A Formal Specification of the Haskell 98 Module System

Iavor S. Diatchki, Mark P. Jones, Thomas Hallgren
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Rd, Beaverton, Oregon, USA
{diatchki,mpj,hallgren}@cse.ogi.edu

Abstract

Many programming languages provide means to split large programs into smaller modules. The module system of a language specifies what constitutes a module and how modules interact.

This paper presents a formal specification of the module system for the functional programming language Haskell. Although many aspects of Haskell have been subjected to formal analysis, the module system has, to date, been described only informally as part of the Haskell language report. As a result, some aspects of it are not well understood or are under-specified; this causes difficulties in reasoning about Haskell programs, and leads to practical problems such as inconsistencies between different implementations. One significant aspect of our work is that the specification is written in Haskell, which means that it can also be used as an executable test-bed, and as a starting point for Haskell implementers.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics, Haskell*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages, Recursion, Haskell*

General Terms

Documentation,Standardization,Languages

Keywords

Specification, Haskell, module system

1 Introduction

Many programming languages claim some kind of *module system* as part of their definition. In each case, the module system is intended to provide support for modular construction of software systems, but the precise interpretation of the term can vary quite significantly from one language to the next. In some languages, the module system provides a powerful mechanism for creating, using, and reusing *programming abstractions*. Standard ML, for example, has one of the most powerful module systems of this kind [9]. In other languages, the main purpose of the module system is to support *separate compilation*, motivated by pragmatic issues that arise during the development of large programs. In yet other languages, the module system serves primarily as a mechanism for *namespace management*, allowing programmers to control the visibility of defined names, either to hide implementation-specific details or to access parts of the program that would otherwise be out of scope. Of course, this classification is somewhat subjective, often depending on emphasis and the issues that are most directly targeted; some programming language module systems make a good attempt to serve several of these (or other) goals simultaneously.

The goal of this paper is to provide a formal description for the module system of Haskell 98, which falls most directly into the namespace management category that was described above. Although many aspects of Haskell have been studied previously, we are not aware of any other attempts to formalize its module system. Moreover, as readers of the Haskell mailing list may confirm, the module system is one of the least understood aspects of the language, and one in which some of the greatest variations between different implementations can be found.

In the spirit of the type system specification of Jones [5], this paper is a typeset version of a literate Haskell script, which is an executable specification of the module system. The number of lines of code in the specification is about 200.

In writing the specification, we have focused more on clarity and readability than efficiency. Nevertheless, the code presented in this paper has been developed in the context of a full-scale front-end for Haskell that works well in practice. For example, we have used the front-end to parse and process the complete source for the Alfa proof editor [**?**] (comprising on the order of 50,000 lines in 500 modules) in approximately 90 seconds (on a 1.9GHz Pentium 4).

1.1 Haskell Modules

A Haskell program is a collection of *modules*. A typical module defines a number of *entities* (functions, data types, classes, etc.), imports entities defined in other modules, and export some of the locally available entities for use by other modules. Mutual dependencies between modules are allowed.

Environments

Within the context of a particular Haskell module—or, indeed, at the prompt of an interpreter—there are top-level *environments* (also known as *symbol tables*) that associate names with the entities to which they refer. One of the main goals of this paper is to specify and describe how these environments are constructed. Following conventional wisdom, we might be tempted to use finite maps as the representation for environments. For the semantics of Haskell, however, *finite relations* are more appropriate because they allow us to capture the possibility that a name has zero, one, or multiple interpretations. More specifically, a name *n* with zero or multiple interpretations causes an error only if it is actually used in the scope of the environment. Of course, the type of error that is reported will be different in each case: if there are no interpretations for the name, then a reference to *n* indicates a reference to an unbound name; if there are multiple interpretations, then it indicates an ambiguous reference. In our specification we shall often refer to the relation modeling the symbol table of a module as its *in-scope relation*.

By using relations, we could also give a semantics for the renaming imports found in earlier versions of Haskell. Because we focus on Haskell 98, we do not pursue this idea further here.

Interfaces

Another important aspect of a module is its *interface*. It defines a set of names (with corresponding entities), which are made available for other modules to use. The main use of this feature is to avoid cluttering other modules with spurious names. It also provides a simple abstraction mechanism: by controlling what names are available to other modules, a programmer can enforce abstractions. Because the module interface is essentially a subset of the symbol table of a module, we also model it as a relation. We often refer to this relation as the *export relation* of the module.

1.2 Scope and contributions of this paper

The formal semantics given in this paper captures the semantics of the Haskell 98 module system in the following sense: given a collection of Haskell modules, the semantics

- computes the in-scope and export relations of each module.
- checks the correctness of all import and export specifications in the modules.

By using the computed in-scope relations, one can determine for each name that occurs in the body of a module which entities (zero, one or more) it refers to. The local scoping rules within module bodies are *not* part of the presented semantics, however, so it does not tell you how to detect references to unbound or ambiguous names occurring in module bodies. That check can be done one module at a time, without further reference to the module system semantics.

The specification can thus be seen as partitioning the semantics of Haskell 98 programs into a *module system specific part*, and a *module system independent part*. But apart from determining what names are in scope, module boundaries affect the meaning of Haskell programs in two other ways: the scope of a *default* declaration is limited to the module it occurs in, and type ambiguities caused by the monomorphism restriction[1] are resolved locally within each module.

The starting point for our work was the original Haskell 98 report [10]. It has since, in part as a result of our work, been revised, and the semantics presented in this paper is intended to be consistent with the current version of section 5 of the report [2]. One feature that is not yet covered by our semantics, is the visibility of instances, as described in Section 5.4 of the report.

1.3 Outline of the paper

The rest of paper is organized as follows: Section 2 introduces relations and operations on them; Section 3 gives definitions dealing with names and entities; Section 4 presents an abstract syntax for the module system; Section 5 describes the semantics of Haskell modules, i.e., the meaning of import and export declarations; Section 6 states criteria for the detection of invalid modules. Section 7 glues everything together and discusses some practical issues, such as separate compilation. Related work is discussed briefly in Section 8. Conclusions and further discussion appear in Section 9.

2 Relations

In this section, we present a number of operators for manipulating relations. To represent relations we use the *Set* library provided with the GHC and Hugs Haskell implementations. However, the specification in this paper uses only the operators defined here, so any other representation would do as well.

```
type Rel a b = Set (a,b)
```

Next we describe a number of simple operations on relations. Most of them require the elements to be in the class *Ord*. This is due to the implementation of the *Set* library. A different representation may relax or strengthen these requirements.

The operations *listToRel* and *relToList* allow us to switch between relations represented as sets, and relations represented as association lists.

```
listToRel :: (Ord a,Ord b) ⇒ [(a,b)] → Rel a b
listToRel xs = mkSet xs

relToList :: Rel a b → [(a,b)]
relToList r = setToList r
```

The empty relation is *emptyRel*. It does not relate any elements at all.

```
emptyRel :: Rel a b
emptyRel = emptySet
```

The combinators *restrictDom* and *restrictRng* restrict the domain and range, respectively, of a relation *r*, to the elements satisfying a predicate *p*.

```
restrictDom :: (Ord a, Ord b) ⇒
  (a → Bool) → Rel a b → Rel a b
restrictDom p r = listToRel [(x,y) | (x,y) ← relToList r, p x]
```

[1] More correctly referred to as the *annoying* monomorphism restriction :-)

```
restrictRng :: (Ord a, Ord b) ⇒
  (b → Bool) → Rel a b → Rel a b
restrictRng p r = listToRel [(x,y) | (x,y) ← relToList r, p y]
```

To access the domain and range of a relation, we use the functions *dom* and *rng*, respectively.

```
dom :: Ord a ⇒ Rel a b → Set a
dom r = mapSet fst r

rng :: Ord b ⇒ Rel a b → Set b
rng r = mapSet snd r
```

Sometimes it is useful to apply a function to all elements in the domain or range of a relation. This is the task of *mapDom* and *mapRng*, respectively.

```
mapDom :: (Ord b, Ord x) ⇒
   (a → x) → Rel a b → Rel x b
mapDom f = mapSet (\(x,y) → (f x, y))

mapRng :: (Ord a, Ord x) ⇒
   (b → x) → Rel a b → Rel a x
mapRng f = mapSet (\(x,y) → (x, f y))
```

We also need to be able to compute the intersection and union of relations. Elements are related by the intersection of two relations, if they are related by *both* relations. They are related by the union of two relations, if they are related by *either* one of them.

```
intersectRel :: (Ord a, Ord b) ⇒
  Rel a b → Rel a b → Rel a b
r ‘intersectRel‘ s = r ‘intersect‘ s

unionRels :: (Ord a, Ord b) ⇒ [Rel a b] → Rel a b
unionRels rs = unionManySets rs
```

The function *minusRel* computes the complement of a relation with respect to another relation. The new relation relates all those elements that are related by *r*, but not by *s*.

```
minusRel :: (Ord a, Ord b) ⇒
  Rel a b → Rel a b → Rel a b
r ‘minusRel‘ s = r ‘minusSet‘ s
```

Given a predicate *p* over the domain of a relation *r*, *partitionDom* produces two new relations: the first one is the subset of *r* whose first component satisfies *p*, and the second is the rest of *r*.

```
partitionDom :: (Ord a, Ord b) ⇒
  (a → Bool) → Rel a b → (Rel a b, Rel a b)
partitionDom p r = (restrictDom p r, restrictDom (not . p) r)
```

So far we have been thinking of relations as sets of pairs. An alternative view is to think of them as functions, which given an element of the domain, return all related elements in the range. The function *applyRel* converts a relation to a function form.

```
applyRel :: (Ord a, Ord b) ⇒ Rel a b → a → [b]
applyRel r a = setToList (rng (restrictDom (== a) r))
```

Finally we define the operation *unionMapSet*, which is the “bind” operator of the set monad. It is not an operation on relations, but rather on arbitrary sets. It is missing from the *Set* library, so we define it here.

```
unionMapSet :: Ord b ⇒ (a → Set b) → (Set a → Set b)
unionMapSet f = unionManySets . map f . setToList
```

3 Names and Entities

Having described the Haskell module system as a mechanism for name space management, it is natural for us to begin its specification with a discussion about names and entities.

3.1 Entities

The basic idea is that *names* in a program refer to *entities*. Entities get introduced in a program by *declarations*. For example, a declaration such as *f x = x + 2* will introduce one entity: a function named *f*. For the purposes of this paper, we are only interested in top level entities, as they are manipulated by the module system. There are at least six varieties of entities in Haskell: functions, type constructors, value constructors, field labels, classes, and class methods. One could perhaps also consider class instances to be entities as they are also introduced by declarations. We do not do that here because, in Haskell 98, there is no way to refer to them by name.

The module system specification is parametrized by the type of entities, so we represent it using an abstract data type:

```
data Entity = ...
isCon :: Entity → Bool
owns  :: Entity → Set Entity

instance Ord Entity where ...
```

The function *isCon* is intended to distinguish between value constructors and other entities, as they need to be handled differently in “hiding” imports as opposed to normal imports and exports (see Section 5).

The function *owns* defines the *subordinate* relation between entities. Type constructors “own” their value constructors and field labels; classes “own” their methods.

The requirement that entities are in the *Ord* class is stronger than strictly necessary. For the module system to work, we only need an equality operation. The ordering is required by the implementation of relations as sets (described in Section 2).

Entities will be written in a different font, and annotated with the module where they were originally defined. For example f_M refers to the entity *f* originally defined in module *M*.

3.2 Names

Our specification is also parameterized by the types used to model names. We distinguish between three different kinds of names:

- *simple names* (of type *Name*) are used in declarations, and to name entities exported by a module.
- *program identifiers* (whose type is *QName*) are used in the main text of the program and refer to entities. They may be either qualified or unqualified.
- *module names* and *name qualifiers* (with type *ModName*) are used to name modules, in import declarations, in export lists, and in qualified names.

We use the type *Name* whenever we want to indicate that only simple (i.e. not qualified) names are allowed, and *QName* when both simple and qualified names may be used. As for entities, the module system only needs equality operations on names, but to use the *Set* data structure we require the *Ord* instance.

```
data Name     = ...
data ModName  = ...
data QName    = ...

getQualifier :: QName → Maybe ModName
getQualified :: QName → Name
mkUnqual     :: Name → QName
mkQual       :: ModName → Name → QName

instance Ord Name where ...
instance Ord ModName where ...
instance Ord QName where ...
```

We define a couple of useful functions to manipulate (possibly qualified) names. We note that if *qual* is applied to an already qualified name, it will replace the old qualifier (however in this specification we always apply it to unqualified names).

```
isQual :: QName → Bool
isQual = isJust . getQualifier

qual :: ModName → QName → QName
qual m = mkQual m . getQualified
```

It is also convenient to define an overloaded function *toSimple*, which produces the unqualified part of a name. It does nothing on values of type *Name* as they cannot be qualified. For values of type *QName* it strips the qualifiers.

```
class ToSimple t where
  toSimple :: t → Name

instance ToSimple Name where
  toSimple = id

instance ToSimple QName where
  toSimple = getQualified
```

In examples throughout the paper we shall use *String* for *Name*, *ModName*, and *QName*. This is not the case in our prototype implementation, as it defeats the purpose of having three different types in the first place. We made this choice to keep the examples readable.

4 Abstract Syntax

As described in the Haskell 98 report [2, Section 5.1], a Haskell module consists of a name, an export specification, a number of import declarations and a number of local definitions. We use the following data structure to represent modules:

```
data Module = Module {
   modName    :: ModName,
   modExpList :: Maybe [ExpListEntry],
   modImports :: [Import],
   modDefines :: Rel Name Entity }
```

The concrete syntax of Haskell allows an abbreviated form, where the module name and the export specification are omitted. This is an abbreviation for a module with name “Main” and an export specification exporting a single entity named “main” [2, Section 5.1] and will in our abstract syntax be represented in its expanded form.

An element of the export specification is either an entity name or a module name as described by the data structure *ExpListEntry*. For entities with subordinate names, a programmer may also provide a *subordinate export list*. This list is modeled by the data structure *SubSpec*. It specifies which of the subordinate entities currently in scope are to be exported.

```
data ExpListEntry = EntExp (EntSpec QName)
                  | ModuleExp ModName
data EntSpec j = Ent j (Maybe SubSpec)
data SubSpec   = AllSubs | Subs [Name]
```

Example: For the Haskell module:

```
module A (f,C(..),module M) where ...
```

the field *modExpList* would be:

```
Just [EntExp (Ent “f” Nothing),
      EntExp (Ent “C” (Just AllSubs)),
      ModuleExp “M”]
```

□

The structure *EntSpec* is used in both import and export lists. Because qualified names are allowed in export lists, but not in import lists, we use the parameter *j* to capture the different types of *EntSpec*.

At first it may seem that we may eliminate *AllSubs* by thinking of it as just an abbreviation for all the methods/value constructors of its owner. This however is not the case, as its meaning depends on what entities are currently in scope, and this is one of the things the module system computes.

The lack of an export list is a special form of export specification: one saying that only—and all—locally defined entities are to be exported [2, Section 5.2]. It is *not* an abbreviation for the empty export list, or the export list containing only *module M* (where *M* is the current module). We represent this explicitly by using the *Maybe* type constructor in the *modExpList* field.

To make use of entities defined in other modules, programmers have to supply *import declarations*. Their purpose it to specify what entities are to be imported, which module provides the required entities, and valid ways to refer to the imported entities.

```
data Import = Import {
   impQualified :: Bool,
   impSource    :: ModName,
   impAs        :: ModName,
   impHiding    :: Bool,
   impList      :: [EntSpec Name] }
```

The *impSource* field is the only field that must be specified explicitly in an import specification. It specifies the name of the module from which entities will be imported. All remaining fields take on a default value, if not specified explicitly.

There are two flavors of import declarations: the ones specifying what names are to be imported, and the ones specifying what names are *not* to be imported (sometimes called "hiding" imports). The boolean field *impHiding* distinguishes between those two.

The field *impList* contains the actual specification, which has structure similar to the export list of a module. There are two differences: there are no "module" imports, and all names in the list must be simple. To capture this similarity we reuse the *EntSpec* data type. If this field is omitted the specification is assumed to be *[]*, and the *impHiding* field is set to *True*. This has the effect of importing all exported entities of the source module.

Sometimes it is more convenient to qualify names imported from a module not using the module name, but some other alias instead. This is particularly useful if the name of the source module is quite long and a programmer needs to refer often to imported entities by their qualified names. The field *impAs* stores this alias. If the alias is omitted, this field is assumed to have the same value as the *impFrom* fields (i.e. we use the module name in qualified names).

Finally in some situations it might be preferable to only import entities with their qualified names. This can be done with the so called *qualified* imports. The field *impQualified* distinguishes qualified from normal imports.

Example: The import:

```
import Prelude as P hiding (and,Bool(True))
```

is represented by the data structure:

```
Import {impQualified = False,
        impSource    = "Prelude",
        impAs        = "P",
        impHiding    = True,
        impList      = [Ent "and" Nothing,
                         Ent "Bool" (Just (Subs ["True"]))]
       }
```

□

5 The semantics of imports and exports

Now we are ready to present a method for computing the in-scope and export relations of the modules in a program. The process proceeds in two stages. In this section we compute the relations, ignoring any errors that might occur (e.g. ambiguous or undefined exports). In Section 6, we check that each computed relation satisfies a number of additional requirements and produce appropriate error messages if any problems are detected. This approach simplifies the specification as we can concentrate on a single issue at a time. It also results in better error messages being reported to the users of our prototype system, because the module system analysis can continue, even in the presence of ambiguities. In Section 7 we glue everything together.

The computation phase, described in the remainder of this section, is itself split in three parts: first we describe how to export/import a single entity, next we present how to combine the meaning of entity specifications to obtain the meaning of complete import and export specifications. Finally, we describe how to handle mutually recursive modules.

5.1 Importing or exporting an entity

We already noted the similarity in the abstract syntax for exporting and importing an entity—they both made use of the *EntSpec* data structure. It is therefore not surprising that those two cases are handled in essentially the same manner. Our goal is to determine which name-entity pairs in an in-scope or export relation satisfy a certain *EntSpec* specification. The function *mEntSpec* formalizes this process.

```
mEntSpec :: (Ord j,ToSimple j) ⇒
   Bool →            -- is it a hiding import?
   Rel j Entity →    -- the original relation
   EntSpec j         -- the specification
     → Rel j Entity -- the subset satisfying the specification

mEntSpec isHiding rel (Ent x subspec) =
   unionRels [mSpec, mSub]
   where
   mSpec   = restrictRng consider (restrictDom (== x) rel)
   allSubs = owns ‘unionMapSet‘ rng mSpec
   subs    = restrictRng (‘elementOf‘ allSubs) rel
   mSub    =
      case subspec of
        Nothing        → emptyRel
        Just AllSubs   → subs
        Just (Subs xs) →
          restrictDom ((‘elem‘ xs) . toSimple) subs

   consider
     | isHiding && isNothing subspec = const True
     | otherwise                     = not . isCon
```

Example: Before we describe *mEntSpec* in detail, we present an example of how it is going to be used. Consider the module:

```
module M (f, M.List(..), Show(showList)) where
import A(g)
...
```

The function *mEntSpec* is applied to each of the three entries in the export list, to determine the subset of the in-scope relation each of them matches. Similarly, it is applied to the entry *g* of the import list, to determine which of *A*'s exports come in scope. □

The *EntSpec* structure consists of two components: the "main" specification, and possibly a subordinate specification. The meaning of the entire specification is the union of its components. The subset of *rel* which matches the "main" specification is *mSpec*. It is computed by restricting the domain of the relation to contain only names matching the specification. It turns out we might need to restrict the range of the relation as well, the reasons for this are discussed shortly. Typically (but not always!) *mSpec* will be a relation only relating a single name to an entity, representing the fact that the specification is unambiguous.

Continuing with the example above, *Show* is the "main" specification, and *mSpec* would be:

$$Show \mapsto \mathsf{Show}_{Prelude}$$

The meaning of the subordinate specification depends on the meaning of the "main" specification. The set *allSubs* contains all subordinate entities of all possible interpretations in *mSpec*. To compute the subordinates that are in-scope/exported, we restrict *rel* so that names may only refer to entities in *allSubs* — the result is *subs*. In our example *allSubs* would contain the entities of the methods in the *Show* class:

$$\{\mathsf{show}_{Prelude}, \mathsf{showsPrec}_{Prelude}, \mathsf{showList}_{Prelude}\}$$

Now suppose that $\mathsf{showsPrec}_{Prelude}$ is not in scope, perhaps because the programmer explicitly hid it, when importing the *Prelude*. Then *subs* would be:

$$\begin{aligned} show &\mapsto \mathsf{show}_{Prelude} \\ Prelude.show &\mapsto \mathsf{show}_{Prelude} \\ showList &\mapsto \mathsf{showList}_{Prelude} \\ Prelude.showList &\mapsto \mathsf{showList}_{Prelude} \end{aligned}$$

Finally, we need to compute what part of *subs* matches the subordinate specification. If there was no specification, the result is the empty relation, if the specification was of the form *AllSubs* we return *subs*, as by construction it contains all subordinates which are in-scope or exported. Finally, if a programmer specified an explicit list of subordinates, we need to restrict *subs* so that it only contains names in the explicitly provided list. Since subordinate specifications contain only simple names (i.e. of type *Name*), and in-scope relations contain possibly qualified names (i.e. of type *QName*) we first strip any qualifiers using the method *toSimple*. This has the effect that a subordinate will be exported if it is available in scope with either qualified *or* unqualified name [2, Section 5.2]. In the ongoing example, we had a subordinate name list, containing one name: *"showList"*. So the restricted *subs* relation will be:

$$\begin{aligned} showList &\mapsto \mathsf{showList}_{Prelude} \\ Prelude.showList &\mapsto \mathsf{showList}_{Prelude} \end{aligned}$$

Next we describe a quirk in the Haskell module system, which gives rise to the *isHiding* parameter and the auxiliary predicate *consider*. The entities in Haskell may be grouped in two non-interchangeable groups: classes and type constructors on the one side, all other entities on the other. In the body of a module, it is always possible to determine which type of entity a name refers to. A name may refer to two different entities without the risk of an ambiguity. A problem occurs with import and export lists, as they may not provide enough context.

Example: It is quite common to use the same name for both a type and a value constructor, as types and values do not mix:

```
module A (Env) where
newtype Env a = Env [(String,a)]
```

There is no context in the export list to indicate if the programmer intended to export the type constructor *Env*, the value constructor *Env*, or perhaps both. □

To avoid the potential ambiguity illustrated in the example, Haskell 98 uses the following strategy to decide what is to be imported/exported.

Because only classes and type constructors may "own" other entities, the presence of a subordinate name list indicates that the programmer is referring to the type or class in scope. The situation is more complicated if the name does not have a subordinate list, as then there are two different policies, depending on where the name occurs.

The first policy is that names starting with a capital letter always refer to types or classes. It is used for names occurring in the export list of a module, and in non-hiding imports. So in particular, in the above example, only the type constructor *Env* will be exported, and not the value constructor. This means that in order to export/import a value constructor, a programmer has to include it in the subordinate list of the relevant type constructor. A consequence of this is that it is not possible to export just a value constructor without its type. In practice this is not a very big problem.

The second policy is used for "hiding" imports, and it says that capital names may refer to *both* types/classes and value constructors. One reason for this difference is that it is sometimes useful to *hide* just a value constructor without hiding its type. If the first policy was used for hiding imports this would not be possible. We note that if a name refers to both a type and a value constructor, both of them are hidden.

A (non Haskell 98) alternative to having two separate policies is to have a single more flexible policy that applies in both cases. Later in the paper (Section 9) we will describe a simple modification to the first policy to achieve this.

Here is an example illustrating what the policies do.

Example:

```
import A (Env)             -- import only the type
import A hiding (Env)      -- hide the type and the value
import A hiding (Env())    -- hide only the type
```

□

To implement this rule we defined the auxiliary predicate *consider*. The parameter *isHiding* tells us if we are in a hiding import (the special case). If we are, then we consider all entities as valid interpretations for the name in the "main" specification. However if we are using *mEntSpec* in an export list or a normal import, then we do not consider value constructors.

5.2 Export relations

Now that we know how to handle a single entry in the export list, we are ready to compute the export relation of a module. This is the task of the function *exports*.

```
exports :: Module → Rel QName Entity → Rel Name Entity
exports mod inscp =
  case modExpList mod of
    Nothing → modDefines mod
    Just es → getQualified ‘mapDom‘ unionRels exps
      where
      exps = mExpListEntry inscp ‘map‘ es
```

The parameter *inscp* models the in-scope relation of the module. It is necessary, as the interface of a module is essentially a subset of *inscp*. If the programmer omitted the export specification of a module, we just export the locally defined entities by using the *modDefines* field of the module under consideration. Alternatively if an explicit export list was present, we take the union of the meanings of all listed entries. To obtain a proper export relation we remove all qualifiers. Note that after removing the qualifiers (or even before that) we might end up with a relation containing ambiguous names. This is not valid in Haskell, but we will delay detection of such invalid solutions until a later pass.

We have already seen that there are two forms of specification that may appear in an export list. If we see an entity specification, we just use *mEntSpec* to determine the subset of *inscp* which matches it. The other possibility is that we see an export of the form *module M*. In this case, the Haskell 98 report [2, Section 5.2, item 5] states that the result is the subset of *inscp* containing precisely those entities, which may be named with *both* some simple name *x and* a qualified name *M.x*

```
mExpListEntry ::
    Rel QName Entity → ExpListEntry → Rel QName Entity
mExpListEntry inscp (EntExp it) = mEntSpec False inscp it
mExpListEntry inscp (ModuleExp m) =
    (qual m ‘mapDom‘ unqs) ‘intersectRel‘ qs
    where
    (qs,unqs) = partitionDom isQual inscp
```

5.3 In-scope relations

In this section we specify how to compute the in-scope relation of a module. This is done by the function *inscope*:

```
inscope :: Module → (ModName → Rel Name Entity)
                                  → Rel QName Entity
inscope m expsOf = unionRels [imports, locals]
  where
  defEnts = modDefines m
  locals  = unionRels
               [mkUnqual ‘mapDom‘ defEnts,
                mkQual (modName m) ‘mapDom‘ defEnts]

  imports =
     unionRels $ map (mImp expsOf) (modImports m)
```

An entity is in scope if it is either locally defined, or if it is imported from another module. It is therefore necessary to know what the exports of other modules are. The parameter *expsOf* is a function mapping module names to their exports.

Every locally defined entity may be referred to with at least two names: the simple name used in its definition, and a qualified name, obtained by prefixing with the name of the module. For example if a module *A* defines an entity with simple name *f*, a programmer may refer to it as either *f* or *A.f* [2, Section 5.5.1]. The part of the in-scope relation containing local definitions is *locals*.

Import declarations are cumulative [2, Section 5.3] and so the imported entities (*imports*) are the union of the entities imported by each declaration.

Example: The declarations:

```
import A hiding (f)
import A (f)
```

import everything from module *A*, as the first one imports everything but *f*, while the second one imports just *f*. □

The function *mImp* is used to compute what entities come in scope through a single import declaration.

```
mImp :: (ModName → Rel Name Entity) → Import →
            Rel QName Entity
mImp expsOf imp
   | impQualified imp = qs
   | otherwise        = unionRels [unqs, qs]
   where
   qs   = mkQual (impAs imp) ‘mapDom‘ incoming
   unqs = mkUnqual ‘mapDom‘ incoming

   listed  = unionRels $
                  map (mEntSpec isHiding exps)
                      (impList imp)
   incoming
     | isHiding  = exps ‘minusRel‘ listed
     | otherwise = listed

   isHiding = impHiding imp
   exps     = expsOf (impSource imp)
```

First we define the relation *listed*, which contains exported entities matching *any* of the entity specifications in the list of the import declaration. Next, we compute the subset of the export relation of the source module, which matches the specification (*incoming*). If we have a normal import, *incoming* is exactly *listed*. If however we are dealing with a “hiding” import, we need to take all those entities which are exported, but are *not* in *listed*.

Having computed *incoming*, we now need to convert it to an in-scope relation. To do that, we adjust the names of the entities according to the import declaration. If we have a “qualified” import, we introduce only qualified names for the imported entities (*qs*), otherwise we also add the unqualified names. The qualified names are computed by simply qualifying all names in *incoming* with the *impAs* specification of the declaration.

5.4 Recursive modules

The reader might have noticed that to compute the exports of a module we need to know what is in scope, and to compute what is in scope we need to know certain exports. If we allow for mutually recursive modules, then our equations become mutually recursive. In this section, we show how they are solved.

The idea is that the export and in-scope relations for a group of mutually recursive modules are computed at the same time (as they all depend on each other). This means that the compilation unit of a compiler is not a single module, but a *strongly connected component* of mutually recursive modules. In fact, one could process *all* modules at once and still get the same result, but this will not be very practical for large systems. For this reason, we will work with strongly connected components of modules, and assume that they are processed in dependency order.

The function *computeInsOuts* computes the in-scope and export relations of the modules in a strongly connected component. The argument *otherExps* is a function mapping module names to export relations. It needs to be defined only for modules in earlier (dependency-wise) strongly connected components.

```
computeInsOuts ::
  (ModName → Rel Name Entity) → [Module] →
   [(Rel QName Entity, Rel Name Entity)]
computeInsOuts otherExps mods = inscps ‘zip‘ exps
  where
  inscps = computeIs exps
  exps   = lfpAfter nextExps $
             replicate (length mods) emptyRel

  nextExps = computeEs . computeIs

  computeEs is = zipWith exports mods is
  computeIs es = map (‘inscope‘ toFun es) mods

  toFun es m    = maybe (otherExps m) (es !!)
                           (lookup m mod_ixs)
  mod_ixs       = map modName mods ‘zip‘ [0..]

lfpAfter f x = if fx == x then fx else lfpAfter f fx
  where
  fx = f x
```

The function *computeInsOuts* starts by assuming that the modules in the strongly connected component do not export anything. It then keeps applying the function *nextExps* to obtain successive approximations to the exports of each of the modules until a fixed point is reached. Thus, the export-relations in a strongly connected component are the least fixed point of the function *nextExps*. Similarly, the in-scope relations are the least fixed point of *computeIs . computeEs*, but using the well known fact that $fix(f.g) = f(fix(g.f))$, we obtain the definition of *inscps* used above.

The function *nextExps* computes the next approximation to the exports of each module. Given the current exports of each module, it first determines what are the corresponding new in-scope relations, and based on that computes the new exports. It makes use of the helper functions *computeEs* and *computeIs*, which are generalizations of *exports* and *inscope* respectively, to work with strongly connected components rather than just single modules.

The functions *computeEs* and *computeIs* work in a similar way: they apply *exports* (or *inscope* respectively) to all modules in the strongly connected component. The main difference between the two is that the export relation of a module depends only on the in-scope relation of the same module, while the in-scope relation depends on the export relations of many modules. As a result, before mapping *inscope* over the strongly connected component, we need to compute a function that maps module names to their respective export relations. This is done by *toFun*. If the module is in the current strongly connected component, the result of *toFun* just projects the appropriate export relation. Otherwise, we use the parameter *otherExps* to lookup the exports of a previously processed module.

Example: In this example we consider a module that imports itself, and use the algorithm above to compute its in-scope and export relations:

```
module A (B.f) where
import A as B
f = ...
```

We start with an empty export relation, and then compute the in-scope relation:

$$f \mapsto \mathsf{f}_A, A.f \mapsto \mathsf{f}_A$$

Note that this contains only the locally defined names, because *A* imports only from itself, and we have assumed it does not export anything. Because the in-scope relation does not relate *B.f* to anything, nothing is exported again, and so we reach a fixed point. Thus module *A* does not export anything, and it has one entity in-scope: f_A. This entity may be referred to by either *A.f*, or just *f*. □

Example: This example is a slight variation on the previous one, two modules are involved: *A* and *B*:

```
module A (B.f) where
import A as B
import qualified B
f = ...

module B where
f = ...
```

The dependencies between those two are: *A* depends on *A* and *B*, and *B* does not depend on anything, so we first analyze *B*. Since it has no export specification, it exports all locally defined names and nothing else, so we have the following in-scope and export relations:

in-scope	exports
$f \mapsto \mathsf{f}_B, B.f \mapsto \mathsf{f}_B$	$f \mapsto \mathsf{f}_B$

Next we analyze module *A*. The steps in the fixed point calculation are:

	in-scope	exports
1.		$\emptyset$
2.	$f \mapsto \mathsf{f}_A, A.f \mapsto \mathsf{f}_A, B.f \mapsto \mathsf{f}_B$	$f \mapsto \mathsf{f}_B$
3.	$f \mapsto \{\mathsf{f}_A, \mathsf{f}_B\}, A.f \mapsto \mathsf{f}_A, B.f \mapsto \mathsf{f}_B$	$f \mapsto \mathsf{f}_B$

We start with the empty export relation. The in-scope relation we compute contains the locally defined names and also the imports. In this case the imports are just $B.f \mapsto \mathsf{f}_B$. On the next iteration, $f \mapsto \mathsf{f}_B$ gets exported. So the next in-scope relation contains more imports: $f \mapsto \mathsf{f}_B$, and $B.f \mapsto \mathsf{f}_B$, which came in through the *import A* declaration. These do not add anything new to the exports of the module, so we have reached a fixed point. □

We defined the import and export relations of modules in terms of the least fixed point of a certain function. To ensure that the algorithm above terminates, we need to ensure that this least fixed point exists. Now we examine this issue in more detail.

We used lists to represent the in-scope and export relations of a group of modules. Since relations may be ordered by inclusion, we may also order lists (of equal length) of relations by a pointwise ordering. In a similar fashion we obtain a lattice structure on the lists of relations by using the lattice structure of relations pointwise. The lattice obtained in this way is finite, as each module may only define finitely many entities and each entity may only have finitely many names. Names get associated with entities in two ways: by local declaration, in which case an entity receives two names (qualified and unqualified); and by import declaration, in which case an entity receives either one or two names. Since there are only finitely many local and import declarations, an entity may only have finitely many names.

It then follows from the Knaster-Tarski theorem [1], that *nextExps* has a least fixed point if it is monotonic with respect to the above ordering. The *inscope* and *exports* functions are monotonic with respect to the relation ordering, as they are essentially filters that produce larger outputs when given larger inputs. The same holds for *computeEs*, *computeIs*, and their composition *nextExps*.

6 Error Detection

In the previous section we described how to compute the in-scope and export relations of mutually recursive modules. The algorithm produces a result even for modules containing errors. We now examine what properties need to be satisfied by correct solutions, and how we can detect "bad" solutions.

Even though this specification aims for clarity rather than efficiency or usability, we believe that it is important not only to detect invalid solutions, but also to say *why* they are invalid. The data type *ModSysErr* classifies the different kinds of problems which might occur.

```
data ModSysErr
  = UndefinedModuleAlias ModName
  | UndefinedExport QName
  | UndefinedSubExport QName Name
  | AmbiguousExport Name [Entity]
  | MissingModule ModName
  | UndefinedImport ModName Name
  | UndefinedSubImport ModName Name Name
  deriving Show
```

The meanings of the individual errors are as follows:

- *UndefinedModuleAlias* means that an export list contained an entry of the form *module M*, where *M* is not a valid alias.
- *UndefinedExport* refers to an entry in an export list, for which there is no corresponding entry in the symbol table.
- *UndefinedSubExport* is similar to *UndefinedExport*, except that it also reports the owner of the subordinate name.
- *AmbiguousExport* reports an exported name, together with all the possible entities that it might refer to.
- *MissingModule* is reported when an import declaration refers to a module that is missing.
- *UndefinedImport* is reported when an import declaration attempts to import (or hide) an entity that was not exported by the source module. The name of the source module is part of the error.
- *UndefinedSubImport* is similar to *UndefinedImport*, except that it also reports the owner of the undefined subordinate entity. We report the owner specified by the programmer in the import list.

In this section we preset functions to validate the import and export specifications of a module. The task of the function *chkModule* is to ensure that a module is valid from the point of view of the module system. To achieve this we need to check: (1) that the module interface is unambiguous; (2) that all referenced modules are present, and if so, (a) that each import declaration is valid; (b) that the export specification is valid. If some referenced modules are missing, we report that, but skip the remaining checks, since they might produce bogus error messages.

```
chkModule ::
  (ModName → Maybe (Rel Name Entity)) →
  Rel QName Entity →
  Module →
    [ModSysErr]

chkModule expsOf inscp mod
  = chkAmbigExps mod_exports
     ++ if null missingModules
        then chkExpSpec inscp mod
              ++ [err | (imp,Just exps) ← impSources,
                        err ← chkImport exps imp]
        else map MissingModule missingModules
  where
  Just mod_exports = expsOf (modName mod)

  missingModules =
    nub [impSource imp|(imp,Nothing)←impSources]
  impSources =
    [(imp,expsOf (impSource imp))|imp←modImports mod]
```

The parameter *expsOf* is a function, which maps module names to their export relations. The parameter *inscp* is the in-scope relation of the module we are checking. The result of *chkModule* is a list of errors detected in the module.

The export specification and the import declarations are checked by separate functions. *chkModule* provides the necessary information to each function, and collects their results in a single list of errors.

A module should not contain ambiguities in its interface. It is however possible—in fact quite common—to have the same name refer to a type constructor and a value constructor. As we previously discussed, this is not considered to be an ambiguity as we may determine from the context which one is meant.

```
chkAmbigExps :: Rel Name Entity → [ModSysErr]
chkAmbigExps exps = concatMap isAmbig
                        (setToList (dom exps))
  where
  isAmbig n =
    let (cons,other) = partition isCon (applyRel exps n)
    in ambig n cons ++ ambig n other

  ambig n ents@(_:_:_) = [AmbiguousExport n ents]
  ambig n _            = []
```

The function *chkAmbigExps* detects ambiguities in the export relation of a module (*exps*). For each name in the domain of *exps*, we use *applyRel* to compute the list of entities it may refer to. The function *isAmbig* detects any ambiguities in this list, considering value constructors and other entities separately.

We have already encountered some similarity between import declaration and export specifications. We exploit this again, by using the same function *chkEntSpec* to ensure that entries in export and import lists are defined. The parameters are essentially the same as in the *mEntSpec* function of the previous section, but we shall briefly describe them again. The boolean *isHiding* tells us if we are in the special case of hiding imports. The two functions *errUndef* and *errUndefSub* are new, and are needed so that we can report different errors for the import and export cases. Finally, we have the specification we are checking, and the relation modeling either the exports of the source module, or the symbol table of the current module.

```
chkEntSpec :: (Ord j, ToSimple j) ⇒
  Bool →                          -- is it a hiding import?
  (j → ModSysErr) →               -- report error
  (j → Name → ModSysErr) →        -- report error
  EntSpec j →                     -- the specification
  Rel j Entity →                  -- the relation to check
    [ModSysErr]                   -- detected errors

chkEntSpec isHiding errUndef errUndefSub
           (Ent x subspec) rel =
  case xents of
    []   → [errUndef x]
    ents → concatMap chk ents
  where
  xents = filter consider (applyRel rel x)

  chk ent =
    case subspec of
      Just (Subs subs) →
        map (errUndefSub x)
            (filter (not . (‘elementOf‘ subsInScope)) subs)
        where
        subsInScope =
          mapSet toSimple
             $ dom
             $ restrictRng (‘elementOf‘ owns ent) rel
      _ → []

  consider
    | isHiding && isNothing subspec = const True
    | otherwise                     = not . isCon
```

Despite the large number of arguments, the function is quite simple. We lookup what the name in the specification (*x*) may refer to, and if nothing was found we report an error. In case it was defined we check the subordinate list in two steps. First we compute the names of subordinate entities of *ent* which are also in *rel* (*subsInScope*). Then we make sure that all listed subordinates are in *subsInScope*. We do not consider ambiguities in *chkEntSpec*, as this is the task of the function *chkAmbigExps*. The predicate *consider* has the same role as in *mEntSpec*.

We now describe how to check an export specification. It may be either implicit or explicit. Implicit specifications are always correct. For an explicit specification we need to check all entries in the exports list. For entries of the form *module M* we need to ensure that *M* is a valid alias in this module. An alias is valid, if it is either introduced by an import declaration, or is the name of the current module. For other entries we need to check that the entities they refer to are defined by using the generic *chkEntSpec*.

```
chkExpSpec :: Rel QName Entity → Module → [ModSysErr]
chkExpSpec inscp mod =
    case modExpList mod of
      Nothing   → []
      Just exps → concatMap chk exps
  where
  aliases = modName mod : impAs ‘map‘ modImports mod

  chk (ModuleExp x)
    | x ‘elem‘ aliases = []
    | otherwise        = [UndefinedModuleAlias x]
  chk (EntExp spec) = chkEntSpec False
                  UndefinedExport UndefinedSubExport
                  spec inscp
```

The remaining check we have is the validity of import declarations. The process is quite similar to the checks of the export specification and we have already done all the hard work in *chkEntSpec*. The function *chkImport* just uses *chkEntSpec* to ensure the correctness of the entries in the specification list of the import.

```
chkImport :: Rel Name Entity → Import → [ModSysErr]
chkImport exps imp = concatMap chk (impList imp)
  where
  src     = impSource imp
  chk spec =
    chkEntSpec (impHiding imp)
      (UndefinedImport src) (UndefinedSubImport src)
      spec exps
```

7 The semantics of a Haskell program

Having defined the meaning of imports and exports, and how to detect errors, we can now glue everything together and define the meaning of a Haskell program.

The semantics of a Haskell program (with respect to the module system) is a mapping from a collection of modules to their corresponding in-scope and export relations. It can be given by a function of type:

```
mProgram ::
   [Module] → Either [[ModSysErr]]
                     [(Rel QName Entity,Rel Name Entity)]
```

Given a list of modules, the function either reports a list of errors found in each module, or returns the in-scope and export relations of the modules. There is a one-to-one correspondence between positions in the module list and positions in the resulting lists.

Using the functions defined Sections 5 and 6, we define the function *mProgram* as follows:

```
mProgram modules
   | not (null errs) = Left errs
   | otherwise       = Right rels
   where
   rels = computeInsOuts (const emptyRel) modules
   errs = zipWith (chkModule expsOf) inscps modules

   (inscps,exps) = unzip rels
   expsOf m      = lookup m mod_exps
   mod_exps      = map modName modules ‘zip‘ exps
```

It is assumed that implicit imports of the Prelude [2, Section 5.6.1] have been made explicit before *mProgram* is called. It is also assumed that all modules in the argument list have unique names.

While the function *mProgram* is sufficient to explain the meaning of a Haskell program, it would probably not be very practical in a Haskell implementation, since it does not support separate compilation. Instead of *mProgram*, we have implemented a more sophisticated function based on the same key ingredients: the functions *computeInsOuts* (which supports separate compilation) and *chkModule*, described in sections 5.4 and 6 respectively. Our Haskell front-end processes modules one strongly connected component at a time, caches module interfaces between runs, and has better error handling. We omit the implementation of these practical details from this presentation.

8 Related work

We are aware of at least two attempts to formalize the static semantics of Haskell, but neither of them fully specifies the module system. In the static semantics by Peyton Jones and Wadler [6], the specification of imports was left as future work. The authors rated it as one of the highest-priority items on their todo list. More recently, Faxén also worked on the static semantics of Haskell [3]. In this work, he gave semantics to some parts of the module system, but also deviated from the report, opting for what he considered to be a simpler (although non Haskell 98 compatible) specification (Section 4.2, [3]). Faxén's work is consistent with the report in that it does not specify how to treat mutually recursive modules.

Our specification of the Haskell module system is relatively independent of Haskell itself. In this respect it is similar to Leroy's work on ML's module system [7]. There have been numerous studies on advanced module systems, and the use of type theory to formalize them [4, 8]. In the same spirit, there has recently been a proposal for a replacement of the Haskell module system by Shields and Peyton Jones [11].

9 Conclusions and discussion

We have provided a formal specification of the Haskell 98 module system, based on the Haskell 98 language report. The process of writing the specification was valuable as we identified a number of areas of the report, which were unclear, or underspecified, and as a result the report has been improved. In particular, while the report mentions that mutually recursive modules are allowed, there is no mention of how they should work. Our specification provides a clear semantics for mutually recursive modules, and as far as we are aware, is the only implementation of the Haskell module system that supports this feature. It is possible to compile programs with mutually recursive modules using GHC [12, Section 4.9.7], but the programmer has to provide a special interface file, essentially implementing this aspect of the module system manually.

The Haskell module system aims at simplicity and has a clear goal—to manage name uses in a program. Its design has largely been driven by practical concerns, which has both positive and negative consequences. It works in practice and most of the time it does not place large cognitive overhead on the programmer. Our specification is not too complicated, and the few thorny parts of it point to possibilities for improvements in the design of the module system.

One of the complications is caused by the special rules used to distinguish between type and value constructors in import/export lists. These seem somewhat ad-hoc and are a source of unnecessary complexity. Many of the difficulties arise from the choice of meaning for capitalized names in import/export lists:

- in export lists and "normal" imports they refer to types or classes
- in "hiding" imports they refer to types, classes, or value-constructors

An alternative choice is to make them always refer to value-constructors. The presence or absence of a subordinate list may be used to distinguish types and classes, from value-constructors. Here is a table summarizing the difference between the current and the alternative interpretation:

what to name:	current	alternative
just type or class:	T or T()	T()
just constructor:	-	T

As we see, currently there are two different ways to name just a type or a class, and no way at all to just name a value-constructor. With our alternative interpretation, the meaning of an entry does not vary depending on the context (i.e., no need for special cases for "hiding" imports). This seems like an attractive idea, but unfortunately it changes the meaning of many Haskell programs. As such it is not feasible to introduce in Haskell 98, but perhaps it can be considered in future revisions of the language.

Haskell turned out to be a very suitable language for writing executable specifications. We found its clear syntax to be particularly valuable, providing a high-level of abstraction. The ability to type-check and execute the specification not only improved our confidence in its correctness, but also enabled us to compare it against the behaviors of a number of implementations such as Hugs and GHC.

10 Acknowledgments

We would like to thank the participants on the Haskell mailing list, and in particular Simon Peyton Jones, Simon Marlow, and Malcom Wallace, for valuable clarifications and discussions.

11 References

[1] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[2] K.-F. Faxén. A Static Semantics for Haskell. *Journal of Functional Programming*, 2002. To appear.

[3] T. Hallgren. Home Page of the Proof Editor Alfa. `http://www.cs.chalmers.se/~hallgren/Alfa/`, 1996-2002.

[4] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. of 1994 ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[5] M. P. Jones. Typing Haskell in Haskell. In *Proceedings of the 3rd Haskell Workshop*, Paris, France, October 1999.

[6] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[7] D. B. MacQueen. Using dependent types to express modular structure. In *Proc. of 1986 ACM Symposium on Principles of Programming Languages*, pages 277–286, January 1986. St. Petersburg.

[8] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[9] S. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98. Technical Report YALEU/DCS/RR-1106, Yale University, CS Dept., Feb. 1999.

[10] S. Peyton Jones and P. Wadler. A static semantics for Haskell. Unpublished draft, 1992.

[11] S. Peyton Jones (editor). Report on the programming language Haskell 98. `http://research.microsoft.com/~simonpj/haskell98-revised/`, March 2002.

[12] M. Shields and S. Peyton Jones. First class modules for Haskell. In *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon*, pages 28–40, Jan. 2002.

[13] The GHC team. The Glasgow Haskell Compiler user's guide version 5.04. `http://haskell.org/ghc/`, July 2002.