by the *program* production.

Dependency analysis on value declarations. It is well known that in order to obtain the more general possible types from a group of local bindings in the Hindley-Milner system, it is necessary to perform *dependency analysis* on them so as to make explicit where recursion actually occurs and where it does not.

program	$\rightarrow$	*typedecl* ; *classdecl* ; *instdecl* ; *binds*	
typedecl	$\rightarrow$	$typedecl_1$; $typedecl_2$	Type declarations
	$\mid$	`data` *context* `=>` *tycon* $tyvar_1 \ldots tyvar_k$ `=` *condecl* $(k \geq 0)$	
	$\mid$	`type` *tycon* $tyvar_1 \ldots tyvar_k$ `=` *monotype* $(k \geq 0)$	
condecl	$\rightarrow$	$condecl_1$ `|` $condecl_2$	Constructor bindings
	$\mid$	*con* $monotype_1 \ldots monotype_k$ $(k \geq 0)$	
classdecl	$\rightarrow$	$classdecl_1$ `then` $classdecl_2$	Class declarations
	$\mid$	`class` *context* `=>` *class tyvar* `where` *sigs*	
instdecl	$\rightarrow$	$instdecl_1$; $instdecl_2$	Instance declarations
	$\mid$	`instance` *context* `=>` *class monotype* `where` *binds*	
binds	$\rightarrow$	*bind*	
	$\mid$	*bind* `with` *sigs*	
	$\mid$	$binds_1$ `then` $binds_2$	Nested declarations
sigs	$\rightarrow$	$\{\}$	Type signatures
	$\mid$	$sigs_1$ `and` $sigs_2$	
	$\mid$	*var* :: *polytype*	
bind	$\rightarrow$	*monobinds*	Set of bindings
	$\mid$	`rec` *monobinds*	Recursive set of bindings
monobinds	$\rightarrow$	$monobinds_1$; $monobinds_2$	A set of bindings
	$\mid$	*pat grhs*	A pattern binding
	$\mid$	*var match*	A function binding
	$\mid$	$\{\}$	Empty set of bindings

Figure 3: Abstract syntax for Haskell: declarations and bindings

match	$\rightarrow$	$match_1$ [] $match_2$		Case alternatives
	$\mid$	*mrule*		
mrule	$\rightarrow$	*grhs*		
	$\mid$	*pat mrule*		
grhs	$\rightarrow$	$grhs_1$ [] $grhs_2$		Guarded right-hand side
	$\mid$	`|` *guard* `=` *exp*		
	$\mid$	`=` *exp*		
guard	$\rightarrow$	*exp*		
pat	$\rightarrow$	*var*		Variable
	$\mid$	`_`		Wildcard
	$\mid$	*con* $pat_1 \ldots pat_n$		Compound pattern
	$\mid$	`~`*pat*		Lazy pattern
	$\mid$	*var*`@`*pat*		As-pattern
	$\mid$	`[`$pat_1, \ldots, pat_n$`]`	$(n \geq 0)$	List pattern
	$\mid$	`(`$pat_1, \ldots, pat_n$`)`	$(n \geq 2)$	Tuple pattern
	$\mid$	`()`		Unit pattern
exp	$\rightarrow$	*var*		Variable
	$\mid$	*con*		Constructor
	$\mid$	*integer*		
	$\mid$	*float*		
	$\mid$	*char*		
	$\mid$	*string*		
	$\mid$	`\` *mrule*		Lambda abstraction
	$\mid$	*exp* exp'		Application
	$\mid$	`case` *exp* `of` *match*		Case expression
	$\mid$	`if` exp_1 `then` exp_2 `else` exp_3		Conditional
	$\mid$	`[`*exp*`|`*quals*`]`		List comprehension
	$\mid$	`[`*exp*`..]`		Enumerations
	$\mid$	`[`exp_1`,`exp_2`..]`		
	$\mid$	`[`exp_1`..`exp_2`]`		
	$\mid$	`[`exp_1`,`exp_2`..`exp_3`]`		
	$\mid$	`let` *binds* `in` *exp*		Local definitions
	$\mid$	`[`exp_1`, ... ,`exp_n`]`	$(n \geq 0)$	Lists
	$\mid$	`(`exp_1`, ... ,`exp_n`)`	$(n = 0 \ or \ n \geq 2)$	Tuples
quals	$\rightarrow$	*quals*`,`*quals*		Qualifiers
	$\mid$	*pat*`<-`*exp*		Generator
	$\mid$	*guard*		Filter

Figure 4: Abstract syntax of Haskell: patterns and expressions

polytype	$\rightarrow$	*context* `=>` *monotype*		
	$\mid$	*monotype*		
monotype	$\rightarrow$	*tyvar*		Type variable
	$\mid$	$monotype_1$ `->` $monotype_2$		Function type
	$\mid$	`[`*monotype*`]`		List type
	$\mid$	`(`$monotype_1$ `,` $\ldots$ `,` $monotype_n$`)`	$(n = 0 \text{ or } n \geq 2)$	Tuple type (incl unit)
	$\mid$	*tycon* $monotype_1 \ldots monotype_n$	$(n \geq 0)$	Constructed type
context	$\rightarrow$	`(`$class_1$ $tyvar_1$ `,` $\ldots$ `,` $class_n$ $tyvar_n$`)`		

Figure 5: Abstract syntax of Haskell: types

pat	$\rightarrow$	$var : \tau$	Translated patterns
	$\mid$	$var : \kappa\ \tau$	Dictionary variables
	$\mid$	...other productions as before	
exp	$\rightarrow$	$\Lambda\alpha_1 \ldots \alpha_n$ `->` **exp**	Type abstraction
	$\mid$	**exp** $\tau_1 \ldots \tau_n$	Type application
	$\mid$	...other productions as before	

Figure 6: Extra syntax for translated programs

We assume that this dependency analysis has already taken place before the program is presented to the type checker, so the abstract syntax allows the result of such dependency analysis to be expressed. In particular, groups of value bindings *binds* are (topologically) sorted into nested groups, combined with `then`. Each group is then either a non-recursive group *monobinds* or a recursive group `rec` *monobinds*. Declarations within a *monobinds* are combined together with `;`.

The `then` production is used solely to allow the generation of most general types; it does *not* introduce a nested scope, so that no "shadowing" of names occurs as a result of its use.

We use the keyword `let` instead of `where` in the production for expressions as a reminder that bindings have this structure.

Type signatures. These are assumed to be split up so that each signature gives a type to only one variable.

Polytypes. The syntax for polymorphic types *polytype* does not contain any explicit quantification; instead, every such type is implicitly quantified over its free variables.

Generality. The abstract syntax for several constructs is more permissive than the concrete syntax.

For example, the abstract syntax for lambda abstractions allows multiple guarded bodies. The restriction to a single guarded body in the concrete syntax is done only on stylistic grounds. The syntax is more modular in the more general form we give here, and this gives rise directly to greater modularity in the type inference rules.

Similarly, the alternatives in a `case` expression can only have one pattern in the concrete syntax, rather than several as the above abstract syntax allows.

Similarly, there is an empty production for *monobinds* which is required only for instance declarations with no `where` clause.

The unit expression, written `()` in Haskell, is treated as a zero-tuple.

List comprehensions. The syntax does not permit a list comprehension with an empty qualifier list, whereas the Report does. This is an issue of style only; there is no deep issue here.

Class signatures. Default declarations are not (yet) incorporated, so *csigs* has no productions for them.

Notice also that the punctuation generated by the abstract syntax does not conform precisely to the concrete syntax, because the same nonterminal *grhs* is used for lambda abstractions as for function bindings. Since this is an *abstract* syntax, this does not matter in the slightest.

4.2 Translated syntax

The abstract syntax of the translated program is very similar to that of the source program. We write the nonterminals of translated programs in bold font: thus **exp** is a translated form of *exp*. Figure 6 gives the extra productions present in the translated program; all the existing productions are retained.

A program is translated to a **binds**, with only value bindings remaining. The other forms of declaration (`import`, `class`, `instance`, `type`, `data`, and type signatures) have been eliminated.

There are some extra productions to handle the second-order lambda caclulus constructs. In particular, every variable in a pattern has a type attached to it, and there are extra forms of expression for type abstraction and type application.

In the translation, dictionaries are introduced explicitly. We use **dvar** for variables bound to dictionaries. Dictionary variables are a distinct name-space from ordinary variables, so that there can be no name-clashes between dictionary variables and ordinary ones. The type of a dictionary, needed to annotate a dictionary variable in a pattern, is written $\kappa\ \tau$.

4.3 Notation for translated programs

In constructing translations there are a number of design choices to be made, especially where overloading is concerned. For example, are dictionaries represented as tuples, or trees, or what? To avoid making a premature committments it is convenient to use some extra notation when writing down translations.

We interpret this extra notation as shorthand for a particular representation, rather than as extra productions for the abstract syntax of translations. In this section we summarise the extra notation, together with the design choices they imply.

4.3.1 Applying and constructing overloaded values

When an overloaded value is used, it must be applied to appropriate dictionaries. Dually, to construct an overloaded value, some form of lambda abstraction must be used.

- The notation

  $$\textbf{exp}\ \mathit{dicts}$$

  where *dicts* is a sequence of dictionary names, stands for the application of a (overloaded) expression **exp** to the dictionaries in *dicts*. This notation abstracts away from whether the application is curried or not.

- The notation

  $$\backslash \mathit{dicts} : \theta\ \texttt{->}\ \textbf{exp}$$

  stands for a lambda abstraction of the dictionaries in *dicts*, each of which has the type given by the corresponding element of θ.

4.3.2 Building dictionaries and taking them apart

Recall that a *dictionary* for a particular instance of a class κ is a value which contains

- the methods for each operation of κ at that instance;
- and dictionaries for the superclasses of κ.

We need two constructs, one to build a dictionary, and one to take it apart.

- The notation

  $$\langle \mathit{dicts}, \mathit{methods} \rangle$$

  where *dicts* is a sequence of dictionary names and *methods* is a sequence of method names, stands for a dictionary containing these methods and superclass dictionaries.

  It will typically be represented as a flat tuple, with a component for each method or superclass dictionary. If there is only one method and no superclasses (or vice versa), then it will typically be represented by the method itself. The notation allows us to abstract away from these details.

$$\boxed{E \overset{\text{program}}{\vdash} \mathit{program} \leadsto \textbf{binds} : E'}$$

$$\text{PROGRAM} \quad \frac{\begin{array}{ll} (1) & (TCE \textit{ of } E) \oplus (TCE \textit{ of } TE_T); CE_C \overset{\text{typedecl}}{\vdash} \mathit{typedecl} : (TCE_T, CVE_T) \\ (2) & E \oplus TE_T \oplus CVE_T \overset{\text{classdecl}}{\vdash} \mathit{classdecl} \leadsto \textbf{binds}_C : (GVE_C, CE_C, GIE_C) \\ (3) & E' = (TCE_T, (CVE_T, GVE_C, \{\}), CE_C, (GIE_C, \{\})) \\ (4) & E \oplus E' \oplus GIE_I \oplus GVE_V \overset{\text{inst}}{\vdash} \mathit{instdecl} \leadsto \textbf{monobinds}_I : GIE_I \\ (5) & E \oplus E' \oplus GIE_I \overset{\text{bind}}{\vdash} \mathit{binds} \leadsto \textbf{binds}_V : (\{\}, GVE_V) \end{array}}{\begin{array}{ll} E \overset{\text{program}}{\vdash} & \mathit{typedecl}\,;\mathit{classdecl}\,;\mathit{instdecl}\,;\mathit{binds} \\ & \leadsto \\ & \textbf{binds}_C \texttt{ then } (\texttt{rec } \textbf{monobinds}_I) \texttt{ then } \textbf{binds}_V : E' \oplus GIE_I \oplus GVE_V \end{array}}$$

Figure 7: Rule for programs

- The notation

$$\texttt{\textbackslash}\langle \mathit{dicts} : \theta, \mathit{methods} : GVE \rangle \texttt{ -> } \textbf{exp}$$

  stands for a dictionary abstraction. The θ and GVE give the types for each variable bound by the abstraction.

  This construct is actually used only for the selector functions which take a dictionary apart, in which case **exp** is just a single variable or dictionary variable.

4.3.3 Abstraction over bindings

The syntax in Figure 6, and the shorthand notation of Section 4.3.1, give notation for type and dictionary abstractions over *expressions*. In fact, it turns out that we need to abstract over *sets of bindings* rather than over a single expression, so the expression form is never used directly. Instead the notation

$$\Lambda\alpha_1 \ldots \alpha_n \texttt{ -> \textbackslash} \mathit{dicts} : \theta \texttt{ -> } \textbf{binds} \texttt{ in } \textbf{monobinds}$$

stands for the abstraction of the type variables $\alpha_1 \ldots \alpha_n$ and the dictionaries *dicts* over the bindings in **binds** and **monobinds**. The resulting set of bindings bind only those variables bound in **monobinds**; the bindings in **binds** only scope over **monobinds**. This is rather a complicated construct, and it is discussed in some detail in Section 9.2.

5 Programs

WARNING: I HAVN'T LOOKED AT THIS SECTION FOR A WHILE, PENDING COMPLETION OF THE IMPORTS STUFF.

** Should add stuff to check for cycles in type synonyms and class declarataions.

The inference rule for programs is given in Figure 7. It has six premises:

Premise(1) validates the program's type declarations, and derives a type environment TE_T.

The T subscript identifies this environment as arising from type checking the type declarations. We use subscripts C for class declarations, I for instance declarations and V for value declarations.

Premise (2) deals with the class declarations, producing a some translated bindings, and three environments.

Premise (3) just combines together the environments derived so far into a total environment E'.

Premise (4) deals with instance declarations. Notice that they are all mutually recursive with each other and with the value declarations; this is specified by making GIE_I and GVE_V part of the environment for the judgement. To see that instance declarations can be mutually recursive among themselves, consider the program fragment:

```
class C1 a where
        f1,g1 :: a -> Int
class C2 a where
        f2,g2 :: a -> Int
instance C1 Int where
        f1 x = f2 x
        g1 x = 4
instance C2 Int where
        f2 x = g1 x
        g2 x = f1 x
```

Similarly, the GIE_I environment (obtained from typechecking the instance declarations) must clearly be in scope when typechecking the value declarations; and the GVE_V environment (obtained from typeckecking the value declarations) must be in scope when typechecking the instance declarations, because methods in an instance declarations may refer to values declared by a value declaration.

Premise(5) deals with the value declarations. Mutual recursion within value declarations is handled within the rules for *binds*. An empty *LIE* should be returned.

6 Type declarations

The rules for types and type declarations are given in Figures 8, 9 and 10.

The rules for *monotype*, *context*, and *polytype* look a bit trivial, but they formally specify the following:

- Checking that all uses of type constructors refer to type constructors which are in scope, which are specified by TE.

$$\boxed{TCE; TVE \overset{\text{monotype}}{\vdash} monotype : \tau}$$

$$\text{TYPE-TAUT} \quad \frac{TVE\ tyvar = \alpha}{TCE; TVE \overset{\text{monotype}}{\vdash} tyvar : \alpha}$$

$$\text{TYPE-FUN} \quad \frac{\begin{array}{c} TCE; TVE \overset{\text{monotype}}{\vdash} monotype_1 : \tau_1 \\ TCE; TVE \overset{\text{monotype}}{\vdash} monotype_2 : \tau_2 \end{array}}{TCE; TVE \overset{\text{monotype}}{\vdash} monotype_1 \texttt{ -> } monotype_2 : \tau_1 \rightarrow \tau_2}$$

$$\text{TYPE-LIST} \quad \frac{TCE; TVE \overset{\text{monotype}}{\vdash} monotype : \tau}{TCE; TVE \overset{\text{monotype}}{\vdash} \texttt{[}monotype\texttt{]} : [\tau]}$$

$$\text{TYPE-TUPLE} \quad \frac{TCE; TVE \overset{\text{monotype}}{\vdash} monotype_i : \tau_i \qquad (1 \leq i \leq n)}{TCE; TVE \overset{\text{monotype}}{\vdash} \texttt{(}monotype_1, \ldots, monotype_n\texttt{)} : (\tau_1, \ldots, \tau_n)}$$

$$\text{TYPE-CON} \quad \frac{\begin{array}{c} TCE\ tycon = \Lambda \alpha_1 \ldots \alpha_n . \tau \\ TCE; TVE \overset{\text{monotype}}{\vdash} monotype_i : \tau_i \qquad (1 \leq i \leq n) \end{array}}{TCE; TVE \overset{\text{monotype}}{\vdash} tycon\ monotype_1 \ldots monotype_n : \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}$$

Figure 8: Rules for *monotype*

- Ditto type variables, which are specified by TVE. In type signatures this is not significant, since all the type variables are implicitly universally quantified, but it is significant in `data` and `type` declarations, where the quantification is explicit.
- Ditto class names.
- Expanding type synonyms. (The expansion could contain info about the original synonym too; that is decided by the $\overset{\text{typedecl}}{\vdash}$ rule for `type`.)
- Checking that the type variables mentioned in the *context* part of a *polytype* are a subset of those mentioned in the *monotype* part (see rule POLYTYPE1).
- Checking that the context in a *polytype* predicates only type variables which are not bound in the enclosing scope, as defined by the TVE to the left of the turnstile. This enclosing TVE only contains bindings for type variables bound by a class or instance declaration.

Notice, for example, the use of an explicit environment TVE in the *polytype* judgement (Figure 9). It ensures that the quantification covers all the free variables of both the *context* and the *monotype*. Of course, it could contain some extra unused variables, but that does not matter.

$$\boxed{TVE; CE \overset{\text{context}}{\vdash} context : \theta}$$

$$\text{CONTEXT}\quad \frac{\begin{array}{ll} CE\ class_i = (\kappa_i, \_) & (1 \leq i \leq n) \\ TVE\ tyvar_i = \alpha_i & (1 \leq i \leq n) \end{array}}{TCE; TVE; CE \overset{\text{context}}{\vdash} \begin{array}{l} (class_1\ tyvar_1\ \texttt{,} \ldots \texttt{,} class_n\ tyvar_n) \\ : \\ \langle \kappa_1\ \alpha_1, \ldots, \kappa_n\ \alpha_n \rangle \end{array}}$$

$$\boxed{TCE; TVE; CE \overset{\text{polytype}}{\vdash} polytype : \sigma}$$

$$\text{POLYTYPE1}\quad \frac{\begin{array}{l} \{tyvar_1, \ldots, tyvar_n\} = fv(monotype) \setminus dom(TVE) \\ TVE' = \{tyvar_1 : \alpha_1, \ldots, tyvar_n : \alpha_n\} \\ TVE'; CE \overset{\text{context}}{\vdash} context : \theta \\ TCE; TVE' \oplus TVE \overset{\text{monotype}}{\vdash} monotype : \tau \end{array}}{TCE; TVE; CE \overset{\text{polytype}}{\vdash} context\ \texttt{=>}\ monotype : \forall \alpha_1 \ldots \alpha_n.\ \theta \Rightarrow \tau}$$

$$\text{POLYTYPE2}\quad \frac{\begin{array}{l} \{tyvar_1, \ldots, tyvar_n\} = fv(monotype) \setminus dom(TVE) \\ TVE' = \{tyvar_1 : \alpha_1, \ldots, tyvar_n : \alpha_n\} \\ TCE; TVE' \oplus TVE \overset{\text{monotype}}{\vdash} monotype : \tau \end{array}}{TCE; TVE; CE \overset{\text{polytype}}{\vdash} monotype : \forall \alpha_1 \ldots \alpha_n.\ \langle\rangle \Rightarrow \tau}$$

Figure 9: Rules for *context* and *polytype*

6.1 Contexts in `data` declarations

A *context* may appear just after the keyword `data` in a `data` declaration. Its effect is to make each constructor into an overloaded function, as you can see in the CONSTR rule. For example, consider

```
data Eq a => Set a = EmptySet | Singleton a | Two (Set a) (Set a)
```

The types of the constructors are as follows:

$$\begin{array}{lll} \texttt{EmptySet} & :: & \forall \alpha.\ \texttt{Set}\ \alpha \\ \texttt{Singleton} & :: & \forall \alpha.\ \texttt{Eq}\ \alpha \Rightarrow \alpha \rightarrow \texttt{Set}\ \alpha \\ \texttt{Two} & :: & \forall \alpha.\ \texttt{Eq}\ \alpha \Rightarrow \texttt{Set}\ \alpha \rightarrow \texttt{Set}\ \alpha \rightarrow \texttt{Set}\ \alpha \end{array}$$

Notice that `EmptySet` is not overloaded: only type variables free in the arguments of the constructor are predicated. The notation $\theta|_{\{\alpha_1 \ldots \alpha_m\}}$ denotes the subset of θ which predicates type variables in $\{\alpha_1 \ldots \alpha_m\}$. (The reason Haskell is designed like this is to avoid needless ambiguity errors.)

$$\boxed{TCE; CE \overset{\text{typedecl}}{\vdash} typedecl : (TE, CVE)}$$

$$\text{DATA-DECL}\quad \frac{\begin{array}{l} TVE = \{tyvar_1 : \alpha_1, \ldots, tyvar_n : \alpha_n\} \\ TCE; TVE; \theta; \chi\ \alpha_1 \ldots \alpha_n \overset{\text{condecl}}{\vdash} condecl : CVE \\ TVE; CE \overset{\text{context}}{\vdash} context : \theta \end{array}}{TCE; CE \overset{\text{typedecl}}{\vdash} \begin{array}{l} \texttt{data}\ context\ \texttt{=>}\ tycon\ tyvar_1 \ldots tyvar_n\ \texttt{=}\ condecl \\ : \\ (\{tycon : \Lambda\alpha_1 \ldots \alpha_n\ \texttt{->}\ \chi\ \alpha_1 \ldots \alpha_n\}, GVE) \end{array}}$$

$$\text{SYN-DECL}\quad \frac{TCE; \{tyvar_1 : \alpha_1, \ldots, tyvar_n : \alpha_n\} \overset{\text{monotype}}{\vdash} monotype : \tau}{TCE; CE \overset{\text{typedecl}}{\vdash} \begin{array}{l} \texttt{type}\ tycon\ tyvar_1 \ldots tyvar_n\ \texttt{=}\ monotype \\ : \\ (\{tycon : \Lambda\alpha_1 \ldots \alpha_n .\ \tau\}, \{\}) \end{array}}$$

$$\text{TYPE-DECLS}\quad \frac{\begin{array}{l} TCE; CE \overset{\text{typedecl}}{\vdash} typedecl_1 : (TCE_1, CVE_1) \\ TCE; CE \overset{\text{typedecl}}{\vdash} typedecl_2 : (TCE_2, CVE_2) \end{array}}{TCE; CE \overset{\text{typedecl}}{\vdash} typedecl_1 ; typedecl_2 : (TCE_1 \oplus TCE_2, CVE_1 \oplus CVE_2)}$$

$$\boxed{TCE; TVE; \theta; \tau \overset{\text{condecl}}{\vdash} condecl : CVE}$$

$$\text{CONSTR}\quad \frac{\begin{array}{l} TCE; TVE \overset{\text{monotype}}{\vdash} monotype_i : \tau_i \qquad (1 \leq i \leq n) \\ \alpha_1 \ldots \alpha_m = \bigcup_i fv(\tau_i) \\ \theta' = \theta|_{\{\alpha_1 \ldots \alpha_m\}} \end{array}}{TCE; TVE; \theta; \tau \overset{\text{condecl}}{\vdash} \begin{array}{l} con\ monotype_1 \ldots monotype_n \\ : \\ \{con : \forall\alpha_1 \ldots \alpha_m .\ \theta' \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau\} \end{array}}$$

$$\text{CONBIND}\quad \frac{\begin{array}{l} TCE; TVE; \theta; \tau \overset{\text{condecl}}{\vdash} condecl_1 : CVE_1 \\ TCE; TVE; \theta; \tau \overset{\text{condecl}}{\vdash} condecl_2 : CVE_2 \end{array}}{TCE; TVE; \theta; \tau \overset{\text{condecl}}{\vdash} condecl_1\ \texttt{|}\ condecl_2 : CVE_1 \oplus CVE_2}$$

Figure 10: Rules for type declarations

6.2 A note on notation

A note on notation: in the TYPE-TUPLE rule, the premise above the bar is short for n premises, with i ranging from 1 to n. The constraints on n are given earlier, in the abstract syntax; in this case n must be at least 2. Similarly in the TYPE-CON rule, i varies between 1 and n, but here the abstract syntax tells that $n \geq 0$. The constraint $1 \leq i \leq n$ does *not* imply that $n \geq 1$.

7 Class declarations

The rules for class declarations are shown in Figure 11.

Rule CLASSES is straightforward; observe that the bindings inferred from $classdecl_1$ are in scope for $classdecl_2$ but not vice versa. This ensures that the superclass hierachy cannot contain cycles. Class operation names cannot clash with each other or other in-scope names because of the use of $\oplus$ to combine environments.

Rule CLASS is the interesting one.

Premise (1) validates the context, returning its semantic form, θ, and checking that only the type variable *tyvar* is mentioned.

Premise (2) validates the class signature. This produces GVE, a value environment containing bindings for the class operations.

Notice that the type signatures in a `class` declaration can be *polytype*s. This allows declarations such as

```
class C a where
        f :: D b => a -> b -> a
```

The type variable `b` will be polymorphic and overloaded.

Premise (3) defines *methods*, a sequence of the names of all and only the class operations, obtained by taking the domain of GVE. The significance of *methods* is that it records the order in which the methods must appear in the dictionary. This order is arbitrary, but the dictionaries created by `instance` declarations must agree with the selectors defined by the `class` declaration.

A pragmatic note: in a system supporting separate compilation, there will need to be some canonical ordering of the methods.cd

Premise (4) is a backwards use of *zip* . The "input" is the context θ, and this premise invents new dictionary names *dicts*, and then zips them together with the context to produce a local instance environment LIE.

Conclusion. The translation of a class declaration is a set of bindings which bind the class operations to selector functions which extract the corresponding method from a dictionary for the class. Selectors are also generated to extract the dictionaries for superclasses.

$$\boxed{E \overset{\text{classdecl}}{\vdash} \mathit{classdecl} \leadsto \mathbf{binds} : (GVE, CE, GIE)}$$

$$\text{CLASSES}\quad \frac{\begin{array}{c} E \overset{\text{classdecl}}{\vdash} \mathit{classdecl}_1 \leadsto \mathbf{binds}_1 : (GVE_1, CE_1, GIE_1) \\ E \oplus (GVE_1, CE_1, GIE_1) \overset{\text{classdecl}}{\vdash} \mathit{classdecl}_2 \leadsto \mathbf{binds}_2 : (GVE_2, CE_2, GIE_2) \end{array}}{\begin{array}{ll} E \overset{\text{classdecl}}{\vdash} & \mathit{classdecl}_1 \ \texttt{then}\ \mathit{classdecl}_2 \\ & \leadsto \\ & \mathbf{binds}_1 \ \texttt{then}\ \mathbf{binds}_2 : (GVE_1 \oplus GVE_2, CE_1 \oplus CE_2, GIE_1 \oplus GIE_2) \end{array}}$$

$$\text{CLASS}\quad \frac{\begin{array}{ll} (1) & \{\mathit{tyvar} : \alpha\}; CE \ \mathit{of}\ E \overset{\text{context}}{\vdash} \mathit{context} : \theta \\ (2) & CE\ \mathit{of}\ E; TCE\ \mathit{of}\ E; \{\mathit{tyvar} : \alpha\} \overset{\text{sigs}}{\vdash} \mathit{sigs} : GVE \\ (3) & \mathit{methods} = \mathit{dom}(GVE) \\ (4) & LIE = \mathit{zip}\ \mathit{dicts}\ \theta \end{array}}{\begin{array}{ll} E \overset{\text{classdecl}}{\vdash} & \texttt{class}\ \mathit{context}\ \texttt{=>}\ \mathit{class}\ \mathit{tyvar}\ \texttt{where}\ \mathit{sigs} \\ & \leadsto \\ & \left\{ \begin{array}{l} \mathit{var}\ \texttt{=}\ \Lambda\alpha\ \texttt{->}\ \texttt{\\}\langle \mathit{dicts} : \theta, \mathit{methods} : GVE\rangle\ \texttt{->}\ \mathit{var} \\ \quad \mid \mathit{var} \in \mathit{dom}(LIE) \cup \mathit{dom}(GVE) \end{array} \right\} \\ & : \\ & (\forall\alpha.\ \langle\kappa\ \alpha\rangle \Rightarrow GVE, \\ & \{\mathit{class} : (\kappa, \forall\alpha(\theta, \mathit{methods} : GVE))\}, \\ & \forall\alpha.\ \langle\kappa\ \alpha\rangle \Rightarrow LIE) \end{array}}$$

$$\boxed{CE; TCE; TVE \overset{\text{sigs}}{\vdash} \mathit{sigs} : GVE}$$

$$\text{SIGS}\quad \frac{\begin{array}{c} CE; TCE; TVE \overset{\text{sigs}}{\vdash} \mathit{sigs}_1 : GVE_1 \\ CE; TCE; TVE \overset{\text{sigs}}{\vdash} \mathit{sigs}_2 : GVE_2 \end{array}}{CE; TCE; TVE \overset{\text{sigs}}{\vdash} \mathit{sigs}_1\ \texttt{and}\ \mathit{sigs}_2 : GVE_1 \oplus GVE_2}$$

$$\text{SIG}\quad \frac{TCE; TVE; CE \overset{\text{polytype}}{\vdash} \mathit{polytype} : \sigma}{TCE; TVE; CE \overset{\text{sigs}}{\vdash} \mathit{var} :: \mathit{polytype} : \{\mathit{var} : \sigma\})}$$

$$\text{SIG\_EMPTY}\quad \frac{}{CE; TCE; TVE \overset{\text{sigs}}{\vdash} \{\} : \{\}}$$

Figure 11: Rules for class declarations

$$\boxed{E \overset{\text{instdecl}}{\vdash} \mathit{instdecl} \leadsto \mathit{monobinds} : \mathit{GIE}}$$

$$\text{INSTS}\quad \frac{\begin{array}{c} E \overset{\text{instdecl}}{\vdash} \mathit{instdecl}_1 \leadsto \mathit{monobinds}_1 : \mathit{GIE}_1 \\ E \overset{\text{instdecl}}{\vdash} \mathit{instdecl}_2 \leadsto \mathit{monobinds}_2 : \mathit{GIE}_2 \end{array}}{E \overset{\text{instdecl}}{\vdash} \mathit{instdecl}_1 \;;\; \mathit{instdecl}_2 \leadsto \mathit{monobinds}_1 \;;\; \mathit{monobinds}_2 : \mathit{GIE}_1 \oplus \mathit{GIE}_2}$$

$$\text{INST}\quad \frac{\begin{array}{ll} (1) & (\mathit{CE}\ \mathit{of}\ E)\ \mathit{class} = (\kappa, \forall\alpha(\mathit{superdicts} : \theta_C, \mathit{methods} : \mathit{GVE}_C)) \\ (2) & \mathit{TVE} = \{\mathit{tyvar}_1 : \alpha_1, \ldots, \mathit{tyvar}_n : \alpha_n\} \\ (3) & \mathit{TCE}\ \mathit{of}\ E; \mathit{TVE} \overset{\text{monotype}}{\vdash} \mathit{monotype} : \tau \\ (4) & \tau = \chi\ \alpha_1 \ldots \alpha_n \qquad (\alpha_1, \ldots, \alpha_n \text{ distinct}) \\ (5) & \mathit{TVE}; \mathit{CE}\ \mathit{of}\ E \overset{\text{context}}{\vdash} \mathit{context} : \theta_I \\ (6) & \mathit{LIE}_I = \mathit{zip}\ \mathit{dicts}\ \theta_I \\ (7) & \mathit{GIE}\ \mathit{of}\ E; \mathit{LIE}_I \overset{\text{dicts}}{\vdash} \mathit{zip}\ \mathit{superdicts}\ (\theta_C[\tau/\alpha]) \leadsto \mathbf{binds}_{\mathit{super}} \\ (8) & E \oplus \mathit{TVE} \overset{\text{binds}}{\vdash} \mathit{binds} \leadsto \mathbf{binds}_{\mathit{methods}} : (\mathit{LIE}_I, \mathit{GVE}_C[\tau/\alpha]) \end{array}}{\begin{array}{l} E \overset{\text{instdecl}}{\vdash} \quad \texttt{instance}\ \mathit{context}\ \texttt{=>}\ \mathit{class}\ \mathit{monotype}\ \texttt{where}\ \mathit{monobinds} \\ \qquad \leadsto \\ \qquad \left\{ \mathbf{dfun}\ \texttt{=}\ \begin{array}{l} \Lambda\alpha_1 \ldots \alpha_n\ \texttt{->} \\ \texttt{\textbackslash}\mathit{dicts} : \theta_I\ \texttt{->} \\ \texttt{let}\ \mathbf{binds}_{\mathit{super}}\ \texttt{in} \\ \texttt{let}\ \mathbf{binds}_{\mathit{methods}}\ \texttt{in} \\ \langle \mathit{superdicts}, \mathit{methods} \rangle \end{array} \right\} \\ \qquad : \\ \qquad \{\mathbf{dfun} : \forall\alpha_1 \ldots \alpha_n.\ \theta_I \Rightarrow \kappa\ \tau\} \end{array}}$$

Figure 12: Instance declarations

The existence of these selectors is recorded in the environments returned: the class operation selectors appear in the value environment GVE, while the superclass selectors appear in the instance environment GIE.

8 Instance declarations

Figure 12 gives the rules for instance declarations.

The INSTS rule is straightforward. By contrast with CLASSES, the bindings inferred from $instdecl_1$ are not made in scope in $instdecl_2$. Instead, the top-level rule PROGRAM makes the entire collection of instance declarations mutually recursive.

The INST rule is the most complex single rule in the entire system. It has seven premises, which are listed here:

Premise (1) looks up the class *class* in the class environment, to get its semantic name κ

and its type $\forall\alpha(superdicts : \theta_C, methods : GVE_C)$.

Premise(2) defines a type-variable environment, with bindings for (exactly) the names of the free type variables of *monotype* in the instance declaration.

Premises (3) and (4) validate *monotype*. Premise (4) checks that the form of *monotype* is correct: it must be a type constructor applied to simple type variables.

Premise (5) validates *context*, checking that it mentions no type variables which are not free in *monotype* (which would not make sense).

Premise (6) builds a local instance environment LIE_I from the context θ, generating suitable new dictionary variables *dicts*.

Premise (7) confirms that the superclass dictionaries described by the τ instance of LIE_C can indeed be constructed from dictionaries provided by the instance declaration LIE_I. It witnesses this fact by producing a set of bindings which give the former in terms of the latter. The $\overset{\text{dicts}}{\vdash}$ judgement is discussed in Section 12.

Premise (8) type checks the value bindings in the body of the instance declaration, and verifies that they match the τ instance of GVE_C obtained from the class environment in Premise (1).

The conclusion gives as translation the function which takes dictionaries corresponding to the context θ, and delivers a dictionary for the class, complete with its superclass dictionaries and methods.

Note the type abstraction in the binding for the dictionary function **dfun**: it is polymorphic!

In the INST rule as given, notice that $\kappa\ \tau$ is not in the local instance environment LIE_I. This means that instance declarations such as

```
instance Eq a => Eq (Foo a) where
        (==) p q = ...
        (/=) p q = not (p == q)
```

where one method calls another, will generate a rather inefficient translation. The call to `(==)` in the definition of `(/=)` will result in a second call to the dictionary function for `Eq (Foo a)`. Instead it would have been possible to add **dvar** : $\kappa\ \tau$ to LIE_I, and produce the following translation for the instance declaration:

$$
\begin{array}{l}
\{\mathbf{dfun} = \Lambda\alpha_1 \ldots \alpha_n \texttt{ ->} \\
\qquad \backslash dicts : \theta_I \texttt{ ->} \\
\qquad \texttt{let rec} \\
\qquad\qquad \mathbf{dvar} = \texttt{let } \mathbf{binds}_{super} \texttt{ in} \\
\qquad\qquad\qquad\quad \texttt{let } \mathbf{binds}_{methods} \texttt{ in} \\
\qquad\qquad\qquad\quad \langle superdicts, methods \rangle \\
\qquad \texttt{in} \\
\qquad \mathbf{dvar}\ \}
\end{array}
$$

$$\boxed{\text{Sets of bindings: } E \overset{\text{monobinds}}{\vdash} \textit{monobinds} \leadsto \textbf{monobinds} : (\mathit{LIE}, \mathit{LVE})}$$

$$\text{UNION} \quad \frac{\begin{array}{c} E \overset{\text{monobinds}}{\vdash} \textit{monobinds}_1 \leadsto \textbf{monobinds}_1 : (\mathit{LIE}_1, \mathit{LVE}_1) \\ E \overset{\text{monobinds}}{\vdash} \textit{monobinds}_2 \leadsto \textbf{monobinds}_2 : (\mathit{LIE}_2, \mathit{LVE}_2) \end{array}}{\begin{array}{ll} E \overset{\text{monobinds}}{\vdash} & \textit{monobinds}_1 \,;\textit{monobinds}_2 \\ & \leadsto \\ & \textbf{monobinds}_1 \,; \textbf{monobinds}_2 : (\mathit{LIE}_1 \oplus \mathit{LIE}_2, \mathit{LVE}_1 \oplus \mathit{LVE}_2) \end{array}}$$

$$\text{PATBIND} \quad \frac{\begin{array}{c} \mathit{CVE} \textit{ of } E \overset{\text{pat}}{\vdash} \textit{pat} \leadsto \textbf{pat} : (\tau, \mathit{LVE}) \\ E \overset{\text{grhs}}{\vdash} \textit{grhs} \leadsto \textbf{grhs} : (\mathit{LIE}, \tau) \end{array}}{E \overset{\text{monobinds}}{\vdash} \textit{pat grhs} \leadsto \textbf{pat grhs} : (\mathit{LIE}, \mathit{LVE})}$$

$$\text{FUNBIND} \quad \frac{E \overset{\text{match}}{\vdash} \textit{match} \leadsto \textbf{match} : (\mathit{LIE}, \tau)}{E \overset{\text{monobinds}}{\vdash} \textit{var match} \leadsto \textit{var}\ \textbf{match} : (\mathit{LIE}, \{\textit{var} : \tau\})}$$

$$\text{EMPTYBIND} \quad \frac{}{E \overset{\text{monobinds}}{\vdash} \{\} \leadsto \{\} : (\{\}, \{\})}$$

Figure 13: Rules for *monobinds*

The declaration of the dictionary is made recursive, thus witnessing the addition of the extra assertion to the class declaration. This modification somewhat complicates the rule in exchange for a probable increase in efficiency.

9 Value declarations

The syntax and type rules for value declarations are structured in three layers:

- A *monobinds* is a group of declarations, typechecked with $\overset{\text{monobinds}}{\vdash}$ to give a monomorphic environment (Figure 13). The fact that the envrionment is monomorphic ensures that in the right-hand sides of mutually-recursive definitions the variables being defined have only a monomorphic and non-overloaded type. The judgement

  $$E \overset{\text{monobinds}}{\vdash} \textit{monobinds} \leadsto \textbf{monobinds} : (\mathit{LIE}, \mathit{LVE})$$

  should be read "in environment E, the declarations *monobinds* translate to the declarations **monobinds**, and the types of variables bound in those declarations are given by the type environment LVE, under the class constraints given by the local instance environment LIE". Since pattern bindings may bind zero, one or several variables, there may be more or fewer typings in LVE than declarations in **monobinds**. (Pattern bindings which bind no varibles are not very useful, but it is inconvenient to exclude them and they do no harm.)

  It is always the case that $\mathit{fv}(\mathit{LIE}) \subseteq \mathit{fv}(\mathit{LVE})$.

$$\boxed{\text{Binds: } E \overset{\text{binds}}{\vdash} \mathit{binds} \leadsto \mathbf{binds} : (LIE, GVE)}$$

$$\text{BINDS-TRIV}\quad \frac{E \overset{\text{bind}}{\vdash} \mathit{bind} \leadsto \mathbf{bind} : (LIE, GVE)}{E \overset{\text{binds}}{\vdash} \mathit{bind} \leadsto \mathbf{bind} : (LIE, GVE)}$$

$$\text{BINDS-THEN}\quad \frac{\begin{array}{l} E \overset{\text{binds}}{\vdash} \mathit{binds}_1 \leadsto \mathbf{binds}_1 : (LIE_1, GVE_1) \\ E \overset{\rightarrow}{\oplus} GVE_1 \overset{\text{binds}}{\vdash} \mathit{binds}_2 \leadsto \mathbf{binds}_2 : (LIE_2, GVE_2) \end{array}}{\begin{array}{ll} E \overset{\text{binds}}{\vdash} & \mathit{binds}_1 \ \texttt{then}\ \mathit{binds}_2 \\ & \leadsto \\ & \mathbf{binds}_1 \ \texttt{then}\ \mathbf{binds}_2 : (LIE_1 \oplus LIE_2, GVE_1 \oplus GVE_2) \end{array}}$$

$$\text{BINDS-SIG}\quad \frac{\begin{array}{l} E \overset{\text{bind}}{\vdash} \mathit{bind} \leadsto \mathbf{bind} : (LIE, GVE) \\ CE\ of\ E; TCE\ of\ E; TVE\ of\ E \overset{\text{sigs}}{\vdash} \mathit{sigs} \leadsto GVE : \end{array}}{E \overset{\text{bind}}{\vdash} \mathit{bind}\ \texttt{with}\ \mathit{sigs} \leadsto \mathbf{bind} : (LIE, GVE)}$$

Figure 14: Rules for *binds*

$$\boxed{\text{Bind: } E \overset{\text{bind}}{\vdash} \mathit{bind} \leadsto \mathbf{bind} : (LIE, GVE)}$$

$$\text{BIND-GEN}\quad \frac{\begin{array}{l} E \overset{\text{bind-1}}{\vdash} \mathit{bind} \leadsto \mathbf{bind} : (LIE, GVE) \\ \alpha_1 \ldots \alpha_n \notin VE\ of\ E \\ \alpha_1 \ldots \alpha_n \notin TVE\ of\ E \\ \alpha_1 \ldots \alpha_n \notin LIE \end{array}}{E \overset{\text{bind}}{\vdash} \mathit{bind} \leadsto \Lambda\alpha_1 \ldots \alpha_n\ \texttt{->}\ \mathbf{bind} : (LIE, \forall\alpha_1 \ldots \alpha_n.\ GVE)}$$

$$\text{BIND-PRED}\quad \frac{\begin{array}{l} E \overset{\text{bind-2}}{\vdash} \mathit{bind} \leadsto \mathbf{bind} : LIE, LVE) \\ GIE\ of\ E; (LIE_{encl} \oplus LIE_{bind}) \overset{\text{dicts}}{\vdash} LIE \leadsto \mathbf{monobinds} \\ LIE_{bind} = zip\ dicts\ \theta \\ \mathit{bind} \text{ contains only function bindings} \end{array}}{E \overset{\text{bind-1}}{\vdash} \mathit{bind} \leadsto \texttt{\\}\mathit{dicts} : \theta\ \texttt{->}\ \mathbf{monobinds}\ \texttt{in}\ \mathbf{bind} : (LIE_{encl}, \theta \Rightarrow LVE)}$$

$$\text{BIND-REC}\quad \frac{E \overset{\rightarrow}{\oplus} LVE \overset{\text{monobinds}}{\vdash} \mathit{monobinds} \leadsto \mathbf{monobinds} : (LIE, LVE)}{E \overset{\text{bind-2}}{\vdash} \texttt{rec}\ \mathit{monobinds} \leadsto \texttt{rec}\ \mathbf{monobinds} : (LIE, LVE)}$$

$$\text{BIND-TRIV}\quad \frac{E \overset{\text{monobinds}}{\vdash} \mathit{monobinds} \leadsto \mathbf{monobinds} : (LIE, LVE)}{E \overset{\text{bind-2}}{\vdash} \mathit{monobinds} \leadsto \mathbf{monobinds} : (LIE, LVE)}$$

Figure 15: Rules for *bind*

$$\boxed{\text{Type signatures: } E; GVE \overset{\text{sigs}}{\vdash} sigs \leadsto \mathbf{binds} : GVE}$$

$$\text{SIGS-EMPTY}\quad \frac{}{E; GVE \overset{\text{sigs}}{\vdash} \{\} \leadsto \{\} : \{\}}$$

$$\text{SIGS}\quad \frac{E; GVE \overset{\text{sigs}}{\vdash} sigs_1 \leadsto \mathbf{binds}_1 : GVE_1 \quad E; GVE \overset{\text{sigs}}{\vdash} sigs_2 \leadsto \mathbf{binds}_2 : GVE_2}{E; GVE \overset{\text{sigs}}{\vdash} sigs_1 \texttt{ and } sigs_2 \leadsto \mathbf{binds}_1 \texttt{ then } \mathbf{binds}_2 : GVE_1 \oplus GVE_2}$$

$$\text{SIG}\quad \frac{\begin{array}{l} TCE \textit{ of } E; TVE \textit{ of } E; CE \textit{ of } E \overset{\text{polytype}}{\vdash} polytype : \sigma \\ GVE\ var = \sigma \end{array}}{E; GVE \overset{\text{sigs}}{\vdash} var :: polytype \leadsto \{\} : \{var : \sigma\}}$$

$$\text{SIG1}\quad \frac{\begin{array}{ll} (1) & TCE \textit{ of } E; TVE \textit{ of } E; CE \textit{ of } E \overset{\text{polytype}}{\vdash} polytype : \forall \beta_1 \ldots \beta_m .\ \theta_{sig} \Rightarrow \tau_{sig} \\ (2) & GVE\ var = \forall \alpha_1 \ldots \alpha_n .\ \theta \Rightarrow \tau \\ (3) & \gamma_1, \ldots, \gamma_m \notin E \oplus GVE \\ (4) & \tau_{sig}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \\ (5) & LIE = zip\ dicts\ \theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \\ (6) & LIE_{sig} = zip\ dicts_{sig}\ \theta_{sig}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] \\ (7) & GIE \textit{ of } E; LIE_{sig} \overset{\text{dicts}}{\vdash} LIE \leadsto \mathbf{binds} \end{array}}{\begin{array}{ll} E; GVE \overset{\text{sigs}}{\vdash} & var :: polytype \\ & \leadsto \\ & \left\{ var \texttt{ = } \begin{array}{l} \Lambda \beta_1 \ldots \beta_m \texttt{ ->} \\ \backslash dicts_{sig} : LIE_{sig} \texttt{ ->} \\ \texttt{let } \mathbf{binds} \texttt{ in} \\ var\ \tau_1 \ldots \tau_n\ dicts \end{array} \right\} \\ & : \\ & \{var : \forall \beta_1 \ldots \beta_m .\ \theta_{sig} \Rightarrow \tau_{sig}\} \end{array}}$$

Figure 16: Type signatures

- A *bind* deals with recursion, by optionally adding a `rec` to the *monobinds*. The typing rules are given in Figure 15.

  Once recursion is thus dealt with, the monotyped environment can be overloaded (BIND-PRED) and generalised (BIND-GEN) to give a polymorphic environment GVE.

  BIND-PRED only applies to sets of declarations all of which are function bindings, or in which a simple variable is bound directly to a lambda abstraction. This constraint is the one required to implement Haskell's (rather unsatisfactory) "monomorphic pattern binding" rule. BIND-GEN will still be able to generalise over any type variables not constrained in the instance environment.

  There are several important operational issues here, which are discussed further below in Sections 9.1, 9.2 and 9.3.

- Finally, a *binds* is a nested set of *bind* groups, each of which can optionally have a group of type signatures attached to it, connected with `then`. The typing rules are in Figures 14 and 16.

  The `then` construct is used only to get the most general type, and not for scoping purposes. Hence $\oplus$ is used in conclusion of the BINDS-THEN rule, rather than $\vec{\oplus}$. We still use $\oplus$ in the second premise, because the declarations in $bind_1$ might shadow outer bindings in E.

  The BINDS-SIG rule is rather a cop-out. It simply says that the type environment for the *bind* should be the same as that specified by the *sigs*. This is perfectly correct, but the implementation is much more complicated than the rule implies. A type inference algorithm cannot clairvoyantly infer the (perhaps less than most-general) types for the *bind* specified by *sigs*. Rather, it will infer the most general types, and a separate process must then specialise the inferred types to the specified ones, modifying the translation appropriately.

9.1 Rule order

The three rules BIND-TRIV/BIND-REC, BIND-PRED and BIND-GEN must be applied in an appropriate order:

- First apply BIND-TRIV or BIND-REC, depending on whether the declarations are recursive or not.
- Next use BIND-PRED to discharge all the possible assumptions in the local instance environment.
- Finally use BIND-GEN to bind all free type variables.

The reason for this ordering is to maintain the form of types given in Figure 1. This rule order is indicated by giving numbers to the turnstiles, thus $\overset{\text{bind-1}}{\vdash}$, $\overset{\text{bind-2}}{\vdash}$.

9.2 Translating type and dictionary abstractions

The translations given in BIND-GEN and BIND-PRED involve new constructs in the translated syntax for type and multidictionary abstraction. These constructs can readily be translated into existing ones, but there is more than one way to do, as we will see in this section. By using a new construct for each, we avoid making a premature committment as to the appropriate translation.

We now explore the possible translations of the new constructs. We discuss only the multidictionary abstraction; the type abstraction is analogous but a little simpler. Our discussion here is by example only, but the generalisation is straightforward, if tedious.

Consider the mutually recursive definitions[2]:

```
rec {   f x = g (x==x) x;
        g b x = abs (f x)
}
```

Recall that `abs` is an operation of class `Num`; that `==` is an operation of class `Eq`. Recall also that `Eq` is a superclass for `Num`; that is, the class declaration for `Num` looks like this:

```
class (Eq a) => Num a where ...
```

The fact that `Eq` is a superclass of `Num` means that every `Num` dictionary contains an `Eq` dictionary. This is recorded in the global instance environment with the entry

$$\texttt{eqFromNum} : \forall\alpha.\ \texttt{Num}\ \alpha \Rightarrow \texttt{Eq}\ \alpha$$

which says that `eqFromNum` (the name is arbitrary) maps a `Num` dictionary to an `Eq` dictionary.

Returning the the example, after the recursive binding for `f` and `g` has been typechecked by $\overset{\text{monobinds}}{\vdash}$, the translated binding will be:

```
rec { f x = g ((==) de x x) x; g b x = abs dn (f x) }
```

The inferred local type environment will be $\{\texttt{f} : \alpha \to \alpha,\ \texttt{g} : \texttt{Bool} \to \alpha \to \alpha\}$ and the local instance environment will have the assumption $\{\texttt{dn} : \texttt{Num}\ \alpha, \texttt{de} : \texttt{Eq}\ \alpha\}$, where `dn` is the name of the (`Num`) dictionary passed to `abs`, and `de` is the name of the (`Eq`) dictionary passed to `==` (again, the dictionary names are arbitrary). Now, applying BIND-PRED and then BIND-GEN produces the binding:

```
\\a -> \dn -> {de = eqFromNum a dn} in
rec { f a x:a = g a ((==) a de x x) x;
      g a b:Bool x:a = abs a dn (f a x) }
```

and the type environment now has overloaded types for `f` and `g`, namely $\{\texttt{f} : \forall\alpha.\ \texttt{Num}\ \alpha \Rightarrow \alpha \to \alpha,\ \texttt{g} : \forall\alpha.\ \texttt{Num}\ \alpha \Rightarrow \texttt{Bool} \to \alpha \to \alpha\}$. (Because typewriter font does not include greek letters, we use `\\` for Λ and `a` for α.) Notice the type signatures on bound variables. Notice also that the `Eq` dictionary, `de`, is obtained by applying the global function `eqFromNum` to the `Num` dictionary `dn`.

[2] To keep the example short it is degenerate: `f` and `g` both fail to terminate.

What does this binding abstraction mean? It is a **binds** which gives values for both `f` and `g`, each of which is a function taking a type `a` and a dictionary `dn` as its arguments. If the bindings were not mutually recursive, and if there were no dictionary bindings (in this case the binding for `de`) to take into account this would be easy: just add `a` and `dn` as extra parameters to each definition.

In the case of mutual recursion, this is not possible, because the recursive calls do not pass this extra parameter. Furthermore, there is the question of what is to be done with the dictionary bindings. One alternative is to add the extra parameters to all the recursive calls, and to replicate the dictionary bindings in each right-hand side, thus (we omit the type signatures on arguments from now on, for brevity):

```
rec { f a dn x   = let {de = eqFromNum a dn} in  g a dn ((==) a de x x) x;
        g a dn b x = let {de = eqFromNum a dn} in  abs a dn (f a dn x)
      }
```

There are a number of problems with this translation. Firstly, there may be several dictionaries which need to be repeatedly passed rather than just one. Depending on the implementation this may impose a significant overhead. Secondly, the dictionary bindings are duplicated, perhaps increasing code size. In this case the binding for `de` can be dropped from the right-hand side of `g`, since it is not used, but this may not always be so. Thirdly, and perhaps most importantly, the expression `(eqFromNum a dn)` is recomputed at every recursion. This loses a property known as full laziness, and unfortunately the standard full laziness transformation will not recover it. Finally, it is an untidy transformation to express in the type rules because it involves a "second pass" over the translated expression to replace uses of `f` with `(f a dn)`, and similarly for `g`.

A second possible translation is this:

```
    { f a dn = let {de = eqFromNum a dn} in
               let rec {f x = g ((==) a de x x) x; g b x = abs a dn (f x)} in
               f;
      g a dn = let {de = eqFromNum a dn} in
               let rec {f x = g ((==) a de x x) x; g b x = abs a dn (f x)} in
               g
        }
```

This translation recovers full laziness, because `(eqFromNum a dn)` is only computed once, but at the cost of duplicating the bindings for `f` and `g` as well as those for the dictionaries! However, it is usually the case that of a mutually-recursive group of definitions, only one is used in the subsequent code. In the case of our example, perhaps only `f` is called subsequently. This would mean that the (top-most) definition of `g` could be dropped entirely, which eliminates the duplication. But this single-use property cannot be guaranteed.

A third possible translation, which does not duplicate code, is as follows:

```
    { tup a dn = let {de = eqFromNum a dn} in
                 let rec {f x = g ((==) a de x x) x; g b x = abs a dn (f x)} in
                 (f,g)
```

```
f a dn = case (tup a dn) of (f,g) -> f;
g a dn = case (tup a dn) of (f,g) -> g;
  }
```

It appears that this translation is somewhat less efficient than the previous one, because of the tupling and untupling. However, in the common case where only `f`, say, is used subsequently, the definition of `g` can be dropped; in this case, the definition of `tuple` can safely be substituted for its occurrence in the body of `f`, and after little simplification the second translation (given above) emerges. For this reason, this third alternative seems to be the best.

9.3 Choosing which type variables to generalise over

The rules as given are correct (I hope) regardless of which parts of the *LIE* are selected as LIE_{bind} by BIND-PRED, and which type variables are chosen for generalisation by BIND-GEN. Nevertheless, in a type-inferencer, a suitable set must be chosen, and this section discusses the issues.

9.3.1 Generalisation

Firstly, BIND-GEN will not choose any type variables which are free in *VE*; this is part of the BIND-GEN rule. We call these type variables the *constrained type variables.* Nor is there any point in BIND-GEN choosing any type variables which are not free in the *LVE* being generalised. So we conclude that BIND-GEN should select exactly the type variables which are free in *any of the types in LVE, and which are not constrained by the enclosing VE.* We call these the *target type variables.*

This seems innocuous enough, but consider the bindings

```
{ tup = (\x -> x,  \y -> y) }
then
{ (f,g) = tup }
```

Type inference for the first binding will derive the environment

$$\{\texttt{tup} : \forall\alpha,\beta.\ (\alpha \to \alpha,\ \beta \to \beta\}$$

Type inference for the second binding, prior to BIND-GEN will infer the environment

$$\{\texttt{f} : \alpha \to \alpha,\ \texttt{g} : \beta \to \beta\}$$

Now BIND-GEN will generalise over *both* α *and* β, giving the environment

$$\{\texttt{f} : \forall\alpha,\beta.\ \alpha \to \alpha,\ \texttt{g} : \forall\alpha,\beta.\ \beta \to \beta\}$$

Curious! Both `f` and `g` take two type arguments, though their types only use one of them. There is no real problem here (especially since type abstractions and applications vanish in most real implementations) but it is worth noting.

9.3.2 Predication

The situation with BIND-PRED is more interesting still. Firstly, there is not much point in BIND-PRED selecting any predicates in *LIE* which predicate constrained type variables. (Remember, a predicate in *LIE* can predicate more than one type variable; for example `Eq` (α, β).) Any predicates in *LIE* which predicate only constrained type variables can therefore be eliminated from consideration. Predicates affecting some constrained and some unconstrained type variables can be simplified using $\overset{\text{dicts}}{\vdash}$ until none remain with this property. The remaining predicates affect only unconstrained type variables.

Here's an example of this separation process:

```
f x y = ... where
             g z1 z2 = if (x>y) then z1 else z2
```

When the declaration of `g` is typechecked in an environment which gives `x` and `y` the type α, the (LIE, LVE) pair returned is

$$(\{\mathtt{d} : \mathtt{Ord}\ \alpha\}, \{\mathtt{g} : \beta \rightarrow \beta \rightarrow \beta\})$$

The translation of `g`'s body will mention the dictionary `d` when it makes the comparion of `x` and `y`. Since α is not free in the type of `g`, BIND-PRED should leave the dictionary constraint on `d` in the *LIE* it returns, for later resolution by the BIND-PRED which deals with the declaration of `f`.

9.3.3 Ambiguity

Finally, the situation may arise of predicates which affect type variables which are neither constrained nor target variables. These predicates are *ambiguous*.

Should really say more here.

9.4 Type signatures

Type signatures are attached to possibly-recursive groups of declarations, rather than to individual declarations, because the types given cannot be checked until overloading and generalisation has taken place, which must be done for a group as a whole.

Type signatures are validated by the $\overset{\text{sigs}}{\vdash}$ judgement, which uses the *GVE* for the declaration group to check that the type signatures attached to a declaration group mention only variables declared in the group, and with exactly the correct type (Figure 16).

If this were all it did, the judgement would need to return no result. While this is perfectly legitimate, it is no help at all to a type *inference* engine. It requires that the inference engine "clairvoyantly" derives a perhaps-less-than-fully-general type for the original binding, so that this inferred type precisely matches that given in the type signature.

This is implausible. What will happen in practice is that the most general type will be inferred, and the signature will specialise it. Notice that some translation may be required in this case. For example, suppose we infer that `f` has type $\forall\alpha.$ `Num` $\alpha \Rightarrow \alpha \rightarrow \alpha$. Then given the type signature `f::Int -> Int`, we need to make a new binding for `f` of the form

```
f = f intD
```

where `intD` is the `Num` dictionary for `Int`s. (Note: this is a *non-recursive* binding! The `f` being defined is distinct from, and a specialised form of, the `f` on the right-hand side.) This re-binds `f` to be a version specialised to `Int`.

For this reason, the $\overset{\text{sigs}}{\vdash}$ judgement form takes as returns a set of extra bindings, together with the modified GVE. The rule SIG1 deals with coercing a general type to a more specific one. It is considerably more compliated than the original, because in effect it incorporates DGEN, SPEC, DPRED and REL! It works like this:

Premise (1) validates the *polytype* given in the type signature.

Premise (2) checks that the variable being typed does indeed have a binding in the current group, and finds its inferred type.

Premise (3) grabs a fresh set of type variables.

Premise (4) checks that the most general instance of the given type is an instance of the inferred type. $\tau_1, \ldots, \tau_n$ are the types which instantiate the inferred type to the given type.

Premise (5) generates an LIE from the inferred context θ, inventing new dictionary names.

Premise (6) does the same for the given context θ_{sig}.

Premise (7) finally generates the new bindings which produce the dictionaries needed by (the instance of) θ from those made available by θ_{sig}.

10 Expressions

The rules for typechecking expressions are given in Figures 17, 18, 19, 20, 21 and 22. The expression judgement form is

$$E \overset{\text{exp}}{\vdash} exp \leadsto \mathbf{exp} : (LIE, \sigma)$$

which should be read "in environment E, the expression exp is translated to **exp** and has type σ, under the class constraints given by the local instance environment LIE". It is always the case that

$$fv(LIE) \subseteq fv(\sigma)$$

Some points to notice:

TAUT-LVAR and TAUT-GVAR apply only if there is a binding for the variable in the local or global value environments, so typing fails if the variable is not in scope.

$$\boxed{\text{Expressions: } E \overset{\text{exp}}{\vdash} exp \leadsto \mathbf{exp} : (LIE, \sigma)}$$

$$\text{TAUT-GVAR} \quad \frac{(GVE \textit{ of } E)\ var = \sigma}{E \overset{\text{exp}}{\vdash} var \leadsto var : (\{\}, \sigma)} \quad var \in dom(GVE \textit{ of } E)$$

$$\text{TAUT-LVAR} \quad \frac{(LVE \textit{ of } E)\ var = \tau}{E \overset{\text{exp}}{\vdash} var \leadsto var : (\{\}, \tau)} \quad var \in dom(LVE \textit{ of } E)$$

$$\text{TAUT-CON} \quad \frac{(CVE \textit{ of } E)\ con = \sigma}{E \overset{\text{exp}}{\vdash} con \leadsto con : (\{\}, \sigma)}$$

$$\text{TAUT-INTEGER} \quad \frac{}{E \overset{\text{exp}}{\vdash} integer \leadsto \texttt{fromInteger}\ \langle \mathbf{dvar} \rangle\ integer : (\{\mathbf{dvar} : \texttt{Num}\ \tau\}, \tau)}$$

$$\text{TAUT-FLOAT} \quad \frac{}{E \overset{\text{exp}}{\vdash} float \leadsto \texttt{fromRational}\ \langle \mathbf{dvar} \rangle\ float : (\{\mathbf{dvar} : \texttt{Fractional}\ \tau\}, \tau)}$$

$$\text{TAUT-CHAR} \quad \frac{}{E \overset{\text{exp}}{\vdash} char \leadsto char : (\{\}, \texttt{Char})}$$

$$\text{TAUT-STRING} \quad \frac{}{E \overset{\text{exp}}{\vdash} string \leadsto string : (\{\}, [\texttt{Char}])}$$

Figure 17: Basic rules for expressions

$$\text{SPEC} \quad \frac{E \overset{\text{exp}}{\vdash} var \leadsto \mathbf{exp} : (LIE, \forall \alpha_1 \ldots \alpha_k .\ \rho)}{E \overset{\text{exp}}{\vdash} var \leadsto \mathbf{exp}\ \tau_1 \ldots \tau_k : (LIE, \rho[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k])}$$

$$\text{REL} \quad \frac{\begin{array}{l} E \overset{\text{exp}}{\vdash} exp \leadsto \mathbf{exp} : (LIE, \theta \Rightarrow \tau) \\ LIE' = zip\ dicts\ \theta \end{array}}{E \overset{\text{exp}}{\vdash} exp \leadsto (\mathbf{exp}\ dicts) : (LIE \oplus LIE', \tau)}$$

Figure 18: Non-syntactic rules for expressions

TAUT-INTEGER and TAUT-FLOAT introduce overloaded numeric and fractional constants respectively. It is for this reason that they interact with the instance environment.

TAUT-STRING deals with literal strings. Notice that even the empty string `""` is given the type [`Char`]. This means that, though the empty string is implemented as the empty list, the expression `[1] ++ ""` will fail the type checker (in constrast to the LML compiler, for example, which passes it).

GEN and SPEC manipulate a group of type variables at once, rather than working one type variable at a time as is more conventional. There is no deep reason for this, except the analogy with PRED and REL.

REL. (REL is short for "Release".) Mutter mutter. Need to give some explanation here.

$$\text{EXPSIG}\quad \frac{\mathit{TCE}\ \mathit{of}\ E; \mathit{TVE}\ \mathit{of}\ E; \mathit{CE}\ \mathit{of}\ E \overset{\text{polytype}}{\vdash} \mathit{polytype} : \sigma}{E \overset{\text{exp}}{\vdash} (\mathit{exp}::\mathit{polytype}) \leadsto \mathbf{exp} : (\{\}, \sigma)}$$

where $E \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\{\}, \sigma)$

$$\text{LAMBDA}\quad \frac{E; \{\} \overset{\text{mrule}}{\vdash} \mathit{mrule} \leadsto \mathbf{mrule} : (\mathit{LIE}, \tau)}{E \overset{\text{exp}}{\vdash} \backslash\mathit{mrule} \leadsto \backslash\mathbf{mrule} : (\mathit{LIE}, \tau)}$$

$$\text{COMB}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}, \tau' \to \tau) \\ E \overset{\text{exp}}{\vdash} \mathit{exp}' \leadsto \mathbf{exp}' : (\mathit{LIE}', \tau') \end{array}}{E \overset{\text{exp}}{\vdash} (\mathit{exp}\ \mathit{exp}') \leadsto (\mathbf{exp}\ \mathbf{exp}') : (\mathit{LIE} \oplus \mathit{LIE}', \tau)}$$

$$\text{LET}\quad \frac{\begin{array}{c} E \overset{\text{binds}}{\vdash} \mathit{binds} \leadsto \mathbf{binds} : (\mathit{LIE}, \mathit{GVE}) \\ E \overset{\rightarrow}{\oplus} \mathit{GVE} \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}', \tau) \end{array}}{E \overset{\text{exp}}{\vdash} (\texttt{let}\ \mathit{binds}\ \texttt{in}\ \mathit{exp}) \leadsto (\texttt{let}\ \mathbf{binds}\ \texttt{in}\ \mathbf{exp}) : (\mathit{LIE} \oplus \mathit{LIE}', \tau)}$$

$$\text{CASE}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}, \tau) \\ E \overset{\text{match}}{\vdash} \mathit{match} \leadsto \mathbf{match} : (\mathit{LIE}', \tau \to \tau') \end{array}}{E \overset{\text{exp}}{\vdash} (\texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match}) \leadsto (\texttt{case}\ \mathbf{exp}\ \texttt{of}\ \mathbf{match}) : (\mathit{LIE} \oplus \mathit{LIE}', \tau')}$$

$$\text{IF}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} \mathit{exp}_1 \leadsto \mathbf{exp}_1 : (\mathit{LIE}_1, \texttt{Bool}) \\ E \overset{\text{exp}}{\vdash} \mathit{exp}_2 \leadsto \mathbf{exp}_2 : (\mathit{LIE}_2, \tau) \\ E \overset{\text{exp}}{\vdash} \mathit{exp}_3 \leadsto \mathbf{exp}_3 : (\mathit{LIE}_3, \tau) \end{array}}{E \overset{\text{exp}}{\vdash} \begin{array}{l} \texttt{if}\ \mathit{exp}_1\ \texttt{then}\ \mathit{exp}_2\ \texttt{else}\ \mathit{exp}_3 \\ \leadsto \\ \texttt{if}\ \mathbf{exp}_1\ \texttt{then}\ \mathbf{exp}_2\ \texttt{else}\ \mathbf{exp}_3 : (\mathit{LIE}_1 \oplus \mathit{LIE}_2 \oplus \mathit{LIE}_3, \tau) \end{array}}$$

$$\text{LISTCOMP}\quad \frac{\begin{array}{c} E \overset{\text{qual}}{\vdash} \mathit{quals} \leadsto \mathbf{quals} : (\mathit{LIE}, \mathit{LVE}) \\ E \overset{\rightarrow}{\oplus} \mathit{LVE} \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}', \tau) \end{array}}{E \overset{\text{exp}}{\vdash} [\mathit{exp} \,|\, \mathit{quals}] \leadsto [\mathbf{exp} \,|\, \mathbf{quals}] : (\mathit{LIE} \oplus \mathit{LIE}', [\tau])}$$

Figure 19: Rules for compound expressions

$$\text{LIST}\quad \frac{E \overset{\text{exp}}{\vdash} \mathit{exp}_i \leadsto \mathbf{exp}_i : (\mathit{LIE}_i, \tau) \qquad (0 \le i \le n)}{E \overset{\text{exp}}{\vdash} [\mathit{exp}_1, \ldots, \mathit{exp}_n] \leadsto [\mathbf{exp}_1, \ldots, \mathbf{exp}_n] : (\mathit{LIE}_1 \oplus \ldots \oplus \mathit{LIE}_n, [\tau])}$$

$$\text{TUPLE}\quad \frac{E \overset{\text{exp}}{\vdash} \mathit{exp}_i \leadsto \mathbf{exp}_i : (\mathit{LIE}_i, \tau_i) \qquad (1 \le i \le n)}{E \overset{\text{exp}}{\vdash} (\mathit{exp}_1, \ldots, \mathit{exp}_n) \leadsto (\mathbf{exp}_1, \ldots, \mathbf{exp}_n) : (\mathit{LIE}_1 \oplus \ldots \oplus \mathit{LIE}_n, (\tau_1, \ldots, \tau_n))}$$

Figure 20: Data structures in expressions

$$\text{ENUM-FROM}\quad \frac{E \overset{\text{exp}}{\vdash} exp \leadsto \mathbf{exp} : (LIE, \tau)}{E \overset{\text{exp}}{\vdash} \texttt{[}exp\texttt{..]} \leadsto (\texttt{enumFrom}\ \langle\mathbf{dvar}\rangle\ \mathbf{exp}) : (LIE \oplus \{\mathbf{dvar} : \texttt{Enum}\ \tau\}, [\tau])}$$

$$\text{ENUM-FROM-THEN}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} exp_1 \leadsto \mathbf{exp}_1 : (LIE_1, \tau) \\ E \overset{\text{exp}}{\vdash} exp_2 \leadsto \mathbf{exp}_2 : (LIE_2, \tau) \end{array}}{E \overset{\text{exp}}{\vdash} \begin{array}{l} \texttt{[}exp_1\texttt{,}exp_2\texttt{..]} \\ \leadsto \\ (\texttt{enumFromThen}\ \langle\mathbf{dvar}\rangle\ \mathbf{exp}_1\ \mathbf{exp}_2) \\ : \\ (LIE_1 \oplus LIE_2 \oplus \{\mathbf{dvar} : \texttt{Enum}\ \tau\}, [\tau]) \end{array}}$$

$$\text{ENUM-FROM-TO}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} exp_1 \leadsto \mathbf{exp}_1 : (LIE_1, \tau) \\ E \overset{\text{exp}}{\vdash} exp_2 \leadsto \mathbf{exp}_2 : (LIE_2, \tau) \end{array}}{E \overset{\text{exp}}{\vdash} \begin{array}{l} \texttt{[}exp_1\texttt{..}exp_2\texttt{]} \\ \leadsto \\ (\texttt{enumFromTo}\ \langle\mathbf{dvar}\rangle\ \mathbf{exp}_1\ \mathbf{exp}_2) \\ : \\ (LIE_1 \oplus LIE_2 \oplus \{\mathbf{dvar} : \texttt{Enum}\ \tau\}, [\tau]) \end{array}}$$

$$\text{ENUM-FROM-THEN-TO}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} exp_1 \leadsto \mathbf{exp}_1 : (LIE_1, \tau) \\ E \overset{\text{exp}}{\vdash} exp_2 \leadsto \mathbf{exp}_2 : (LIE_2, \tau) \\ E \overset{\text{exp}}{\vdash} exp_3 \leadsto \mathbf{exp}_3 : (LIE_3, \tau) \end{array}}{E \overset{\text{exp}}{\vdash} \begin{array}{l} \texttt{[}exp_1\texttt{,}exp_2\texttt{..}exp_3\texttt{]} \\ \leadsto \\ (\texttt{enumFromThenTo}\ \langle\mathbf{dvar}\rangle\ \mathbf{exp}_1\ \mathbf{exp}_2\ \mathbf{exp}_3) \\ : \\ (LIE_1 \oplus LIE_2 \oplus LIE_3 \oplus \{\mathbf{dvar} : \texttt{Enum}\ \tau\}, [\tau]) \end{array}}$$

Figure 21: Rules for enumerations expressions

EXPSIG handles type signatures on expressions. Like the SIG rule for bindings, this rule is rather unrealistic, because it requires that the inference engine derive a less-than-most-general type for the expression, so that it exactly matches the signature given.

A more useful rule is given in Figure 23.

List comprehensions and case expressions. Notice that bindings in both list comprehensions and case expressions are *monomorphic*. It is easy to see how to make those in list comprehensions polymorphic, by adding a GEN rule to $\overset{\text{qual}}{\vdash}$, but less easy to see how to do the same for case expressions.

QUALS. Note the use of $\vec{\oplus}$ in the conclusion of this rule, expressing the fact that that bindings within a list comprehension can shadow each other. For example `{xs | xs <- ys; xs <- xs}` is allowed.

$$\boxed{E \overset{\text{qual}}{\vdash} quals \rightsquigarrow \mathbf{quals} : (LIE, LVE)}$$

$$\text{QUALS} \quad \frac{\begin{array}{c} E \overset{\text{qual}}{\vdash} quals_1 \rightsquigarrow \mathbf{quals}_1 : (LIE, LVE_1) \\ E \overset{\rightarrow}{\oplus} LVE_1 \overset{\text{qual}}{\vdash} quals_2 \rightsquigarrow \mathbf{quals}_2 : (LIE_2, LVE_2) \end{array}}{E \overset{\text{qual}}{\vdash} quals_1\, \mathbf{,}\, quals_2 \rightsquigarrow \mathbf{quals}_1\, \mathbf{,}\, \mathbf{quals}_2 : (LIE_1 \oplus LIE_2, LVE_1 \oplus LVE_2)}$$

$$\text{FILTER} \quad \frac{E \overset{\text{exp}}{\vdash} exp \rightsquigarrow \mathbf{exp} : (LIE, \texttt{Bool})}{E \overset{\text{qual}}{\vdash} exp \rightsquigarrow \mathbf{exp} : (LIE, \{\})}$$

$$\text{GENERATOR} \quad \frac{\begin{array}{c} CVE\ of\ E \overset{\text{pat}}{\vdash} pat \rightsquigarrow \mathbf{pat} : (\tau, LVE) \\ E \overset{\text{exp}}{\vdash} exp \rightsquigarrow \mathbf{exp} : (LIE, [\tau]) \end{array}}{E \overset{\text{qual}}{\vdash} pat\texttt{<-}exp \rightsquigarrow \mathbf{pat}\texttt{<-}\mathbf{exp} : (LIE, LVE)}$$

Figure 22: Rules for *quals*

$$\text{EXPSIG1} \quad \frac{\begin{array}{ll} (1) & E \overset{\text{exp}}{\vdash} exp \rightsquigarrow \mathbf{exp} : (LIE, \tau) \\ (2) & \alpha_1 \ldots \alpha_n \notin E \\ (3) & TCE\ of\ E; TVE\ of\ E; CE\ of\ E \overset{\text{polytype}}{\vdash} polytype : \forall \beta_1 \ldots \beta_m\ \theta_{sig} \Rightarrow \tau_{sig} \\ (4) & \gamma_1 \ldots \gamma_m \notin E \cup LIE \cup \tau \\ (5) & \tau_{sig}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \\ (6) & LIE_{sig} = zip\ dicts_{sig}\ \theta_{sig}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] \\ (7) & GIE\ of\ E; LIE_{sig} \overset{\text{dicts}}{\vdash} LIE \rightsquigarrow \mathbf{binds} \end{array}}{E \overset{\text{exp}}{\vdash} exp\texttt{::}polytype \rightsquigarrow \texttt{let}\ \mathbf{binds}\ \texttt{in}\ \mathbf{exp} : (LIE_{sig}, \tau_{sig}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m])}$$

Figure 23: More realistic expression-signature rule

11 Pattern matching

Patterns are type-checked using the judgement form $\overset{\text{pat}}{\vdash}$, whose typing rules are given in Figure 24. The judgement form is written thus

$$CVE \overset{\text{pat}}{\vdash} pat \rightsquigarrow \mathbf{pat} : (\tau, LVE)$$

and should should be read: "in the constructor value environment CVE, the pattern *pat* is translated to **pat** with type τ, and the bindings of variables within it is given by LVE." The judgement guarantees that LVE contains bindings for all and only the variables of *pat*, and that the pattern is linear (ie contains no repeated variables). The translated pattern **pat** gives an explicit type to each variable, which is necessary in the second-order lambda calculus.

The $\overset{\text{mrule}}{\vdash}$ judgement type-checks *mrules*, which are sequences of patterns followed by a sequence of guarded expressions. *mrules* are used by both function bindings and lambda ab-

$$\boxed{\text{Patterns: } CVE \overset{\text{pat}}{\vdash} pat \leadsto \mathbf{pat} : (\tau, LVE)}$$

TAUT-PAT
$$\frac{}{CVE \overset{\text{pat}}{\vdash} var \leadsto (var : \tau) : (\tau, \{var : \tau\})}$$

CONSTR-PAT
$$\frac{\begin{array}{l} CVE\ con = \sigma \\ \sigma \leq \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \\ CVE \overset{\text{pat}}{\vdash} pat_i \leadsto \mathbf{pat}_i : (\tau_i, LVE_i) \qquad (1 \leq i \leq n) \end{array}}{CVE \overset{\text{pat}}{\vdash} \begin{array}{l} con\ pat_1 \ldots pat_n \\ \leadsto \\ con\ \mathbf{pat}_1 \ldots \mathbf{pat}_n : (\tau, LVE_1 \oplus \ldots \oplus LVE_n) \end{array}}$$

AS-PAT
$$\frac{CVE \overset{\text{pat}}{\vdash} pat \leadsto \mathbf{pat} : (\tau, LVE)}{CVE \overset{\text{pat}}{\vdash} var\texttt{@}pat \leadsto var\texttt{@}\mathbf{pat} : (\tau, LVE \oplus \{var : \tau\})}$$

LAZY-PAT
$$\frac{CVE \overset{\text{pat}}{\vdash} pat \leadsto \mathbf{pat} : (\tau, LVE)}{CVE \overset{\text{pat}}{\vdash} \texttt{\~{}}pat \leadsto \texttt{\~{}}\mathbf{pat} : (\tau, LVE)}$$

WILD-PAT
$$\frac{}{CVE \overset{\text{pat}}{\vdash} \_ \leadsto \_ : (\tau, \{\})}$$

LIST-PAT
$$\frac{CVE \overset{\text{pat}}{\vdash} pat_i \leadsto \mathbf{pat}_i : (\tau, LVE_i) \qquad (0 \leq i \leq n)}{CVE \overset{\text{pat}}{\vdash} \texttt{[}pat_1\texttt{,} \ldots \texttt{,}pat_n\texttt{]} \leadsto \texttt{[}\mathbf{pat}_1\texttt{,} \ldots \texttt{,}\mathbf{pat}_n\texttt{]} : ([\tau], LVE_1 \oplus \ldots \oplus LVE_n)}$$

TUPLE-PAT
$$\frac{CVE \overset{\text{pat}}{\vdash} pat_i \leadsto \mathbf{pat}_i : (\tau_i, LVE_i) \qquad (1 \leq i \leq n)}{CVE \overset{\text{pat}}{\vdash} \begin{array}{l} \texttt{(}pat_1\texttt{,} \ldots \texttt{,}pat_n\texttt{)} \\ \leadsto \\ \texttt{(}\mathbf{pat}_1\texttt{,} \ldots \texttt{,}\mathbf{pat}_n\texttt{)} : ((\tau_1, \ldots, \tau_n), LVE_1 \oplus \ldots \oplus LVE_n) \end{array}}$$

Figure 24: Rules for patterns

stractions.

Finally, the $\overset{\text{grhs}}{\vdash}$ judgement type-checks a sequence of guarded expressions. The typing rules for this and the *mrule* judgement form are given in Figure 25.

The following points are worth noting:

Linearity is ensured by the use of $\oplus$ in AS-PAT, LIST-PAT, TUPLE-PAT and CONSTR-PAT.

MRULE-ABS and MRULE-GRHS. The $\overset{\text{mrule}}{\vdash}$ judgement carries down a monomorphic value environment LVE, which contains bindings for all the variables bound by the patterns in the current sequence of patterns. This environment is initialised empty by the MATCH rule, and augmented by the MRULE-ABS rule using a the non-shadowing $\oplus$ operation, thus checking for linearity among the patterns.

$$\boxed{\text{Matches: } E \overset{\text{match}}{\vdash} \mathit{match} \leadsto \mathbf{match} : (\mathit{LIE}, \tau)}$$

$$\text{MATCHES}\quad \frac{\begin{array}{c} E \overset{\text{match}}{\vdash} \mathit{match}_1 \leadsto \mathbf{match}_1 : (\mathit{LIE}_1, \tau) \\ E \overset{\text{match}}{\vdash} \mathit{match}_2 \leadsto \mathbf{match}_2 : (\mathit{LIE}_2, \tau) \end{array}}{E \overset{\text{match}}{\vdash} \mathit{match}_1 \,[]\, \mathit{match}_2 \leadsto \mathbf{match}_1 \,[]\, \mathbf{match}_2 : (\mathit{LIE}_1 \oplus \mathit{LIE}_2, \tau)}$$

$$\text{MATCH}\quad \frac{E; \{\} \overset{\text{mrule}}{\vdash} \mathit{mrule} \leadsto \mathbf{mrule} : (\mathit{LIE}, \tau)}{E \overset{\text{match}}{\vdash} \mathit{mrule} \leadsto \mathbf{mrule} : (\mathit{LIE}, \tau)}$$

$$\boxed{\text{Mrules: } E; \mathit{LVE} \overset{\text{mrule}}{\vdash} \mathit{mrule} \leadsto \mathbf{mrule} : (\mathit{LIE}, \tau)}$$

$$\text{MRULE-GRHS}\quad \frac{E \overset{\rightarrow}{\oplus} \mathit{LVE} \overset{\text{grhs}}{\vdash} \mathit{grhs} \leadsto \mathbf{grhs} : (\mathit{LIE}, \tau)}{E; \mathit{LVE} \overset{\text{mrule}}{\vdash} \mathit{grhs} \leadsto \mathbf{grhs} : (\mathit{LIE}, \tau)}$$

$$\text{MRULE-ABS}\quad \frac{\begin{array}{c} \mathit{CVE\ of\ E} \overset{\text{pat}}{\vdash} \mathit{pat} \leadsto \mathbf{pat} : (\tau, \mathit{LVE}') \\ E; \mathit{LVE} \oplus \mathit{LVE}' \overset{\text{mrule}}{\vdash} \mathit{mrule} \leadsto \mathbf{mrule} : (\mathit{LIE}, \tau') \end{array}}{E; \mathit{LVE} \overset{\text{mrule}}{\vdash} \mathit{pat\ mrule} \leadsto \mathbf{pat\ mrule} : (\mathit{LIE}, \tau \to \tau')}$$

$$\boxed{\text{Guarded right-hand sides: } E \overset{\text{grhs}}{\vdash} \mathit{grhs} \leadsto \mathbf{grhs} : (\mathit{LIE}, \tau)}$$

$$\text{GUARD-OR}\quad \frac{\begin{array}{c} E \overset{\text{grhs}}{\vdash} \mathit{grhs}_1 \leadsto \mathbf{grhs}_1 : (\mathit{LIE}_1, \tau) \\ E \overset{\text{grhs}}{\vdash} \mathit{grhs}_2 \leadsto \mathbf{grhs}_2 : (\mathit{LIE}_2, \tau) \end{array}}{E \overset{\text{grhs}}{\vdash} \mathit{grhs}_1 \,[]\, \mathit{grhs}_2 \leadsto \mathbf{grhs}_1 \,[]\, \mathbf{grhs}_2 : (\mathit{LIE}_1 \oplus \mathit{LIE}_2, \tau)}$$

$$\text{GUARD}\quad \frac{\begin{array}{c} E \overset{\text{exp}}{\vdash} \mathit{guard} \leadsto \mathbf{guard} : (\mathit{LIE}_1, \texttt{Bool}) \\ E \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}_2, \tau) \end{array}}{E \overset{\text{grhs}}{\vdash} \texttt{|}\mathit{guard}\texttt{=}\mathit{exp} \leadsto \texttt{|}\mathbf{guard}\texttt{=}\mathbf{exp} : (\mathit{LIE}_1 \oplus \mathit{LIE}_2, \tau)}$$

$$\text{GUARD-DEFAULT}\quad \frac{E \overset{\text{exp}}{\vdash} \mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}, \tau)}{E \overset{\text{grhs}}{\vdash} \texttt{=}\mathit{exp} \leadsto \mathbf{exp} : (\mathit{LIE}, \tau)}$$

Figure 25: Rules for *match*, *mrule* and *grhs*

$$\boxed{\text{Dictionary equivalence: } \mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \mathit{LIE}' \rightsquigarrow \mathbf{binds}}$$

$$\text{DICT-TRIV} \quad \frac{}{\mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \{\} \rightsquigarrow \{\}}$$

$$\text{DICTS} \quad \frac{\begin{array}{c} \mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \mathit{LIE}_1 \rightsquigarrow \mathbf{binds}_1 \\ \mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \mathit{LIE}_2 \rightsquigarrow \mathbf{binds}_2 \end{array}}{\mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \mathit{LIE}_1 \oplus \mathit{LIE}_2 \rightsquigarrow \mathbf{binds}_1 \ \mathtt{then}\ \mathbf{binds}_2}$$

$$\text{DICT-GLOBAL} \quad \frac{\begin{array}{l} \mathit{GIE}\ \mathbf{dfun} = \forall \alpha_1 \ldots \alpha_n .\ \theta \Rightarrow \kappa\ \tau \\ \mathit{LIE}' = \mathit{zip}\ \mathit{dicts}\ \theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \\ \mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \mathit{LIE}' \rightsquigarrow \mathbf{binds} \end{array}}{\begin{array}{ll} \mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} & \{\mathbf{dvar} : \kappa\ \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\} \\ & \rightsquigarrow \\ & \mathbf{binds}\ \mathtt{then}\ \{\mathbf{dvar}\ \mathtt{=}\ \mathbf{dfun}\ \tau_1 \ldots \tau_n\ \mathit{dicts}\} \end{array}}$$

$$\text{DICT-DUP} \quad \frac{\mathit{LIE}\ \mathbf{dvar}' = \kappa\ \tau}{\mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \{\mathbf{dvar} : \kappa\ \tau\} \rightsquigarrow \{\mathbf{dvar}\ \mathtt{=}\ \mathbf{dvar}'\}}$$

Figure 26: Dictionary manipulation

Finally, MRULE-GRHS typechecks the guarded right-hand side in a type environment *LVE* formed by combining the environment derived from the patterns to the main one *E*.

12 Dictionary manipulation

The $\overset{\text{dicts}}{\vdash}$ judgement form tells how to express some dictionaries in terms of others. It takes the form

$$\mathit{GIE}; \mathit{LIE} \overset{\text{dicts}}{\vdash} \mathit{LIE}' \rightsquigarrow \mathbf{binds}$$

Here, LIE' is a local instance environment containing assumptions of form $\mathbf{dvar} : \kappa\ \tau$, and this judgement shows how to translate LIE' to a collection of bindings **monobinds** which give values to the dictionaries mentioned in LIE' in terms of those mentioned in *GIE* and *LIE*.

There are four main places $\overset{\text{dicts}}{\vdash}$ is used:

- In the INST rule, it shows how to construct dictionaries for the superclasses in terms of those provided by the context in the instance declaration.
- Many judgements return a local instance environment *LIE* as part of their result, which tracks the names of all the dictionaries used in the translation of the construct, together with their types. These *LIE*s are merged together up the expression tree, thereby

producing a merged *LIE*. The latter may be highly redundant, in the sense that some dictionaries can be produced from others, and the $\stackrel{\text{dicts}}{\vdash}$ judgement allows this redundancy to be eliminated.

- Type signatures on declarations.
- Type signatures on expressions.

There are three ways in which $\stackrel{\text{dicts}}{\vdash}$ can give a binding for a dictionary **dvar** with type $\kappa\ \tau$:

- If *LIE* contains a dictionary with identical type, a trivial binding will suffice (DICTS-DUP).
- If τ is of form $\chi\ \tau_1 \ldots \tau_n$ then there should have been an instance declaration for the χ instance of κ. In this case, *GIE* will contain a matching assumption, which is used in the DICTS-GLOBAL rule to translate **dvar** : $\kappa\ \tau$ to a binding and a simpler collection of *LIE* assumptions.
- Finally, if τ is just a type variable α, it may nevertheless be that there is another assumption in *LIE* of form $\mathbf{dvar}'$ :: $\kappa'\ \alpha$ where κ is a superclass of κ'. In this case it is possible to express **dvar** in terms of $\mathbf{dvar}'$. The DICTS-GLOBAL rule copes with this too, but it in implementation terms this case is the trickiest to handle, because it involves conditionally using the superclass hierachy.

13 Implementation notes

When generalising (DGEN) it is sufficient (I think) to look at *LVE*, and not *GVE*, because while *GVE* may contain free type variables, they are only free because they are also free in *LVE* (otherwise they would have been $\forall$-bound).

14 References

R Harper, R Milner & M Tofte [May 1989], "The definition of Standard ML (Version 3)," ECS-LFCS-89-81, LFCS, Dept of Computer Science, University of Edinburgh.

P Hudak, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, SL Peyton Jones, M Reeve, D Wise & J Young [April 1990], "Report on the functional programming language Haskell," Department of Computing Science, Glasgow University.

P Wadler & S Blott [Jan 1989], "How to make ad-hoc polymorphism less ad hoc," in *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, ACM.

Index