JFP **12** (4 & 5): 295–357, July & September 2002.
DOI: 10.1017/S0956796802004380 Printed in the United Kingdom

A static semantics for Haskell

KARL-FILIP FAXÉN
KTH/IMIT/LECS, Electrum 229, S-164 40 Kista, Sweden
(*e-mail:* kff@it.kth.se)

Abstract

This paper gives a static semantics for Haskell 98, a non-strict purely functional programming language. The semantics formally specifies nearly all the details of the Haskell 98 type system, including the resolution of overloading, kind inference (including defaulting) and polymorphic recursion, the only major omission being a proper treatment of ambiguous overloading and its resolution. Overloading is translated into explicit dictionary passing, as in all current implementations of Haskell. The target language of this translation is a variant of the Girard–Reynolds polymorphic lambda calculus featuring higher order polymorphism and explicit type abstraction and application in the term language. Translated programs can thus still be type checked, although the implicit version of this system is impredicative. A surprising result of this formalization effort is that the monomorphism restriction, when rendered in a system of inference rules, compromises the principal type property.

1 Introduction

It is now more than ten years since the first version of Haskell was made public (Hudak & Wadler, 1990), and Haskell is now one of two widely used non strict functional languages (the other being Clean). During this time, Haskell has been defined by a succession of Reports, the latest one defining Haskell 98 (Peyton Jones *et al*., 1999). While the Reports have provided formal descriptions of Haskell syntax, the static and dynamic semantics has only been treated informally. This is especially unfortunate for the static semantics since the type system, in particular, is both complex and innovative. Several research papers describe various aspects of it in a simplified form (Wadler & Blott, 1989; Jones, 1995; Hall *et al*., 1996; Jones, 1999), and still more propose extensions. In particular, Peyton Jones *et al*. (1997) discuss a number of minor variations of the type class system, though only informally. None of the formal papers have dealt with the full Haskell language, the one getting closest being Hall *et al*. (1996). This situation has had the concrete consequence that different Haskell implementations actually differ in the programs they consider legal.

This paper aims to provide most of what has previously been missing. We describe the latest version of Haskell, called Haskell 98. While the language is expected to continue evolving by incorporating the more useful of the proposed extensions, Haskell 98 is special in that implementations are expected to continue supporting it even when subsequent versions of the language have been defined. Thus Haskell 98

is a stable target for application writers and authors of text books (and semantices). In the rest of the paper we will refer to Haskell 98 simply as Haskell.

We follow previous formalizations of type classes in providing a translation into a language without overloading. Our target language is explicitly typed and similar to the Girard-Reynolds polymorphic lambda calculus (Girard, 1972; Reynolds, 1974), although we do not use the full power of that system.

1.1 Sources

The main source for this work has of course been the Haskell Report (Peyton Jones *et al*., 1999); in the rest of the paper we will refer to this document simply as the Report, and we will frequently refer to it when discussing our formalizations of various features. When the Report has been unclear, we have sometimes consulted (Jones, 1999) which gives a type inference algorithm for a subset of Haskell (basically the expression sublanguage, including bindings). We have also considered (Jones, 1995) to be a source with respect to constructor classes and higher order polymorphism.

Another important starting point has been Peyton Jones and Wadler's unpublished draft semantics for Haskell 1.2 (Peyton Jones & Wadler, 1991). While that report describes an earlier version of the language, is incomplete and was never formally published, it is still the closest ancestor of the present work.

We have not considered the various existing implementations as sources, however. Where an implementation differs from the Report, we think that the Report is correct rather than the implementation. Where the Report is vague, we have still not consulted any implementation but resolved the ambiguity both with an eye to what appears most useful to the programmer and to what gives the simplest formal description.

1.2 Scope and contributions

Since this semantics aims to be complete, it deals with many issues which have not been formalized before. These issues include:

Modules. We deal with import and export specifications (section 4), but the influence of the module system is pervasive in the semantics. We translate Haskell into a language using *original names* to refer to entities defined in other modules.

Kind inference. Kinds can be seen as types for types and are used in Haskell to structure its higher order type system (type variables range not only over types, but also over type constructors). Kinds are not explicit in the source language but are inferred by the type checker. Sometimes, more than one kind can be assigned to a class or type. Since the kind system is monomorphic, kinds are defaulted in these situations. Kind defaulting is only sketched in the Report, and its integration into an inference system is not entirely trivial (section 3).

Default methods. None of the previous formalizations deal with the typing of default method bindings in class declarations, and the Report is not very explicit. There does however seem to be only one sensible choice, which we present in section 6.1.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

Labelled fields. Constructors with labelled fields (records) were introduced in Haskell 1.3 and are absent from previous formal descriptions. Much of their semantics is only described indirectly in the Report, and it turns out that the interaction with overloading and polymorphism is non obvious (section 5.2.2).

In addition, there are various other constructs, including for instance $n+k$ patterns, which have not been given formal typing rules previously.

1.3 Omissions

The major omission in this paper is a proper treatment of ambiguity. We have not been able to formulate inference rules which disallow ambiguity since ambiguity can always be hidden by making a less than maximally general derivation, resolving the ambiguous overloading in an arbitrary way (section 10.2).

In addition, we do not deal with strictness flags, nor do we treat `newtype` declarations (these are indistinguishable from `data` declarations from a typing point of view). A more substantial omission is `deriving` clauses in algebraic data type declarations. A proper treatment of derived instances would take up a lot of space, mainly for specifying the dynamic semantics. Fortunately, generating derived instances can be done before type checking, and is described in the Report (Appendix D).

We do not describe the semantics of mutually recursive modules since their semantics is effectively left up to each implementation by the Report (section 5.7).

2 Notation and an introduction to the formalization

This section discusses the source and target languages, the syntax of types and judgments, typing environments and some other issues which cut across the entire system.

2.1 Abstract syntax of Haskell

Since the program has already been parsed, as described in the Report, we will not work with the concrete syntax but with a slightly simplified abstract syntax which is quite close to the data type one would use to represent a Haskell program in a compiler. This syntax differs from the concrete one in the following respects:

- Infix operators are assumed to be compiled to function applications, and fixity declarations are eliminated. Operator sections are translated to lambda abstractions.
- List expression and patterns of the form $[e_1, \ldots, e_n]$ and $[p_1, \ldots, p_n]$, respectively, are assumed to be written as `(:)` e_1 `(...((:)` e_n `[])...)` and similarly for list patterns.
- Likewise, types are written in prefix form. We thus have the following

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$
\begin{array}{rll}
\mathit{mod} \in \text{Module} & \to \texttt{module}\ M\ \texttt{(}\mathit{ent}_1, \ldots, \mathit{ent}_k\texttt{)}\ \texttt{where}\ \mathit{imp}_1\,; \ldots ; \mathit{imp}_n\,;\ \mathit{body} & k, n \geqslant 0 \\
\mathit{imp} \in \text{Import} & \to \texttt{import}\ \mathit{qualifier}\ M\ \texttt{as}\ M'\ \mathit{implist} & \\
\\
\mathit{qualifier} \in \text{Qualifier} & \to \texttt{[qualified]} & \\
\\
\mathit{implist} \in \text{Import list} & \to \texttt{[[hiding]}\ \texttt{(}\mathit{ent}_1, \ldots, \mathit{ent}_n\texttt{)]} & n \geqslant 0 \\
\\
\mathit{ent} \in \text{Entity} & \to x & \\
& \mid K & \\
& \mid T\ \texttt{(}x_1, \ldots, x_k, K_1, \ldots, K_n\texttt{)} & k, n \geqslant 0 \\
& \mid T\ \texttt{(..)} & \\
& \mid C\ \texttt{(}x_1, \ldots, x_k\texttt{)} & k \geqslant 0 \\
& \mid C\ \texttt{(..)} & \\
& \mid \texttt{module}\ M & \\
\\
\mathit{body} \in \text{Module body} & \to \mathit{ctDecls}\,;\ \mathit{instDecls}\,;\ \mathit{binds} & \\
\\
\mathit{ctDecls} \in \text{Classes and types} & \to [\mathit{ctDecl}_1\,; \ldots ; \mathit{ctDecl}_n\ \texttt{then}\ \mathit{ctDecls}] & n \geqslant 1 \\
\\
\mathit{ctDecl} \in \text{Class or type} & \to \texttt{type}\ S\ u_1 \ldots u_k = t & k \geqslant 0 \\
& \mid \texttt{data}\ \mathit{cx}\ \texttt{=>}\ S\ u_1 \ldots u_k = \mathit{conDecls} & k \geqslant 0 \\
& \mid \texttt{class}\ \mathit{cx}\ \texttt{=>}\ B\ u\ \texttt{where}\ \mathit{sigs}\,;\ \mathit{bind}_1\,; \ldots ; \mathit{bind}_n & n \geqslant 0 \\
\\
t \in \text{Type expression} & \to u & \\
& \mid T & \\
& \mid t_1\ t_2 & \\
\\
\mathit{cx} \in \text{Context} & \to \texttt{(}\mathit{class}_1, \ldots, \mathit{class}_k\texttt{)} & k \geqslant 0 \\
\\
\mathit{class} \in \text{Class assertion} & \to C\ \texttt{(}u\ t_1 \ldots t_k\texttt{)} & k \geqslant 0 \\
\\
\mathit{conDecls} \in \text{Constructor decls} & \to \mathit{conDecl}_1 \mid \ldots \mid \mathit{conDecl}_n & n \geqslant 1 \\
\\
\mathit{conDecl} \in \text{Constructor decl} & \to J\ t_1 \ldots t_k & k \geqslant 0 \\
& \mid J\ \{v_1\ \texttt{::}\ t_1, \ldots, v_k\ \texttt{::}\ t_k\} & k \geqslant 0 \\
\\
\mathit{instDecls} \in \text{Instance decls} & \to \mathit{instDecl}_1\,; \ldots ; \mathit{instDecl}_n & n \geqslant 0 \\
\\
\mathit{instDecl} \in \text{Instance decl} & \to \texttt{instance}\ \mathit{cx}\ \texttt{=>}\ C\ t\ \texttt{where}\ \mathit{bind}_1\,; \ldots ; \mathit{bind}_n & n \geqslant 0 \\
\\
\mathit{sigs} \in \text{Signatures} & \to \mathit{sig}_1\,; \ldots ; \mathit{sig}_n & n \geqslant 0 \\
\\
\mathit{sig} \in \text{Signature} & \to v\ \texttt{::}\ \mathit{cx}\ \texttt{=>}\ t &
\end{array}
$$

Fig. 1. Abstract syntax, part 1.

equalities:

$$
\begin{array}{rll}
\texttt{(->)}\ t_1\ t_2 = t_1\ \texttt{->}\ t_2 & & \text{Function types} \\
\texttt{[]}\ t = \texttt{[}t\texttt{]} & & \text{Lists} \\
\texttt{(}k\texttt{)}\ t_1 \ldots t_k = \texttt{(}t_1\texttt{,} \ldots \texttt{,}\ t_k\texttt{)} & & k\text{-tuples}
\end{array}
$$

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

Table 1. *Lexical syntax*

	Unqualified form	Qualified form	
Variables	v	x	
Constructors	J	K	(including `()`, (k), `[]` and `(:)`)
Type names	S	T	(including `()`, (k), `[]` and `(->)`)
Type variables	u	M	
Class names	B	C	
Module names	M	M	

We also use the syntax (k) rather than `(,...,)` with $k-1$ commas for the curried k-tuple constructor.

- The same syntax is used for lists of case alternatives and for function bindings; the different alternatives for the same function is collected into the same *match*. This syntax also allows case alternatives with a sequence of patterns, something which will not be generated by a translation from Haskell source.
- All pattern matching is guarded; an unguarded (source level) match $p = e$ can be translated into p `|` `True` $= e$. Similarly, all guarded expression lists have `where` clauses; a source level list without a `where` clause can be translated to $gde_1 \ldots gde_n$ `where` ϵ.
- A type signature gives a type for one variable only; source level signatures can easily be split into this form. Similarly for constructor declarations with labelled fields.
- We mark labelled record updates with an explicit $\Leftarrow$ to distinguish them from labelled constructions.
- Haskell has special syntax for bindings in instance declarations since these may bind qualified names. Our abstract syntax instead allows for qualified names in all function bindings and patterns.
- Binding groups are explicitly nested in our abstract syntax; a *binds* consists of a sequence of *bindG*s. This allows the result of dependency analysis to be made explicit, which is important for type checking.

We give the abstract syntax in figures 1, 2 and 3 where we use [*phrase*] for an optional occurrence of *phrase* (either *phrase* or ϵ).

2.2 Name spaces

Section 1.4 of the Report states that class names and type names share a single name space, so that the environment determines whether a given identifier refers to a class or a type. In this paper we will instead assume that all name spaces are distinct and that type and class names are syntactically distinguished.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

binds ∈ Binds	→	[*sigs* ; *bindG* `then` *binds*]	
bindG ∈ Bind group	→	$bind_1 ; \ldots ; bind_n$	$n \geqslant 1$
bind ∈ Binding	→	$x\ match_1$ [] … [] $match_n$	$n \geqslant 1$
	|	p *gdes*	
match ∈ Match	→	$p_1 \ldots p_k$ *gdes*	$k \geqslant 1$
gdes ∈ Guarded exprs	→	$gde_1 \ldots gde_n$ `where` *binds*	$n \geqslant 1$
gde ∈ Guarded exp	→	`|` e_1 `=` e_2	
e ∈ Expression	→	x	
	|	*literal*	
	|	K	
	|	`\` $p_1 \ldots p_k$ `->` e	$k \geqslant 1$
	|	$e_1\ e_2$	
	|	`let` *binds* `in` e	
	|	`case` e `of` $match_1$ [] … [] $match_n$	$n \geqslant 1$
	|	`do` *stmts*	
	|	`[`e `|` *quals*`]`	
	|	`[`e_1 `[,`e_2`] . .[`e_3`]]`	
	|	$e \Leftarrow \{fbind_1, \ldots, fbind_k\}$	$k \geqslant 0$
	|	$K\ \{fbind_1, \ldots, fbind_k\}$	$k \geqslant 0$
stmts ∈ Statements	→	p `<-` e ; *stmts*	
	|	`let` *binds* ; *stmts*	
	|	e ; *stmts*	
	|	e	
quals ∈ Qualifiers	→	p `<-` e, *quals*	
	|	`let` *binds*, *quals*	
	|	e, *quals*	
	|	ϵ	
fbind ∈ Field binding	→	x `=` e	
p ∈ Pattern	→	x	
	|	$K\ p_1 \ldots p_k$	$k \geqslant 0$
	|	$K\ \{fp_1, \ldots, fp_k\}$	$k \geqslant 0$
	|	v`@`p	
	|	`~`p	
	|	`_`	
	|	*literal*	
	|	v`+`*integer*	
fp ∈ FieldPattern	→	x `=` p	

Fig. 2. Abstract syntax, part 2.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$T \in$ Qualified type constructor $\rightarrow S \mid M.S \mid \Sigma$

$C \in$ Qualified class name $\rightarrow B \mid M.B$

$\Sigma \in$ Special type constructor $\rightarrow$ `()` $\mid$ `(`k`)` $\mid$ `[]` $\mid$ `(->)`

$K \in$ Qualified data constructor $\rightarrow J \mid M.J \mid \Delta$

$\Delta \in$ Special data constructor $\rightarrow$ `()` $\mid$ `(`k`)` $\mid$ `[]` $\mid$ `(:)`

$S \in$ Type constructor
$u \in$ Type variable
$B \in$ Class name
$J \in$ Data constructor
$M \in$ Module name

literal $\in$ Literal $\rightarrow$ *char* $\mid$ *string* $\mid$ *integer* $\mid$ *float*

Fig. 3. Abstract syntax, part 3.

$\sigma \in$ Type scheme $\rightarrow \forall\alpha_1 \ldots \alpha_k . \theta \Rightarrow \tau$

$\theta \in$ Context $\rightarrow (\Gamma_1\,\tau_1, \ldots, \Gamma_n\,\tau_n)$

$\tau \in$ Type $\rightarrow \alpha \mid \chi \mid \tau_1\,\tau_2$

$\chi \in$ Type constructor $\rightarrow \mathsf{T}^\kappa$

$\alpha \in$ Type variable $\rightarrow u^\kappa$

$\Gamma \in$ Class name $\rightarrow \mathsf{C}^\kappa$

$\mathsf{T} \in$ Original type name $\rightarrow \mathsf{S} \mid \mathsf{M!S} \mid \Sigma$

$\mathsf{C} \in$ Original class name $\rightarrow B \mid \mathsf{M!}B$

$\kappa \in$ Kind $\rightarrow * \mid \kappa_1 \rightarrow \kappa_2$

Fig. 4. Semantic types.

2.3 Qualified and original names

In Haskell, `import` declarations are used to bring names defined in other modules into scope. To resolve name clashes, *qualified names* may be used. A qualified name has the form *M.name* where *M* is either the name of the module that *name* is imported from or an arbitrary module identifier given in the `import` declaration (using the `as` form). A module may also export a name it has itself imported from another module, so the qualifier in a qualified name does not necessarily correspond to the module containing the definition of the name.

It is also possible that the same definition is referred to by names with different

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

qualifiers. To avoid gratuitous name clash errors, it is legal to import the same name by different routes (through different import declarations) as long as all of the routes lead to the same definition in the same module.

To formalize these requirements, our semantics uses *original names* which refer to the module containing the definition of the name rather than to the `import` declaration bringing the name into scope. A *local* original name looks like an unqualified name (*e.g.* `foo`) and always refers to a definition in the current module. A *global* original name is syntactically similar to a qualified name, using "`!`" rather than "`.`" (*e.g.* `Foo!foo`) and always refers to a definition in the indicated module. We will sometimes regard a local name as a global name with ϵ as the qualifier (*e.g.* ϵ`!foo`), although we never write them that way. In the translated program, references to imported entities use global original names whereas locally defined names are local. The same rule holds for entities used in the inference rules, like type and class names, which are always global if imported. Consider the following example modules:

```
module Foo where
foo = 1
x = True

module Bar(foo,bar,x)
import Foo(foo)
bar = 2
x = "Hello, world!\n"

module Main where
import Foo as F
import Bar
y = 42
main = ...
```

In the definition of `main`, the various qualified names are related to original names as follows:

- `bar` and `Bar.bar` correspond to the original name `Bar!bar` and thus have the value `2`.
- `foo`, `F.foo` and `Bar.foo` correspond to the original name `Foo!foo` and consequently refer to the definition in `Foo` with value `1`. The `F` comes from the `as` clause, and `Bar.foo` is legal since `Bar` imports `foo` and reexports it. The unqualified `foo` is legal since both of the `import` declarations yield the same original name for `foo`.
- `F.x` corresponds to `Foo!x` and has value `True` and `Bar.x` corresponds to `Bar!x` and has the value `"Hello, world!\n"`. The unqualified `x` is illegal since the two `import` declarations yield different original names for `x`.
- `y` corresponds to `y` (or ϵ`!y`) with value `42` since we omit the module name part in locally defined original names.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

x ∈ Original variable	→ M!v	v ∈ Target variable	→ *v*
	| v	J ∈ Target constructor	→ *J*
		M ∈ Target module name	→ *M*
K ∈ Original constructor	→ M!J		
	| J	S ∈ Target type name	→ *S*
	| Δ		| *B*

Fig. 5. Lexical syntax of the target language.

In this way, the module system of the target language is much simplified: Both `import` declarations and exports are eliminated, so that a module becomes just a named set of type declarations and bindings.

We will sometimes want to remove a possible qualifier from a qualified name, and we write $unQual(x)$ for this purpose (*e.g.* $unQual(\texttt{Bar.foo}) = \texttt{foo}$).

2.4 Types

We distinguish between the type expressions of the source program and the types derived by the inference rules. We call the latter *semantic types* and we give their syntax in figure 4. The main differences between syntactic and semantic types are that semantic type and class names and type variables are explicitly kinded and that type and class names are original. Semantic contexts, types and type schemes are always assumed to be well-kinded. Whenever these entities occur in an inference rule, it is an implicit side condition that the entity is well-kinded.

We will not assume that the type system knows about the special type constructors Σ (`()`, `[]`, etc) and data constructors Δ. Instead we will have information about these in an initial environment given in figure 16.

From time to time, we will help ourselves to some syntactic sugar: When a context in a type scheme is empty, we will write the type scheme as $\forall\alpha_1 \ldots \alpha_k.\tau$ and if the list of quantified type variables is empty as well we will simply write τ. We will also use the familiar notation for some types, for instance $[\tau]$ instead of $[]^{*\to *}\ \tau$ for lists and $\tau_1 \to \tau_2$ rather than $(\texttt{->})^{*\to *\to *}\ \tau_1\ \tau_2$ for functions.

2.5 The target language

The target language of the translation is rather similar to the source language, differing mainly in things related to the module system, type classes and types in general, and bindings. The module system is much simplified due to the use of original names for imported entities, obviating the need for import and export specifications. The constructs related to type classes (class and instance declarations, contexts, `do` expressions and arithmetic sequences) are removed by the translation and do not occur in the target language. The types of all variables are explicit and universal quantification and instantiation are also explicit in the term language. Type and variable names are annotated with kinds. The explicit type information on variable bindings also makes the nesting of binding sets unnecessary since that

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

mod ∈ Module	→	`module` M `where` typeDecls; binds	
typeDecls ∈ Type declarations	→	$\mathsf{typeDecl}_1; \ldots; \mathsf{typeDecl}_n$	$n \geqslant 0$
typeDecl ∈ Type declaration	→	`data` $\chi\ \alpha_1 \ldots \alpha_k = \mathsf{conDecl}_1 \mid \ldots \mid \mathsf{conDecl}_n$	$k \geqslant 0$ $n \geqslant 1$
conDecl ∈ Constructor decl	→	$\mathsf{J}\ \sigma_1 \ldots \sigma_k$	$k \geqslant 0$
	|	$\mathsf{J}\{\mathsf{v}_1 : \sigma_1, \ldots, \mathsf{v}_k : \sigma_k\}$	$k \geqslant 0$
binds ∈ Bindings	→	$\mathsf{bind}_1; \ldots; \mathsf{bind}_n$	$n \geqslant 0$
	|	`rec` $\mathsf{bind}_1; \ldots; \mathsf{bind}_n$	$n \geqslant 1$
bind ∈ Binding	→	$\mathsf{v} : \sigma\ \mathsf{match}_1\ [] \ldots [] \mathsf{match}_n$	$n \geqslant 1$
	|	p gdes	
match ∈ Match	→	$\mathsf{p}_1 \ldots \mathsf{p}_k$ gdes	$k \geqslant 1$
gdes ∈ Guarded expressions	→	$\mathsf{gde}_1 \ldots \mathsf{gde}_n$ `where` binds	$n \geqslant 1$
gde ∈ Guarded expression	→	$\mid \mathsf{e}_1 = \mathsf{e}_2$	
e ∈ Expression	→	x	
	|	literal	
	|	K	
	|	$\lambda\, \mathsf{p}_1 \ldots \mathsf{p}_k$ -> e	$k \geqslant 1$
	|	$\mathsf{e}_1\ \mathsf{e}_2$	
	|	`let` binds `in` e	
	|	`case` e `of` $\mathsf{match}_1\ [] \ldots [] \mathsf{match}_n$	$n \geqslant 1$
	|	[e | quals]	
	|	$\mathsf{e} \Leftarrow \{\mathsf{fbind}_1, \ldots, \mathsf{fbind}_k\}$	$k \geqslant 0$
	|	$\mathsf{e}\{\mathsf{fbind}_1, \ldots, \mathsf{fbind}_k\}$	$k \geqslant 0$
	|	$\mathsf{e}\ \tau_1 \ldots \tau_k$	$k \geqslant 1$
	|	$\Lambda\alpha_1 \ldots \alpha_k.\, \mathsf{e}$	$k \geqslant 1$
quals ∈ Qualifiers	→	p <- e, quals	
	|	`let` binds, quals	
	|	e, quals	
	|	ϵ	
fbind ∈ Field binding	→	x = e	
p ∈ Pattern	→	$\mathsf{v} : \sigma$	
	|	$\mathsf{K}\ \mathsf{p}_1 \ldots \mathsf{p}_k$	$k \geqslant 0$
	|	$\mathsf{K}\ \{\mathsf{fp}_1, \ldots, \mathsf{fp}_k\}$	$k \geqslant 0$
	|	$\mathsf{v} : \sigma$@p	
	|	~p	
	|	_	
	|	{e}	
	|	$\mathsf{v} : \sigma\{\mathsf{e}_1, \mathsf{e}_2\}$	
fp ∈ Field pattern	→	x = p	

Fig. 6. Target syntax.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$\widehat{(\Gamma_1\,\tau_1,\ldots,\Gamma_k\,\tau_k)} = (\widehat{\Gamma}_1\,\tau_1,\ldots,\widehat{\Gamma}_k\,\tau_k)$	Context to dictionaries
$\widehat{\mathsf{C}^\kappa} = \mathsf{C}^{\kappa\to *}$	Class to dictionary type
$(\mathsf{v}_1,\ldots,\mathsf{v}_k)\,\hat{:}\,(\Gamma_1\,\tau_1,\ldots,\Gamma_k\,\tau_k) = (\mathsf{v}_1 : \widehat{\Gamma}_1\,\tau_1,\ldots,\mathsf{v}_k : \widehat{\Gamma}_k\,\tau_k)$	Vars and context to typed dictionary pattern
$(\mathsf{x}_1,\ldots,\mathsf{x}_k)\,\tilde{:}\,(\Gamma_1\,\tau_1,\ldots,\Gamma_k\,\tau_k) = \{\mathsf{x}_1 : \Gamma_1\,\tau_1,\ldots,\mathsf{x}_k : \Gamma_k\,\tau_k\}$	instance environment

Fig. 7. Correspondence between classes and types.

nesting only serves to allow more polymorphic types to be derived in the absence of explicit types.

The semantics implements overloading by explicit dictionary passing. This translation affects types as follows: An overloaded value with type $\forall\alpha_1\ldots\alpha_k.\,\theta \Rightarrow \tau$ will be translated to a function with type $\forall\alpha_1\ldots\alpha_k.\,\widehat{\theta}\to\tau$ where $\widehat{\theta}$ is the type of a dictionary tuple corresponding to the context θ. The basic names in the target language are taken from those in the source language (*e.g.* a v is a v), as shown in figure 5 on page 303, with the extra feature that a target type name S can also be a source class name B, which is convenient for this translation.

On the term level, we will use the $\hat{:}$ operator to construct typed dictionary patterns from a tuple of dictionary variables $(\mathsf{v}_1,\ldots,\mathsf{v}_k)$ (which we will often write as vs) and a context θ. We will also use $\tilde{:}$ to make an overloading environment (section 2.7.4) from the same ingredients.

A class declaration for class Γ will be translated to an algebraic type declaration declaring $\widehat{\Gamma}$ to be a single constructor type with labelled fields for the class operations and super classes. Since class methods may be polymorphic in Haskell, we allow polymorphic fields in constructor declarations.

There are two unusual forms of pattern in the target language. They are used to implement overloaded literal patterns and n+k-patterns and are discussed in more detail in section 9.1.

2.6 Judgment forms

The task of the inference rules presented in the following sections is in general three-fold:

- To check that the program is well-formed,
- to specify a translation of the program into a language without overloading and
- to derive some information about the program.

These requirements carry over to the smaller syntactic units and give the judgments in the system the general form

$$\mathit{environments} \overset{\mathit{judgement\ name}}{\vdash} \mathit{source\ phrase} \rightsquigarrow \mathit{target\ phrase} : \mathit{derived\ information}$$

where *environments* contain various contextual information, the *target phrase* is the translation of *source phrase* and the *derived information* may be the type of

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

source phrase (if it is an expression) or for instance one or more environments describing names defined in the phrase. If no translation of *source phrase* occurs in the target program, the *target phrase* is omitted, giving the following form of judgment:

$$environments \overset{judgement\ name}{\vdash} source\ phrase : derived\ information$$

This form is used for instance to derive a semantic type from a syntactic one.

Sometimes, when judgments are used in different contexts, not all parts of the judgment always have a meaningful function. In such situations we will write an "_" as a wildcard with the understanding that any value will do in this context.

2.7 Environments

There are several forms of environment used in the inference rules presented in the following sections. An environment is a set of pairs of the form *name* : *information* where *name* can be *e.g.* a variable and *information* is in general a tuple containing various pieces of information about *name*.

The environments are all formalized as sets but they will almost always have the (partial) function-like property that there should be at most one item which carries information about each name.

This motivates certain operations on environments. If E_1 and E_2 are environments, then

- $dom(E_1) = \{name \mid name : information \in E_1\}$ is the set of names in E_1,
- $E_1 \setminus names = \{name : information \mid name : information \in E_1 \wedge name \notin names\}$ is E_1 with the names in the set *names* removed,
- $E_1|_{names} = \{name : information \mid name : information \in E_1 \wedge name \in names\}$ is the part of E_1 which contains information about names in the set *names* only,
- $E_1 \oplus E_2$ is $E_1 \cup E_2$ with the side condition that $dom(E_1) \cap dom(E_2) = \emptyset$,
- $E_1 \bar{\oplus} E_2$ is $E_1 \cup E_2$ with the side condition that $E_1|_{names} = E_2|_{names}$ where $names = dom(E_1) \cap dom(E_2)$,
- $E_1 \vec{\oplus} E_2 = (E_1 \setminus dom(E_2)) \cup E_2$ is an asymmetric version of $\oplus$ where entries in E_2 hide entries with the same *name* in E_1,
- $E_1 \tilde{\oplus} E_2 = E_1 \cup E_2$ without any side conditions, so there may be multiple conflicting entries,
- $unQual(E_1) = \{unQual(name) : information \mid name : information \in E_1\}$ removes any qualifiers among the names in E_1,
- $justQs(E_1) = \{M.uname : information \mid M.uname : information \in E_1\}$ includes only the information about qualified names in E_1,
- $justSingle(E_1) = \{name : information \mid name : information \in E_1 \wedge (name : information' \in E_1 \Rightarrow information = information')\}$ only retains information about names with a single entry.

We will often use tuples of environments, and we extend the above operations componentwise to such tuples. Note that $\oplus$, $\bar{\oplus}$ and $\tilde{\oplus}$ have slightly different definitions on instance environments (see section 2.7.4).

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

2.7.1 The class environment

The *class environment*, ranged over by CE, contains information about type classes. This information is derived from class declarations and is used in instance declarations and contexts. An item in the class environment has the general form

$$C : \langle \Gamma, h, \mathsf{x}_{\mathsf{def}}, \alpha, IE_{sup} \rangle$$

where Γ is the annotated original name of the class, h is a positive integer used to express the acyclicality of the superclass relation, $\mathsf{x}_{\mathsf{def}}$ is the name of the default dictionary for the class, α is the class variable (occurs free in IE_{sup}) and IE_{sup} is an instance environment (see section 2.7.4) giving the names of the fields the superclasses are stored in in dictionaries for the class. There is an entry of the form $\mathsf{x} : \Gamma' \alpha$ in IE_{sup} for every superclass Γ' of Γ. Information about the types of the operations of the class is not stored in the class environment but in the top-level value environment. This simplifies the CLASS SOME rule on page 321 for selective import or export of only a subset of the class methods.

2.7.2 The type environment

The *type environment*, ranged over by TE, contains information about type constructors and type variables. The type constructor information is derived from type declarations in the program and the type variable information records in-scope type variables. The information is used to check type signatures, type declarations and instance declarations. An item in the type environment is of one of the general forms below:

$T : \chi$	Algebraic data type name
$T : \langle \chi, h, \Lambda\alpha_1 \ldots \alpha_k.\tau \rangle$	Type synonym
$u : \alpha$	Type variable

The name of an algebraic type constructor is associated with just the annotated original version of the name χ whereas a type synonym is also associated with a positive integer h (expressing the acyclicality of dependencies among type synonyms) and an expansion of the synonym (Λ is abstraction on the type level). A type variable is associated with its annotated counterpart α. See figure 4 for the syntax of types.

2.7.3 The data constructor environment

The *data constructor environment*, ranged over by DE, contains information about data constructors and named fields. The information is derived from algebraic data type declarations and is used to type constructors, field construction, field updates and constructor and field patterns. An item in the constructor environment is of one of the general forms below:

$K : \langle \mathsf{K}, \chi, \sigma \rangle$	Constructor
$x : \langle \mathsf{x}, \chi, LE \rangle$	Labelled field

Here K and x are the original names of K and x, respectively, χ is the annotated original name of the type constructor in whose declaration the data constructor K

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

is defined, σ is a type scheme giving the type of K, and LE records local contexts and labels with their types for all constructors which have the field x.

The *label environment*, ranged over by LE, contains information about named fields of constructors. The information is used when typing field updates, field construction and field patterns. An entry in the label environment is of the form

$$\mathsf{K} : \varphi$$

where φ is of the form $\forall\alpha_1 \ldots \alpha_k.\, \theta \Rightarrow UE \rightarrow \tau$. Here the α_i are the parameters of the algebraic type containing the constructor K, θ is the part of the context of the type pertaining to K and UE is an *update environment*, so called because it records the types in a field update. An entry in the update environment is of the form

$$\mathsf{x} : \tau$$

where x is a field name and τ a type. Both of these two forms of environment are looked up with original names, and they are unaffected by selective import, so they can contain information about constructors and fields which are not visible. Where needed, the data constructor environment is used to look up a source name to find the original name to use for accessing the label or update environments. See section 5.2.1 for further discussion of the typing of labeled fields.

2.7.4 The instance and overloading environments

The *instance environment*, ranged over by IE, contains information about which dictionary variable is bound to a dictionary of which instance of which class. The information in the instance environment is in part derived from instance declarations and in part related to dictionary variables bound in dictionary abstractions. The information is used to construct dictionaries for occurrences of overloaded variables and constants. An entry in the instance environment has one of the general forms below:

$\mathsf{v} : \Gamma\,(\alpha\ \tau_1 \ldots \tau_k)$	v is bound in a dictionary abstraction
$\mathsf{x} : \Gamma\,\alpha$	x represents a superclass in classinfo
$\mathsf{x} : \forall\alpha_1 \ldots \alpha_k\,.\,\theta \Rightarrow \Gamma\,(\chi\,\alpha_1 \ldots \alpha_k)$	x is a dictionary from an instance declaration
$\mathsf{x} : \forall\alpha\,.\,\Gamma'\,\alpha \Rightarrow \Gamma\,\alpha$	x extracts a dictionary for the superclass Γ

Note that v and x are variables that do not occur in the source program but only in the target program. In the last three cases above, x is bound to a dictionary or function returning a dictionary for some instance of the class Γ while v is bound in a dictionary abstraction.

We will write $\forall\alpha_1 \ldots \alpha_k.\, \theta \,\widetilde{\Rightarrow}\, IE$ for $\{\mathsf{x} : \forall\alpha_1 \ldots \alpha_k.\, \theta \Rightarrow \Gamma\,\tau \mid \mathsf{x} : \Gamma\,\tau \in IE\}$.

An *overloading environment* OE is an instance environment which only contains entries of the first kind above. We use overloading environments in rules which deal with dictionary abstractions; the context abstracted over must be the right-hand sides of an overloading environment.

One extra condition pertains to instance (and overloading) environments: Not only is such an environment ill-formed if the same *name* occurs with different *information*,

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

but it is also ill-formed if the same *information* occurs with different *names*. Thus we require that there is only one dictionary for each instance of each class. This rule also has consequences for the definition of the environment combining operators $\oplus$ and $\bar{\oplus}$ over instance and overloading environments; they get the implicit side condition that the resulting environment must have unique right hand sides. For these environments, $\tilde{\oplus}$ is defined to mean the same as $\bar{\oplus}$.

2.7.5 The variable environment

The *variable environment*, ranged over by VE, contains information about in-scope variables. The information comes from several different sources; algebraic type declarations with labeled fields define the field names as selector functions, class declarations introduce overloaded operations and ordinary binding constructs yield ordinary variables. The information is used to type variable occurrences in expressions as well as to find class operations in instance declarations. An entry in the variable environment has one of the general forms below:

$x : \langle \mathsf{x}, \sigma \rangle$	Field selectors and ordinary variables
$x : \langle \mathsf{x}, \forall\alpha\,.\,\Gamma\,\alpha \Rightarrow_c \sigma \rangle$	Class methods of class Γ

In both cases, x is the original name of x and σ is a type scheme. Note that a type scheme is necessary to the right of $\Rightarrow_c$ (which we distinguish syntactically from $\Rightarrow$ in order to mark class methods) since class operations may be polymorphic and overloaded in type variables other than α. An example of this is the `ceiling` method in the class `RealFrac` from the Haskell Prelude (discussed in section 6.4.6 of the Report). Its type will be recorded as

$$\forall a^*\,.\,\texttt{Prelude!RealFrac}^*\,a^* \Rightarrow_c \forall b^*\,.\,\texttt{Prelude!Integral}^*\,b^* \Rightarrow a^* \to b^*$$

in the variable environment.

We will write $\forall\alpha_1 \ldots \alpha_k\,.\,\theta \mathrel{\widetilde{\Rightarrow}} VE$ for $\{x : \forall\alpha_1 \ldots \alpha_k\,.\,\theta \Rightarrow \langle \mathsf{x}, \tau \rangle \mid x : \langle \mathsf{x}, \tau \rangle \in VE\}$ and similarly for $\widetilde{\Rightarrow_c}$.

2.7.6 The kind environment

The *kind environment*, ranged over by KE, contains information about the kinds of class and type names and type variables. It is used in the derivation of kinds for types and classes as well as when checking type signatures. An entry in the kind environment has the general form

$$name : \kappa$$

where *name* is C, T or u. See section 3 for more details of the Haskell kind system.

We will write $kindsOf(CE, TE)$ for the kind environment having the same kind information as the union of CE and TE. For classes, type names and type variables, the kind is taken from the semantic class name, type name or type variable in the environment, respectively.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

2.7.7 The source environment

The *source environment*, ranged over by SE, contains information about which names are imported from which modules. It is a tuple $\langle CS, TS, DS, VS \rangle$ of basic source environments containing information about class names, type names, constructor and field names and variable names. An entry in a basic source environment has the form

$$name : M$$

signifying that *name* is imported from module M (there are in general several entries with the same *name*). We will write $names : M$ for $\{name : M \mid name \in names\}$ indicating a source environment associating each *name* in *names* with M.

2.7.8 Environment tuples

The *global environment*, ranged over by GE, is a tuple of the form $\langle CE, TE, DE \rangle$ containing information from global declarations which is not affected (hidden or augmented) by bindings or expressions. Global environments are used in judgments for bindings, expressions and patterns.

The *full environment*, ranged over by FE, is a tuple $\langle CE, TE, DE, IE, VE \rangle$ and is used in module environments and in connection with imports and exports. The *entity environment*, ranged over by EE, is a tuple of the form $\langle CE, TE, DE, VE \rangle$ and is also used in this context.

2.7.9 The module environment

The *module environment*, ranged over by ME, contains the environments exported from each module. An entry in the module environment has the form

$$M : FE$$

where FE is the full environment exported by the module named M.

3 Kind inference

Haskell uses a system of *kinds* to classify type expressions much in the same way as a type system is used to classify expressions. Indeed, kinds are best thought of as "types for types". The kind system, seen as a type system, is very simple: There is only one base kind, written as $*$, and there are no kind variables and consequently no polymorphism. The grammar of kinds is:

$$\kappa \in \text{Kind} \rightarrow * \mid \kappa_1 \rightarrow \kappa_2$$

The base kind $*$ is the kind of ordinary types, like `Int`, `Char` or `[Char]`. A kind of the form $\kappa_1 \rightarrow \kappa_2$ represents a function from types to types. For instance, `[]` (the type constructor for lists) is of kind $* \rightarrow *$ since it represents a function from ordinary types to ordinary types.

Prior to type inference and resolution of overloading, *kind inference* is performed

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

in order to determine kinds for all type and class names as well as for all type variables. The kind inference system has two major groups of judgments:

1. $KE \overset{ktype}{\vdash} t : \kappa$. The type expression t has kind κ in kind environment KE.
2. $KE \overset{ksig}{\vdash} sig$. The signature *sig* is well-kinded in kind environment KE.

In both cases, KE is a kind environment as discussed in section 2.7.6.

3.1 Kind defaulting

There are some cases where unique kinds cannot be inferred. Consider a type signature

```
x :: a b
```

where the type variables `a` and `b` can be kinded as

$$\texttt{a} : \kappa \to * \qquad \texttt{b} : \kappa$$

for any kind κ. This situation is well-known from type inference where some terms can be assigned more than one type. Polymorphic type systems capture the sets of possible types using type variables and quantification, but the kind system of Haskell is monomorphic, so this option is not available. Instead, kinds are *defaulted* to $*$ in these situations, giving

$$\texttt{a} : * \to * \qquad \texttt{b} : *$$

in our example. A polymorphic kind system is quite possible, but was not deemed worth the added complexity during the design of Haskell 98.

Kind defaulting is not only used for type variables in signatures, but also for type and class names. Consider the algebraic type declaration below:

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
```

The type constructor `Tree` can be assigned kind $\kappa \to *$ for any kind κ, and is defaulted to $* \to *$. Since class declarations and algebraic type declarations can be mutually recursive, kind inference is performed one mutually recursive declaration group at a time.

The Report is not entirely explicit about what constitutes a mutually recursive declaration group since it does not detail what constitutes a dependence. We propose the following rule:

Definition 1 (*Kind dependence*)
Any occurrence of a class or type name within an algebraic data type declaration or type synonym declaration, except in a `deriving` clause[1], counts as a dependence. In a class declaration, occurrences in the superclass context or in the signatures

[1] Since `deriving` clauses may only refer to certain predefined classes, mutual dependence through `deriving` clauses is not an issue anyway.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

defining the class methods count, but occurrences in type signatures inside any default method bindings do not count.

The occurrences that count for dependency analysis are exactly those that are touched by the *kctDecl* judgement on page 314.

The rationale for this choice is that the occurrences that count contribute to the definition of the type or class, whereas the default methods and derived classes are logically related to instance declarations. This means that adding or removing a derived instance in an algebraic datatype declaration or a type signature in a default method binding does not affect kind inference. The following somewhat contrived example illustrates the difference:

```
class C a where
  op :: d a
  op = let x = undefined
           x :: C b => T b c
       in undefined x

data C a => T a b = MkT (a b)
```

The kind of `C` can be any κ, whereas the type `T` uses `C` in such a way that `C` must have a kind of the form $\kappa \to *$. By our definition of dependence, the type signature in the default method for `op` does not count, so the class declaration will be kinded first, defaulting the kind of `C` to $*$ which makes the type declaration ill-kinded. If the occurrence counts, on the other hand, the two declarations are kinded together and the kind of `C` is defaulted to $* \to *$ instead, eliminating the kind error. But in that case, removing the signature, removing the default method or even rewriting the class declaration as

```
class C a where
  op :: d a
  op = foo

foo = let x = undefined
          x :: C b => T b c
      in undefined x
```

would make the program fragment ill-kinded. We feel that such a simple syntactic change should not have such a subtle consequence.

3.1.1 Kind ordering

The above examples indicate that when several kinds are possible, we choose the kind that is in some sense the simplest one. We formalize this intuition using the relation $\prec$, defined by these two inference rules:

$$* \prec \kappa \qquad \frac{\kappa_1 \prec \kappa_1' \qquad \kappa_2 \prec \kappa_2'}{\kappa_1 \to \kappa_2 \prec \kappa_1' \to \kappa_2'}$$

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

The first rule says that $*$ is the simplest of all kinds and the second rule says that a function kind is simpler than another function kind if its argument and result kinds are simpler than those of the other. Note the covariance in both argument and result kinds in this definition. This ordering is extended to environments by the following rule:

$$\frac{\forall(name : \kappa) \in KE_1 . \exists \kappa' . (name : \kappa') \in KE_2 \wedge \kappa \prec \kappa'}{KE_1 \prec KE_2}$$

One consequence of this rule is that if $KE_1 \subseteq KE_2$, then $KE_1 \prec KE_2$.

We refer to this ordering when kinding groups of mutually recursive type and class declarations in the KCTDECLS rule on page 314 and when kinding signatures in the SIG rule on page 330. In these cases we seek the smallest kinding environment such that certain judgments can be derived. These premises are a bit special since there are no inference rules which allow us to infer this minimality; this is an additional proof obligation. In practice there are no particular problems since the main purpose of the inference rules in this paper is to be a specification of an inference algorithm, and the extra proof obligation can be taken care of when proving the correctness of that algorithm.

3.2 Kind inference rules

The kind inference rules are mostly straight forward. The KCTDECLS rule on page 314 traverses the nested lists of class or type declarations and applies kind defaulting to every mutually recursive group of declarations. The defaulting, expressed using the kind ordering, makes this judgment deterministic; given a kind environment KE and a declaration nest *ctDecls* there is at most one *KE'* such that

$$KE \overset{kctDecls}{\vdash} ctDecls : KE'.$$

In the KIND DATA and KIND TYPE rules (for algebraic type declarations and type synonyms), the kinds chosen for the type variables are reflected in the kind of the type name. Observe that an algebraic data type applied to all of its type parameters must have kind $*$ since it is the type of a data object, whereas a type synonym applied to all of its parameters may still have a higher kind. Looking at the rules for the *kcondecl* judgement on page 315 we see that the arguments to the constructors must all have kind $*$.

In the KIND CLASS rule, the context and signatures are checked in an environment extended with the kind of the class variable. Referring to the KIND SIG rule on page 315 we find that the kinding for the class variable is not hidden by the kindings of other type variables, ensuring that the same kind is assumed for the class variable in all of the signatures for the class operations. The other type variables are entirely local to the signature in which they appear and will be given minimal kinds when a semantic type is derived by the SIG rule on page 330.

The analogy of kinds as types for types is very striking in the definition of the

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{KE \overset{kctDecls}{\vdash} ctDecls : KE'}$$

$$\frac{KE_{decls} = \min\{KE' \mid KE \oplus KE' \overset{kgroup}{\vdash} ctDecl_1 ; \ldots ; ctDecl_n : KE'\} \qquad KE \oplus KE_{decls} \overset{kctDecls}{\vdash} ctDecls : KE_{groups}}{KE \overset{kctDecls}{\vdash} ctDecl_1 ; \ldots ; ctDecl_n \texttt{ then } ctDecls : KE_{decls} \oplus KE_{groups}} \quad \text{KCTDECLS}$$

$$KE \overset{kctDecls}{\vdash} \epsilon : \{\} \quad \text{KCTEMPTY}$$

$$\boxed{KE \overset{kgroup}{\vdash} ctDecl_1 ; \ldots ; ctDecl_n : KE'}$$

$$\frac{i \in [1,n] : KE \overset{kctDecl}{\vdash} ctDecl_i : KE_i}{KE \overset{kgroup}{\vdash} ctDecl_1 ; \ldots ; ctDecl_n : KE_1 \oplus \ldots \oplus KE_n} \quad \text{KGROUP}$$

$$\boxed{KE \overset{kctDecl}{\vdash} ctDecl : KE'}$$

$$\frac{\begin{array}{c}\kappa = \kappa_1 \to \ldots \to \kappa_k \to * \\ KE \oplus \{u_1 : \kappa_1, \ldots, u_k : \kappa_k\} \overset{kctx}{\vdash} cx \\ i \in [1,n] : KE \oplus \{u_1 : \kappa_1, \ldots, u_k : \kappa_k\} \overset{kconDecl}{\vdash} conDecl_i\end{array}}{KE \overset{kctDecl}{\vdash} \texttt{data } cx \texttt{ => } S\ u_1 \ldots u_k \texttt{ = } conDecl_1 \texttt{ | } \ldots \texttt{ | } conDecl_n \ : \ \{S : \kappa\}} \quad \text{KIND DATA}$$

$$\frac{KE \oplus \{u_1 : \kappa_1, \ldots, u_k : \kappa_k\} \overset{ktype}{\vdash} t : \kappa}{KE \overset{kctDecl}{\vdash} \texttt{type } S\ u_1 \ldots u_k \texttt{ = } t \ : \ \{S : \kappa_1 \to \ldots \to \kappa_k \to \kappa\}} \quad \text{KIND TYPE}$$

$$\frac{KE \oplus \{u : \kappa\} \overset{kctx}{\vdash} cx \qquad KE \oplus \{u : \kappa\} \overset{ksigs}{\vdash} sigs}{KE \overset{kctDecl}{\vdash} \texttt{class } cx \texttt{ => } B\ u \texttt{ where } sigs ; bind_1 ; \ldots ; bind_n \ : \ \{B : \kappa\}} \quad \text{KIND CLASS}$$

Fig. 8. Kind inference, top level declarations.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$\boxed{KE \overset{kconDecl}{\vdash} conDecl}$

$$\frac{i \in [1,k] : KE \overset{ktype}{\vdash} t_i : *}{KE \overset{kconDecl}{\vdash} J\ t_1 \ldots t_k} \quad \text{KIND POSCON}$$

$$\frac{i \in [1,k] : KE \overset{ktype}{\vdash} t_i : *}{KE \overset{kconDecl}{\vdash} J\ \{v_1 :: t_1, \ldots, v_k :: t_k\}} \quad \text{KIND LABCON}$$

$\boxed{KE \overset{ksigs}{\vdash} sigs}$

$$\frac{i \in [1,n] : KE \overset{ksig}{\vdash} sig_i}{KE \overset{ksigs}{\vdash} sig_1 ; \ldots ; sig_n} \quad \text{KIND SIGS}$$

$\boxed{KE \overset{ksig}{\vdash} sig}$

$$\frac{KE' = \{u_1 : \kappa_1, \ldots, u_n : \kappa_n\} \qquad KE \oplus KE' \overset{kctx}{\vdash} cx \qquad KE \oplus KE' \overset{ktype}{\vdash} t : *}{KE \overset{ksig}{\vdash} v :: cx \Rightarrow t} \quad \text{KIND SIG}$$

$\boxed{KE \overset{kctx}{\vdash} cx}$

$$\frac{\{C_1 : \kappa_1, \ldots, C_n : \kappa_n\} \subseteq KE \qquad i \in [1,n] : KE \overset{ktype}{\vdash} t_i : \kappa_i}{KE \overset{kctx}{\vdash} C_1\ t_1, \ldots, C_n\ t_n} \quad \text{KIND CTX}$$

Fig. 9. Kind inference, signatures and contexts.

kinding rules for type expressions given in figure 10. Note that type synonyms are not treated specially in kind inference.

4 Modules

A module consists of a module header, specifying the name of the module and the entities to be exported from the module, a set of imports specifying the entities to be imported from other modules, and a body consisting of top-level declarations.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{KE \overset{ktype}{\vdash} t : \kappa}$$

$$\frac{u : \kappa \in KE}{KE \overset{ktype}{\vdash} u : \kappa} \quad \text{KIND TVAR}$$

$$\frac{T : \kappa \in KE}{KE \overset{ktype}{\vdash} T : \kappa} \quad \text{KIND TCON}$$

$$\frac{KE \overset{ktype}{\vdash} t_1 : \kappa_1 \to \kappa_2 \qquad KE \overset{ktype}{\vdash} t_2 : \kappa_1}{KE \overset{ktype}{\vdash} t_1\, t_2 : \kappa_2} \quad \text{KIND APP}$$

Fig. 10. Kind inference, type expressions.

$$\boxed{ME \overset{module}{\vdash} mod \rightsquigarrow \mathsf{mod} : ME'}$$

$$\frac{\begin{array}{l} i \in [1,n] : ME \overset{import}{\vdash} imp_i : FE_i, SE_i \\ \quad FE_{imp} = justSingle(FE_1 \tilde{\oplus} \ldots \tilde{\oplus} FE_n) \\ \quad SE_{imp} = SE_1 \tilde{\oplus} \ldots \tilde{\oplus} SE_n \\ \quad M, FE_{imp} \overset{body}{\vdash} body \rightsquigarrow \mathsf{typeDecls; binds} : FE, SE \\ i \in [1,k] : FE_{imp} \vec{\oplus} [FE]_M, SE_{imp} \vec{\oplus} SE \overset{export}{\vdash} ent_i : FE'_i \\ \quad FE_{exp} = FE'_1 \bar{\oplus} \ldots \bar{\oplus} FE'_k \end{array}}{ME \overset{module}{\vdash} \left\{\begin{array}{l}\texttt{module}\ M\texttt{(}ent_1, \ldots, ent_k\texttt{)}\ \texttt{where} \\ imp_1; \ldots; imp_n; body\end{array}\right\} \rightsquigarrow \left\{\begin{array}{l}\texttt{module}\ M\ \texttt{where} \\ \mathsf{typeDecls; binds}\end{array}\right\} : \{M : FE_{exp}\}} \quad \text{MODULE}$$

Fig. 11. Modules.

4.1 The top level judgment

The *module* judgment is the root of this semantics. Given a module environment ME and a module *mod* it allows us to derive a target module mod and a module environment ME' containing information about the entities exported by *mod*.

The MODULE rule on page 316 deals with creating an environment FE_{imp}, to be used for typing the body of the module, from the environments exported from other modules. It also forms an environment FE_{exp} to be exported from the module. This

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

environment is built from the imported information (FE_{imp}) and the environment derived from the module body (FE).

The scoping rules for Haskell are rather complex, in particular the rules concerned with name clashes between imported entities. The imported environment FE_{imp} is formed by unioning the environments yielded by different `import` declarations, followed by discarding entries for names which occur multiple times with conflicting information. Since the original name of the entity is included in the information about it, this rule enforces the requirement from section 5.5.2 of the Report that a name may only be imported by multiple `import` declarations if all of the imports refer to the same original definition. If that is not the case, a *name clash* is said to occur, but this only makes the program invalid if the multiply-defined name is actually used.

The source environments SE and SE_{imp} indicate from which module(s) a name is imported. It is used to determine what information to export for an export entity of the form `module` M'. We use the operator $\bar{\oplus}$ to combine environments from different export entities. This allows the same name to be exported with the same information by different export entities, but ensures that the program is rejected if unique information can not be found. Note that the FE'_i only contain information about unqualified names since all qualifiers have been stripped by the *export* judgement. Any qualifiers will be added by the `import` declarations of importing modules.

The environment FE derived from the body of the module may contain unqualified (local) original names which can not be exported since the same local name may occur in other modules. Therefore, FE is *globalized* before being exported. We write this as $[FE]_M$ with the meaning that every local original name *name* occurring in FE is replaced by M `!`*name*. For most environments, these names occur only to the right, but in instance environments, label environments and update environments, they may also occur to the left. Whether occurring to the right or to the left, they are globalized in the same way. One way to think about globalization is to regard every unqualified name which occurs in a place where an original name is expected (these are the local original names) as implicitly qualified with ϵ, in which case $[FE]_M$ is FE with M substituted for ϵ. This corresponds to the requirement in section 5.5.3 of the Report that any entity needed for type checking must be automatically imported even if it is not explicitly imported.

4.2 Scoping of imported names

The scoping rules formalized in this paper deviates from those in the Report (section 5.3) in the following respects:

- In our semantics it is not possible to use qualified names to refer to top-level declarations and bindings in the same module. The following module is legal according to the Report but illegal in our semantics:

```
module Foo where
foo = 1
bar = Foo.foo
```

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{FE, SE \overset{export}{\vdash} ent : FE'}$$

$$\frac{\begin{array}{l} Cs = \{C \mid C : M \in CS\} \\ Ts = \{T \mid T : M \in TS\} \\ Ks = \{K \mid K : M \in DS\} \\ xs = \{x \mid x : M \in VS\} \cup \{x \mid x : M \in DS\} \\ FE = \langle CE|_{Cs}, TE|_{Ts}, DE|_{Ks \cup xs}, IE, VE|_{xs} \rangle \end{array}}{\langle CE, TE, DE, IE, VE \rangle, \langle CS, TS, DS, VS \rangle \overset{export}{\vdash} \texttt{module}\ M : unQual(FE)} \quad \text{EXPORT MODULE}$$

$$\frac{\langle CE, TE, DE, VE \rangle \overset{entity}{\vdash} ent : \langle CE', TE', DE', VE' \rangle}{\langle CE, TE, DE, IE, VE \rangle, SE \overset{export}{\vdash} ent : unQual(\langle CE', TE', DE', IE, VE' \rangle)} \quad \text{EXPORT ENTITY}$$

Fig. 12. Export.

- In our semantics, top-level bindings and declarations hide imported names whereas they give rise to name clashes according to the Report (although an error only occurs if the name is actually used). The following module is illegal according to the Report but legal in our semantics:

```
module Foo where
import Bar(bar)
bar = 1
foo = bar
```

We feel that our rule is simpler to understand and formalize and, in addition, a Haskell module that is legal according to the Report can be made legal in our semantics by replacing each reference *M .name* to the local top-level name *name* with *name* and renaming any non top-level binding of *name* in scope at the occurrence of *M .name*.

4.3 Import declarations

Haskell `import` declarations allow rather fine-grained control over what gets imported. The *import* judgement on page 319 fetches the environment exported by the imported module from the module environment. What actually gets imported is determined by a two-stage process. First the import list is consulted using the *implist* judgement. In the LIST SOME rule rule an explicit list of entities to import is used to filter the imported environment. Whether this filtering should also affect the qualified names is not entirely clear from the Report; we have chosen to filter them as well. In the HIDE SOME rule the import list contains entities which will not be imported from the module. In this case qualified imports are *not* affected. An

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{ME \overset{import}{\vdash} imp : FE, SE}$$

$$\frac{\begin{array}{l} M : \langle CE, TE, DE, IE, VE\rangle \in ME \\ M', \langle CE, TE, DE, VE\rangle \overset{implist}{\vdash} implist : EE \\ EE \overset{qualifier}{\vdash} qualifier : \langle CE', TE', DE', VE'\rangle \\ SE = \langle dom(CE'), dom(TE'), dom(DE'), dom(VE')\rangle : M \end{array}}{ME \overset{import}{\vdash} \texttt{import}\ qualifier\ M\ \texttt{as}\ M'\ implist : \langle CE', TE', DE', IE, VE'\rangle, SE} \quad \text{IMPORT}$$

$$\boxed{M, EE \overset{implist}{\vdash} implist : EE'}$$

$$\frac{\begin{array}{c} i \in [1, n] : EE \overset{entity}{\vdash} ent_i : EE_i \\ EE' = EE_1 \cup \ldots \cup EE_n \end{array}}{M, EE \overset{implist}{\vdash} (ent_1, \ldots, ent_n) : EE' \tilde{\oplus} M.EE'} \quad \text{LIST SOME}$$

$$\frac{\begin{array}{l} i \in [1, n] : EE \overset{entity}{\vdash} ent_i : EE_i \\ \langle CE, TE, DE, VE\rangle = EE \setminus (EE_1 \cup \ldots \cup EE_n) \\ Ks = \{K \mid K \in \{ent_1, \ldots, ent_n\}\} \\ EE' = \langle CE, TE, DE \setminus Ks, VE\rangle \end{array}}{M, EE \overset{implist}{\vdash} \texttt{hiding}\ (ent_1, \ldots, ent_n) : EE' \tilde{\oplus} M.EE'} \quad \text{HIDE SOME}$$

$$M, EE \overset{implist}{\vdash} \epsilon : EE \tilde{\oplus} M.EE \quad \text{ALL}$$

$$\boxed{EE \overset{qualifier}{\vdash} qualifier : EE'}$$

$$EE \overset{qualifier}{\vdash} \texttt{qualified} : justQs(EE) \quad \text{QUALIFIED}$$

$$EE \overset{qualifier}{\vdash} \epsilon : EE \quad \text{UNQUALIFIED}$$

Fig. 13. Import declarations.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

entity of the form K hides any data constructor with that name; otherwise data constructors only appear in import lists together with the name of the algebraic data type they are part of. Finally, the ALL rule imports the entire environment, both qualified and unqualified, exported by the imported module. Note that the exported environment of a module only contains unqualified names; the qualified names are added by the rules for the *implist* judgement.

The second stage of import processing deals with the possibility of only importing the qualified names. In the QUALIFIED rule for the *qualifier* judgement the unqualified names are removed from the imported environment.

In all cases, the instance environment is unaffected by the filtering performed by the *implist* judgement as well as the *qualifier* judgement.

4.4 Entities

Both import and export lists consist of *entities*. The *entity* judgement on page 321 filters out the information associated with a particular entity from an export environment. The filtering functions *constrs* and *fields* finds which constructors and fields of a given type χ are in the domain of the data constructor environment, and the *ops* function finds all methods of a given class Γ which are in the domain of the variable environment. Recall that variables may occur both in the variable environment and in the data constructor environment. In the latter case, they must be field names.

An entity of the form C`(..)` or T`(..)` refers to all in-scope methods or field and constructor names associated with the class or type. Entities with an explicit enumeration only refer to the enumerated names; it is an error if some of these names are not in scope. It is possible to mention some constructors, field names or methods in one entity in an import or export list, and also mention other constructors, field names or methods of the same type or class in another entity in the same list. This motivates the choice of not keeping information about the visibility of such names in the information recorded about the type or class. The consequences of this choice are apparent in the rule for instance declarations on page 332, among other places.

4.5 Module bodies

The BODY rule on page 322 takes the name of the module, the imported environment and the module body and yields the environment produced from the declarations and bindings in the body together with the data type declarations and bindings that form the translation of the body.

The rule applies kind inference, using the *kctDecls* judgement on page 314, to the type and class declarations of the module starting from kind information extracted from the imported class and type environments. The kind information derived in this premise must agree with the kind information in the environments CE' and TE' derived from the module's own declarations. Since kind inference, as embodied in the KCTDECLS rule on page 314, is deterministic, this definition corresponds to performing the kind inference before type inference.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{EE \overset{entity}{\vdash} ent : EE'}$$

$$\frac{x \in dom(VE)}{\langle CE, TE, DE, VE\rangle \overset{entity}{\vdash} x : \langle \{\}, \{\}, DE|_{\{x\}}, VE|_{\{x\}}\rangle} \quad \text{VAR ENT}$$

$$\frac{T : \chi \in TE \qquad xs \subseteq fields(DE, \chi) \qquad Ks \subseteq constrs(DE, \chi)}{\langle CE, TE, DE, VE\rangle \overset{entity}{\vdash} T\,(xs, Ks) : \langle \{\}, TE|_{\{T\}}, DE|_{xs \cup Ks}, VE|_{xs}\rangle} \quad \text{TYPE SOME}$$

$$\frac{T : \chi \in TE \qquad xs = fields(DE, \chi) \qquad Ks = constrs(DE, \chi)}{\langle CE, TE, DE, VE\rangle \overset{entity}{\vdash} T\,(..) : \langle \{\}, TE|_{\{T\}}, DE|_{xs \cup Ks}, VE|_{xs}\rangle} \quad \text{TYPE ALL}$$

$$\frac{T : \langle \chi, h, \Lambda\alpha_1 \ldots \alpha_k . \tau\rangle \in TE}{\langle CE, TE, DE, VE\rangle \overset{entity}{\vdash} T : \langle \{\}, TE|_{\{T\}}, \{\}, \{\}\rangle} \quad \text{TYPE SYN}$$

$$\frac{C : \langle \Gamma, h, \mathsf{x}_{\mathsf{def}}, \alpha, IE_{sup}\rangle \in CE \qquad xs \subseteq ops(VE, \Gamma)}{\langle CE, TE, DE, VE\rangle \overset{entity}{\vdash} C\,(xs) : \langle CE|_{\{C\}}, \{\}, \{\}, VE|_{xs}\rangle} \quad \text{CLASS SOME}$$

$$\frac{C : \langle \Gamma, h, \mathsf{x}_{\mathsf{def}}, \alpha, IE_{sup}\rangle \in CE \qquad xs = ops(VE, \Gamma)}{\langle CE, TE, DE, VE\rangle \overset{entity}{\vdash} C\,(..) : \langle CE|_{\{C\}}, \{\}, \{\}, VE|_{\{x_1, \ldots, x_k\}}\rangle} \quad \text{CLASS ALL}$$

$$fields(DE, \chi) = \{x \mid x : \langle \mathsf{x}, \chi', LE\rangle \in DE \wedge \chi' = \chi\}$$
$$constrs(DE, \chi) = \{K \mid K : \langle \mathsf{K}, \chi', \sigma\rangle \in DE \wedge \chi' = \chi\}$$
$$ops(VE, \Gamma) = \{x \mid x : \langle \mathsf{x}, \forall\alpha . \Gamma'\,\alpha \Rightarrow_c \sigma\rangle \in VE \wedge \Gamma' = \Gamma\}$$

Fig. 14. Import-export entities.

The top level environments GE_{top}, IE_{top} and VE_{top} are formed by combining the environments derived from the declarations and bindings in the module, the imported environments and the initial environment containing information about the special type and data constructors (given in figure 16).

The translated bindings represent bindings of default and instance dictionaries

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{M, FE \overset{body}{\vdash} body \rightsquigarrow \mathsf{typeDecls}; \mathsf{binds} : FE', SE}$$

$$\frac{\begin{array}{l} kindsOf(CE, TE) \overset{kctDecls}{\vdash} ctDecls : kindsOf(CE', TE') \\ GE_{top}, IE_{top}, VE_{top} \overset{ctDecls}{\vdash} ctDecls \rightsquigarrow \mathsf{typeDecls}; \mathsf{binds}_{\mathsf{tycl}} : \langle CE', TE', DE', IE', VE' \rangle \\ GE_{top}, IE_{top}, VE_{top} \overset{instDecls}{\vdash} instDecls \rightsquigarrow \mathsf{binds}_{\mathsf{inst}} : IE'' \\ GE_{top}, IE_{top}, VE \mathbin{\vec{\oplus}} VE' \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds}_{\mathsf{bds}} : VE'' \\ GE' = \langle CE', TE', DE' \rangle \\ GE_{top} = GE_{init} \oplus (\langle CE, TE, DE \rangle \mathbin{\vec{\oplus}} GE') \\ IE_{top} = IE \oplus IE' \oplus IE'' \\ VE_{top} = VE \mathbin{\vec{\oplus}} (VE' \oplus VE'') \\ FE = \langle CE', TE', DE', IE' \oplus IE'', VE' \oplus VE'' \rangle \\ SE = \langle dom(CE') : M, dom(TE') : M, dom(DE') : M, dom(VE') \cup dom(VE'') : M \rangle \\ \mathsf{binds} = \mathtt{rec}\ \mathsf{binds}_{\mathsf{tycl}}; \mathsf{binds}_{\mathsf{inst}}; \mathsf{binds}_{\mathsf{bds}} \end{array}}{M, \langle CE, TE, DE, IE, VE \rangle \overset{body}{\vdash} ctDecls; instDecls; binds \rightsquigarrow \mathsf{typeDecls}, \mathsf{binds} : FE, SE}\ \text{BODY}$$

Fig. 15. Module bodies.

$$\begin{array}{rl} GE_{init} = & \langle CE_{init}, TE_{init}, DE_{init} \rangle \\ CE_{init} = & \{\,\} \\ TE_{init} = & \{\mathtt{()} : \mathtt{()}^*, \mathtt{[]} : \mathtt{[]}^{*\to *}, \mathtt{(->)} : \mathtt{(->)}^{*\to *\to *}\} \\ & \cup \{\mathtt{(}k\mathtt{)} : \mathtt{(}k\mathtt{)}^{*_1\to \ldots \to *_k \to *} \mid k > 1\} \\ DE_{init} = & \{\mathtt{()} : \langle \mathtt{()}, \mathtt{()}^*, \mathtt{()}^* \rangle, \\ & \mathtt{[]} : \langle \mathtt{[]}, \mathtt{[]}^{*\to *}, \forall u^*.\, [u^*] \rangle, \\ & \mathtt{(:)} : \langle \mathtt{(:)}, \mathtt{[]}^{*\to *}, \forall u^*.\, u^* \to [u^*] \to [u^*] \rangle \\ & \} \\ & \cup \{\mathtt{(}k\mathtt{)} : \langle \mathtt{(}k\mathtt{)}, \mathtt{(}k\mathtt{)}^{*_1\to \ldots \to *_k \to *}, \forall u_1^* \ldots u_k^*.\, u_1^* \to \ldots \to u_k^* \to (u_1^*, \ldots, u_k^*) \rangle \\ & \quad \mid k > 1\} \end{array}$$

Fig. 16. The initial global environment.

as well as translations of the top-level *binds*. They are all joined in one mutually recursive group of bindings in the translated program since the dictionary bindings may refer to variables bound in the top-level bindings and the latter in general will refer to the dictionaries. Since the target program contains explicit type information, it is not necessary to maintain the dependency sorting present in the source program.

The rules for the *ctDecls* judgment are not very interesting and only traverse the nested structure which is necessary to get the right defaulting in the kind inference. The same environments are passed down to all of the type and class declarations.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{ctDecls}{\vdash} ctDecls \rightsquigarrow \mathsf{typeDecls}; \mathsf{binds} : FE}$$

$$\frac{\begin{array}{l} i \in [1,n] : GE, IE, VE \overset{ctDecl}{\vdash} ctDecl_i \rightsquigarrow \mathsf{typeDecls}_i; \mathsf{binds}_i : FE_i \\ GE, IE, VE \overset{ctDecls}{\vdash} ctDecls \rightsquigarrow \mathsf{typeDecls}; \mathsf{binds} : FE \\ \mathsf{typeDecls}' = \mathsf{typeDecls}; \mathsf{typeDecls}_1, \ldots, \mathsf{typeDecls}_n \\ \mathsf{binds}' = \mathsf{binds}; \mathsf{binds}_1, \ldots, \mathsf{binds}_n \\ FE' = FE \oplus FE_1 \oplus \ldots \oplus FE_n \end{array}}{GE, IE, VE \overset{ctDecls}{\vdash} ctDecl_1; \ldots; ctDecl_n \texttt{ then } ctDecls \rightsquigarrow \mathsf{typeDecls}'; \mathsf{binds}' : FE'} \text{ CTDECL}$$

$$GE, IE, VE \overset{ctDecls}{\vdash} \epsilon \rightsquigarrow \epsilon; \epsilon : \langle \{\}, \{\}, \{\}, \{\}, \{\} \rangle \quad \text{EMPTY CTDECL}$$

Fig. 17. Type and class declaration groups.

5 Types and type declarations

This section deals with type expressions and type declarations. The latter are top-level declarations and fall in three groups; `data`, `newtype` (omitted) and `type` declarations.

5.1 *Type synonym declarations*

Since type synonyms are expanded during type inference, Haskell does not allow type synonym declarations to be mutually recursive without an intervening algebraic datatype. We express this acyclicality using the integer h which is included in the information about a type synonym in the type environment. The height of the synonym S is constrained to be larger than the height of any synonym occurring in the expansion (right hand side) of S. The *ktype* premise only serves to find the kind κ for the expansion of the synonym.

5.2 *Algebraic datatype declarations*

Typing a `data` declaration yields a translated declaration as well as some environments. The type environment records the semantic name of the declared type, the constructor environment records information about the constructors and labelled fields of the type and the variable environment records the types of the field names when used as selector functions.

5.2.1 *Typing of labelled fields*

If the typing rule for algebraic datatype definitions looks forbidding, it is because of the indirect way the semantics of operations on datatypes with field labels is defined

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{TE, h \overset{type}{\vdash} t : \tau}$$

$$\frac{u : \alpha \in TE}{TE, h \overset{type}{\vdash} u : \alpha} \quad \text{TVAR}$$

$$\frac{T : \chi \in TE}{TE, h \overset{type}{\vdash} T : \chi} \quad \text{TCON}$$

$$\frac{T : \langle \chi, g, \Lambda\alpha_1 \ldots \alpha_k.\tau \rangle \in TE \qquad g < h \qquad i \in [1,k] : TE, h \overset{type}{\vdash} t_i : \tau_i}{TE, h \overset{type}{\vdash} T\ t_1 \ldots t_k\ :\ \tau[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]} \quad \text{TSYN}$$

$$\frac{TE, h \overset{type}{\vdash} t_1 : \tau_1 \qquad TE, h \overset{type}{\vdash} t_2 : \tau_2}{TE, h \overset{type}{\vdash} t_1\ t_2\ :\ \tau_1\ \tau_2} \quad \text{TAPP}$$

Fig. 18. Type expressions.

in the Report; by translating operations involving fields into operations using the underlying positional constructors, which have their semantics specified explicitly. For this reason, we will discuss an example of a data type with field labels, to point out some of the issues before tackling the inference rules.

```
data (Monad m, Eq (m a)) => Foo a m = ConA {x :: a, w :: Int}
                                    | ConB {y :: m Int, w :: Int}
                                    | ConC {z :: m a}
```

A `data` declaration may contain a context *cx*. Any type variables occurring in *cx* must be parameters of the `data` declaration. This context will affect creation of objects of the declared type through the types derived for the constructors by the *conDecl* judgement on page 327. Each constructor *J* acquires the part of the context *cx* that mentions only variables in the types of the arguments to *J*. This context is used both for construction and for pattern matching. The contexts associated with the constructors above is empty for `ConA`, `Monad m` for `ConB` and `(Monad m, Eq (m a))` for `ConC`.

The above declaration defines exactly the same type and the same constructor functions as the corresponding declaration without labels:

```
data (Monad m, Eq (m a)) => Foo a m = ConA a Int
                                    | ConB (m Int) Int
                                    | ConC (m a)
```

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

In addition, it becomes possible to use the labels `x`, `y`, `z` and `w` as selector functions with the following types:

$$\begin{aligned}
&\texttt{x} : \forall\, a\ m.\, \texttt{Foo}\ a\ m \rightarrow \texttt{Int}\\
&\texttt{y} : \forall\, a\ m.\, \texttt{Monad}\ m \texttt{=>} \texttt{Foo}\ a\ m \rightarrow m\ \texttt{Int}\\
&\texttt{z} : \forall\, a\ m.\, (\texttt{Monad m, Eq (m a)}) \texttt{=>} \texttt{Foo}\ a\ m \rightarrow m\ a\\
&\texttt{w} : \forall\, a\ m.\, \texttt{Monad}\ m \texttt{=>} \texttt{Foo}\ a\ m \rightarrow \texttt{Int}
\end{aligned}$$

These types, and in particular the contexts, are motivated by their translation to selector functions using pattern matching. For example, here is the definition of `w`:

```
w v = case v of
        ConA a b -> b
        ConB a b -> b
        _        -> undefined
```

In general, the `case` will enumerate the constructors which have the selected field (in this case `w`). Here we see that `ConA`, which has no context, and `ConB`, which has context `Monad` *m*, are mentioned, yielding the context `Monad` *m* for the type of the selector function.

In addition to selector functions, constructors with labelled fields provide us with special syntax for construction and non destructive update of objects built with these constructors. The labelled constructions are fairly straightforward, but the update construct has extremely tricky typing rules. Using the above type, we can write for instance:

```
aFoo {x = 'A'}
```

This will translate into a `case` expression with one branch for every constructor containing all of the fields in the update (in this case just `x`):

```
case aFoo of
  ConA a b -> ConA 'A' b
  _        -> undefined
```

If we encountered this expression during type inference, we would conclude that the variable `aFoo` must have some type `Foo` τ_1 τ_2 with an empty context (since `ConA` has an empty context) and the expression as a whole would have type `Foo Char` τ_3. These two types are not the same, which has the implication that the expression `aFoo {x = 'A'}` does not necessarily have the same type as the variable `aFoo`.

Sometimes the types *are* the same, though. Consider instead `aFoo {w = 3}` which translates into

```
case aFoo of
  ConA a b -> ConA a 3
  ConB a b -> ConB a 3
  _        -> undefined
```

where the first arguments of the two constructors are copied and thus force the type of the result of the expression the be the same as the type of `aFoo`. Given these subtle differences in typing, it should not come as a surprise that the typing rules for algebraic data types essentially have to anticipate the translation.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$GE, IE, VE \overset{ctDecl}{\vdash} ctDecl \rightsquigarrow \mathsf{typeDecls}; \mathsf{binds} : \langle CE', TE', KE', IE', VE' \rangle$$

$$\begin{array}{ll}
 & \chi = S^{\kappa_1 \to \ldots \to \kappa_k \to *} \\
i \in [1,k] : & \alpha_i = u_i^{\kappa_i} \\
 & TE' = \{u_1 : \alpha_1\} \oplus \ldots \oplus \{u_k : \alpha_k\} \\
 & CE, TE \oplus TE', _ \overset{context}{\vdash} cx : \theta \\
i \in [1,n] : & TE \oplus TE', \theta, \tau \overset{conDecl}{\vdash} conDecl_i \rightsquigarrow \mathsf{conDecl}_i : DE_i, VE_i, LE_i, \theta_i \\
 & DE' = DE_1 \oplus \ldots \oplus DE_n \oplus DE_{fields} \\
 & \{v_1 : \langle \mathsf{v}_1, \tau_1 \rangle, \ldots, v_m : \langle \mathsf{v}_m, \tau_m \rangle\} = VE_1 \bar{\oplus} \ldots \bar{\oplus} VE_n \\
i \in [1,m] : & LE'_i = \cup \{LE_j \mid v_i \in dom(VE_j)\} \\
 & DE_{fields} = \{v_1 : \langle \mathsf{v}_1, \chi, LE'_1 \rangle, \ldots, v_m : \langle \mathsf{v}_m, \chi, LE'_m \rangle\} \qquad \text{DATA DECL} \\
i \in [1,m] : & \sigma_i = \forall \alpha_1 \ldots \alpha_k. \sqcup \{\theta_j \mid v_i \in dom(VE_j)\} \Rightarrow \tau \to \tau_i \\
 & VE' = \{v_1 : \langle \mathsf{v}_1, \sigma_1 \rangle, \ldots, v_m : \langle \mathsf{v}_m, \sigma_m \rangle\} \\
 & \tau = \chi\, \alpha_1 \ldots \alpha_k \\
 & GE = \langle CE, TE, DE \rangle
\end{array}$$

$$\begin{array}{l}
GE, IE, VE \overset{ctDecl}{\vdash} \mathtt{data}\ cx \Rightarrow S\ u_1 \ldots u_k = conDecl_1 \mid \ldots \mid conDecl_n \\
\quad \rightsquigarrow \mathtt{data}\ \chi\, \alpha_1 \ldots \alpha_k = \mathsf{conDecl}_1 \mid \ldots \mid \mathsf{conDecl}_n ; \epsilon \\
\quad : \langle \{\}, \{S : \chi\}, DE', \{\}, VE' \rangle
\end{array}$$

$$\begin{array}{ll}
 & kindsOf(\{\}, TE \oplus TE_1 \oplus \ldots \oplus TE_k) \overset{ktype}{\vdash} t : \kappa \\
 & TE \oplus TE_1 \oplus \ldots \oplus TE_k, h \overset{type}{\vdash} t : \tau \\
i \in [1,k] : & TE_i = \{u_i : u_i^{\kappa_i}\} \\
 & TE' = \{S : \langle S^{\kappa_1 \to \ldots \to \kappa_k \to \kappa}, h, \Lambda u_1^{\kappa_1} \ldots u_k^{\kappa_k}.\, \tau \rangle\} \qquad \text{TYPE DECL} \\
 & GE = \langle CE, TE, DE \rangle
\end{array}$$

$$GE, IE, VE \overset{ctDecl}{\vdash} \mathtt{type}\ S\ u_1 \ldots u_k = \tau \rightsquigarrow \epsilon; \epsilon : \langle \{\}, TE', \{\}, \{\}, \{\} \rangle$$

Fig. 19. Type declarations.

5.2.2 *Formalizing algebraic datatype declarations*

With the above remarks in mind, we will now discuss the formalization. The type environment TE' records the kinds of the type parameters. These kinds are constrained by the kinds derived by kind inference and recorded for the type S in TE. The type parameters in a `data` declaration must be distinct, hence the use of $\oplus$ to combine the type environments corresponding to the individual type parameters.

The constructor declarations are processed by the *conDecl* judgement which yields a data constructor environment DE containing information about the constructor and a variable environment VE, a label environment LE and a context θ which are used to construct the information about the labelled fields of the constructor. The variable environment simply maps the labels to their original names and types. Combining the VE_i from different constructor declarations using $\bar{\oplus}$ (in the DATA

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{TE, \theta, \tau \overset{conDecl}{\vdash} conDecl \rightsquigarrow \mathsf{conDecl} : DE, VE, LE, \theta}$$

$$\frac{\begin{array}{l} i \in [1,n] : TE, \_ \overset{type}{\vdash} t_i : \tau_i \\ \sigma = \forall \alpha_1 \ldots \alpha_k . \theta' \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \chi\, \alpha_1 \ldots \alpha_k \\ DE = \{J : \langle J, \chi, \sigma \rangle\} \\ \theta' = \theta\,|_{\tau_1, \ldots, \tau_n} \end{array}}{TE, \theta, \chi\, \alpha_1 \ldots \alpha_k \overset{conDecl}{\vdash} J\ t_1 \ldots t_n \rightsquigarrow J\ \tau_1 \ldots \tau_n\ :\ DE, \{\}, \{\}, \theta'} \quad \text{POSCON}$$

$$\frac{\begin{array}{l} i \in [1,n] : TE, \_ \overset{type}{\vdash} t_i : \tau_i \\ DE = \{J : \langle J, \chi, \sigma \rangle\} \\ VE = \{v_1 : \langle v_1, \tau_1 \rangle\} \oplus \ldots \oplus \{v_n : \langle v_n, \tau_n \rangle\} \\ LE = \{J : \forall \alpha_1 \ldots \alpha_k . \theta' \Rightarrow \{v_1 : \tau_1, \ldots, v_n : \tau_n\} \rightarrow \tau\} \\ \sigma = \forall \alpha_1 \ldots \alpha_k . \theta' \Rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \\ \theta' = \theta\,|_{\tau_1, \ldots, \tau_n} \\ \tau = \chi\, \alpha_1 \ldots \alpha_k \end{array}}{TE, \theta, \tau \overset{conDecl}{\vdash} J\ \{v_1 :: t_1, \ldots,\ v_n :: t_n\} \rightsquigarrow J\ \{v_1 : \tau_1, \ldots,\ v_n : \tau_n\}\ :\ DE, VE, LE, \theta} \quad \text{LABCON}$$

Fig. 20. Type declarations continued; constructor declarations.

$$\begin{array}{l} \mathtt{x} : \langle \mathtt{x}, \mathtt{Foo}, \{\mathtt{ConA} : \forall\, a\ m . \{\mathtt{x} : a, \mathtt{w} : \mathtt{Int}\} \rightarrow \mathtt{Foo}\ a\ m\} \rangle \\ \mathtt{y} : \langle \mathtt{y}, \mathtt{Foo}, \{\mathtt{ConB} : \forall\, a\ m . \mathtt{Monad}\ m \Rightarrow \{\mathtt{y} : m\ \mathtt{Int}, \mathtt{w} : \mathtt{Int}\} \rightarrow \mathtt{Foo}\ a\ m\} \rangle \\ \mathtt{z} : \langle \mathtt{z}, \mathtt{Foo}, \{\mathtt{ConC} : \forall\, a\ m . (\mathtt{Monad}\ m, \mathtt{Eq}\ (m\ a)) \Rightarrow \{\mathtt{z} : m\ a\} \rightarrow \mathtt{Foo}\ a\ m\} \rangle \\ \mathtt{w} : \langle \mathtt{w}, \mathtt{Foo}, \{\mathtt{ConA} : \forall\, a\ m . \{\mathtt{x} : a, \mathtt{w} : \mathtt{Int}\} \rightarrow \mathtt{Foo}\ a\ m, \\ \qquad\qquad \mathtt{ConB} : \forall\, a\ m . \mathtt{Monad}\ m \Rightarrow \{\mathtt{x} : m\ \mathtt{Int}, \mathtt{w} : \mathtt{Int}\} \rightarrow \mathtt{Foo}\ a\ m\} \rangle \end{array}$$

Fig. 21. Examples of information about labelled fields.

DECL rule) means that different constructors may have identically named fields if they have identical type. The label environments LE_i each contain information about one constructor and are used to construct the LE'_j which contain, for each field label, information about each constructor having a field with that label. For each constructor, the types of the labels are given together with the type of the constructed value. The contexts θ_i, which are the same contexts that go into the types of the constructors in DE_i, are used for getting the right context in the selector functions using $\sqcup$ as a union operator on contexts. Figure 21 gives the entries for the field labels added to the global data constructor environment for the example data type `Foo`. Note how, in each case, the constructors included are those that would be used in the generated selector functions. In this example we have omitted the kind annotations and we have not used original names for entities imported from the Standard Prelude.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{IE, \varphi \overset{lcon}{\vdash} \tau_{old}, UE, \tau_{new}}$$

$$\frac{\begin{array}{l} UE[\tau'_1/\alpha_1, \ldots, \tau'_k/\alpha_k] = UE[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k] \dot{\oplus} UE' \\ IE \overset{dict}{\vdash} \_ : \theta[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k] \\ IE \overset{dict}{\vdash} \_ : \theta[\tau'_1/\alpha_1, \ldots, \tau'_k/\alpha_k] \end{array}}{IE, \forall \alpha_1 \ldots \alpha_k . \theta \Rightarrow UE \to \tau \overset{lcon}{\vdash} \tau[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k], UE', \tau[\tau'_1/\alpha_1, \ldots, \tau'_k/\alpha_k]} \quad \text{LCON}$$

Fig. 22. Instantiation for constructors and labelled fields.

We also use the *lcon* judgement defined in figure 22 when typing expressions and patterns involving labelled fields. This judgment implements the instantiation of the universal quantification in the φ associated with one constructor in the label environment. It also arranges for the type of the updated value and the result of the update to be suitably similar depending on which fields are updated. There are two reasons why these types need not be identical. First, some type parameters may not occur in the type of any field of the constructor, in which case the type variable does not occur in UE. This is the case for m in the example update `aFoo {x = 'A'}` since only `ConA` has the field `x` and m does not appear in the type of any field of `ConA`. Second, all fields which mention a type parameter might be assigned a new value in the update. This is formalized using the $\dot{\oplus}$ operator; types in the update hide those in the original value. Again this is the case in the example, for a this time, since the field `x` is the only field whose type mentions a in any constructor which has a field named `x`.

This judgment also checks that the instance environment entails the context of the constructor, both when used in pattern matching with the type of the value being updated and when it is used to construct the new value with the new instance. See also the UPD rule on page 347 for an example of how the *lcon* judgment is used. Labelled construction (the LABCON rule on page 347) and labelled patterns (the PLAB rule on page 351) also use this judgment, but do not use all of its features.

6 Class and instance declarations

In this section we present the rules concerning the class system, including rules for `class` and `instance` declarations, type signatures (since they are used in `class` declarations), contexts, method bindings and dictionary construction.

6.1 Class declarations

A class declaration is translated to an algebraic datatype declaration for the type of dictionaries for the class and a function constructing a dictionary containing the default methods, to be used in instance declarations. Three environments are

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$GE, IE, VE \overset{ctDecl}{\vdash} ctDecl \rightsquigarrow \mathsf{typeDecls}; \mathsf{binds} : FE$$

$$\begin{array}{l}
CE, \{u : \alpha\}, h \overset{context}{\vdash} cx : \theta \\
IE'_{sup} = \mathsf{vs} \,\tilde{:}\, \theta \\
IE_{sup} = \forall \alpha . \Gamma\, \alpha \,\tilde{\Rightarrow}\, IE'_{sup} \\
\langle CE, TE \cup \{u : \alpha\}, DE \rangle \overset{sigs}{\vdash} sigs : VE_{sigs} \\
i \in [1,n] : GE, IE \oplus \{\mathsf{v_d} : \Gamma\, \alpha\}, VE \overset{method}{\vdash} bind_i \rightsquigarrow \mathsf{fbind}_i : VE_i \\
VE_1 \oplus \ldots \oplus VE_n \subseteq VE_{sigs} \\
\alpha = u^{\kappa} \\
\Gamma = B^{\kappa} \\
CE' = \{B : \langle \Gamma, h, \mathsf{v_{def}}, \alpha, IE'_{sup} \rangle\} \\
VE' = \forall \alpha . \Gamma\, \alpha \,\tilde{\Rightarrow}_{\mathsf{c}}\, VE_{sigs} \\
GE = \langle CE, TE, DE \rangle \\
\mathsf{J_{dict}}, \mathsf{vs}, \mathsf{v_{def}} \text{ fresh}
\end{array} \qquad \text{CLASS DECL}$$

$$\begin{array}{l}
GE, IE, VE \overset{ctDecl}{\vdash} \mathtt{class}\ cx\ \mathtt{=>}\ B\ u\ \mathtt{where}\ sigs; bind_1; \ldots; bind_n \\
\quad \rightsquigarrow \left\{ \begin{array}{l} \mathtt{data}\ \widehat{\Gamma}\, \alpha\ \mathtt{=}\ \mathsf{J_{dict}}\, \{IE'_{sup}, VE_{sigs}\}; \\ \mathsf{v_{def}} : (\forall \alpha . \widehat{\Gamma}\, \alpha \rightarrow \widehat{\Gamma}\, \alpha) \\ \qquad \mathtt{=}\ \Lambda \alpha . \lambda \mathsf{v_d} : (\widehat{\Gamma}\, \alpha) . \mathsf{J_{dict}}\, \alpha\, \{\mathsf{fbind}_1, \ldots, \mathsf{fbind}_n\} \end{array} \right\} \\
\quad : \langle CE', \{\,\}, \{\,\}, IE_{sup}, VE' \rangle
\end{array}$$

Fig. 23. Class declarations

also returned: A class environment containing information about the defined class, an instance environment indicating how superclasses may be accessed and a value environment giving types to the class operations.

The context cx in a class declaration gives the (immediate) superclasses of the class. The class assertions in cx all have the form $C\, u$ where u is the class variable (no type name or other type variable is included in the type environment used to check cx). The superclass relation must be acyclic, a requirement enforced by the integer h in the *context* judgment; all classes mentioned in cx must have $h' < h$ recorded in the class environment.

The instance environment IE'_{sup} associates a fresh target variable with each superclass. These variables (vs) become the field names for the superclass fields in dictionaries for the class, as IE'_{sup} is used in the generation of the new type declaration. The returned instance environment IE_{sup} is formed by adding the context $\Gamma\, \alpha$ (where Γ is the semantic name of the declared class) to every item in IE'_{sup} and quantifying over α. Since the class variable u is included in the type environment used to check the superclass context cx, α can occur free in θ.

The class operations and their types are given by the signatures in the class declaration. These are checked by the *sigs* judgment which returns a variable environment, the semantic counterpart to a set of type signatures. This premise also ensures that the class variable u (α) must occur in the type part of each signature but not in any

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE \overset{sigs}{\vdash} sigs : VE}$$

$$\frac{i \in [1,n] : GE \overset{sig}{\vdash} sig_i : VE_i}{GE \overset{sigs}{\vdash} sig_1 ; \ldots ; sig_n : VE_1 \oplus \ldots \oplus VE_n} \quad \text{SIGS}$$

$$\boxed{GE \overset{sig}{\vdash} sig : VE}$$

$$\frac{\begin{array}{l} KE = kindsOf(TE, CE) \\ \{u_1 : \kappa_1, \ldots, u_k : \kappa_k\} = \min\{KE' \mid KE \oplus KE' \overset{kctx}{\vdash} cx \ \wedge \ KE \oplus KE' \overset{ktype}{\vdash} t : *\} \\ CE, TE \oplus \{u_1 : u_1^{\kappa_1}, \ldots, u_k : u_k^{\kappa_k}\}, \_ \overset{context}{\vdash} cx : \theta \\ TE \oplus \{u_1 : u_1^{\kappa_1}, \ldots, u_k : u_k^{\kappa_k}\}, \_ \overset{type}{\vdash} t : \tau \\ fv(cx) \subseteq fv(t) \\ \{u \mid u \in dom(TE)\} \subseteq fv(t) \setminus fv(cx) \end{array}}{\langle CE, TE, DE \rangle \overset{sig}{\vdash} v :: cx \Rightarrow t : \{v : \langle v, \forall u_1^{\kappa_1} \ldots u_k^{\kappa_k} . \theta \Rightarrow \tau \rangle\}} \quad \text{SIG}$$

Fig. 24. Type signatures.

context part, and that it may not be generalized over in VE_{sigs}. The requirement that the class variable must occur in the types disallows class operations which can not be used without causing ambiguity.

The algebraic datatype declaration generated introduces the the type of dictionaries of the class. Explicitly defining such a type has the advantage that types in the target program become similar to the types in the source program. For instance, the `elem` function from the Haskell Prelude has the type $\forall a.\, \texttt{Eq}\ a \Rightarrow a \rightarrow [a] \rightarrow \texttt{Bool}$ whereas its translation will have type $\forall a^*.\, \texttt{EqD}^{*\rightarrow *}\ a^* \rightarrow a^* \rightarrow [a^*] \rightarrow \texttt{Bool}^*$ where `EqD` is the name of the type for `Eq` dictionaries (ignoring original names).

The declaration uses the labelled field syntax of the target language, which carries over from that of the source. The methods of the class become field labels for the corresponding fields of the dictionary. Occurrences of the methods will be translated to applications of these selector functions to dictionaries. The superclasses will also be associated with field labels, but these labels (`vs`) are fresh and do not occur in the source program. The association between field label and superclass is recorded in IE_{sup} and IE'_{sup}.

The $bind_i$ part of a class declaration gives default implementations of the class methods, to be used when an `instance` declaration does not give implementations for all class methods. These default methods are collected in a default dictionary function $\mathsf{v}_{\mathsf{def}}$ which, given a dictionary for a particular instance of the class, returns

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{CE, TE, h \overset{context}{\vdash} cx : \theta}$$

$$\frac{i \in [1,n] : CE, TE, h \overset{class}{\vdash} class_i : \Gamma_i\ \tau_i}{CE, TE, h \overset{context}{\vdash} (class_1, \ldots, class_n) : \Gamma_1\ \tau_1, \ldots, \Gamma_n\ \tau_n} \quad \text{CONTEXT}$$

$$\boxed{CE, TE, h \overset{class}{\vdash} class : \Gamma\ \tau}$$

$$\frac{\begin{array}{l} C : \langle \Gamma, h', \mathsf{x}, \alpha, IE_{sup} \rangle \in CE \\ h' < h \\ TE, \_ \overset{type}{\vdash} u\ t_1 \ldots t_k : \tau \end{array}}{CE, TE, h \overset{sclass}{\vdash} C\ (u\ t_1 \ldots t_k) : \Gamma\ \tau} \quad \text{CLASS}$$

Fig. 25. Validation of contexts used in class and instance declarations.

a dictionary containing the default methods. The bindings are typed in an instance environment associating the formal parameter of this dictionary function with $\Gamma\ \alpha$. See also the discussion of the use of this function in the code produced from an instance declaration.

6.2 Type signatures

The judgment for type signatures, given in figure 24, is used when typing class declarations and binding groups. For its former use, it has extra functionality for checking that the class variable is handled in the right way. It must occur in the type but may not occur in the context. Any variable in the domain of the type environment TE is considered to be the class variable; when typing bindings, the type environment will only contain information about type names.

Type signatures also require kind inference since they introduce new type variables by quantification. These type variables can be defaulted independently for each signature, except for the class variable, which already is associated with a kind in the type environment.

6.3 Contexts

Figure 25 shows the rules for validating contexts. The integer h in the judgments is used to express the acyclicality of the superclass relation.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{instDecls}{\vdash} instDecls \rightsquigarrow \mathsf{binds} : IE'}$$

$$\frac{i \in [1,n] : GE, IE, VE \overset{instDecl}{\vdash} instDecl_i \rightsquigarrow \mathsf{binds}_i : IE_i}{GE, IE, VE \overset{instDecls}{\vdash} \left\{ \begin{array}{l} instDecl_1 ; \\ \ldots ; \\ instDecl_n \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \mathsf{binds}_1 ; \\ \ldots ; \\ \mathsf{bind}_n \end{array} \right\} : IE_1 \oplus \ldots \oplus IE_n} \quad \text{INST DECLS}$$

$$\boxed{GE, IE, VE \overset{instDecl}{\vdash} instDecl \rightsquigarrow \mathsf{binds} : IE'}$$

$$\frac{\begin{array}{l} T : \chi \in TE \\ i \in [1,k] : \ \alpha_i = u_i^{\kappa_i} \\ C : \langle \Gamma, h, \mathsf{x}_{\mathsf{def}}, \alpha, IE_{sup} \rangle \in CE \\ CE, \{u_1 : \alpha_1\} \oplus \ldots \oplus \{u_k : \alpha_k\}, \_ \overset{context}{\vdash} cx : \theta \\ i \in [1,m] : GE, IE \oplus \mathsf{vs} \tilde{:} \theta, VE \overset{method}{\vdash} bind_i \rightsquigarrow \mathsf{fbind}_i : VE_i \\ VE_{ops}[\chi\, \alpha_1 \ldots \alpha_k / \alpha] = VE_1 \oplus \ldots \oplus VE_m \\ (\forall \alpha . \Gamma\, \alpha \tilde{\Rightarrow}_c VE_{ops}) \subseteq VE \\ (\mathsf{x}_1, \ldots, \mathsf{x}_n) \tilde{:} \theta_{sup} = IE_{sup} \\ IE \oplus \mathsf{vs} \tilde{:} \theta \overset{dict}{\vdash} (\mathsf{e}_1, \ldots, \mathsf{e}_n) : \theta_{sup}[\chi\, \alpha_1 \ldots \alpha_k / \alpha] \\ GE = \langle CE, TE, DE \rangle \\ IE_{inst} = \{\mathsf{v}_{\mathsf{dict}} : \forall \alpha_1 \ldots \alpha_k . \theta \Rightarrow \Gamma\, (\chi\, \alpha_1 \ldots \alpha_k)\} \\ \mathsf{vs}, \mathsf{v}_{\mathsf{dict}} \text{ fresh} \end{array}}{GE, IE, VE \overset{instDecl}{\vdash} \texttt{instance}\ cx\ \texttt{=>}\ C\ (T\, u_1 \ldots u_k)\ \texttt{where}\ bind_1 ; \ldots ; bind_m \rightsquigarrow \left\{ \begin{array}{l} \mathsf{v}_{\mathsf{dict}} : \forall \alpha_1 \ldots \alpha_k . \widehat{\theta} \rightarrow \widehat{\Gamma}\, (\chi\, \alpha_1 \ldots \alpha_k) \\ \quad = \Lambda \alpha_1 \ldots \alpha_k . \lambda \mathsf{vs} \hat{:} \theta . \\ \quad\ \texttt{let rec}\ \mathsf{v}_{\mathsf{d}} : \widehat{\Gamma}\, (\chi\, \alpha_1 \ldots \alpha_k) \\ \qquad\qquad = (\mathsf{x}_{\mathsf{def}}\, (\chi\, \alpha_1 \ldots \alpha_k)\, \mathsf{v}_{\mathsf{d}})\ \{\mathsf{x}_1 = \mathsf{e}_1, \ldots, \mathsf{x}_n = \mathsf{e}_n, \\ \qquad\qquad\qquad\qquad \mathsf{fbind}_1, \ldots, \mathsf{fbind}_m\} \\ \quad\ \texttt{in}\ \mathsf{v}_{\mathsf{d}} \end{array} \right\} : IE_{inst}} \quad \text{INST DECL}$$

Fig. 26. Instance declarations.

6.4 Instance declarations

An instance declaration is translated into a binding for a dictionary function which, given dictionaries as defined in the context of the instance declaration, constructs a dictionary for the declared instance. An instance environment associating this function with the class and type of the instance declaration is also returned.

The rule for instance declarations is one of the most complex rules in the system. The first premise validates the instance type $T\, u_1 \ldots u_k$, checking that T is not a

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

type synonym. The instance type also defines which type variables may appear in the instance context cx, which is validated, ignoring the height part of the judgment which is only interesting in class declarations. The method bindings are typed in an extended instance environment which also includes the assertions about the u_i expressed by cx. The dictionary function $\mathsf{v}_{\mathsf{dict}}$ produced from the instance declaration should be passed dictionaries corresponding to these assertions.

Because of selective import of class operations, some operations of an imported class may not be visible, or only visible in qualified form. Since an instance declaration may only provide bindings for in-scope methods, the method bindings are checked against the global variable environment. This checking is accomplished in two steps: First, the information derived from the method bindings is checked against an instantiation of a hypothetical variable environment VE_{ops} which gives the same types to the methods as VE_{sigs} does in the CLASS DECL rule on page 329. Second, a generalization of VE_{ops} is required to be included in the global variable environment VE just like VE' is formed from VE_{sigs} in the CLASS DECL rule. Note that VE_{ops} associates the the class operations with type schemes, so that adding the extra quantification $\forall\alpha.\,\Gamma\,\alpha$ brings the types in VE_{ops} into the same form that types of class operations have in the global variable environment.

The dictionary function produced by an instance declaration of class C must also construct dictionaries for the corresponding instance of the immediate superclasses of C. These are listed in IE_{sup}, associated with the names of the fields they are to be stored in in the dictionary. The same extended instance environment that was used for the method bindings is used to construct dictionary expressions corresponding to the context part of IE_{sup}, yielding the required superclass dictionaries. In contrast to the method fields, all of the superclass fields in a dictionary will always be defined.

The dictionary function produced from an instance declaration takes a tuple of dictionaries corresponding to a particular instance of the instance context and returns an appropriate dictionary. This dictionary is built using a circular binding where the resulting dictionary is passed to the default dictionary function produced from the class declaration. The result of that application is updated with the information from the instance declaration, the $\mathsf{fbind}_\mathsf{i}$ of the methods and the e_i of the superclasses. In this way references in the default method bindings to class operations defined in the instance declaration are directed to the correct bindings.

6.5 Method bindings

Method bindings (figure 27) occur in both class declarations (giving default methods) and instance declarations. They are the only forms of nonrecursive bindings in Haskell in that occurrences of the bound variables in the right hand sides do not refer directly to the bindings of those variables (after dictionary conversion, they may of course refer to those bindings indirectly). Rather, they are considered bound by the corresponding class declaration.

Method bindings are translated to fbinds since dictionaries are implemented as constructors with labelled fields. These fbinds are used in constructions (class dec-

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{method}{\vdash} bind \rightsquigarrow \mathsf{fbind} : VE'}$$

$$\frac{\begin{array}{l} GE, IE \oplus \mathsf{vs}\,\tilde{:}\,\theta, VE \overset{bind}{\vdash} bind \rightsquigarrow \mathsf{bind} : \{x : \langle \_, \tau \rangle\} \\ \{\alpha_1, \ldots, \alpha_k\} \cap (fv(IE) \cup fv(VE)) = \emptyset \\ \mathsf{vs}\ \text{fresh} \end{array}}{GE, IE, VE \overset{method}{\vdash} bind \rightsquigarrow \left\{ \begin{array}{l} \mathsf{x} = \Lambda\alpha_1 \ldots \alpha_k . \lambda \mathsf{vs}\,\hat{:}\,\theta. \\ \quad \mathtt{let}\ \mathsf{bind}' \\ \quad \mathtt{in}\ unQual(x) \end{array} \right\} : \{x : \langle \mathsf{x}, \forall\alpha_1 \ldots \alpha_k . \theta \Rightarrow \tau \rangle\}} \text{ METHOD}$$

Fig. 27. Method bindings.

$$\boxed{IE \overset{dict}{\vdash} \mathsf{e} : (\Gamma_1\ \tau_1, \ldots, \Gamma_n\ \tau_n)}$$

$$\frac{i \in [1, n] : IE \overset{dict}{\vdash} \mathsf{e}_i : \Gamma_i\ \tau_i}{IE \overset{dict}{\vdash} (\mathsf{e}_1, \ldots, \mathsf{e}_n) : (\Gamma_1\ \tau_1, \ldots, \Gamma_n\ \tau_n)} \text{ DICT TUPLE}$$

$$\frac{\mathsf{v} : \Gamma\ (\alpha\ \tau_1 \ldots \tau_k) \in IE}{IE \overset{dict}{\vdash} \mathsf{v} : \Gamma\ (\alpha\ \tau_1 \ldots \tau_k)} \text{ DICT VAR}$$

$$\frac{\mathsf{x} : \forall\alpha_1 \ldots \alpha_k . \theta \Rightarrow \Gamma\ (\chi\ \alpha_1 \ldots \alpha_k) \in IE \qquad IE \overset{dict}{\vdash} \mathsf{e} : \theta[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]}{IE \overset{dict}{\vdash} \mathsf{x}\ \tau_1 \ldots \tau_k\ \mathsf{e} : \Gamma\ (\chi\ \tau_1 \ldots \tau_k)} \text{ DICT INST}$$

$$\frac{\mathsf{x} : \forall\alpha . \Gamma'\ \alpha \Rightarrow \Gamma\ \alpha \in IE \qquad IE \overset{dict}{\vdash} \mathsf{e} : \Gamma'\ \tau}{IE \overset{dict}{\vdash} \mathsf{x}\ \tau\ \mathsf{e} : \Gamma\ \tau} \text{ DICT SUPER}$$

Fig. 28. Dictionary construction.

larations) and updates (instance declarations). The returned variable environment must agree with that derived from the signatures in the class declaration; in particular, the original name of the method, which becomes the field name in the returned fbind, is obtained in that way.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds} : VE_{binds}}$$

$$\frac{GE, IE, VE \overset{bindG}{\vdash} sigs; bindG \rightsquigarrow \mathsf{binds} : VE_{bindG} \qquad GE, IE, VE \oplus VE_{bindG} \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds}' : VE_{binds}}{GE, IE, VE \overset{binds}{\vdash} sigs; bindG \ \mathtt{then}\ binds \rightsquigarrow \mathsf{binds}; \mathsf{binds}' : VE_{bindG} \oplus VE_{binds}} \quad \text{BINDS}$$

$$GE, IE, VE \overset{binds}{\vdash} \epsilon \rightsquigarrow \epsilon : \{\} \quad \text{EMPTY BINDS}$$

Fig. 29. Bindings, part 1 (dependency analysis).

6.6 *Dictionary construction*

Dictionaries are built where they are needed, at occurrences of overloaded variables and constructors. Dictionary building is formalized in figure 28 using judgments of the form

$$IE \overset{dict}{\vdash} \mathsf{e} : (\Gamma_1\ \tau_1, \ldots, \Gamma_n\ \tau_n)$$

where IE is an instance environment, e is an expression evaluating to the required dictionary tuple, the Γ_i are the classes of the dictionaries and the types τ_i give the required instances. This judgment closely parallels the entailment judgments found in more theoretical accounts such as Jones (1995).

The DICT TUPLE rule builds a tuple of dictionaries, one for each component of the context. In the rule DICT VAR there is a variable (v) that is already bound to the right dictionary by an enclosing dictionary abstraction. In the rule DICT INST we use a dictionary function derived from an `instance` declaration for the class Γ and type χ which constructs a dictionary for $\chi\,\tau_1 \ldots \tau_k$ given dictionaries for the τ_i as indicated by the context in the `instance` declaration (these dictionaries may be for classes other than Γ). The variable x is the dictionary function. Finally, the DICT SUPER rule extracts a Γ dictionary for τ from a Γ' dictionary for τ where Γ is a superclass of Γ'. Here, x is a field name used as a selector function, defined in the algebraic datatype for Γ' dictionaries.

This translation is inefficient if the same dictionary is used in several places or if the dictionary building occurs inside a recursive binding (in which case it is executed repeatedly). However, standard compiling techniques can be used to optimize the code. Specifically, common subexpression elimination solves the first problem while the full laziness transformation takes care of the second situation.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

7 Bindings

Binding sets occur at the top level of a module, in `let` expressions, `let` qualifiers and `let` statements (in list comprehensions and `do` expressions) and finally in `where` clauses in guarded alternatives (in function bindings and case expressions). Binding sets are always recursive in Haskell; the bindings shadow all outer bindings of the same variable also in the right hand sides of the bindings.

The typing rules for bindings deal with the generalization aspects of polymorphism and overloading. The rules are complicated by the interaction of polymorphism and recursion. This interaction is treated rather briefly in the Report, and implementations differ in the programs they consider legal.

7.1 Dependency analysis and polymorphism

The Report states (section 4.5.1) that dependency analysis should be used prior to type checking. Type inference is then applied to one strongly connected group of mutually recursive bindings at a time. The order in which the bindings are type checked influence the degree of polymorphism in references to the bound variables. A slightly simplified rule is that references to variables bound in the same binding group (recursive references) have a monomorphic type, whereas references to variables defined in earlier binding groups can be polymorphic.

We express this nesting in the traditional way, as in Peyton Jones & Wadler (1991), for example, by using a language where the dependency analysis is made explicit. A set of *bindings* (*binds*), which occur in the `let` and `where` constructs as well as on the top level of a module, consists of a sequence of *binding groups* (*bindG*). The nesting of binding groups correspond to the result of dependency analysis. The front end of the compiler is responsible for performing the dependency sort of the source-level binding set.

When typing a *bindG*, using the BINDG rule on page 337, we use the polymorphic types given by type signatures for those of the bound variables that have type signatures and monomorphic types for the rest of the bound variables. Using these assumptions we derive monomorphic types for the bindings and hence for the bound variables. Finally, we generalize the derived types, checking that we get what we expected from the signatures.

7.2 Typing binding groups

The typing rule for binding groups is fairly complicated, mainly due to the interaction between polymorphism and overloading on the one hand and recursion on the other. The result of this interaction is that it is difficult to separate polymorphism from recursion, so the traditional division into three different judgements, one dealing with recursion, one with overloading and the last with generalization, becomes fairly contorted. Instead we have a single rule which we will now discuss part by part.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{bindG}{\vdash} sigs; bindG \rightsquigarrow \mathsf{binds} : VE_{bindG}}$$

$$\begin{array}{l}
GE, IE \oplus OE, VE \hat{\oplus} (VE_{sigs} \oplus VE_{rec}) \overset{monobinds}{\vdash} bindG \rightsquigarrow \mathsf{binds} : VE_{monobs} \\
GE \overset{sigs}{\vdash} sigs : VE_{sigs} \\
VE_{sigs} \subseteq VE_{bindG} \\
VE_{rec} = VE_{monobs} \setminus dom(VE_{sigs}) \\
\{\alpha_1, \ldots, \alpha_m\} \cap (fv(IE) \cup fv(VE)) = \emptyset \\
\mathrm{MonoRes}(bindG, dom(VE_{sigs}), \{\alpha_1, \ldots, \alpha_m\} \cap fv(OE)) \\
VE_{bindG} = \forall \alpha_1 \ldots \alpha_m . \theta \Rightarrow VE_{monobs} \\
OE = \mathsf{vs} : \theta \\
VE_{monobs} = \{v_1 : \langle \mathsf{v}_1, \tau_1 \rangle, \ldots, v_n : \langle \mathsf{v}_n, \tau_n \rangle\} \\
VE_{rec} = \{v'_1 : \langle \mathsf{v}'_1, \tau'_1 \rangle, \ldots, v'_k : \langle \mathsf{v}'_k, \tau'_k \rangle\} \\
\mathsf{vs}, \mathsf{v}_{\mathsf{binds}} \text{ fresh}
\end{array}$$

$$\begin{array}{l}
GE, IE, VE \overset{bindG}{\vdash} sigs; bindG \\
\quad \rightsquigarrow \left\{ \begin{array}{l}
\mathsf{v}_{\mathsf{binds}} : \forall \alpha_1 \ldots \alpha_m . \widehat{\theta} \to (\tau_1, \ldots, \tau_n) \\
\quad = \Lambda \alpha_1 \ldots \alpha_m . \lambda \mathsf{vs} \hat{:} \theta . \\
\quad\quad \mathtt{let}\ \mathsf{v}'_1 : \tau'_1 = \mathsf{v}'_1\ \alpha_1 \ldots \alpha_m\ \mathsf{vs}; \\
\quad\quad\quad \ldots; \\
\quad\quad\quad \mathsf{v}'_k : \tau'_1 = \mathsf{v}'_k\ \alpha_1 \ldots \alpha_m\ \mathsf{vs} \\
\quad\quad \mathtt{in\ let\ binds} \\
\quad\quad \mathtt{in}\ (\mathsf{v}_1, \ldots, \mathsf{v}_n); \\
\mathsf{v}_1 : \forall \alpha_1 \ldots \alpha_m . \widehat{\theta} \to \tau_1 \\
\quad = \Lambda \alpha_1 \ldots \alpha_m . \lambda \mathsf{vs} \hat{:} \theta . \mathtt{case}\ \mathsf{v}_{\mathsf{binds}}\ \alpha_1 \ldots \alpha_m\ \mathsf{vs}\ \mathtt{of} \\
\quad\quad\quad\quad\quad\quad (\mathsf{v}_1 : \tau_1, \ldots, \mathsf{v}_n : \tau_n)\ \mathtt{->}\ \mathsf{v}_1; \\
\ldots; \\
\mathsf{v}_n : \forall \alpha_1 \ldots \alpha_m . \widehat{\theta} \to \tau_n \\
\quad = \Lambda \alpha_1 \ldots \alpha_m . \lambda \mathsf{vs} \hat{:} \theta . \mathtt{case}\ \mathsf{v}_{\mathsf{binds}}\ \alpha_1 \ldots \alpha_m\ \mathsf{vs}\ \mathtt{of} \\
\quad\quad\quad\quad\quad\quad (\mathsf{v}_1 : \tau_1, \ldots, \mathsf{v}_n : \tau_n)\ \mathtt{->}\ \mathsf{v}_n;
\end{array} \right\} \\
\quad : VE_{bindG}
\end{array} \quad \text{BINDG}$$

Fig. 30. Bindings, part 2 (generalization and recursion).

7.2.1 Overloading

Binding groups and method bindings introduce overloading in Haskell. The overloading is expressed by typing the bindings in the instance environment IE extended with an overloading environment OE where the context part θ will become the context part of the type schemes in the returned variable environment VE_{bindG}. Looking at the translation we see that the variable part of OE becomes lambda bound. In the translated program, the variables bound in the *bindG* are bound to functions taking dictionaries corresponding to the context θ above. Since this context also occurs in the type schemes for the variables, the correct dictionaries will always be passed at occurrences of the variables.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

The restricted form of the entries in OE force the eager context reduction mandated in Haskell (section 4.5.3 of the Report).

7.2.2 Recursion

Binding groups are always recursive in Haskell. Thus, we type the bindings in a variable environment extended with information about the variables bound in the group. For variables with explicit type signatures, we can assume polymorphic types when typing the bindings (if the types attributed by the signatures are polymorphic, of course). The use of explicit signatures makes polymorphic recursion decidable, something it in general is not (Kfoury *et al.*, 1993). The types given in the signatures (VE_{sigs}) are checked against the types derived after generalization (VE_{bindG}).

For variables lacking explicit type signatures we assume monomorphic types which we check against the types derived from the bindings before generalization (VE_{monobs}). It is evident from the typing rules for the *bind* judgement that these types are monomorphic. Since VE_{rec} contains information about all variables lacking type signatures, outer bindings of these variables are completely hidden.

7.2.3 Generalization and the monomorphism restriction

As in all systems based on Hindley-Milner polymorphism, we are not allowed to generalize over type variables free in the variable environment VE. Since the instance environment IE plays a similar role, we are not allowed to generalize over type variables free in IE either. We may however generalize over variables free in the overloading environment OE since (the context part of it) will end up inside the type schemes after generalization. Since it is not forbidden to generalize over variables not occurring in the variable environment VE_{monobs} derived from the bindings, it is possible to construct ambiguous type schemes. This is one place where the inference rules presented in this paper clearly deviates from the (informal) specification in the Report (section 4.3.4). Ambiguity is further discussed in section 10.2.

One further limitation on generalization comes from Haskell's *monomorphism restriction* (section 4.5.5 in the Report). This rule restricts the amount of overloading that is allowed for variables bound in a declaration group in order to preserve sharing and avoid spurious ambiguity errors. Overloading is controlled on a per-*bindG* basis. If a *bindG* only contains function bindings and pattern bindings binding single variables, and if there is a type signature for each of the latter variables, then overloading is allowed. Otherwise, the type schemes formed by generalization must have empty contexts. This condition is checked by the MonoRes predicate, defined below.

Definition 2 (*Monomorphism restriction*)
MonoRes($bindG, sigvs, congenvs$) is satisfied iff either the set *congenvs* (the constrained generic type variables) is empty or *bindG* only contains function bindings and pattern bindings where the pattern is a single variable included in the set *sigvs* (the set of variables with type signatures).

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

```
f :: Show a => a -> String
f x = show x ++ g x
g x = f (show x)

~~>

v: ∀a. P!ShowD a → (a → [P!Char], a → [P!Char])
  = Λa. λd : P!ShowD a.
    let g : (a → [P!Char]) = g a d
    in let f : (a → [P!Char]) (x : a) = (P!++) P!Char (P!show a d x) (g x);
           g : (a → [P!Char]) (x : a) = f [P!Char] (P!showListD P!Char P!showCharD)
                                                  (P!show a d x)
       in (f,g);

f: ∀a. P!ShowD a → a → [P!Char]
  = Λa. λd : P!ShowD a. case v a d of
                        (f : a → [P!Char], g : a → [P!Char]) -> f;

g: ∀a. P!ShowD a → a → [P!Char]
  = Λa. λd : P!ShowD a. case v a d of
                        (f : a → [P!Char], g : a → [P!Char]) -> g
```

Kind annotations are omitted and the module name `Prelude` has been abbreviated to `P`.

Fig. 31. Example of translation by the *bindG* judgement.

7.2.4 *The translation*

The translation part of the rules is quite involved because of the interaction between the explicit type and dictionary abstractions in our target language and the fact that we generalize over binds rather than over expressions. We could of course have added generalization over binds to the target language, but that would simply move the problem to that of giving the construct a semantics.

Therefore, we have chosen a translation which only needs abstraction over expressions. Figure 31 shows an example which involves most of the features of this translation. In this example we have omitted all kind annotations and we have abbreviated the name of the `Prelude` module to `P`. The basic strategy is to bind a fresh variable $\mathsf{v}_{\mathsf{binds}}$ (`v` in the example) to a type and dictionary abstraction which, when applied, yields a tuple of the values of the bound variables (`f` and `g`) of the *bindG*. Each of the bound variables is then bound to an overloaded and polymorphic function which instantiates the tuple (`v`) and extracts the appropriate component. In this way we avoid duplicating the entire target binds. This set of bindings, one for each bound variable and one for $\mathsf{v}_{\mathsf{binds}}$, will be wrapped in a `rec` wherever it is used (in the LET rule on page 344, the GDES rule on page 343, the QLET rule on page 348 and the SLET rule on page 349). In the target program, all references, recursive as well as nonrecursive, will go through these bindings since the `let`s inside the tuple function are nonrecursive.

To construct the tuple, we wrap the translated bindings obtained from the

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

```
f1: ∀a. P!ShowD a → a → [P!Char]
   = Λa. λ(d : P!ShowD a) (x : a). (P!++) P!Char (P!show a d x) (g2 a d x);

g1: ∀a. P!ShowD a → a → [P!Char]
   = Λa. λ(d : P!ShowD a) (x : a).
        f [P!Char] (P!showListD P!Char P!showCharD) (P!show a d x)

g2: ∀a. P!ShowD a → a → [P!Char]
   = Λa. λ(d : P!ShowD a). g a d

v  : ∀a. P!ShowD a → (a → [P!Char], a → [P!Char])
   = Λa. λ(d : P!ShowD a).(f1 a d, g1 a d);

f  : ∀a. P!ShowD a → a → [P!Char]
   = Λa. λ(d : P!ShowD a). case v a d of
                           (f : a → [P!Char], g : a → [P!Char]) -> f;

g  : ∀a. P!ShowD a → a → [P!Char]
   = Λa. λ(d : P!ShowD a). case v a d of
                           (f : a → [P!Char], g : a → [P!Char]) -> g
```

Fig. 32. Example of optimization of the translation.

monobinds judgement in a `let` expression, the body of which is the tuple. Because of polymorphic recursion, some of the bound variables may be polymorphically typed (`f` is, in this case), so occurrences of these variables in the source *bindG* (there is one in the body of `g`) are translated to type applications in the target binds. This agrees with the bindings of these variables to type and dictionary abstractions. Variables lacking type signatures (like `g`) are associated with monomorphic types during typing of the *bindG* and their occurrences are therefore not translated to type applications even though they are also bound to type and dictionary abstractions in the outer bindings. The `let` expression containing the translation of the *bindG* is therefore wrapped in an outer `let` which binds these variables to monomorphic values, shadowing the outer, polymorphic, bindings.

This translation is reasonable in that it avoids code duplication, but it is not ideal in that it requires construction of the tuple for every reference to one of the bound variables. If the monomorphism restriction applies to the *bindG*, so that the variables bound in the *bindG* have non overloaded types, the problem is solved, at least for an implementation which at some point switches to an untyped intermediate language. This is the case since the type abstraction and application disappears at that point and there were no dictionary abstractions to begin with. Otherwise, there is still the dictionary abstractions and applications to take care of. Fortunately, there are some relatively simple optimizations which can be used.

For instance, lambda lifting (Johnsson, 1985) can be used to move the dictionary abstractions into the right hand sides of each of the bindings. Since the monomorphism restriction did not apply, all of these are either function bindings or simple pattern bindings, guaranteeing that the transformation is type correct. Each of the

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

```
f: ∀a. P!ShowD a → a → [P!Char]
  = Λa. λ(d : P!ShowD a) (x : a). (P!++) P!Char (P!show a d x) (g a d x);

g: ∀a. P!ShowD a → a → [P!Char]
  = Λa. λ(d : P!ShowD a) (x : a).
        f [P!Char] (P!showListD P!Char P!showCharD) (P!show a d x)
```

Fig. 33. Example of optimization of the translation; the final code.

bindings, which no longer have free dictionary variables, can then be lifted out of the tuple function which becomes small and is subsequently inlined and simplified away. Figure 32 shows the result of applying this transformation to the example. The situation just after lifting (and renaming) the nested definitions is shown, with `f1` and `g1` being the lifted versions of the original bindings and `g2` being the lifted conversion for `g`. Note that `v` can now be inlined since the tuple is exposed, and so can `f1`, `g1` and `g2`, leaving the rather reasonable code in figure 33.

Note that this lambda lifting is unusual in that it may lift bindings which are not function bindings, like the conversion for `g`, leading to a potential loss of sharing. The requirement that variables bound in pattern bindings must have explicit type signatures serves to make the programmer aware of this fact. Effectively, by supplying a signature with a non-empty context, the programmer has sanctioned this loss of sharing.

7.3 Function and pattern bindings

Figures 34 and 35 give the rules for function and pattern bindings. These rules are fairly uncomplicated. One thing to note, though, is that in the FUNBIND rule on page 342, the function variable x might be qualified (hence an x rather than a v) since this judgment is used to process bindings in instance declarations (the *method* judgement on page 334). The translation always binds the unqualified version of the name, and that version is also used in the METHOD rule on page 334.

8 Expressions

The typing judgement for expressions has the form $GE, VE, IE \overset{exp}{\vdash} e \rightsquigarrow \mathtt{e} : \tau$. As discussed earlier, generalization is done in connection with bindings rather than with expressions, hence the monotype τ. The rules for this judgement are syntax directed in that there is only one rule which can apply in each situation (although there are two rules for variable references in figure 36, only one can be used depending on the form of the type for the variable). Instantiation is built into the rules that need it (VAR-I and VAR-II in figure 36 and CON, UPD and LABCON in figure 39).

There are many rules spread over four figures, and we will not comment extensively on all of them. Some comments are in order, however, and we will organize them in one subsection for each figure.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{monobinds}{\vdash} bindG \rightsquigarrow \mathsf{binds} : VE_{binds}}$$

$$\frac{i \in [1,n] : GE, VE, IE \overset{bind}{\vdash} bind_i \rightsquigarrow \mathsf{bind}_i : VE_i}{GE, IE, VE \overset{monobinds}{\vdash} \left\{ \begin{array}{l} bind_1; \\ \ldots; \\ bind_n \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \mathsf{bind}_1; \\ \ldots; \\ \mathsf{bind}_n \end{array} \right\} : VE_1 \oplus \ldots \oplus VE_n} \quad \text{MONOBINDS}$$

$$\boxed{GE, IE, VE \overset{bind}{\vdash} bind \rightsquigarrow \mathsf{bind} : VE_{bind}}$$

$$\frac{i \in [1,n] : GE, IE, VE \overset{match}{\vdash} match_i \rightsquigarrow \mathsf{match}_i : \tau}{GE, IE, VE \overset{bind}{\vdash} \left\{ \begin{array}{c} x\ match_1 \\ [] \ldots [] \\ match_n \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{c} unQual(x)\ \mathsf{match}_1 \\ [] \ldots [] \\ \mathsf{match}_n \end{array} \right\} : \{x : \langle x, \tau \rangle\}} \quad \text{FUNBIND}$$

$$\frac{\begin{array}{l} GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau \\ GE, IE, VE \overset{gdes}{\vdash} gdes \rightsquigarrow \mathsf{gdes} : \tau \end{array}}{GE, IE, VE \overset{bind}{\vdash} p\ gdes \rightsquigarrow \mathsf{p\ gdes} : VE_p} \quad \text{PATBIND}$$

Fig. 34. Bindings, part 3.

8.1 Expressions part one: Variables

Variables are either class methods or ordinary variables. The two forms are distinguished by the form of typing information in the variable environment. Both variable rules instantiate any polymorphism and the translations show the explicit type and dictionary applications.

In the rule VAR-II, the substitution $[\tau/\alpha]$ is not applied to θ since α is not allowed to occur in θ by the rules for class declarations in section 4.3.1 of the Report. The last premise of the SIG rule on page 330 formalizes this requirement.

8.2 Literals

Literals are overloaded in Haskell, and are translated to applications of the overloaded conversion functions `fromInteger` for integer literals and `fromRational` for floating point literals. These choices of types for literals enable very large integers and very precise floating point numbers to be written in the code, but since such

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{match}{\vdash} match \rightsquigarrow \mathsf{match} : \tau}$$

$$\frac{i \in [1,k] : GE, IE \overset{pat}{\vdash} p_i \rightsquigarrow \mathsf{p}_i : VE_i, \tau_i \qquad GE, IE, VE \vec{\oplus} (VE_1 \oplus \ldots \oplus VE_k) \overset{gdes}{\vdash} gdes \rightsquigarrow \mathsf{gdes} : \tau}{GE, IE, VE \overset{match}{\vdash} p_1 \ldots p_k \; gdes \rightsquigarrow \mathsf{p}_1 \ldots \mathsf{p}_k \; \mathsf{gdes} : \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \tau} \quad \text{MATCH}$$

$$\boxed{GE, IE, VE \overset{gdes}{\vdash} gdes \rightsquigarrow \mathsf{gdes} : \tau}$$

$$\frac{i \in [1,n] : GE, IE, VE \vec{\oplus} VE_{binds} \overset{gde}{\vdash} gde_i \rightsquigarrow \mathsf{gde}_i : \tau \qquad GE, IE, VE \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds} : VE_{binds}}{GE, IE, VE \overset{gdes}{\vdash} gde_1 \ldots gde_n \; \texttt{where} \; binds \rightsquigarrow \mathsf{gde}_1 \ldots \mathsf{gde}_n \; \texttt{where rec} \; \mathsf{binds} : \tau} \quad \text{GDES}$$

$$\boxed{GE, IE, VE \overset{gde}{\vdash} gde \rightsquigarrow \mathsf{gde} : \tau}$$

$$\frac{GE, IE, VE \overset{exp}{\vdash} e_1 \rightsquigarrow \mathsf{e}_1 : \texttt{Prelude!Bool}^* \qquad GE, IE, VE \overset{exp}{\vdash} e_2 \rightsquigarrow \mathsf{e}_2 : \tau}{GE, IE, VE \overset{gde}{\vdash} \mid e_1 = e_2 \rightsquigarrow \mid \mathsf{e}_1 = \mathsf{e}_2 : \tau} \quad \text{GDE}$$

Fig. 35. Pattern matches and guarded expressions.

conversions are expensive, there are programs where the bulk of the execution time is spent in constructing literals. Augustsson gives examples and suggests some improvements in Augustsson (1993).

8.3 Expressions part two

In lambda expressions (rule LAMBDA), the bound variables must have monomorphic types. This is ensured by the *pat* judgement on pages 351 and 352 which always returns monomorphic types. The information from the pattern hides any outer bindings for the same variables; hence the use of $\vec{\oplus}$.

In the LET rule, the translated bindings are wrapped in a `rec` as discussed elsewhere, and the bindings hide outer bindings.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau}$$

$$\frac{\begin{array}{l} x : \langle \mathsf{x}, \forall \alpha_1 \ldots \alpha_n.\theta \Rightarrow \tau \rangle \in VE \\ IE \overset{dict}{\vdash} \mathsf{e} : \theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \end{array}}{GE, IE, VE \overset{exp}{\vdash} x \rightsquigarrow \mathsf{x}\ \tau_1 \ldots \tau_n\ \mathsf{e} : \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]} \quad \text{VAR I}$$

$$\frac{\begin{array}{l} x : \langle \mathsf{x}, \forall \alpha.\, \Gamma\, \alpha \Rightarrow_c \forall \alpha_1 \ldots \alpha_n.\theta \Rightarrow \tau \rangle \in VE \\ IE \overset{dict}{\vdash} \mathsf{e}_1 : \Gamma\, \tau \\ IE \overset{dict}{\vdash} \mathsf{e}_2 : \theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \end{array}}{GE, IE, VE \overset{exp}{\vdash} x \rightsquigarrow \mathsf{x}\ \tau\ \mathsf{e}_1\ \tau_1 \ldots \tau_n\ \mathsf{e}_2 : \tau[\tau/\alpha, \tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]} \quad \text{VAR II}$$

Fig. 36. Expressions, part 1.

The CASE rule uses the *match* judgement on page 343. The same judgement is used in the rule for function bindings in figure 34 on page 342. From the point of the CASE rule alone, there is nothing that prohibits the $match_i$ to have more than one pattern each. Since this does not correspond to legal Haskell syntax, such `case` expressions do not occur in practice, but the prohibition lies outside this semantics.

8.4 Expressions part 3

List comprehensions are translated to target list comprehensions; we do only that desugaring which is directly related to resolution of overloading or to modules. The expression part of a comprehension is typed in a variable environment extended with the types derived from the qualifiers. Looking at the typing rules for qualifiers in figure 40 on page 348, we see that the types derived for variables bound in `let` qualifiers (in the QLET rule) have polymorphic types, both in the rest of the qualifiers and in the expression. The types of variables bound in generators (QGEN) are monomorphic, however. Since the QGEN rule uses $\dot{\oplus}$ to combine environments, later qualifiers hide earlier.

In the DO rule, the requirement that its type be monadic is enforced in the typing rules for statements. The same rules for polymorphism and monomorphism as for list comprehensions apply to `do` expressions. Note that the rule for typing an expression statement, STHEN, is very different from the QFILTER rule for typing an expression in a qualifier list. The similarities between list comprehensions and `do` expressions are somewhat misleading; the qualifier is a boolean whereas the statement is monadic. A related syntactic difference is that a statement list must always be terminated by an expression. In particular, it may not be empty.

In the CON, UPD and LABCON rules, the dictionaries derived are not used in the

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{IE \overset{literal}{\vdash} literal \rightsquigarrow \mathtt{e} : \tau}$$

$$IE \overset{literal}{\vdash} char \rightsquigarrow char : \mathtt{Prelude!Char}^* \qquad \text{LIT CHAR}$$

$$IE \overset{literal}{\vdash} string \rightsquigarrow string : [\mathtt{Prelude!Char}^*] \qquad \text{LIT STRING}$$

$$\frac{IE \overset{dict}{\vdash} \mathtt{e} : \mathtt{Prelude!Num}^* \tau}{IE \overset{literal}{\vdash} integer \rightsquigarrow \mathtt{Prelude!fromInteger}\ \tau\ \mathtt{e}\ integer : \tau} \qquad \text{LIT INTEGER}$$

$$\frac{n/d = float \qquad IE \overset{dict}{\vdash} \mathtt{e} : \mathtt{Prelude!Fractional}^* \tau}{IE \overset{literal}{\vdash} float \rightsquigarrow \mathtt{Prelude!fromRational}\ \tau\ \mathtt{e}\ (\mathtt{(Ratio.\%)}\ n\ d) : \tau} \qquad \text{LIT FLOAT}$$

Fig. 37. Literals

translation. The only reason to derive them is to force the instance environment to entail the context from the data type declaration. Thus constructors from types with nonempty contexts are not overloaded in any semantic or operational sense. Their overloading is closer to an assertion about their intended use, with a type error signaled if the assertion is invalidated. The value of this feature is dubious and the price in terms of spurious complexity is rather high, as especially the rules for typing labeled updates show.

The UPD rule mimics the translation of the update construct. The fields used in the expression are looked up in the data constructor environment DE to find their label environments which for each field name lists all constructors which have a field of that name. Thus the K_j are the constructors which occur in all of the LE_i. For each of these constructors, its associated φ will be used with the *lcon* judgement to find the relation between τ_{new} and τ_{old} and the τ_i. The condition that $k > 0$ ensures that there is some constructor which has all of the fields mentioned in the update. In the LABCON rule, there is no τ_{old} since the value constructed is new. The constructor K is looked up in DE to find its original name, which must be in the domain of the intersection of the LE_i, ensuring that K has all of the required fields.

8.5 Expressions part four: Sequences

The rules for arithmetic sequences translate these to applications of methods of the `Enum` class from the Prelude.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau}$$

$$\frac{IE \overset{literal}{\vdash} literal \rightsquigarrow \mathsf{e} : \tau}{GE, IE, VE \overset{exp}{\vdash} literal \rightsquigarrow \mathsf{e} : \tau} \quad \text{LITERAL}$$

$$\frac{\begin{array}{c} i \in [1,k] : GE, IE \overset{pat}{\vdash} p_i \rightsquigarrow \mathsf{p}_i : VE_i, \tau_i \\ GE, IE, VE \vec{\oplus}(VE_1 \oplus \ldots \oplus VE_k) \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} \backslash p_1 \ldots p_k \texttt{->} e \rightsquigarrow \backslash \mathsf{p}_1 \ldots \mathsf{p}_k \texttt{->} \mathsf{e} : \tau_1 \to \ldots \to \tau_k \to \tau} \quad \text{LAMBDA}$$

$$\frac{\begin{array}{c} GE, IE, VE \overset{exp}{\vdash} e_1 \rightsquigarrow \mathsf{e}_1 : \tau' \to \tau \\ GE, IE, VE \overset{exp}{\vdash} e_2 \rightsquigarrow \mathsf{e}_2 : \tau' \end{array}}{GE, IE, VE \overset{exp}{\vdash} e_1\ e_2 \rightsquigarrow \mathsf{e}_1\ \mathsf{e}_2 : \tau} \quad \text{APP}$$

$$\frac{\begin{array}{c} GE, IE, VE \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds} : VE_{binds} \\ GE, IE, VE \vec{\oplus} VE_{binds} \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} \texttt{let}\ binds\ \texttt{in}\ e \rightsquigarrow \texttt{let rec}\ \mathsf{binds}\ \texttt{in}\ \mathsf{e} : \tau} \quad \text{LET}$$

$$\frac{\begin{array}{c} GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau' \\ i \in [1,n] : GE, IE, VE \overset{match}{\vdash} match_i \rightsquigarrow \mathsf{match}_i : \tau' \to \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} \texttt{case}\ e\ \texttt{of}\ match_1\ [] \ldots [] match_n \rightsquigarrow \texttt{case}\ \mathsf{e}\ \texttt{of}\ \mathsf{match}_1\ [] \ldots [] \mathsf{match}_n : \tau} \quad \text{CASE}$$

Fig. 38. Expressions, part 2.

9 Patterns

Type checking a pattern yields a variable environment giving the types of the variables in the pattern and a type for the pattern itself. This type should match the type of whatever the pattern is matched against.

For the benefit of method bindings in instance declarations, the abstract syntax of patterns and the PVAR rule allow single variables to be qualified. The translated pattern is still unqualified but the returned environment associates the type with the qualified name since it is checked against the (possibly imported) variable

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{quals}{\vdash} quals \rightsquigarrow \mathsf{quals} : VE_{quals} \\ GE, IE, VE \overset{\rightarrow}{\oplus} VE_{quals} \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} [e \mid quals] \rightsquigarrow [\mathsf{e} \mid \mathsf{quals}] : [\tau]} \quad \text{LIST COMP}$$

$$\frac{GE, IE, VE \overset{stmts}{\vdash} stmts \rightsquigarrow \mathsf{e} : \tau}{GE, IE, VE \overset{exp}{\vdash} \mathsf{do}\ stmts \rightsquigarrow \mathsf{e} : \tau} \quad \text{DO}$$

$$\frac{\begin{array}{l} K : \langle \mathsf{K}, \chi, \forall \alpha_1 \ldots \alpha_n . \theta \Rightarrow \tau \rangle \in DE \\ GE = \langle CE, TE, DE \rangle \\ IE \overset{dict}{\vdash} \mathsf{e} : \theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \end{array}}{GE, IE, VE \overset{exp}{\vdash} K \rightsquigarrow \mathsf{K}\ \tau_1 \ldots \tau_n : \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]} \quad \text{CON}$$

$$\frac{\begin{array}{l} i \in [1,n] : GE, IE, VE \overset{exp}{\vdash} e_i \rightsquigarrow \mathsf{e}_i : \tau_i \\ i \in [1,n] : x_i : \langle \mathsf{x}_i, \chi, LE_i \rangle \in DE \\ \qquad \{\mathsf{K}_1 : \varphi_1, \ldots, \mathsf{K}_k : \varphi_k\} = LE_1 \cap \ldots \cap LE_n \\ i \in [1,k] : IE, \varphi_i \overset{lcon}{\vdash} \tau_{old}, \{\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n\}, \tau_{new} \\ \qquad k > 0 \\ \qquad GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau_{old} \\ \qquad GE = \langle CE, TE, DE \rangle \end{array}}{GE, IE, VE \overset{exp}{\vdash} e \Leftarrow \{x_1 = e_1, \ldots, x_n = e_n\} \rightsquigarrow \mathsf{e} \Leftarrow \{\mathsf{x}_1 = \mathsf{e}_1, \ldots, \mathsf{x}_n = \mathsf{e}_n\} : \tau_{new}} \quad \text{UPD}$$

$$\frac{\begin{array}{l} i \in [1,n] : GE, IE, VE \overset{exp}{\vdash} e_i \rightsquigarrow \mathsf{e}_i : \tau_i \\ i \in [1,n] : x_i : \langle \mathsf{x}_i, \chi, LE_i \rangle \in DE \\ \qquad K : \langle \mathsf{K}, \chi, \sigma \rangle \in DE \\ \qquad \mathsf{K} : \varphi \in LE_1 \cap \ldots \cap LE_n \\ \qquad IE, \varphi \overset{lcon}{\vdash} \_, \{\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n\}, \tau \\ \qquad \tau = \chi\, \tau'_1 \ldots \tau'_k \\ \qquad GE = \langle CE, TE, DE \rangle \end{array}}{GE, IE, VE \overset{exp}{\vdash} K\ \{x_1 = e_1, \ldots, x_n = e_n\} \rightsquigarrow \mathsf{K}\ \tau'_1 \ldots \tau'_k\ \{\mathsf{x}_1 = \mathsf{e}_1, \ldots, \mathsf{x}_n = \mathsf{e}_n\} : \tau} \quad \text{LABCON}$$

Fig. 39. Expressions, part 3.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{quals}{\vdash} quals \rightsquigarrow \mathsf{quals} : VE_{quals}}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : [\tau] \\ GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau \\ GE, IE, VE \mathbin{\vec{\oplus}} VE_p \overset{quals}{\vdash} quals \rightsquigarrow \mathsf{quals} : VE_{quals} \end{array}}{GE, IE, VE \overset{quals}{\vdash} p \texttt{ <- } e, quals \rightsquigarrow \mathsf{p} \texttt{ <- } \mathsf{e}, \mathsf{quals} : VE_p \mathbin{\vec{\oplus}} VE_{quals}} \quad \text{QGEN}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds} : VE_{binds} \\ GE, IE, VE \mathbin{\vec{\oplus}} VE_{binds} \overset{quals}{\vdash} quals \rightsquigarrow \mathsf{quals} : VE_{quals} \end{array}}{GE, IE, VE \overset{quals}{\vdash} \texttt{let } binds, \ quals \rightsquigarrow \texttt{let rec } \mathsf{binds}, \ \mathsf{quals} : VE_{binds} \mathbin{\vec{\oplus}} VE_{quals}} \quad \text{QLET}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \texttt{Prelude!Bool}^* \\ GE, IE, VE \overset{quals}{\vdash} quals \rightsquigarrow \mathsf{quals} : VE_{quals} \end{array}}{GE, IE, VE \overset{quals}{\vdash} e, quals \rightsquigarrow \mathsf{e}, \mathsf{quals} : VE_{quals}} \quad \text{QFILTER}$$

$$GE, IE, VE \overset{quals}{\vdash} \epsilon \rightsquigarrow \epsilon : \{\} \quad \text{QEMPTY}$$

Fig. 40. Qualifiers.

environment in the INST rule on page 332. Variables also occur in as-patterns (v`@`p) and n+k patterns where only unqualified variables are allowed.

9.1 Overloaded patterns

Literal patterns and n+k patterns are overloaded in Haskell, and in the Report (section 3.17.3) they are translated to conditional expressions as part of the translation of pattern matching. Since we do not want to give that translation, which entails a fair amount of desugaring and has little to do with the type system, in this paper, we have invented alternative constructs in the target language into which overloaded literals and n+k patterns can be translated.

- A pattern of the form $\{\mathsf{e}\}$ matches values z for which $f\ z$ is `True` where f is the semantics of e.
- Similarly, patterns of the form $\mathsf{v} : \tau\{\mathsf{e}_1, \mathsf{e}_2\}$ match values z such that $f\ z$ is true where f is the semantics of e_1, binding v to $g\ z$ where g is the semantics of e_2.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{stmts}{\vdash} stmts \rightsquigarrow \mathsf{e} : \tau}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e}_1 : \tau\,\tau_1 \\ GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau_1 \\ GE, IE, VE \oplus VE_p \overset{stmts}{\vdash} stmts \rightsquigarrow \mathsf{e}_2 : \tau\,\tau_2 \\ IE \overset{dict}{\vdash} \mathsf{e_d} : \mathtt{Prelude!Monad}^{*\to *}\,\tau \end{array}}{GE, IE, VE \overset{stmts}{\vdash} p \texttt{ <- } e; stmts \rightsquigarrow \left\{\begin{array}{l} \texttt{let x p = e}_2 \\ \quad \texttt{[] \_ = Prelude!fail } \tau\ \mathsf{e_d}\ \texttt{"..."} \\ \texttt{in (Prelude!>>=) } \tau\ \mathsf{e_d}\ \tau_1\ \tau_2\ \mathsf{e}_1\ \texttt{x} \end{array}\right\} : \tau\,\tau_2} \quad \text{SBIND}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{binds}{\vdash} binds \rightsquigarrow \mathsf{binds} : VE_{binds} \\ GE, IE, VE \oplus VE_{binds} \overset{stmts}{\vdash} stmts \rightsquigarrow \mathsf{e} : \tau \end{array}}{GE, IE, VE \overset{stmts}{\vdash} \texttt{let } binds\texttt{; } stmts \rightsquigarrow \texttt{let rec binds in e} : \tau} \quad \text{SLET}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e}_1 : \tau\,\tau_1 \\ GE, IE, VE \overset{stmts}{\vdash} stmts \rightsquigarrow \mathsf{e}_2 : \tau\,\tau_2 \\ IE \overset{dict}{\vdash} \mathsf{e_d} : \mathtt{Prelude!Monad}^{*\to *}\,\tau \end{array}}{GE, IE, VE \overset{stmts}{\vdash} e\texttt{; } stmts \rightsquigarrow \texttt{(Prelude!>>) } \tau\,\mathsf{e_d}\,\tau_1\,\tau_2\,\mathsf{e}_1\,\mathsf{e}_2 : \tau\,\tau_2} \quad \text{STHEN}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau\,\tau_1 \\ IE \overset{dict}{\vdash} \texttt{\_} : \mathtt{Prelude!Monad}^{*\to *}\,\tau \end{array}}{GE, IE, VE \overset{stmts}{\vdash} e \rightsquigarrow \mathsf{e} : \tau\,\tau_1} \quad \text{SRET}$$

Fig. 41. Statements.

10 A Critique of Haskell 98

Writing down these inference rules has exposed several places where the Report has been vague but also some places where it has expressed rules which have been very difficult to formalize. We are now also in the position to study the formal properties of the Haskell type system, and we offer some initial comments on the subject.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e_1 \rightsquigarrow \mathsf{e}_1 : \tau \\ GE, IE, VE \overset{exp}{\vdash} e_2 \rightsquigarrow \mathsf{e}_2 : \tau \\ GE, IE, VE \overset{exp}{\vdash} e_3 \rightsquigarrow \mathsf{e}_3 : \tau \\ IE \overset{dict}{\vdash} \mathsf{e} : \texttt{Prelude!Enum}^* \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} [e_1, e_2 .. e_3] \rightsquigarrow \texttt{Prelude!enumFromThenTo}\ \tau\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3 : [\tau]} \quad \text{ENUM FROM THEN TO}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e_1 \rightsquigarrow \mathsf{e}_1 : \tau \\ GE, IE, VE \overset{exp}{\vdash} e_2 \rightsquigarrow \mathsf{e}_2 : \tau \\ IE \overset{dict}{\vdash} \mathsf{e} : \texttt{Prelude!Enum}^* \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} [e_1 .. e_2] \rightsquigarrow \texttt{Prelude!enumFromTo}\ \tau\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2 : [\tau]} \quad \text{ENUM FROM TO}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e_1 \rightsquigarrow \mathsf{e}_1 : \tau \\ GE, IE, VE \overset{exp}{\vdash} e_2 \rightsquigarrow \mathsf{e}_2 : \tau \\ IE \overset{dict}{\vdash} \mathsf{e} : \texttt{Prelude!Enum}^* \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} [e_1, e_2 ..] \rightsquigarrow \texttt{Prelude!enumFromThen}\ \tau\ \mathsf{e}\ \mathsf{e}_1\ \mathsf{e}_2 : [\tau]} \quad \text{ENUM FROM THEN}$$

$$\frac{\begin{array}{l} GE, IE, VE \overset{exp}{\vdash} e_1 \rightsquigarrow \mathsf{e}_1 : \tau \\ IE \overset{dict}{\vdash} \mathsf{e} : \texttt{Prelude!Enum}^* \tau \end{array}}{GE, IE, VE \overset{exp}{\vdash} [e_1 ..] \rightsquigarrow \texttt{Prelude!enumFrom}\ \tau\ \mathsf{e}\ \mathsf{e}_1 : [\tau]} \quad \text{ENUM FROM}$$

Fig. 42. Expressions, part 4 (sequences).

10.1 Principality and the monomorphism restriction

Polymorphic types are related by an instantiation ordering which tells when a type is more general than another. For instance, $\forall\alpha\,\alpha'.(\alpha, \alpha')$ is more general than $\forall\alpha'.(\texttt{Int}, \alpha')$ since the latter type can be obtained by substituting `Int` for α in (α, α'). The definition of the generic instance relation for Haskell is complicated by the context parts of the type schemes and must in general be defined relative to the class and instance declarations that are in scope. Any generic instance relation must however insist that if σ is less general than σ', then type part of σ must be a

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau}$$

$$GE, IE \overset{pat}{\vdash} x \rightsquigarrow unQual(x) : \tau : \{x : \langle x, \tau \rangle\}, \tau \qquad \text{PVAR}$$

$$\frac{GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau}{GE, IE \overset{pat}{\vdash} v\,@p \rightsquigarrow v : \tau @ \mathsf{p} : VE_p \oplus \{v : \langle v, \tau \rangle\}, \tau} \qquad \text{PAS}$$

$$\frac{GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau}{GE, IE \overset{pat}{\vdash} {\sim}p \rightsquigarrow {\sim}\mathsf{p} : VE_p, \tau} \qquad \text{PIRR}$$

$$GE, IE \overset{pat}{\vdash} \_ \rightsquigarrow \_ : \{\}, \tau \qquad \text{PWILD}$$

$$\frac{\begin{array}{l} i \in [1,n] : GE, IE \overset{pat}{\vdash} p_i \rightsquigarrow \mathsf{p}_i : VE_i, \tau_i[\tau'_1/\alpha_1, \ldots, \tau'_k/\alpha_k] \\ K : \langle \mathsf{K}, \chi, \forall \alpha_1 \ldots \alpha_k . \theta \Rightarrow \tau_1 \to \ldots \to \tau_n \to \chi\,\alpha_1 \ldots \alpha_k \rangle \in DE \\ IE \overset{dict}{\vdash} \mathsf{e} : \theta[\tau'_1/\alpha_1, \ldots, \tau'_k/\alpha_k] \\ GE = \langle CE, TE, DE \rangle \end{array}}{GE, IE \overset{pat}{\vdash} K\ p_1 \ldots p_n \rightsquigarrow \mathsf{K}\,\mathsf{p}_1 \ldots \mathsf{p}_n : VE_1 \oplus \ldots \oplus VE_n, \chi\,\tau'_1 \ldots \tau'_k} \qquad \text{PCON}$$

$$\frac{\begin{array}{l} i \in [1,n] : GE, IE \overset{pat}{\vdash} p_i \rightsquigarrow \mathsf{p}_i : VE_i, \tau_i \\ i \in [1,n] : x_i : \langle \mathsf{x}_i, \chi, LE_i \rangle \in DE \\ \qquad K : \langle \mathsf{K}, \chi, \sigma \rangle \in DE \\ \qquad \mathsf{K} : \varphi \in LE_1 \cap \ldots \cap LE_n \\ \qquad IE, \varphi \overset{lcon}{\vdash} \_, \{\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n\}, \tau \\ \qquad \tau = \chi\,\tau'_1 \ldots \tau'_k \\ \qquad GE = \langle CE, TE, DE \rangle \end{array}}{GE, IE \overset{pat}{\vdash} K\ \{x_1 = p_1, \ldots, x_n = p_n\} \rightsquigarrow \mathsf{K}\ \{\mathsf{x}_1 = \mathsf{p}_1, \ldots, \mathsf{x}_n = \mathsf{p}_n\} : VE_1 \oplus \ldots \oplus VE_n, \tau} \qquad \text{PLAB}$$

Fig. 43. Patterns, part 1.

substitution instance of the type part of σ', in addition to any requirements on the context parts.

Given an expression e and a typing environment E, the *principal type* of e in E is a type scheme σ which is a valid type of e in E and which is more general

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

$$\boxed{GE, IE \overset{pat}{\vdash} p \rightsquigarrow \mathsf{p} : VE_p, \tau}$$

$$\frac{\begin{array}{l} IE \overset{literal}{\vdash} integer \rightsquigarrow \mathsf{e} : \tau \\ IE \overset{dict}{\vdash} \mathsf{e}_{\mathsf{ord}} : \mathtt{Prelude.Ord}^* \ \tau \\ IE \overset{dict}{\vdash} \mathsf{e}_{\mathsf{num}} : \mathtt{Prelude.Num}^* \ \tau \\ IE \overset{dict}{\vdash} \_ : \mathtt{Prelude.Integral}^* \ \tau \\ \mathsf{x} \text{ fresh} \end{array}}{\begin{array}{l} GE, IE \overset{pat}{\vdash} v{+}integer \\ \quad \rightsquigarrow v : \tau \, \{ \mathtt{(Prelude.<=)} \ \tau \ \mathsf{e}_{\mathsf{ord}} \ \mathsf{e}, \ \lambda \mathsf{x} : \tau . \, \mathtt{(Prelude.-)} \ \tau \ \mathsf{e}_{\mathsf{num}} \ \mathsf{x} \ \mathsf{e} \} \\ \quad : \{ v : \langle v, \tau \rangle \}, \tau \end{array}} \quad \text{PNPK}$$

$$GE, IE \overset{pat}{\vdash} char \rightsquigarrow char : \{\}, \mathtt{Prelude.Char}^* \quad \text{PCHAR}$$

$$GE, IE \overset{pat}{\vdash} string \rightsquigarrow string : \{\}, [\mathtt{Prelude.Char}^*] \quad \text{PSTRING}$$

$$\frac{\begin{array}{l} IE \overset{literal}{\vdash} integer \rightsquigarrow \mathsf{e} : \tau \\ IE \overset{dict}{\vdash} \mathsf{e}_{\mathsf{d}} : \mathtt{Prelude.Eq}^* \ \tau \end{array}}{GE, IE \overset{pat}{\vdash} integer \rightsquigarrow \{ \mathtt{(Prelude.==)} \ \tau \ \mathsf{e}_{\mathsf{d}} \ \mathsf{e} \} : \{\}, \tau} \quad \text{PINTEGER}$$

$$\frac{\begin{array}{l} IE \overset{literal}{\vdash} float \rightsquigarrow \mathsf{e} : \tau \\ IE \overset{dict}{\vdash} \mathsf{e}_{\mathsf{d}} : \mathtt{Prelude.Eq}^* \ \tau \end{array}}{GE, IE \overset{pat}{\vdash} float \rightsquigarrow \{ \mathtt{(Prelude.==)} \ \tau \ \mathsf{e}_{\mathsf{d}} \ \mathsf{e} \} : \{\}, \tau} \quad \text{PFLOAT}$$

Fig. 44. Patterns, part 2 (literals).

than any other type scheme that can be derived for e in E. A type system has the principal type property if there is a principal type for every pair of program and typing environment for which the program is well typed.

It has generally been believed that Haskell has the principal type property (for instance, in section 4.1.4 of the Report it is stated that a Haskell type inferencer can find the principal type of every expression). We have however encountered a counter example involving the monomorphism restriction. Consider the following

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

code:

```
class IsNil a where
  isNil :: a -> Bool

instance IsNil [b] where
  isNil [] = True
  isNil _  = False

f x y = let g = isNil
        in (g x, g y)
```

The monomorphism restriction applies to the binding of g since there is no type signature. Thus it is not legal to derive $\forall a.\texttt{IsNil}\ a \Rightarrow a \to \texttt{Bool}$, but two legal possibilities are $\forall b.[b] \to \texttt{Bool}$ and $a \to \texttt{Bool}$ (without quantification and with $\texttt{IsNil}\ a$ in the instance environment). These two choices lead to different types for f:

- $\forall a\, b.[a] \to [b] \to (\texttt{Bool}, \texttt{Bool})$, and
- $\forall a.\texttt{IsNil}\ a \Rightarrow a \to a \to (\texttt{Bool}, \texttt{Bool})$.

These two are incomparable since there are neither any types τ_1 and τ_2 such that $([a] \to [b] \to (\texttt{Bool}, \texttt{Bool}))[\tau_1/a, \tau_2/b] = a \to a \to (\texttt{Bool}, \texttt{Bool})$ nor any type τ such that $(a \to a \to (\texttt{Bool}, \texttt{Bool}))[\tau/a] = [a] \to [b] \to (\texttt{Bool}, \texttt{Bool})$.

Although we have not formally proved so, we conjecture that there is no legal type for g that will allow a type for f that is more general than both of the above.

One way of giving Haskell principal types (unless there are further problems) would be to remove the monomorphism restriction. However, it serves a useful purpose by ensuring that sharing is preserved in bindings, a property that is pragmatically useful. One possibility would be to have two forms of binding, one which preserves sharing but is monomorphic and one which is polymorphic but does not necessarily preserve sharing. This has been suggested by Hughes (2001).

10.2 Ambiguity

Ambiguity is related to an important property called *coherence*. A type and translation system is coherent if, whenever it is possible to derive more than one translation for a term at a certain type, all of the derivable translations have the same semantics. Obviously, we would like the Haskell type system to have this property. It doesn't. For a counterexample, consider the following situation, where we have omitted kind annotations to reduce clutter: Let IE be an instance environment at least containing

```
readIntD  : P!Read P!Int
showIntD  : P!Show P!Int
readIntD  : P!Read P!Bool
showIntD  : P!Show P!Bool
```

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

and let VE be a variable environment at least containing

$$\begin{array}{ll} \texttt{read} & : \langle \texttt{P!read}, \forall a.\, \texttt{P!Read}\ a \Rightarrow_c [\texttt{P!Char}] \rightarrow a \rangle \\ \texttt{show} & : \langle \texttt{P!show}, \forall a.\, \texttt{P!Show}\ a \Rightarrow_c a \rightarrow [\texttt{P!Char}] \rangle \end{array}$$

and let GE be any global environment. Then both of the judgements below are derivable:

$GE, IE, VE \overset{exp}{\vdash}$ `show (read "True")`
$\rightsquigarrow$ `P!show P!Int showIntD (P!read P!Int readIntD "True")`
: `[P!Char]`

$GE, IE, VE \overset{exp}{\vdash}$ `show (read "True")`
$\rightsquigarrow$ `P!show P!Bool showBoolD (P!read P!Bool readBoolD "True")`
: `[P!Char]`

but the two translations have different semantics although the derived type is `[P!Char]` in both cases.

This should not come as too much of a surprise, though. If we look at the expression `show (read "True")`, it is clear that, given that `show` is overloaded on its argument and `read` is overloaded on its result, the intermediate result `read "True"` could have any type whatsoever.

Fortunately, there is a simple way to detect when incoherence might occur. For a smaller language, it is shown in Jones (1993) that if the principal type of an expression is *unambiguous*, all translations of the expression have the same semantics. A type scheme of the form $\forall \alpha_1, \ldots, \alpha_n.\, \theta \Rightarrow \tau$ is unambiguous if every α_i occurring in θ also occurs in τ. In the example above, the most general type of the expression is $\forall a.$ (`P!Show` a, `P!Read` a) $\Rightarrow$ `[P!Char]` (modulo the fact that we do not generalize over expressions, only at bindings). The type variable a clearly makes the type scheme ambiguous. While Jones' result is for a simpler language (essentially the lambda calculus with `let`), we conjecture that it carries over to Haskell (provided that principal types can be recovered).

If a type inference algorithm computes principal types it is very simple to include a check for ambiguity each time a type scheme is constructed. Unfortunately, it turns out that it is not so easy to extend a set of inference rules in the same way. The problem is that the check must be made on the principal type of (in our case) a set of bindings and a set of inference rules can be used not only to derive that type, but also every substitution instance of it. We have seen an example of this already; not only can we derive the type $\forall a.$ (`P!Show` a, `P!Read` a) $\Rightarrow$ `[P!Char]` for `show (read "True")`, but we can also derive just `[P!Char]`, with several semantically different translations.

One might think that the problem can be solved by insisting on deriving the most general type, with type schemes ordered according to the generic instance relation. Unfortunately, the type schemes `[P!Char]` (which is a type scheme with no quantified variables and empty context) and $\forall a.$ (`P!Show` a, `P!Read` a) $\Rightarrow$ `[P!Char]` are both generic instances of each other according to the standard definition of generic instance for qualified types (Jones, 1993).

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

One further complication is that ambiguity is not always illegal in Haskell. It occurs so often in connection with numbers that Haskell includes a *defaulting* mechanism used to resolve it. The programmer can provide a list of numeric types to use for defaulting. An ambiguous type scheme $\forall\alpha_1,\ldots,\alpha_n.\theta \Rightarrow \tau$ can be defaulted by instantiating some of the α_i as follows: If α_i occurs in θ but not in τ and the part of θ which mentiones α_i is $\Gamma_1\,\alpha_i,\ldots,\Gamma_k\,\alpha_i$ where all of the Γ_j are classes defined in the Prelude or the standard libraries, at least one of the Γ_j is a numeric class, and at least one of the types in the default list is an instance of all of the Γ_i. If several types satisfy that condition, the first one in the list is chosen.

We will now discuss some of the options we see for the formalization of ambiguity detection and resolution.

10.2.1 Using an inference algorithm

The approach taken by Jones (1999) is to use a type inference algorithm as a specification. It has the advantage that it either gives a program a unique type and translation or rejects it, effectively sidestepping the entire issue of coherence. Since the algorithm is the only description of the type system, the question of principal types also becomes moot. The major disadvantage is complexity; an inference algorithm is necessarily less abstract than a set of inference rules. In particular, it must deal with the details of computing the types, for instance using unification and substitutions. The description therefore becomes larger and perhaps less readable.

10.2.2 Using deterministic inference rules

Deterministic inference rules is an intriguing idea which may or may not be feasible. It involves designing a set of inference rules such that given a global environment GE, a variable environment VE and an instance environment IE containing only the information from (possibly imported) class and instance declarations and an expression e, there would only be one triple OE, e, τ such that $GE, IE \oplus OE, VE \overset{exp}{\vdash} e \rightsquigarrow \mathsf{e} : \tau$. The rules given in this paper may form a good starting point since they are mostly syntax directed. One issue would be the *dict* judgement since there might be situations where both the DICT INST and the DICT SUPER rules are applicable.

The largest hurdle to overcome, however, is that some rules (e.g. the APP rule) have premises where types occur which do not occur in the conclusion. In general these types can be chosen in different ways, yielding different translations, and some way of picking one of the candidates is necessary. The natural choice is to pick one that is in some sense the most general, but it is not clear how to define this formally.

If this approach is feasible, it would make it possible to outlaw ambiguity since there would not be a way to cheat by choosing a less general derivation. In this respect, deterministic inference rules are similar to an inference algorithm as specification. The main difference would be that the deterministic inference rules would not necessarily be easy to translate into an implementation which might allow a more abstract description.

https://doi.org/10.1017/S0956796802004380 Published online by Cambridge University Press

10.3 Other issues

In this section, we will give some, necessarily somewhat subjective, comments on the language. The greatest positive surprise was that kind defaulting (section 3.1) could be handled in such a relatively straightforward way, and we had expected worse problems. That said, it is clear that the choince of interpretation of the kinding dependency relation was influenced by what was simplest to formalize. Counting occurrences inside default methods would have lead to a large number of extra inference rules.

The pervasive influence of the module system was also a surprise. Most of the decisions about what information to keep in which kind of environment (section 2.7) was motivated by issues arising from selective import of class methods, data constructors and field labels. On the other hand, the fine-grained scope control offered by Haskell's `import` declarations is clearly useful on occasion, especially when using several large and independently developed libraries. We can see no easy way around this complexity.

The ugliest part of this formalization is the rules for algebraic datatypes with named fields (section 5.2.1). There are two sources of this ugliness, contexts in type declarations and the indirect specification in the Report. Contexts in algebraic datatypes play no essential role in the language, only moving type errors from one part of the program to another. It is doubtful whether that feature is worth the considerable added complexity. As for the other source, the inference rules would clearly be simplified if in an expression $e \Leftarrow \{fbind_1, \ldots, fbind_n\}$ the expression as a whole always had the same type as the subexpression e.

11 Conclusions

We have presented a set of inference rules which formalize the description in the Haskell Report. This can be used by programmers and implementors as well as form a basis for further formal investigations into the type system of Haskell. Such investigation is indeed needed before these inference rules can be used as a specification of the language.

One preliminary result of such an investigation is that Haskell does not have principal types due to a subtle problem with the monomorphism restriction. The lack of this property is troublesome since the rules in this paper admit more than one typing derivation for a given module while a compiler must generate only one type for every exported identifier.

Acknowledgements

Starting from Simon Peyton-Jones' and Phil Wadler's draft semantics provided a valuable head start, although the final product was substantially different and more complicated. I have also benefited from correspondance with them and with Mark Jones and by the comments of an anonymous referee.

References

Augustsson, L. (1993) Implementing Haskell overloading. *Fpca*. Functional Programming Languages and Computer Architecture.

Girard, J.-Y. (1972) *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur*. These D'Etat, Universite Paris VII.

Hall, C. V., Hammond, K., Peyton Jones, S. L. and Wadler, P. (1996) Type classes in Haskell. *Trans. Program. Lang. & Syst*. **18**(2).

Hudak, P, and Wadler, P. (eds) (1990) *Report on the programming language Haskell: A non-strict, purely functional language*. Version 1.0.

Hughes, J. (2001) Message to the Haskell mailing list.

Johnsson, T. (1985) Lambda lifting: transforming programs to recursive equations. *Functional Programming Languages and Computer Architecture*, Nancy, France. Springer-Verlag.

Jones, M. (1995) A system of constructor classes: overloading and implicit higher-order polymorphism. *J. Functional Programming*, **5**(1).

Jones, M. (1999) Typing Haskell in Haskell. *Proceedings 3rd Haskell Workshop*. Available from `www.cse.ogi.edu/~mpj/thih/`

Jones, M. P. (1993) *Coherence for qualified types*. Technical report YALEU/DCS/RR-989, Yale University, New Haven, CT, USA.

Kfoury, A. J., Tiuryn, J. and Urzyczyn, P. (1993) Type recursion in the presence of polymorphic recursion. *Trans. Program. Lang. & Syst*. **15**(2), 290–311.

Peyton Jones, S. and Wadler, P. (1991) *A static semantics for Haskell*. Draft.

Peyton Jones, S., Jones, M. and Meijer, E. (1997) Type classes: exploring the design space. *Haskell Workshop*.

Peyton Jones, S., Hughes, J. *et al*. (1999) *Report on the programming language Haskell 98*. Available from `www.haskell.org`.

Reynolds, J. C. (1974) Towards a theory of type structure. *Paris Colloq. on Programming: Lecture Notes in Computer Science 19*, pp. 408–425. Springer-Verlag.

Wadler, P. and Blott, S. (1989) How to make *ad-hoc* polymorphism less *ad-hoc*. *Conference Record 16th Annual ACM Symposium on Principles of Programming Languages*.