A static semantics for Haskell

Simon L. Peyton Jones
Philip Wadler
Department of Computing Science, University of Glasgow, G12 8QQ
(email: {simonp}, 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

1 Introduction 4

2 A sketch of how overloading is resolved 6

3 Notation 7

4 Abstract syntax 13

5 Programs 19

6 Type declarations 21

7 Class declarations 24

8 Instance declarations 26

9 Value declarations 28

10 Expressions 37
11 Pattern matching

12 Dictionary manipulation

13 Implementation notes

14 References

List of Figures

1 Syntax of semantic types ........................................ 10
2 Environments ...................................................... 11
3 Abstract syntax for Haskell: declarations and bindings ............. 14
4 Abstract syntax of Haskell: patterns and expressions ............... 15
5 Abstract syntax of Haskell: types ................................ 16
6 Extra syntax for translated programs ................................ 16
7 Rule for programs .................................................. 20
8 Rules for monotype .................................................. 21
9 Rules for context and polytype .................................... 22
10 Rules for type declarations ........................................ 23
11 Rules for class declarations ...................................... 25
12 Instance declarations ............................................. 27
13 Rules for monobinds ................................................. 29
14 Rules for binds .................................................... 30
15 Rules for bind ...................................................... 31
16 Type signatures .................................................... 32
17 Basic rules for expressions ....................................... 38
18 Non-syntactic rules for expressions ................................ 38
19 Rules for compound expressions .................................. 39
20 Data structures in expressions .................................... 40
21 Rules for enumerations expressions ................................ 41
22 Rules for qals ....................................................... 42
23 More realistic expression-signature rule ................................ 42
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 19). 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 ∨ 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 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 those 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:

\[ f :: (\text{Ord} \ a, \text{Num} \ a) \Rightarrow a \rightarrow a \rightarrow a \]

\[ f \ x \ y = \text{if} \ (x>y) \ \text{then} \ x+y \ \text{else} \ x \]

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

The translation of \(f\) looks like this:

\[ f \ \text{dord} \ dnum \ x \ y = \text{if} \ ((>) \ \text{dord} \ x \ y) \ \text{then} \ ((+) \ \text{dnum} \ x \ y) \ \text{else} \ x \]

\(f\) takes two extra dictionary arguments \text{dord} and \text{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 \(f\):

\[ g :: \text{Int} \rightarrow \text{Int} \]

\[ g \ x = f \ x \ x \]

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

\[ g \ x = f \ \text{ordInt} \ \text{numIntD} \ x \ x \]

Two extra parameters are supplied to \(f\), namely the \text{Ord} dictionary and the \text{Num} dictionary for the \text{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:

```haskell
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

```haskell
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:

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

we translate to the following

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

3 Notation

This section summarises our notational conventions.

3.1 Sequences

The notation

```latex
\langle x_1, \ldots, x_n \rangle
```
stands an ordered sequence of the identifiers $x_1, \ldots, x_n$. Sequences can be concatenated with $\cdot$.

### 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:

\[ \text{inherited} \quad \text{nonterminal} \quad \text{construct} : \text{synthesised} \]

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:

\[ \text{environment} \quad \text{nonterminal} \quad \text{construct} : \text{type} \]

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:

\[ \text{inherited} \quad \text{nonterminal} \quad \text{construct} \rightarrow \text{construct} : \text{synthesised} \]

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 italics for source language nonterminals (nonterminal), and bold face 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 $\tau$. The correspondence between the two is given by this table:
Monotypes $\tau \rightarrow \alpha$

| $\chi \tau_1 \ldots \tau$ $(k \geq 0)$ | Type variables |
| $\tau_1 \rightarrow \tau_2$ | Structuredtype |
| $\tau$ | Functiontype |
| $(\tau_1, \ldots, \tau_k)$ $(k = 0 \text{ or } k \geq 2)$ | Listtype |
| Bool | Tupletype |
| Boolean | Boolean type |

Polytypes $\sigma \rightarrow \tau$

| $\forall \alpha_1 \ldots \alpha_k, \sigma$ | Universal quantification |
| $\theta \Rightarrow \sigma$ | Predication |

Contexts $\theta \rightarrow \langle \kappa_l, \tau_1, \ldots, \kappa_n, \tau_n \rangle$

Figure 1: Syntax of semantic types

<table>
<thead>
<tr>
<th>Syntactic form</th>
<th>Semantic form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type variable</td>
<td>tyvar</td>
</tr>
<tr>
<td>Monotype</td>
<td>monotype</td>
</tr>
<tr>
<td>Overloaded type</td>
<td>(none)</td>
</tr>
<tr>
<td>Polytpe</td>
<td>polytype</td>
</tr>
<tr>
<td>Type constructor</td>
<td>tycon</td>
</tr>
<tr>
<td>Class</td>
<td>class</td>
</tr>
<tr>
<td>Context</td>
<td>context</td>
</tr>
</tbody>
</table>

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, $\tau$, 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 $\chi$. 
<table>
<thead>
<tr>
<th>Environment</th>
<th>Notation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total environment</td>
<td>$E$</td>
<td>$(TCE, TVE, VE, CE, IE)$</td>
</tr>
<tr>
<td>Value environment</td>
<td>$VE$</td>
<td>$(CVE, GVE, LVE)$</td>
</tr>
<tr>
<td>Constructor value environment</td>
<td>$CVE$</td>
<td>$con \mapsto \sigma$</td>
</tr>
<tr>
<td>Global value environment</td>
<td>$GVE$</td>
<td>$var \mapsto \sigma$</td>
</tr>
<tr>
<td>Local value environment</td>
<td>$LVE$</td>
<td>$var \mapsto \tau$</td>
</tr>
<tr>
<td>Type-constructor environment</td>
<td>$TCE$</td>
<td>$tycon \mapsto \Lambda \alpha_1, ..., \alpha_n, \tau$</td>
</tr>
<tr>
<td>Type-variable environment</td>
<td>$TVE$</td>
<td>$tyvar \mapsto \alpha$</td>
</tr>
<tr>
<td>Class environment</td>
<td>$CE$</td>
<td>$class \mapsto (\kappa, \forall \alpha(\theta, \text{methods : GVE}))$</td>
</tr>
<tr>
<td>Instance environment</td>
<td>$IE$</td>
<td>$(GIE, LIE)$</td>
</tr>
<tr>
<td>Global instance environment</td>
<td>$GIE$</td>
<td>$dfun \mapsto \forall \alpha_1, ..., \alpha_n, \theta \mapsto \kappa \tau$</td>
</tr>
<tr>
<td>Local instance environment</td>
<td>$LIE$</td>
<td>$dvar \mapsto \kappa \tau$</td>
</tr>
</tbody>
</table>

Figure 2: Environments

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

```cpp
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 `fu()` 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: $GVE_1 = \{\text{var}_1 : \sigma_1, \text{var}_2 : \sigma_2, \ldots, \text{var}_n : \sigma_n\}$. An empty environment is denoted $\{\}$.  


The domain and range of an environment may be extracted using the functions \( \text{dom}() \) and \( \text{ran}() \) respectively. Thus \( \text{dom}(GVE_2) = \{ \text{var}_1, \text{var}_2 \} \) and \( \text{ran}(GVE_2) = \{ \sigma_1, \sigma_2 \} \).

An environment is applied to a member of its domain to return the corresponding element of the range. For example, \( GVE_1 \var_1 = \sigma_1 \).

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:

\[
\begin{align*}
(\forall \alpha. \ GVE) \ var &= \forall \alpha. (GVE \ var) \\
(\theta \rightarrow GVE) \ var &= \theta \rightarrow (GVE \ var)
\end{align*}
\]

The operations \( \oplus \) and \( \ominus \) 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:

\[
\begin{align*}
(GVE_1 \oplus GVE_2) \ var &= \begin{cases} 
GVE_1 \ var & \text{if } \var \in \text{dom}(GVE_1) \text{ and } \var \notin \text{dom}(GVE_2) \\
GVE_2 \ var & \text{if } \var \notin \text{dom}(GVE_1) \text{ and } \var \notin \text{dom}(GVE_2)
\end{cases} \\
(GVE_1 \ominus GVE_2) \ var &= \begin{cases} 
GVE_2 \ var & \text{if } \var \in \text{dom}(GVE_2) \text{ and } \var \notin \text{dom}(GVE_1) \\
GVE_1 \ var & \text{if } \var \in \text{dom}(GVE_1) 
\end{cases}
\end{align*}
\]

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 \( E \ominus CE \) 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 \( \text{dom}(E_1) \cap \text{dom}(E_2) = \emptyset \) is implied.
- Similarly, if \( E \ var \) appears in a rule, then the side condition \( \var \in \text{dom}(E) \) is implied.

### 3.4.2 Value environments

Value environments are quite conventional, mapping variable and constructor names to types. We distinguish the local value environment, 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 \texttt{type} and those introduced by \texttt{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 \( \chi \) 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

\[ (\kappa, \forall \alpha, \theta, \text{methods} : \text{GVE}) \]

Here \( \kappa \) is the semantic name of the class and \( \alpha \) is the type variable with respect to which overloading takes place. \( \theta \) specifies its superclasses, in the order they must appear in a dictionary for the class. Finally, \( \text{GVE} \) gives the type signature of the class, while `methods` is a sequence of the variable names in `dom`(`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:

\[ \forall \alpha_1 \ldots \alpha_n. \theta \Rightarrow \kappa \left( \chi \beta_1 \ldots \beta_m \right) \]

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

It is important that each type constructor \( \chi \) is an instance of a given class \( \kappa \) 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 $\kappa$ and same type constructor $\chi$. 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 $(\kappa, \chi)$ pairs rather
than the dictionary function names.

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

3.4.7 Local instance environment

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

The function $\text{zip}$ constructs a local instance environment from a sequence of dictionary names
dicts and a context $\theta$, thus

$$\{d_1 : \kappa_1 \tau_1, \ldots, d_n : \kappa_n \tau_n\} - \text{zip} (\text{dvar}_1, \ldots, \text{dvar}_n) (\kappa_1 \tau_1, \ldots, \kappa_n \tau_n)$$

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 paren-
theses 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 \text{typedcl; classdecl; instdecl; binds} &  
| typedcl \rightarrow \text{typedcl}_1 ; \text{typedcl}_2 & \text{Type declarations} 
| \quad \mid \text{data context} => \text{tycon tyvar}_1 \ldots \text{tyvar}_k 
| \quad \quad = \text{condecl} \quad (k \geq 0) & 
| \quad \mid \text{type tycon tyvar}_1 \ldots \text{tyvar}_k 
| \quad \quad = \text{monotype} \quad (k \geq 0) & 
| condecl \rightarrow \text{condecl}_1 \mid \text{condecl}_2 & \text{Constructor bindings} 
| \quad \mid \text{con monotype}_1 \ldots \text{monotype}_k \quad (k \geq 0) & 
| classdecl \rightarrow \text{classdecl}_1, \text{then classdecl}_2 & \text{Class declarations} 
| \quad \mid \text{class context} => \text{class tyvar} 
| \quad \quad \quad \text{where sigs} & 
| instdecl \rightarrow \text{instdecl}_1 ; \text{instdecl}_2 & \text{Instance declarations} 
| \quad \mid \text{instance context} => \text{class monotype} 
| \quad \quad \text{where binds} & 
| binds \rightarrow \text{bind} & \text{Nested declarations} 
| \quad \mid \text{bind with sigs} & 
| \quad \mid \text{binds}_1 \text{ then binds}_2 & 
| \text{sigs} \rightarrow \{\} & \text{Type signatures} 
| \quad \mid \text{sigs}_1 \text{ and sigs}_2 & 
| \quad \mid \text{var} :: \text{polyscale} & 
| \text{bind} \rightarrow \text{monobinds} & \text{Set of bindings} 
| \quad \mid \text{rec monobinds} & \text{Recursive set of bindings} 
| \text{monobinds} \rightarrow \text{monobinds}_1, \text{; monobinds}_2 & \text{A set of bindings} 
| \quad \mid \text{pat grhs} & \text{A pattern binding} 
| \quad \mid \text{var match} & \text{A function binding} 
| \quad \mid \{\} & \text{Empty set of bindings} 

Figure 3: Abstract syntax for Haskell: declarations and bindings
match → match₁ [] match₂
    | mrule
    | Case alternatives

mrule → grhs
    | pat mrule
    | Guarded right-hand side

grhs → grhs₁ [] grhs₂
    | 1 guard = exp
    | = exp

guard → exp

pat → var
    | -
    | con pat₁ ... patₙ
    | "pat
    | var@pat
    | [pat₁, ..., patₙ] (n ≥ 0)
    | (pat₁, ..., patₙ) (n ≥ 2)
    | ()
    | Variable
    | Wildcard
    | Compound pattern
    | Lazy pattern
    | As-pattern
    | List pattern
    | Tuple pattern
    | Unit pattern

exp → var
    | con
    | integer
    | float
    | char
    | string
    | \ mrule
    | exp exp'
    | case exp of match
    | if exp₁ then exp₂ else exp₃
    | [exp|quals]
    | [exp..]
    | [exp₁, exp₂..]
    | [exp₁..exp₃]
    | [exp₁, exp₂..exp₃]
    | let binds in exp
    | [exp₁, ..., expₙ] (n ≥ 0)
    | (exp₁, ..., expₙ) (n = 0 or n ≥ 2)
    | Lambda abstraction
    | Application
    | Case expression
    | Conditional
    | List comprehension
    | Enumerations
    | Local definitions
    | Qualifiers
    | Generator
    | Filter

Figure 4: Abstract syntax of Haskell: patterns and expressions
\[
\text{polytype} \rightarrow \text{context} \Rightarrow \text{monotype} \\
| \text{monotype}
\]

\[
\text{monotype} \rightarrow \text{tyvar} \\
| \text{monotype}_1 \rightarrow \text{monotype}_2 \\
| [\text{monotype}] \\
| (\text{monotype}_1, \ldots, \text{monotype}_n) \quad (n = 0 \text{ or } n \geq 2) \\
| \text{tycon} \text{ monotype}_1 \ldots \text{monotype}_n \quad (n \geq 0) \\
\]

\[
\text{context} \rightarrow (\text{class}_1, \text{tyvar}_1, \ldots, \text{class}_n, \text{tyvar}_n)
\]

Figure 5: Abstract syntax of Haskell: types

\[
\text{pat} \rightarrow \text{var} : \tau \\
| \text{var} : \kappa \tau \\
| \ldots \text{other productions as before}
\]

\[
\text{exp} \rightarrow \Lambda \alpha_1 \ldots \alpha_n \rightarrow \text{exp} \\
| \text{exp} \tau_1 \ldots \tau_n \\
| \ldots \text{other productions as before}
\]

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 \texttt{binds} are (topologically) sorted into nested groups, combined with \texttt{then}. Each group is then either a non-recursive group \texttt{monobinds} or a recursive group \texttt{rec monobinds}. Declarations within a \texttt{monobinds} are combined together with \texttt{;}

The \texttt{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 \texttt{let} instead of \texttt{where} in the production for expressions as a reminder that bindings have this structure.

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

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

\textbf{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 bod-
ies. 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 produc-
tions for them.

Notice also that the punctuation generated by the abstract syntax does not conform precisely
to the concrete syntax, because the same nonterminal <br>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 elimi-
nated.

There are some extra productions to handle the second-order lambda calculus 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 κ × r.

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 commitments 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:

\[
\exp \ \dicts
\]

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:

\[
\\lambda \dicts : \theta \to \exp
\]

stands for a lambda abstraction of the dictionaries in \( \dicts \), each of which has the type given by the corresponding element of \( \theta \).

4.3.2 Building dictionaries and taking them apart

Recall that a dictionary for a particular instance of a class \( \kappa \) is a value which contains

- the methods for each operation of \( \kappa \) at that instance;
- and dictionaries for the superclasses of \( \kappa \).

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

- The notation:

\[
\{ \dicts, \text{methods} \}
\]

where \( \dicts \) is a sequence of dictionary names and \text{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.
The notation

\( \setminus (\text{dicts} : \theta, \text{methods} : \GVE) \rightarrow \exp \)

stands for a dictionary abstraction. The \( \theta \) 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

\( A\alpha_1 \ldots \alpha_n \rightarrow \setminus \text{dicts} : \theta \rightarrow \text{binds in monobinds} \)

stands for the abstraction of the type variables \( \alpha_1 \ldots \alpha_n \) and the dictionaries \( \text{dicts} \) over the bindings in \( \text{binds} \) and \( \text{monobinds} \). The resulting set of bindings bind only those variables bound in \( \text{monobinds} \); the bindings in \( \text{binds} \) only scope over \( \text{monobinds} \). 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 declarations.

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 $TE_\gamma$.

The $\gamma$ 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:

```haskell
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 type checking the instance declarations) must clearly be in scope when type checking the value declarations; and the $GVE_V$ environment (obtained from type checking the value declarations) must be in scope when type checking 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 $TE$. 

20
### 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 \(\models{\text{ty-syn}}\) 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 POLYTYPE\text{E1}).

- 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.
$$\text{TVE; CE} \vdash \text{context : } \theta$$

| CONTEXT |
|------------------|------------------|------------------|
| \( CE \) class\(_i\) = (\( \kappa_i \), \( \_ \)) \( (1 \leq i \leq n) \) |
| \( TVE \) tyvar\(_i\) = \( \alpha_i \) \( (1 \leq i \leq n) \) |

$$\text{TCE; TVE; CE} \vdash (\text{class\(_1\) tyvar\(_1\), \ldots, class\(_n\) tyvar\(_n\)})$$

$$\vdash \langle \kappa_1 \alpha_1, \ldots, \kappa_n \alpha_n \rangle$$

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

$$\{\text{tyvar}\(_1\), \ldots, \text{tyvar}\(_n\)\} = \text{fv(\text{monotype})} \setminus \text{dom(TVE)}$$

$$\text{TVE'} = \{\text{tyvar}\(_1\) : \alpha_1, \ldots, \text{tyvar}\(_n\) : \alpha_n\}$$

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

$$\text{TCE; TVE' + TVE} \vdash \text{monotype : } \tau$$

$$\text{POLYTYPE1}$$

$$\text{TCE; TVE; CE} \vdash \text{context} \Rightarrow \text{monotype : } \forall \alpha_1 \ldots \alpha_n. \theta \Rightarrow \tau$$

$$\{\text{tyvar}\(_1\), \ldots, \text{tyvar}\(_n\)\} = \text{fv(\text{monotype})} \setminus \text{dom(TVE)}$$

$$\text{TVE'} = \{\text{tyvar}\(_1\) : \alpha_1, \ldots, \text{tyvar}\(_n\) : \alpha_n\}$$

$$\text{TCE; TVE' + TVE} \vdash \text{monotype : } \tau$$

$$\text{POLYTYPE2}$$

$$\text{TCE; TVE; CE} \vdash \text{monotype : } \forall \alpha_1 \ldots \alpha_n. \langle \_ \rangle \Rightarrow \tau$$

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:

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

The types of the constructors are as follows:

- **EmptySet** :: \( \forall \alpha. \text{Set } \alpha \)
- **Singleton** :: \( \forall \alpha. \text{Eq } \alpha \Rightarrow \alpha \rightarrow \text{Set } \alpha \)
- **Two** :: \( \forall \alpha. \text{Eq } \alpha \Rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha \rightarrow \text{Set } \alpha \)

Notice that EmptySet is not overloaded: only type variables free in the arguments of the constructor are predicative. The notation \( \delta|_{\alpha_1, \ldots, \alpha_n} \) denotes the subset of \( \delta \) which predicates type variables in \( \{\alpha_1 \ldots \alpha_n\} \). The reason Haskell is designed like this is to avoid needless ambiguity errors.)
\[ TCE; CE \vdash \text{typedecl} : (TE, CVE) \]

\[
TVE = \{\text{tyvar}_1 : \alpha_1, \ldots, \text{tyvar}_n : \alpha_n\}
\]

\[
TCE; TVE; \theta; \chi \alpha_1 \ldots \alpha_n \vdash \text{condecl} : CVE
\]

\[
TVE; CE \vdash \text{context} : \theta
\]

**DATA-DECL**

\[
TCE; CE \vdash \text{data context} \Rightarrow \text{tycon tyvar}_1 \ldots \text{tyvar}_n = \text{condecl} : \\
\{\{\text{tycon} : \Lambda \alpha_1 \ldots \alpha_n \Rightarrow \chi \alpha_1 \ldots \alpha_n\}, GVE\}
\]

**SYN-DECL**

\[
TCE; CE \vdash \text{type tycon tyvar}_1 \ldots \text{tyvar}_n = \text{monotype} : \tau
\]

\[
\{\{\text{tycon} : \Lambda \alpha_1 \ldots \alpha_n, \tau\}, \{\}\}
\]

\[
TCE; CE \vdash \text{condecl}_1 : (TCE_1, CVE_1)
\]

\[
TCE; CE \vdash \text{condecl}_2 : (TCE_2, CVE_2)
\]

\[
TCE; CE \vdash \text{condecl}_1; \text{condecl}_2 : (TCE_1 \oplus TCE_2, CVE_1 \oplus CVE_2)
\]

**TYPE-DECLS**

\[
TCE; TVE; \theta; \tau \vdash \text{condecl} : CVE
\]

\[
TVE; TVE; \theta; \tau \vdash \text{monotype}_i : \tau_i \quad (1 \leq i \leq n)
\]

\[
\alpha_1 \ldots \alpha_m = \bigcup_i \text{fr}(\tau_i)
\]

\[
\theta' = \theta_{\{\alpha_1 \ldots \alpha_m\}}
\]

**CONSTR**

\[
TCE; TVE; \theta; \tau \vdash \text{con monotype}_1 \ldots \text{monotype}_n : \\
\{\text{con} : \forall \alpha_1 \ldots \alpha_m, \theta' \Rightarrow \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau\}
\]

**CONBIND**

\[
TCE; TVE; \theta; \tau \vdash \text{condecl}_1 : CVE_1
\]

\[
TCE; TVE; \theta; \tau \vdash \text{condecl}_2 : CVE_2
\]

\[
TCE; TVE; \theta; \tau \vdash \text{condecl}_1; \text{condecl}_2 : CVE_1 \oplus CVE_2
\]

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 \( n \geq i \).

7 Class declarations

The rules for class declarations are shown in Figure 11.

Rule CLASSES is straightforward; observe that the bindings inferred from classdecl\(_i\) are in scope for classdecl\(_{i+1}\) but not vice versa. This ensures that the superclass hierarchy 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, \( \theta \), and checking that only the type variable \( tvar \) 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 polytypes. This allows declarations such as

\[
\text{class } C \text{ a where}
\]

\[
f :: D \ b \Rightarrow a \rightarrow b \rightarrow a
\]

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.cn.

**Premise (4)** is a backwards use of \( \text{zip} \). The “input” is the context \( \theta \), and this premise invents new dictionary names \( \text{dicts} \), and then zips them together with the context to produce a local instance environment \( IIE \).

**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.
\[
E \vdash \text{classdecl} \leadsto \text{binds} : (\text{GVE}, \text{CE}, \text{GIE})
\]

### CLASSES

\[
E \vdash \text{classdecl}_1 \leadsto \text{binds}_1 : (\text{GVE}_1, \text{CE}_1, \text{GIE}_1)
\]

\[
E \oplus (\text{GVE}_1, \text{CE}_1, \text{GIE}_1) \vdash \text{classdecl}_2 \leadsto \text{binds}_2 : (\text{GVE}_2, \text{CE}_2, \text{GIE}_2)
\]

\[
E \vdash \text{classdecl}_1 \text{ then classdecl}_2 \leadsto \text{binds}_1 \text{ then binds}_2 : (\text{GVE}_1 \oplus \text{GVE}_2, \text{CE}_1 \oplus \text{CE}_2, \text{GIE}_1 \oplus \text{GIE}_2)
\]

### CLASS

\[
E \vdash \text{class context} \Rightarrow \text{class tyvar where signs} \leadsto
\]

\[
\{\text{var} = \Delta_{\alpha} \rightarrow \{\text{dicts} : \theta, \text{methods} : \text{GVE}\} \rightarrow \text{var}\}
\]

\[
\{\text{var} \in \text{dom}(\text{LIE}) \cup \text{dom}(\text{GVE})\}
\]

\[
(\forall \alpha. (\alpha \Rightarrow \text{GVE}),
\{\text{class} : (\alpha, \forall \alpha(\theta, \text{methods} : \text{GVE}))\},
\forall \alpha. (\alpha \Rightarrow \text{LIE})
\]

### SIGS

\[
\text{CE}; \text{TCE}; \text{TVE} \vdash \text{signs} : \text{GVE}
\]

\[
\text{CE}; \text{TCE}; \text{TVE} \vdash \text{signs}_1 : \text{GVE}_1
\]

\[
\text{CE}; \text{TCE}; \text{TVE} \vdash \text{signs}_2 : \text{GVE}_2
\]

\[
\text{CE}; \text{TCE}; \text{TVE} \vdash \text{signs}_1 \text{ and signs}_2 : \text{GVE}_1 \oplus \text{GVE}_2
\]

\[
\text{TCE}; \text{TVE}; \text{CE} \vdash \text{polytype} : \sigma
\]

\[
\text{TCE}; \text{TVE}; \text{CE} \vdash \text{var} : \text{polytype} : \{\text{var} : \sigma\}
\]

\[
\text{CE}; \text{TCE}; \text{TVE} \vdash \{\} : \{\}
\]

Figure 11: Rules for class 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 $\text{instdecl}_2$ are not made in scope in $\text{instdecl}_1$. 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 $\text{class}$ in the class environment, to get its semantic name \( \kappa \)
and its type $\forall \alpha (\text{superdicts : } \theta_C, \text{methods : } CVE_C)$.

**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 $\theta$, generating suitable new dictionary variables $dicts$.

**Premise (7)** confirms that the superclass dictionaries described by the $\tau$ 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 $\tau$ 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 $\tau$ instance of $CVE_C$ obtained from the class environment in Premise (1).

The conclusion gives as translation the function which takes dictionaries corresponding to the context $\theta$, 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 $\text{dfun}$: 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

$$\text{instance Eq a => Eq (Foo a) where}
\begin{align*}
(\approx) & \ p \ q = \ldots \\
(\neq) & \ p \ q = \text{not} \ (p \neq q)
\end{align*}$$

where one method calls another, will generate a rather inefficient translation. The call to $\approx$ in the definition of $\neq$ will result in a second call to the dictionary function for Eq $\langle \text{Foo a} \rangle$. Instead it would have been possible to add $\text{dvar} : \kappa \tau$ to $LIE_I$, and produce the following translation for the instance declaration:

$$\langle \text{dfun = } A\alpha_1 \ldots \alpha_n \rightarrow \\
\didts : \theta_I \rightarrow \\
\text{let rac = let } \text{binds}_\text{super} \text{ in } \\
\text{let } \text{binds}_\text{method} \text{ in } \\
\langle \text{superdicts, methods} \rangle \\
\text{in } \\
\text{dvar } \rangle$$

27
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 \textit{monobinds} is a group of declarations, typechecked with \( \vdash \) to give a monomorphic environment (Figure 13). The fact that the environment 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

\[
E \monobinds \vdash \monobinds \sim \monobinds : (LIE, LVE)
\]

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 variables are not very useful, but it is inconvenient to exclude them and they do no harm.)

It is always the case that \( \mathit{fe}(LIE) \subseteq \mathit{fe}(LVE) \).
Binds: $E \vdash binds \leadsto binds : (LIE, GVE)$

\[
\begin{align*}
\text{BINDS-TRIV} & \quad \frac{E \vdash bind \leadsto bind : (LIE, GVE)}{E \vdash binds \leadsto binds : (LIE, GVE)} \\
\text{BINDS-THEN} & \quad \frac{E \vdash binds_1 \leadsto binds_1 : (LIE_1, GVE_1) \\ E \oplus GVE_1 \vdash binds_2 \leadsto binds_2 : (LIE_2, GVE_2)}{E \vdash binds_1 \text{ then } binds_2 \leadsto binds_1 \text{ then } binds_2 : (LIE_1 \oplus LIE_2, GVE_1 \oplus GVE_2)} \\
\text{BINDS-SIG} & \quad \frac{CE \text{ of } E; TCE \text{ of } E; \text{TV} \text{E of } E \vdash \text{sigs} \leadsto GVE :}{E \vdash bind \text{ with } \text{sigs} \leadsto bind : (LIE, GVE)}
\end{align*}
\]

Figure 14: Rules for binds

Bind: $E \vdash bind \leadsto bind : (LIE, GVE)$

\[
\begin{align*}
\text{BIND-GEN} & \quad \frac{E \vdash bind \leadsto bind : (LIE, GVE) \\ \alpha_1 \ldots \alpha_n \notin VE \text{ of } E \\ \alpha_1 \ldots \alpha_n \notin TV \text{E of } E \\ \alpha_1 \ldots \alpha_n \notin LIE}{E \vdash \Lambda \alpha_1 \ldots \alpha_n \rightarrow bind : (LIE, \forall \alpha_1 \ldots \alpha_n, GVE)} \\
\text{BIND-PRED} & \quad \frac{E \vdash bind \leadsto bind : LIE, LVE) \\ GIE \text{ of } E; (LIE_{encl} \oplus LIE_{bind}) \vdash LIE \leadsto \text{monobinds}}{E \vdash \text{bind contains only function bindings}} \\
\text{BIND-REC} & \quad \frac{E \oplus LVE \vdash \text{monobinds} \leadsto \text{monobinds} : (LIE, LVE) \\ E \vdash \text{rec monobinds} \leadsto \text{rec monobinds} : (LIE, LVE)}{E \vdash \text{monobinds} \leadsto \text{monobinds} : (LIE, LVE)} \\
\text{BIND-TRIV} & \quad \frac{E \vdash \text{monobinds} \leadsto \text{monobinds} : (LIE, LVE)}{E \vdash \text{monobinds} \leadsto \text{monobinds} : (LIE, LVE)}
\end{align*}
\]

Figure 15: Rules for bind
Type signatures: $E; GVE \vdash \text{signs} \leadsto \text{binds} : GVE$

**SIGS-EMPTY**

$E; GVE \vdash \text{signs} \leadsto \{ \} : \{ \}$

**SIGS**

$E; GVE \vdash \text{signs}_1 \leadsto \text{binds}_1 : GVE_1; E; GVE \vdash \text{signs}_2 \leadsto \text{binds}_2 : GVE_2$

$E; GVE \vdash \text{signs}_1 \text{ and signs}_2 \leadsto \text{binds}_1 \text{ then binds}_2 : GVE_1 \oplus GVE_2$

**SIG**

$E; GVE \vdash \var : \text{polytype} \leadsto \{ \} : \{ \var : \sigma \}$

1. TCE of $E$; TVE of $E$; CE of $E$ \text{ polytype} $\vdash \var \rightarrow \text{polytype} : \forall \beta_1 \ldots \beta_m \cdot \theta_{\text{sig}} \Rightarrow \tau_{\text{sig}}$
2. $GVE \var = \forall \alpha_1 \ldots \alpha_n, \theta \Rightarrow \tau$
3. $\gamma_1, \ldots, \gamma_m \notin E \oplus GVE$
4. $\tau_{\text{sig}}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] = \tau_1[\alpha_1/\gamma_1, \ldots, \alpha_n/\gamma_n]$
5. $\text{LIE} = \text{zip} \ \text{dicts} \ \theta [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$
6. $\text{LIE}_{\text{sig}} = \text{zip} \ \text{dicts}_{\text{sig}} \ \theta_{\text{sig}} [\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m]$
7. GIE of $E$; $\text{LIE}_{\text{sig}} \vdash \text{LIE} \leadsto \text{binds}$

**SIG1**

$E; GVE \vdash \var : \text{polytype}$

$\vdash$

\begin{align*}
\{ \var : \forall \beta_1 \ldots \beta_m, \theta_{\text{sig}} \Rightarrow \tau_{\text{sig}} \}
\end{align*}

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 monotypes 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 **bind** 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 $\otimes$. We still use $\otimes$ in the second premise, because the declarations in **bind** match 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 $\vdash^1 \cdots \vdash^3$.
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 so, as we will see in this section. By using a new construct for each, we avoid making a premature commitment 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:

\[
\text{rec } \{ \begin{array}{l}
    f \ x = g \ (x=x) \ x;
    g \ b \ x = \text{abs} \ (f \ x);
\end{array} \}
\]

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:

\[
\text{class } (\text{Eq } a) \Rightarrow \text{Num } a \text{ where } \ldots
\]

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

\[
\text{eqFromNum } : \forall a. \text{Num } a \rightarrow \text{Eq } a
\]

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

Returning to the example, after the recursive binding for \(f\) and \(g\) has been typechecked by

\[
\vdash
\]

the translated binding will be:

\[
\text{rec } \{ f \ x - g \ ((\rightarrow) \ de \ x \ x) \ x; g \ b \ x - \text{abs} \ dn \ (f \ x) \}
\]

The inferred local type environment will be \(\{f : \alpha \rightarrow \alpha, g : \text{Bool } \rightarrow \alpha \rightarrow \alpha\}\) and the local instance environment will have the assumption \(\text{dn : Num } \alpha, \text{de : Eq } \alpha\)\). where \(\text{dn}\) is the name of the (Num) dictionary passed to abs, and \(\text{de}\) 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:

\[
\\{ a \rightarrow \text{dn } \rightarrow \{ \text{de } = \text{eqFromNum } a \text{ dn } \} \} \in
\text{rec } \{ f \ a \ x : a = g \ a \ ((\rightarrow) \ a \ de \ x \ x) \ x;
    g \ a \ b : \text{Bool } x : a - \text{abs } a \ dn \ (f \ a \ x) \}
\]

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

\[\text{To keep the example short it is degenerate: } f \text{ and } g \text{ both fail to terminate.}\]
What does this binding abstraction mean? It is a \texttt{binds} which gives values for both \texttt{f} and \texttt{g}, each of which is a function taking a type \texttt{a} and a dictionary \texttt{dn} as its arguments. If the bindings were not mutually recursive, and if there were no dictionary bindings (in this case the binding for \texttt{de}) to take into account this would be easy: just add \texttt{a} and \texttt{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):

```latex
\texttt{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 \texttt{de} can be dropped from the right-hand side of \texttt{g}, since it is not used, but this may not always be so. Thirdly, and perhaps most importantly, the expression \texttt{(eqFromNum \ a \ dn)} is recompiled 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 \texttt{f} with \texttt{(f \ a \ dn)}, and similarly for \texttt{g}.

A second possible translation is this:

```latex
\{ f \ a \ dn = let \{ de = eqFromNum \ a \ dn\} in \\
             let rec \{ f x = g \ ((==) \ a \ de \ x \ x) \ x; \\
             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 \\
             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 \texttt{(eqFromNum \ a \ dn)} is only computed once, but at the cost of duplicating the bindings for \texttt{f} and \texttt{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 \texttt{f} is called subsequently. This would mean that the (top-most) definition of \texttt{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:

```latex
\{ 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)
```
\[
\begin{align*}
\text{f a dn} &= \text{case (tup a dn) of (f,g) > f; } \\
\text{g a dn} &= \text{case (tup a dn) of (f,g) -> g; }
\end{align*}
\]

It appears that this translation is somewhat less efficient than the previous one, because of the tupling and untpuling. 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 \( \text{tup} \) 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 \( LIE_{\text{kind}} \) by \( \text{BIND-PRED} \), and which type variables are chosen for generalisation by \( \text{BIND-GEN} \). Nevertheless, in a type-inferencer, a suitable set must be chosen, and this section discusses the issues.

#### 9.3.1 Generalisation

Firstly, \( \text{BIND-GEN} \) will not choose any type variables which are free in \( VE \); this is part of the \( \text{BIND-GEN} \) rule. We call these type variables the *constrained type variables*. Nor is there any point in \( \text{BIND-GEN} \) choosing any type variables which are not free in the \( LVE \) being generalised. So we conclude that \( \text{BIND-GEN} \) 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

\[
\{ \text{tup} = (\lambda x \to x. \lambda y \to y) \}
\]

then

\[
\{ (f,g) = \text{tup} \}
\]

Type inference for the first binding will derive the environment

\[
\{ \text{tup} : \forall \alpha, \beta. (\alpha \to \alpha, \beta \to \beta) \}
\]

Type inference for the second binding, prior to \( \text{BIND-GEN} \) will infer the environment

\[
\{ f : \alpha \to \alpha, \ g : \beta \to \beta \}
\]

Now \( \text{BIND-GEN} \) will generalise over both \( \alpha \) and \( \beta \), giving the environment

\[
\{ f : \forall \alpha, \beta. \alpha \to \alpha, \ g : \forall \alpha, \beta. \beta \to \beta \}
\]

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 \((\alpha, \beta)\).) 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 \(\Box\) until none remain with this property. The remaining predicates affect only unconstrained type variables.

Here's an example of this separation process:

\[
f x y = \ldots \text{ where }
\begin{align*}
g z1 z2 &= \text{if} (x>y) \text{ then} z1 \text{ else} z2
\end{align*}
\]

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

\[
\{(d : \text{ord} \alpha), (g : \beta \rightarrow \beta \rightarrow \beta)\}
\]

The translation of \(g\)'s body will mention the dictionary \(d\) when it makes the comparison of \(x\) and \(y\). Since \(\alpha\) 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 \(\vdash\) 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 \( \forall \alpha. \text{Num} \alpha \Rightarrow \alpha \rightarrow \alpha \). Then given the type signature \( f : \text{Int} \rightarrow \text{Int} \), we need to make a new binding for \( f \) of the form

\[
f = f \infD
\]

where \( \infD \) is the \text{Num} dictionary for Ints. (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 \( \vdash \) 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 complicated than the original, because in effect it incorporates DGEN, SPEC, DPRED and REI! 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 \( \theta \), inventing new dictionary names.

Premise (6) does the same for the given context \( \theta_{\text{sig}} \).

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

10 Expressions

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

\[
E \vdash \exp \rightarrow \exp : (\text{LIE}, \sigma)
\]

which should be read "in environment \( E \), the expression \( \exp \) is translated to \( \exp \) and has type \( \sigma \), under the class constraints given by the local instance environment \text{LIE}". It is always the case that

\[
fv(\text{LIE}) \subseteq fv(\sigma)
\]

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.
Expressions: \[ E \vdash \text{exp} \sim \text{exp} : (LIE, \sigma) \]

- **TAUT-GVAR**: \[ E \vdash \text{var} \sim \text{var} : (\{\}, \sigma) \] \[ \text{(GVE of } E) \text{ var } = \sigma \text{ var } \in \text{dom(GVE of } E) \]

- **TAUT-LVAR**: \[ E \vdash \text{var} \sim \text{var} : (\{\}, \tau) \] \[ \text{(LVE of } E) \text{ var } = \tau \text{ var } \in \text{dom(LVE of } E) \]

- **TAUT-CON**: \[ E \vdash \text{con} \sim \text{con} : (\{\}, \sigma) \] \[ \text{(CVE of } E) \text{ con } = \sigma \]

- **TAUT-INTEGER**: \[ E \vdash \text{integer} \sim \text{fromInteger} \langle \text{dvar} \rangle \text{ integer} : (\{\text{dvar} : \text{Num } \tau \}, \tau) \]

- **TAUT-FLOAT**: \[ E \vdash \text{float} \sim \text{fromRational} \langle \text{dvar} \rangle \text{ float} : (\{\text{dvar} : \text{Fractional } \tau \}, \tau) \]

- **TAUT-CHAR**: \[ E \vdash \text{char} \sim \text{char} : (\{\}, \text{Char}) \]

- **TAUT-STRING**: \[ E \vdash \text{string} \sim \text{string} : (\{\}, [\text{Char}]) \]

**Figure 17**: Basic rules for expressions

**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 contrast to the LML compiler, for example, which passes it).

**GEN** and **SPEC** manipulate a group of type variables at once, rather than working with 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.
\[
\begin{align*}
E \vdash \text{exp} & \leadsto \text{exp} : (\{\}, \sigma) \\
\text{EXPSIG} & \\
& \frac{TCE \ of \ E; \ TVE \ of \ E; \ CE \ of \ E \vdash \text{polytype}}{E \vdash (\text{exp} :: \text{polytype}) \leadsto \text{exp} : (\{\}, \sigma)} \\
\text{LAMBDA} & \\
& \frac{E \vdash \{\} \vdash \text{mrule} \leadsto \text{mrule} : (\text{LIE}, \tau)}{E \vdash \text{\textbackslash mrule} \leadsto \text{\textbackslash mrule} : (\text{LIE}, \tau)} \\
& \frac{E \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}, \tau' \rightarrow \tau)}{E \vdash \text{exp}' \leadsto \text{exp}' : (\text{LIE}', \tau')}
\end{align*}
\]

\[
\begin{align*}
E \vdash (\text{exp} \ text{exp}') & \leadsto (\text{exp} \ text{exp}') : (\text{LIE} \oplus \text{LIE}', \tau) \\
\text{COMB} & \\
& \frac{E \vdash \text{binds} \leadsto \text{binds} : (\text{LIE}, \text{GVE})}{E \oplus \text{GVE} \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}', \tau)} \\
\text{LET} & \\
& \frac{E \vdash (\text{let binds in exp}) \leadsto (\text{let binds in exp}) : (\text{LIE} \oplus \text{LIE}', \tau)}{E \vdash \text{exp} \leadsto (\text{exp} \text{match}) : (\text{LIE}', \tau' \rightarrow \tau')} \\
\text{CASE} & \\
& \frac{E \vdash (\text{case exp of match}) \leadsto (\text{case exp of match}) : (\text{LIE} \oplus \text{LIE}', \tau')}{E \vdash \text{exp}\text{\textunderscore}1 \leadsto \text{exp}\text{\textunderscore}1 : (\text{LIE}\text{\textunderscore}1, \text{Bool})} \\
& \frac{E \vdash \text{exp}\text{\textunderscore}2 \leadsto \text{exp}\text{\textunderscore}2 : (\text{LIE}\text{\textunderscore}2, \tau)}{E \vdash \text{exp}\text{\textunderscore}3 \leadsto \text{exp}\text{\textunderscore}3 : (\text{LIE}\text{\textunderscore}3, \tau)} \\
\text{IF} & \\
& \frac{E \vdash \text{if exp\textunderscore}1 \ then \ exp\textunderscore}2 \ else \ exp\textunderscore}3 \leadsto (\text{if exp\textunderscore}1 \ then \ exp\textunderscore}2 \ else \ exp\textunderscore}3 : (\text{LIE}\text{\textunderscore}1 \oplus \text{LIE}\text{\textunderscore}2 \oplus \text{LIE}\text{\textunderscore}3, \tau)}{E \vdash \text{qual} \ leadsto \text{qual} : (\text{LIE}, \text{LVE})} \\
\text{LISTCOMP} & \\
& \frac{E \vdash [\text{exp} | \text{qual}] \leadsto [\text{exp} | \text{qual}] : (\text{LIE} \oplus \text{LIE}', [\tau])}{E \vdash [\text{exp} | \text{qual}] : (\text{LIE} \oplus \text{LIE}', [\tau])}
\end{align*}
\]

Figure 19: Rules for compound expressions

\[
\begin{align*}
E \vdash \text{exp}\_i & \leadsto \text{exp}\_i : (\text{LIE}\_i, \tau) \quad (0 \leq i \leq n) \\
\text{LIST} & \\
& \frac{E \vdash [\text{exp}\_i, \ldots, \text{exp}\_n] \leadsto [\text{exp}\_i, \ldots, \text{exp}\_n] : (\text{LIE}\_i \oplus \ldots \oplus \text{LIE}\_n, [\tau])}{E \vdash [\text{exp}\_i, \ldots, \text{exp}\_n] : (\text{LIE}\_i \oplus \ldots \oplus \text{LIE}\_n, (\tau_1, \ldots, \tau_n))}
\end{align*}
\]

Figure 20: Data structures in expressions

38
\[
\begin{align*}
\text{ENUM-FROM} & \quad \frac{E \vdash \text{exp} \leadsto \text{exp : } (LIE, \tau)}{E \vdash \text{[exp...]} \leadsto (\text{enumFrom} \ (\text{dvar}) \ \text{exp}) : (LIE \oplus \{\text{dvar : Enum } \tau\}, [\tau])} \\
\text{ENUM-FROM-THEN} & \quad \frac{\begin{array}{c}
E \vdash \text{exp}_1 \leadsto \text{exp}_1 : (LIE_1, \tau) \\
E \vdash \text{exp}_2 \leadsto \text{exp}_2 : (LIE_2, \tau)
\end{array}}{E \vdash \text{[exp}_1, \text{exp}_2...] \leadsto (\text{enumFromThen} \ (\text{dvar}) \ \text{exp}_1 \ \text{exp}_2) : (LIE_1 \oplus LIE_2 \oplus \{\text{dvar : Enum } \tau\}, [\tau])} \\
\text{ENUM-FROM-TO} & \quad \frac{\begin{array}{c}
E \vdash \text{exp}_1 \leadsto \text{exp}_1 : (LIE_1, \tau) \\
E \vdash \text{exp}_2 \leadsto \text{exp}_2 : (LIE_2, \tau)
\end{array}}{E \vdash \text{[exp}_1...\text{exp}_2] \leadsto (\text{enumFromTo} \ (\text{dvar}) \ \text{exp}_1 \ \text{exp}_2) : (LIE_1 \oplus LIE_2 \oplus \{\text{dvar : Enum } \tau\}, [\tau])} \\
\text{ENUM-FROM-THEN-TO} & \quad \frac{\begin{array}{c}
E \vdash \text{exp}_1 \leadsto \text{exp}_1 : (LIE_1, \tau) \\
E \vdash \text{exp}_2 \leadsto \text{exp}_2 : (LIE_2, \tau) \\
E \vdash \text{exp}_3 \leadsto \text{exp}_3 : (LIE_3, \tau)
\end{array}}{E \vdash \text{[exp}_1, \text{exp}_2...\text{exp}_3] \leadsto (\text{enumFromThenTo} \ (\text{dvar}) \ \text{exp}_1 \ \text{exp}_2 \ \text{exp}_3) : (LIE_1 \oplus LIE_2 \oplus LIE_3 \oplus \{\text{dvar : Enum } \tau\}, [\tau])}
\end{align*}
\]

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 \( \downarrow \) , but less easy to see how to do the same for case expressions.

**QUALS.** Note the use of \( \ominus \) in the conclusion of this rule, expressing the fact that that bindings within a list comprehension can shadow each other. For example \{\text{xs} \mid \text{xs} <- \text{ys} ; \text{xs} <- \text{xs}\} is allowed.
\[
\begin{align*}
E \vdash \text{quals} & \leadsto \text{quals} : (\text{LIE}, \text{LVE}) \\
E \vdash \text{quals}_1 \leadsto \text{quals}_1 : (\text{LIE}, \text{LVE}_1) \\
E \oplus \text{LVE}_1 \vdash \text{quals}_2 \leadsto \text{quals}_2 : (\text{LIE}_2, \text{LVE}_2) \\
E \vdash \text{quals}_1, \text{quals}_2 \leadsto \text{quals}, \text{quals}_2 : (\text{LIE}_1 \oplus \text{LIE}_2, \text{LVE}_1 \oplus \text{LVE}_2) \\
E \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}, \text{Bool}) \\
E \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}, \{\}) \\
\text{CVE of } E \vdash \text{pat} \leadsto \text{pat} : (\tau, \text{LVE}) \\
E \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}, [\tau]) \\
E \vdash \text{pat} \leftarrow \text{exp} \leadsto \text{pat} \leftarrow \text{exp} : (\text{LIE}, \text{LVE})
\end{align*}
\]

Figure 22: Rules for qual

\[
\begin{align*}
(1) & \quad E \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}, \tau) \\
(2) & \quad \alpha_1, \ldots, \alpha_n \not\in E \\
(3) & \quad \text{TCE of } E; \text{TVE of } E; \text{CE of } E \vdash \text{polytype} : \forall \beta_1, \ldots, \beta_m \theta_{\text{sig}} \Rightarrow \tau_{\text{sig}} \\
(4) & \quad \gamma_1, \ldots, \gamma_m \not\in E \cup \text{LIE} \cup \tau \\
(5) & \quad \tau_{\text{sig}}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] = \tau[\alpha_1, \ldots, \alpha_n] \\
(6) & \quad \text{LIE}_{\text{sig}} = \text{zip} \quad \text{dicts}_{\text{sig}} \quad \theta_{\text{sig}}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m] \\
(7) & \quad \text{GIE of } E; \text{LIE}_{\text{sig}} \vdash \text{LIE} \leadsto \text{binds} \\
E \vdash \text{exp} : \text{polytype} \leadsto \text{let binds in exp} : (\text{LIE}_{\text{sig}}, \tau_{\text{sig}}[\gamma_1/\beta_1, \ldots, \gamma_m/\beta_m])
\end{align*}
\]

Figure 23: More realistic expression signature rule

11 Pattern matching

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

\[
\text{CVE} \vdash \text{pat} \leadsto \text{pat} : (\tau, \text{LVE})
\]

and should should be read: “in the constructor value environment CVE, the pattern pat is translated to pat with type \( \tau \), 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 \( \vdash \) judgement type-checks \( \text{mrules} \), which are sequences of patterns followed by a sequence of guarded expressions. \( \text{mrules} \) are used by both function bindings and lambda ab-
<table>
<thead>
<tr>
<th>Patterns: ( CVE \vdash pat \sim \text{pat} : (\tau, LVE) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TAUT-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash pat \sim \var : (\tau, {\text{var} : \tau}) )</td>
</tr>
<tr>
<td>( CVE \vdash \text{con} = \sigma )</td>
</tr>
<tr>
<td>( \sigma \leq \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau )</td>
</tr>
<tr>
<td><strong>CONSTR-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash \text{pat}_i \sim \text{pat}_i : (\tau_i, LVE_i) ) (1 ( \leq i \leq n ))</td>
</tr>
<tr>
<td>( CVE \vdash \text{con pat}_1 \ldots \text{pat}_n \sim \text{con pat}_1 \ldots \text{pat}_n : (\tau, LVE_1 \oplus \ldots \oplus LVE_n) )</td>
</tr>
<tr>
<td><strong>AS-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash \text{pat} \sim \text{pat} : (\tau, LVE) )</td>
</tr>
<tr>
<td>( CVE \vdash \text{var@pat} \sim \text{var@pat} : (\tau, LVE \oplus {\text{var} : \tau}) )</td>
</tr>
<tr>
<td><strong>LAZY-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash \text{pat} \sim \text{pat} : (\tau, LVE) )</td>
</tr>
<tr>
<td>( CVE \vdash \neg \text{pat} \sim \neg \text{pat} : (\tau, LVE) )</td>
</tr>
<tr>
<td><strong>WILD-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash _ \sim _ : (\tau, {}) )</td>
</tr>
<tr>
<td><strong>LIST-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash \text{pat}_i \sim \text{pat}_i : (\tau, LVE_i) ) (0 ( \leq i \leq n ))</td>
</tr>
<tr>
<td>( CVE \vdash [\text{pat}_1, \ldots, \text{pat}_n] \sim [\text{pat}_1, \ldots, \text{pat}_n] : ([\tau], LVE_1 \oplus \ldots \oplus LVE_n) )</td>
</tr>
<tr>
<td><strong>TUPLE-PAT</strong></td>
</tr>
<tr>
<td>( CVE \vdash \text{pat}_i \sim \text{pat}_i : (\tau_i, LVE_i) ) (1 ( \leq i \leq n ))</td>
</tr>
<tr>
<td>( CVE \vdash (\text{pat}_1, \ldots, \text{pat}_n) \sim (\text{pat}_1, \ldots, \text{pat}_n) : ((\tau_1, \ldots, \tau_n), LVE_1 \oplus \ldots \oplus LVE_n) )</td>
</tr>
</tbody>
</table>

Figure 24: Rules for patterns

stractions.

Finally, the \( \vdash \) judgement type-checks a sequence of guarded expressions. The typing rules for this and the \textit{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 \( \vdash \) 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.
Matches: $E \vdash \text{match} \leadsto \text{match} : (\text{LIE}, \tau)$

$$
\begin{align*}
E & \vdash \text{match}_1 \leadsto \text{match}_1 : (\text{LIE}_1, \tau) \\
E & \vdash \text{match}_2 \leadsto \text{match}_2 : (\text{LIE}_2, \tau) \\
E & \vdash \text{match}_1 \parallel \text{match}_2 \leadsto \text{match}_1 \parallel \text{match}_2 : (\text{LIE}_1 \oplus \text{LIE}_2, \tau)
\end{align*}
$$

MATCH

$$
E: \{\} \vdash \text{mrule} \leadsto \text{mrule} : (\text{LIE}, \tau)
$$

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

MRULE-GRHS

$$
\begin{align*}
E & \vdash \text{LVE} \vdash \text{grhs} \leadsto \text{grhs} : (\text{LIE}, \tau) \\
E; \text{LVE} & \vdash \text{grhs} \leadsto \text{grhs} : (\text{LIE}, \tau)
\end{align*}
$$

MRULE-ABS

$$
\begin{align*}
\text{CVE of } E & \vdash \text{pat} \leadsto \text{pat} : (\tau, \text{LVE}') \\
E; \text{LVE} & \vdash \text{LVE}' \vdash \text{mrule} \leadsto \text{mrule} : (\text{LIE}, \tau')
\end{align*}
$$

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

GUARD-OR

$$
\begin{align*}
E & \vdash \text{grhs}_1 \leadsto \text{grhs}_1 : (\text{LIE}_1, \tau) \\
E & \vdash \text{grhs}_2 \leadsto \text{grhs}_2 : (\text{LIE}_2, \tau) \\
E & \vdash \text{grhs}_1 \parallel \text{grhs}_2 \leadsto \text{grhs}_1 \parallel \text{grhs}_2 : (\text{LIE}_1 \oplus \text{LIE}_2, \tau)
\end{align*}
$$

GUARD

$$
\begin{align*}
E & \vdash \text{guard} \leadsto \text{guard} : (\text{LIE}_1, \text{Bool}) \\
E & \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}_2, \tau)
\end{align*}
$$

GUARD-DEFAULT

$$
\begin{align*}
E & \vdash \text{exp} \leadsto \text{exp} : (\text{LIE}, \tau) \\
E & \vdash \text{grhs} \leadsto \text{exp} \leadsto \text{exp} : (\text{LIE}, \tau)
\end{align*}
$$

Figure 25: Rules for match, mrule and grhs
Dictionary equivalence: $GIE; LIE \vdash \text{dicts} LIE' \leadsto \text{binds}$

**DICT-TRIV**

\[
GIE; LIE \vdash \{\} \leadsto \{\}
\]

**DICTS**

\[
\begin{align*}
GIE; LIE \vdash & LIE_1 \leadsto \text{binds}_1 \\
GIE; LIE \vdash & LIE_2 \leadsto \text{binds}_2
\end{align*}
\]

\[
GIE; LIE \vdash LIE_1 \oplus LIE_2 \leadsto \text{binds}_1 \text{ then } \text{binds}_2
\]

**DICT-GLOBAL**

\[
\begin{align*}
GIE; LIE \vdash & \{\text{dvar : } \kappa \tau | \tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\} \\
& \leadsto \text{binds then } \{\text{dvar = dfun } \tau_1 \ldots \tau_n \text{ dicts}\}
\end{align*}
\]

**DICT-DUP**

\[
GIE; LIE \vdash \{\text{dvar : } \kappa \tau\} \leadsto \{\text{dvar = dvar}'\}
\]

Figure 26: Dictionary manipulation

Finally, MRULE-GRIFS 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 $\vdash$ judgement form tells how to express some dictionaries in terms of others. It takes the form:

\[
GIE; LIE \vdash LIE' \leadsto \text{binds}
\]

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

There are four main places $\vdash$ 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 \( \vdash \) judgement allows this redundancy
to be eliminated.

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

There are three ways in which \( \vdash \) can give a binding for a dictionary \( \texttt{dvar} \) with type \( \kappa \ k \): 

- If \( \texttt{LIE} \) contains a dictionary with identical type, a trivial binding will suffice (DICTS-DUP).
- If \( \tau \) is of form \( \chi \tau_1 \ldots \tau_n \) then there should have been an instance declaration for the \( \chi \) instance of \( \kappa \). In this case, \( \texttt{LIE} \) will contain a matching assumption, which is used in
the DICTS-GLOBAL rule to translate \( \texttt{dvar} \ : \ \kappa \ \tau \) to a binding and a simpler collection
of \( \texttt{LIE} \) assumptions.
- Finally, if \( \tau \) is just a type variable \( \alpha \), it may nevertheless be that there is another
assumption in \( \texttt{LIE} \) of form \( \texttt{dvar}' \ : \ \kappa' \ \alpha \) where \( \kappa' \) is a superclass of \( \kappa' \). In this case it
is possible to express \( \texttt{dvar} \) in terms of \( \texttt{dvar}' \). 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 hierarchy.

13 Implementation notes

When generalising (DGEN) it is sufficient (I think) to look at \( \texttt{LVE} \), and not \( \texttt{GVE} \), because
while \( \texttt{GVE} \) may contain free type variables, they are only free because they are also free in
\( \texttt{LVE} \) (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 John-
son, R Kieburtz, RS Nihkil, 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 \( \texttt{ad-hoc} \) polymorphism less \( \texttt{ad-hoc} \)." in Proc 16th
ACM Symposium on Principles of Programming Languages, Austin, Texas. ACM.
Index

AS-PAT, 43
bind, judgement form, 20, 30, 31, 31
bind-1, judgement form, 31
bind-2, judgement form, 31
BIND-GEN, 3
BIND-RED, 31
BIND-RET, 31
binds, judgement form, 27, 30, 30, 30, 39
BINDS-SIG, 30
BINDS-THEN, 30
BINDS-TRIV, 30
CASE, 39
CLASS, 25
classdec, judgement form, 20, 25, 25, 25
CLASSES, 25
COMB, 39
CONBIND, 23
condecl, judgement form, 23, 23
CONSTR, 23
CONSTR-PAT, 43
CONTEXT, 22
context, judgement form, 22, 22, 23, 25, 27
DATA-DEC, 23
DICT-TYP, 46
DICT GLOBAL, 46
DICT-TRIV, 46
DICTS, 46
dicts, judgement form, 27, 31, 32, 36, 42, 45, 46, 46
EMPTYBIND, 29
ENUM-FROM, 42
ENUM-FROM-THEN, 42
ENUM-FROM-THEN-TO, 41
ENUM-FROM-TO, 41
exp, judgement form, 37, 38, 38, 39–42, 44
EXP Sig, 39
EXP Sig1, 42
FILTER, 42
FUNBIND, 29
GENERATOR, 42
grhs, judgement form, 29, 44, 44, 44, 45
GUARD, 44
GUARD-DEFAULT, 44
GUARD-OR, 44
IF, 38
INST, 27
inst, judgement form, 20
instdecl, judgement form, 27, 27
INSTS, 27
judgement form
bind, 20, 30, 31, 31
bind-1, 31
bind-2, 31
binds, 27, 30, 30, 30
classdec, 20, 25, 25
condecl, 23, 23, 23
context, 22, 22, 23, 25, 27
dicts, 27, 31, 32, 36, 42, 45, 46, 46, 46
exp, 37, 38, 38, 39–42, 44
grhs, 29, 44, 44, 44, 45
inst, 20
instdecl, 27, 27
match, 29, 39, 44, 44
monominds, 28, 29, 29, 31, 33
monotype, 21, 21, 22, 23, 27
monoids, 39, 44, 44, 45
nonterminal, 8
pat, 5, 29, 42, 43, 43, 43–45
polytype, 22, 22, 25, 32, 39, 42
program, 20, 20
qual, 39, 40, 42, 42
sigs, 25, 25, 25, 30, 32, 32, 32, 36, 37
typedec, 20, 22, 23, 23, 23
LAMBDA, 38
LAZY-PAT, 45
LET, 30
LIST, 40
LIST-PAT, 45
LISTCOMP, 39
MATCH, 44

45
match, judgement form, 20, 39, 44, 44
MATCHES, 44
monobinds, judgement form, 28, 29, 29, 31, 33
monotype, judgement form, 21, 21, 22, 23, 27
Mrule, judgement form, 39, 44, 44, 45
MRULE-ABS, 44
MRULE-GRHS, 44
nonterminal, judgement form, 8
pat, judgement form, 5, 29, 42, 43, 43, 44, 45
PATBIND, 29
polytype, judgement form, 22, 22, 25, 32, 39, 42
POLYTYPE1, 22
POLYTYPE2, 22
PROGRAM, 20
program, judgement form, 20, 20
qual, judgement form, 39, 40, 42, 42
QUALS, 42
REL, 38
SIG, 25, 32
SIG1, 32
x _EMPTY, 25
SIGS, 25, 32
SIGS, judgement form, 25, 25, 25, 30, 32, 32, 36, 37
SIGS-EMPTY, 32
SPEC, 38
SYN-DECL, 23
TAUT CHAR, 38
TAUT-CON, 38
TAUT FLOAT, 38
TAUT-GVAR, 38
TAUT-INTEGER, 35
TAUT-JVAR, 38
TAUT-PAT, 43
TAUT STRING, 38
TUPLE, 40
TUPLE PAT, 43
TYPE-DECLS, 23
TYPE-FUN, 21
TYPE-LIST, 21
TYPE-TAUT, 21
TYPE-TUPLE, 21
typedecl, judgement form, 20, 22, 23, 25, 29
UNIÓN, 29
WILDPAT, 43