An Exploration of Modular Programs

Jan Nicklisch
Department of Computing Science, University of Glasgow
jan@dcs.gla.ac.uk

Simon Peyton Jones
Department of Computing Science, University of Glasgow
simonpj@dcs.gla.ac.uk

October 11, 1996

Abstract

Recently, Mark Jones introduced first class structures as a means to express modular structure. In this paper we elaborate on this idea by comparing the module systems of Standard ML and Haskell 1.3, two widely used functional languages, and a Haskell variant equipped with such first class structures. Moreover, we look at another obvious and well-known extension to Hindley-Milner type systems, namely higher order type variables, to explore its usefulness in solving problems occuring when one attempts to structure larger programs into maintainable pieces.

We argue that there are surprisingly few applications where the module system currently provided by Haskell cannot keep pace with Standard ML's expressiveness. When one adds first class structures to Haskell, the module system reaches the expressiveness of Standard ML and even exceeds it.

1 Preliminaries

It is widely agreed that modular programming (in the sense of using small program units, or *modules*, to construct more sophisticated units) is one of the key points in the successful construction of reusable and maintainable software. Literature on software engineering or module systems in general (for instance [3]) explains this in more detail.

In this text we compare the module systems of two popular functional languages, Standard ML and Haskell. We first relate one to the other, and then we compare them with a Haskell variant extended with so called first class structures, which we call Haskell* just to avoid confusion. First class structures were recently proposed by Jones, see [7]. Our goal is to show that such first class structures, also referred to as XHM-structures (to stand for extended Hindley-Milner structures), offer a range of interesting applications and may in particular be used to model the module system of Standard ML. While there are still several questions to answer and design decisions to make, we believe that first class structures add significant expressiveness and usefulness to the language and may even be considered as a future language extension for Haskell. We aim to justify this claim by showing that Haskell's module system in connection with first class structures is even more powerful than the module system of Standard ML.

Surprisingly enough, when one compares Standard ML's module system with the one of Haskell 1.3 it turns out that, from a purely application point of view, both seem to be equally powerful. It is hard to find convincing, real-life examples of larger ML programs which cannot be expressed in Haskell. The reason is that many of the programs using structures and functors exploit these constructs just to support modular programming, but scarcely for expressing generic algorithms. Therefore, even though the constructs for modular programming provided by Standard ML, i.e. structures and functors, are significantly more complex than the simple name space control system used to support modules in Haskell 1.3, many example programs can easily be translated. Especially larger, not purely academic ML programs do not seem to make use of all the possibilities offered by the language, in particular they do not apply the same functor more than once in one program. But as explained in Section 2.3, several applications of one functor to different sets of arguments are the only use of ML's module system which cannot be quite satisfactory simulated in Haskell.

Of course, in a comparison like this we neglect all issues concerning Standard ML features which cannot be straightforwardly simulated in a pure language, like exceptions and reference types. The main objective is to see how the structuring mechanisms of both languages relate. Our result is that the Haskell 1.3 module system allows for similar expressiveness as ML's module system. An important difference is to be found more on a software engineering rather than on a technical level: the use of signatures to explicitly describe the interface of a structure, i.e. a module, enhances readability and type safety, but cannot be modeled in Haskell.

To explore how the increased power of a Hindley-Milner type system equipped with first class structures adds to the usability of a functional programming language we show how first class structures can express modular systems. As a case study we translated a somewhat larger ML program, intended to serve as a showcase for the structuring mechanisms provided by ML structures and functors, into Haskell 1.3 equipped with first class structures, i.e. into Haskell*. As Jones in [7] points out, such a translation is straightforward and has merely to explain how type components of ML structures are described in a language where there are only polymorphic values to be found in (first class) structures, but no types. It turns out that the lifting of type definitions together with the original Haskell's name space control mechanism is sufficient to simulate all applications of ML's module systems, even for instance multiple functor applications. Additionally, Haskell's module system can truly hide the identity of a structure's internal types, which is – against the merit of abstract data types – revealed in Standard ML. Furthermore, there are obvious examples which cannot be expressed in Standard ML, but can be written using first class structures, e.g. higher order functors and recursive structures.

Since first class structures allow for local universal quantification it stands to reason to also allow for local existential quantification. Hence we have experimented with another extension of Haskell's type system, namely with the use of existential quantification on the level of data types. This is obvious in an exploration of modules and abstract data types, where proper information hiding is essential.

But our generall impression is that existential types merely serve as a means to model heterogeneous collections, as Läufer explains (see [9]), or to support idioms of object oriented programming in a functional setting in a style described by Pierce and Turner (see [11]). The conjecture one might make in the first place, namely that existential types prove to be useful for the modeling of abstract data types (where a hidden implementation type is desired), did not hold in our experiments. The simple and very syntactical support given by Haskell's module system, the mere name space management, is sufficient to hide implementation types. While there are neat ways of describing abstract data types using existential types, similar expressiveness can be achieved using standard data types in connection with a proper name space management. However, it is too early to state firm results.

In Section 2 we explain the module systems of Standard ML and Haskell 1.3 (i.e. Standard Haskell) and relate them to each other. We show the usefulness of higher order type variables in Section 3 Section 4 explores how first class structures improve the expressiveness of a language based on a Hindley-Milner type system, and Section 5 concludes.

2 Standard Haskell and Standard ML

In the core part of this paper we will show how some extensions of the Hindley-Milner type system support modular programming in a functional language like Haskell. But first we summarize the language features as they are currently present. We start by giving a short review of the module systems of Standard ML and Haskell 1.3. After this we show up to which extent the elaborated module system of ML may be modeled by Haskell's much simpler module system.

2.1 Modules in Standard ML

Standard ML has one of the most elaborated module systems available in widely used programming languages. We summarize the main points briefly here and refer to the literature ([10]).

The module system as a whole makes a distinction between the 'core language' and the 'module language' in the sense that the constructs used to express modularity add on the level of simple values. The two worlds cannot be mixed, this means that for instance structures cannot serve as function parameters or results.

An SML module is either a structure or a functor, where structures are basically collections of type and value definitions, and functors are like functions which act on such structures instead of values. There are no higher order functors, i.e. functors can not be arguments or results of other functors.

Signatures summarize type information by collecting specifications just as structures collect definitions. Tofte gives a readable and comprehensive summary of ML's module system, see [12] for more on this. Below is an example of a signature `ITEM`, a structure `IntItem`, and a functor `SquareItem` which takes any argument structure matching signature `ITEM` into another `ITEM`-structure.

```
signature ITEM =
sig
  type item
  val leq: item * item -> bool
  val initial: item
end;

structure IntItem =
struct
  type item = int
  fun leq(i:item, j): bool =
      i <= j
  val initial = 0
end;

functor SquareItem(i:ITEM): ITEM =
struct
  type item = i.item * i.item
  fun leq((x1, y1), (x2, y2)) = ...
  val initial = (i.initial, i.initial)
end;
```

As an example for a functor application we define the following structure `SquaredIntItem`.

```
structure SquaredIntItem = SquareItem(IntItem)
```

During almost a decade of its use, several shortcomings of the ML module system have been reported, for instance the lack of true separate compilation [1], the lack of certain kinds of polymorphic modules [8], and the lack of recursive structures and higher order functors [2].

2.2 Modules in Haskell 1.3

At present, the module system of Haskell is just name space management. Although during its design no attempts have been made at all to make it as powerful or sophisticated as the system of Standard ML, it has not yet been proven to be particularly useless. So the question arises how powerful it actually is, and whether it should be considered poor in comparison to ML.

Akin to SML, the 'module language' in Haskell acts on top of the 'ordinary language', i.e. modules and values cannot be mixed.

As practical experience with Haskell suggests, to describe a larger piece of software as a simple, hierarchical composition of smaller pieces, it is sufficient to provide means of naming items from imported modules. When it comes to readability and type safety, Standard ML may win over Haskell through its use of signatures which concisely describe the interfaces of modules, but this is all. In Haskell one cannot give the type signatures for a module's items, so some programmers just add them as comments. Bu of course it would be much better if the system actually helped the programmer by allowing type signatures for exported items to be given and to be checked according to their usage.

Notice that the naming of imported and exported items in the spirit of Modula-2's qualified names neatly reflects the programmer's intention.

```
module A ( f ) where
data T = ...
f :: T -> T
f = ...

module B ( f, g ) where
import A
g = ... f ...

module C where
import qualified B
f = 5
h = ... B.f ... B.g
```

In the code fragment above showing three separate modules it is only convenient to refer to imported items with names reflecting the direct origin, but not with their original names (as long as there is no confusion). So, in module `C` one refers to the imported function `f` with a qualified name reflecting where `f` was obtained from.

In the compilers view, which is shown below, all qualified names are resolved such that they consist of the name of the original module of definition and the defined identifier.

```
data A.T = ...
A.f :: A.T -> A.T
A.f = ...

B.g = ... A.f ...

C.f = 5
C.h = ... A.f ... B.g
```

2.3 How Haskell 1.3 simulates SML modules

This section explains how, to a limited extent though, Haskell modules may be used to express modular program structures in the spirit of SML. As a case study we translated an SML program, deliberately designed to make much use of structures and functors, into an equivalent Haskell 1.3 program. The interpreter and type checker described in [13] appeared to us as a good candidate since it is comprehensive in the sense that is makes use of several signatures, structures and functors, and it is understandable, since it is still an academic (and therefore smaller) example. Similar to his well known 'Four Lectures on Standard ML', Tofte explains in this more recent work how to use the SML module system to structure a larger piece of software concisely, while maintaining flexibility.

The mapping we give is rather straightforward and relies on the following translation scheme.

```
signature SIG =
sig
  type A
  val  v: A
  ...
end;

structure Struct:SIG =
struct
   (* structure defs *)
end;

functor FUN (S : SIG') : SIG =
struct
   (* functor defs *)
```

```
end;

M = FUN(Struct)
```

A simple SML structure `Struct` with (target) signature `SIG` is described as a Haskell module called `Struct`. The structure resulting from applying the functor `FUN` to `Struct` becomes a module importing `Struct`.

```
module Struct  ( A, v, ... ) where
-- structure defs

module M where
import Struct
-- structure defs
```

Obviously, a translation like this is simple only due to the simple structure of the original ML program. However, since in all the larger applications we have looked at there is only one structure or functor for any given signature, this straightforward mapping lays on hand. Translating the mentioned type checker from Tofte's tutorial program along the lines explained above was simple, since, as in any typical application program, all functors are applied only once. Here we give only a short example to show the translation described above in a concrete setting.

```
signature TYPE =
sig
  type Type
  exception Type
  val mkTypeInt: Type
  val prType: Type -> string
  ...
end;
```

Above, the signature for the type checking functor is given; below you see a concrete implementation.

```
functor Type():TYPE =
struct
  datatype Type = INT | BOOL
  fun mkTypeInt = INT
  ...
end;
```

We have removed all the declarations which deal with exception handling, since they are not particularly interesting in this context. Now see below the accompanying Haskell module.

```
module TYPE ( mkTypeInt, prType, Type ) where

data Type = INT | BOOL
     deriving Eq

mkTypeInt   :: Type
mkTypeInt   = INT
...
```

Finally, see how the complete system, the type checker and interpreter, is composed first in Standard ML and then in Haskell. In intermediate steps appropriate structures are created by applying functors (with no arguments at the bottom level) only, as seen below.

```
signature INTERPRETER=
sig
  val interpret: string -> string
  val eval: bool ref
  and tc  : bool ref
end;

structure Parser= Parser(Expression);
(* ... *)

functor Interpreter
   (structure Ty: TYPE
    structure Value : VALUE
    structure Parser: PARSER
    structure TyCh: TYPECHECKER
    structure Evaluator:EVALUATOR
      sharing Parser.E = TyCh.Exp = Evaluator.Exp
          and TyCh.Type = Ty
          and Evaluator.Val = Value
   ): INTERPRETER=
struct (* ... *) end;

structure Interpreter= Interpreter
   (structure Ty = Ty
    structure Value = Value
    structure Parser = Parser
    structure TyCh = TyCh
    structure Evaluator = Evaluator
   );
```

Notice that the sharing constraints from the functor `Interpreter` above have no equivalent in the Haskell code below, they simply disappear. This is because all the types used inside ML structures can be simulated with data type declarations in a Haskell module and – depending on whether the type should be visible to other modules or not – exposed via the exports of the module.

```
module Interpreter ( interpret ) where
import Type
import Value
import Parser
import Typechecker
import Evaluator

eval = True
tc   = True

interpret = \ str ->
  let abstSyn  = parse str
      typeStr  = if tc then
                     prType (typeCheck abstSyn)
  ...
```

One drawback in a translation of this kind is the lack of signature information in Haskell. While the SML program explicitly defines the interfaces of all structures and functors, in the Haskell equivalent there is actually no way to check whether the implementation of an exported value matches the intended type (as used somewhere else in another module), until the calling module is compiled as well. From a software engineering point of view it seems preferable to support a kind of independence and separate compilation, as Standard ML does.

Hence, the main difference between SML's and Haskell's module system is the use of explicit interface information. While programmers in ML write signatures to provide explicit information about a class of structures, Haskell people have no way to do so, apart from giving type signatures as comments.

3 Higher order type variables

Languages based on Hindley-Milner type systems allow programmers to define new data types parameterized on type variables. For instance, the Haskell declaration below defines the type of (binary) trees with arbitrary elements.

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

In this definition, `a` is a placeholder for concrete types in particular concrete definitions of trees. Hence, when one writes

```
l = Leaf 7
m = Leaf 'y'
s = Node m 'x' m
t = Node l 6 s
```

the values `l` and `m` get types `Tree Int` and `Tree Char`, respectively, and value `s` is a non-leaf member of type `Tree Char`. Value `t` is ill-typed, but the definition `t = Node l 6 t` would be fine.

An obvious extension of this standard mechanism was proposed by Jones (see [4] and [6]. Originally inspired by an extension of Haskell's type class system to allow for the use of type constructors, Jones described constructor classes and implemented them in the Gofer functional programming system. Due to their usefulness they became very well known soon.

Here, we make no use of either type or constructor classes, but only of higher order type variables. Imagine, for instance, that the type definition above should be generalized to allow not only for binary (always two subtrees), but also for ternary (always three subtrees) trees or rose trees (with an arbitrary number of subtrees represented by a list of subtrees). One way to achieve this is to define appropriate data structures for every kind of such trees, but one could also use a type constructor, as the example below shows.

```
data Tree t a = Leaf
              | Node a (t (Tree t a))

data Tup a  = Tup a a
data Trip a = Trip a a a

l = Leaf
s = Node 'x' (Tup l l)
```

Here, `s` has type `Tree Tup Char`.

The advantage of this approach is that it leaves the choice for the type constructor `t` open. Therefore, the data types `Tup` or `Trip` can be also used for other purposes, and the definition of `Tree` does not have to be changed for a new kind of trees. So one simply defines a rose tree like

```
t = Node 8 [t,l,t,t,Node 7 [t]]
```

which gets type `Tree [] Int`.

In the following we use such a type constructor based approach to solve a problem recently described by Brodal and Okasaki in [2], a work on functional data structures. In their work the authors show how ML functors are subsequently applied to simple data structures to create more complex ones, in which certain algorithms can be expressed more efficiently. They make use of both higher order functors, which are only available in some ML implementations, and recursive structures, which are not supported by any implementation.

The goal is to describe priority queues over ordered elements. Consider first the following ML signatures for a data type with ordering relation

```
signature ORDERED =
sig
    type T
    val leq : T * T -> bool
end;
```

and priority queues over such data.

```
signature PRIORITY_QUEUE =
sig
    structure Elem : ORDERED
    type T  (* type of priority queues *)
    val empty     : T
    val isEmpty   : T -> bool
    val insert    : Elem.T * T -> T
    val meld      : T * T -> T
    val findMin   : T -> Elem.T
    val deleteMin : T -> T
 end;
```

We can express this similar in Haskell by defining a record data type for structures with a comparison operation `leq`

```
data Ordered a = Ordered {
  leq :: a -> a -> Bool
}
```

and a data type for priority queues.

```
data PriorityQueue q a =
  PriorityQueue (q a) (PQSig q a)

data PQSig q a = PQSig {
  empty      :: q a,
  isEmpty    :: q a -> Bool,
  insert     ::   a -> q a -> q a,
  meld       :: q a -> q a -> q a,
  findMin    :: q a -> a,
  deleteMin  :: q a -> q a
}
```

Here, priority queues are represented as pairs consisting of the actual queue and a package of methods to access the queue. It is the same modeling technique as the one used in one standard approach to object-oriented programming in a pure functional language (see [11]). There, objects are described as pairs of an internal state and a method package to access that state. The representation and the transformations described in the following depend on the use of `q` as a higher order type variable.

Convenient access to such structures is gained through functions like

```
isEmptyQ :: PriorityQueue q a -> Bool
isEmptyQ (PriorityQueue r m) = (isEmpty m) r

insertQ :: a -> PriorityQueue q a
             -> PriorityQueue q a
insertQ a (PriorityQueue r m) =
  PriorityQueue (insert m a r) m
```

and so on.

An ML functor taking an ordered structure into a queue structure might be given by the following definition.

```
functor BinomialQueue (E:ORDERED):
  PRIORITY_QUEUE =
struct
  structure Elem = E
  type Rank = int
  datatype Tree =
    Node of Elem.T * Rank * Tree list
  type T = Tree list
  val empty = []
  fun isEmpty ts = null ts
  (* insert ... meld ... findMin ... *)
end;
```

Another implementation (skew binomial queues) is described in [2], and further different implementations are conceivable.

```
functor SkewBinomialQueue (E:ORDERED):
  PRIORITY_QUEUE =
struct
  (* ... *)
end;
```

Here is how such functors are translated into Haskell.

```
type Rank = Int
type PrioQueueFunctor q a =
  Ordered a -> PriorityQueue q a

data Tree a = Node a Rank [Tree a]
data TL a = TL [Tree a]

binomialQueue :: PrioQueueFunctor TL a
binomialQueue o = PriorityQueue (TL []) PQSig{
  empty      = TL [],
  isEmpty    = \ (TL ts)           -> null ts,
  -- insert ... meld ... findMin ..
}
```

The only problem in this is related to the internal type definitions in Standard ML structures, as for instance `datatype Tree`. Since record data types in Haskell cannot contain type declarations, we have to move them outside the signature. This is not a problem, however, because Haskell's module system allows to control the export of details of these data types. If desired, the constructors of data type `Tree` or `TL` can be hidden.

The important point is to show how recursion on the level of ML structures can be expressed as recursion on Haskell data types. As an example we use a construction described in [2]. There, a recursive structure is needed to elegantly express a process known as data structural bootstrapping which may be used to decrease the running time of certain operations on data types. Consult [2] to understand the data structures involved, and notice that the following is not valid ML code.

```
functor Bootstrap(
  functor MakeQ (E:ORDERED):
     sig
       include PRIORITY_QUEUE
       sharing Elem = E
     end)
  (E:ORDERED): PRIORITY_QUEUE =
struct
  structure Elem = E

  (* recursive structures not
     supported in SML! *)
  structure rec RootedQ =
      struct
        datatype T = Root of Elem.T * Q.T
        fun leq (Root (x1,q1), Root (x2,q2)) =
              Elem.leq (x1,x2)
      end
  and Q = MakeQ (RootedQ)

  open RootedQ  (* expose Root constructor *)

  datatype T = Empty | NonEmpty of RootedQ.T

  val empty = Empty
  fun isEmpty Empty = true
    | isEmpty (NonEmpty _) = false
  (* ... *)
```

The important bit is the recursion between structures `RootedQ` and `Q`, which is used to create a element structure (with signature `ELEM`) in which the elements consist of a pair of an original element and a priority queue over the newly created elements.

Recursive structures are not supported by any implementation of Standard ML so far. This is due to the possible presence of side effects in Standard ML structures, which make a general treatment of recursion in this context unlikely to be possible. Okasaki therefore expresses the desired structure by manually inlining the appropriate code, a process which he understandably describes as tedious and error prone. In Haskell, where there are no language-inherent side effects anyway, the recursion on the level of ML structures is translated into recursion on the level of data types, where it is easy to deal with.

```
data BootStrap q a = E
                   | NE (q (BootStrap q a)) a

bootStrap :: PrioQueueFunctor q (BootStrap q a)
  -> PrioQueueFunctor (BootStrap q) a
bootStrap makeQ = \ord ->
  let resRepO = Ordered { leq = leqB }
      leqB (NE q1 x1) (NE q2 x2) = leq(ord) x1 x2
      (PriorityQueue _ methods) = makeQ resRepO
      isEmptyB E        = True

      -- ... isEmptyB ... insertB ... findMinB ...

      meldB (NE q1 x1) (NE q2 x2)
        | leq ord x1 x2 =
            NE (insert methods (NE q2 x2) q1) x1
        | otherwise     =
            NE (insert methods (NE q1 x1) q2) x2
  in
  PriorityQueue
    E
    PQSig{empty=E, isEmpty=isEmptyB,
          insert=insertB, meld=meldB,
          findMin=findMinB, deleteMin=deleteMinB}
```

4 Local universal quantification

4.1 Overview

First class structures were described by M. Jones in [5] and [7]. In this work it is explained that first class structures allow for the description of modules on the level of ordinary values, making the treatment of data packages, or abstract data types, more natural. There is, from a conceptional point of view, no need for a second level language to describe the structuring operations acting on smaller program units.

First class structures extend the Hindley-Milner type system underlying popular functional languages like Standard ML and Haskell by supporting a simple form of higher order polymorphism.

```
type Stack a = [a]

data Stck = Stck {
  empty :: Stack a,
  push  :: a -> Stack a  -> Stack a,
  pop   :: Stack a -> Stack a,
  top   :: Stack a -> a
}

stack :: Stck
stack = Stck {
  empty = [],
  push  = (:),
  top   = \ l -> case l of
            []     -> error "stack:top"
            (x:xs) -> x,
  pop   = \ l -> case l of
            []     -> []
            (x:xs) -> xs
}

n = (top stack) ((push stack) 3 (empty stack))
```

Above is an example for the use of record types with polymorphic fields (parameterized signatures) and values of such types (first class structures) with the syntax supported by the Chalmers Haskell Compiler (*hbc-0.9999.0*). The type variable `a` occurring free in the data type definition signals polymorphic types for the fields in `Stack`. The last line shows how to access the fields of the record and how such a structure may be used.

The main advantage over ordinary record types in Hindley-Milner type systems is the polymorphism of the single fields. Since values of data type `Stack` may be passed to functions or returned by them just like any other values, a certain form of higher order polymorphism is achieved. And since the type signatures of the polymorphic fields are explicitly given, also type inference is still possible. For a detailed description of the theoretic properties see the original work [7].

4.2 An example

In the following we explain the translation from Standard ML structures to Haskell, augmented with first class structures. First have a look at the ML code.

```
signature GROUP =
sig
  type G
  val e : G
  val bullet: G * G -> G
  val inv: G -> G
```

```
end;

structure Z : GROUP =
struct
  type G = int
  val e = 0
  fun bullet(n:int,m) = n+m
  fun inv(n:int) = ~n
end;

functor Sq(G: GROUP) : GROUP =
struct
  type G = G.G * G.G
  val e =(G.e, G.e)
  fun bullet((a1,b1),(a2,b2)) =
           (G.bullet(a1,a2),
            G.bullet(b1,b2))
  fun inv(a,b) = (G.inv a, G.inv b)
end;

functor Try(Gr:
       sig
          type G
          sharing type G = int
          val e: G
          val bullet: G*G->G
          val inv: G -> G
       end) =
struct
   val x = Gr.inv(Gr.bullet(9, 7))
end;

structure S = Try(Z);
structure SqZ = Sq(Z);
structure SqSqZ = Sq(SqZ);
```

Signature `GROUP` declares the ubiquitous group interface. As an example of a concrete group, structure `Z` describes the group of integers $(Z, +)$. The functor `Sq` defines the 'square' of a given group, i.e. the binary product of one and the same group, which can be expressed in terms of the underlying group's operations. Finally, functor `Try` shows what is necessary to constrain a functor's argument structure to a particular group, using a type sharing constraint.

Here is the equivalent Haskell code.

```
module Grp where
infix 9 .
r.f = f r

data GROUP g = GROUP {
     e      :: g,
     bullet :: (g,g) -> g,
     inv    :: g -> g
}

grpZ = GROUP {
     e = 0,
     bullet = \ (n, m) -> n+m,
     inv    = \ n -> (-n)
}

sqFunctor :: GROUP g -> GROUP (g,g)
sqFunctor grp = GROUP {
  e      = (grp.e, grp.e),
  bullet = \ ((a1,b1), (a2,b2)) ->
    ((grp.bullet) (a1,a2),
     (grp.bullet) (b1,b2)),
  inv    = \ (a,b) ->
    ((grp.inv) a, (grp.inv) b)
}

data Try = Try {
  valX :: Int
}

tryFunctor :: GROUP Int -> Try
tryFunctor grp = Try {
  x = (grp.inv) ((grp.bullet) (7,9))
}

s     = tryFunctor grpZ
sqZ   = sqFunctor grpZ
sqsqZ = sqFunctor (sqFunctor grpZ)
```

Note that this example does not really require first class structures at all, but is expressible with simple record data types as they are part of the Haskell 1.3 language definition. This is because none of the fields is expected to have a polymorphic type. However, there are some interesting observations to make.

- In the presence of first class structures, Standard ML structures are expressible as values of record types, and functors are translated into functions. It is therefore simple to apply the same functor to different arguments, in contrast to what we observed in Section 2.3.

- While Standard ML allows programmers to use anonymous structures, as seen in the example of functor `Try` above, a new data type has to be defined in Haskell. In our opinion this should not be considered a drawback since it improves the readability of the program and the overall understanding, even though being somewhat tedious. Apart from that the example is certainly fairly unlikely to be of much practical use.

- In Standard ML types can be specified in signatures and defined in structures, as seen above. Since there are no local type definitions in record type definitions, even not in the presence of first class structures, we have to give them on a global level. If a polymorphic type is required, as in the example above, type parameters are given in the definition of the record data type. This lifting of type definitions as part of the conversion process is described in [7]. We return to it in the next section.

- Translating larger examples in the same manner, for instance the interpreter mentioned in Section 2.3, is a straightforward exercise, as far as the modular structure of the program is concerned. Somewhat less obvious is how to simulate local state, i.e. values of reference types in structures. The program described in [13] makes use of state, for instance to switch the type checking or evaluation of terms on or off, respectively. Also, the name supply for polymorphic type checking can easily be provided using local state. The standard and famous Haskell way of doing this is to use a monad, but this makes programs more complicated and changes the involved types substantially. Since the main goal of this comparison is not to deliberately translate the type checker, but to see whether Haskell's module system is in principle equally powerful as Standard ML's one, we do not further elaborate on this point.

4.3 Advantages of first class structures

In this section we give an example for the use of first class structures which shows how their expressiveness can be used to surpass the module system of Standard ML. We describe how higher order functors can easily be expressed using first class structures.

For this we return to the example of priority queues from Section 3. See below the header that is suggested for a higher order functor transforming any priority queue functor into one that adds a global root to the priority queue. The point in this transformation is that the running time of `findMin` can be reduced to $O(1)$, see [2] for more details.

```
(* Higher order functors are not part of
   Standard ML and only supported by some
   implementations!

functor AddRootToFun(
  functor MakeQ(E: ORDERED):
    sig
      include PRIORITY_QUEUE
      sharing Elem = E
    end             )
  (E: ORDERED):
    PRIORITY_QUEUE =
AddRoot(MakeQ (E));                     *)

functor AddRoot (Q:PRIORITY_QUEUE):
  PRIORITY_QUEUE =
    struct
      structure Elem = Q.Elem
      datatype T = Empty
                 | Root of Elem.T * Q.T
      val empty = Empty
      (* isEmpty ... meld ... etc. *)
    end;
```

`AddRoot` is a functor that takes a `PRIORITY_QUEUE` structure and adds a global root. The functor `AddRootToFun` is actually curried, so that the second argument structure `E` serves as argument for the resulting functor.

The equivalent Haskell code shows how to directly express a higher order functor.

```
data R q a = Empty | Root a q

addRootFun :: PrioQueueFunctor q a
  -> PrioQueueFunctor (R (q a)) a

addRootFun makeQ  = \ ord ->
  let (PriorityQueue rep meths) = makeQ ord
      isEmpt r = case r of
                   Empty -> True
                   Root _ _ -> False
      insrt y Empty = Root y (empty meths)
      insrt y (Root x q) =
        if leq(ord) y x
           then Root y (insert(meths) x q)
           else Root x (insert(meths) y q)
      -- ... mld ... findMn ...
  in
     PriorityQueue Empty PQSig{
       empty=Empty,
        isEmpty=isEmpt,
        insert=insrt,
        -- ...
    }
```

Since SML functors boil down to Haskell functions involving first class structures, there is no difficulty in expressing higher order functors in the Haskell equivalent.

5 Conclusion

In this paper we have first discussed how the module systems of Standard ML and Haskell relate. We have seen that the simple name space control mechanism provided by Haskell can in many applications compete with ML's advanced module system. Many applications are composed by subsequently applying functors and constructing larger and larger structures, but the single functors are applied only once in this process. In such cases a direct mapping to the Haskell module system is possible.

Then we have seen how higher order type variables substantially increase the expressiveness of the language. We demonstrated how recursive ML structures, which are useful for the process of data structural bootstrapping, can be modeled in Haskell 1.3.

We have further investigated how the recently proposed first class structures (see [7]) can be used to make an even more powerful translation from Standard ML to Haskell* possible. Standard ML structures and functors, as well as functor applications, can be easily and concisely modeled. Also higher order functors, which are not part of Standard ML so far, can be expressed.

First class structures are a fairly new and unexplored terrain, therefore experience how to use them is lacking. Several options seem possible, but before firm design decisions for a language extension can be made, more examples have to be investigated. The structures incorporating local universal quantification, as described in [7], may naturally be extended to allow for existential quantification. Using this, hiding issues can be rethought, and objects in the sense of [11] can be encoded. First class structures may also neatly cooperate with Haskell type classes, if one allows for local context annotations.

6 Acknowledgments

Many thanks to the attendants of the workshop who gave useful comments and helped to improve the quality of this paper, namely Enno Scholz and Laszlo Nemeth.

References

[1] Andrew W. Appel and David B. MacQueen. Separate Compilation for Standard ML. *SIGPLAN Notices*, 29(6):13–23, June 94.

[2] Gerth Stølting Brodal and Chris Okasaki. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6(6), December 1996.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer, Berlin, 1985, 1985. volume 6 of *EATCS Monographs on Theoretical Computer Science*.

[4] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Functional Programming & Computer Architecture*, Copenhagen, Denmark, June 1993.

[5] Mark P. Jones. From Hindley-Milner Types to First-Class Structures. In *Proceedings of the Haskell Workshop*, June 1995.

[6] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *First International Spring School on Advanced Functional Programming Techniques, Springer Lecture Notes in Computer Science 925*, Båstad, Sweden, May 1995.

[7] Mark P. Jones. Using Parameterized Signatures to Express Modular Structure. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, January 1996.

[8] S. Kahrs. First-class polymorphism for ML. In *Programming languages and system - ESOP '94. Springer Verlag. Lecture Notes in Computer Science,788*, 1994.

[9] Konstantin Läufer. Combining Type Classes and Existential Types. In *Proceedings of the Latin American Informatics Conference*, 1994.

[10] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1992.

[11] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993.

[12] M. Tofte. Four Lectures on Standard ML. Technical Report ECS LFCS 89 73, Laboratory for Foundations of Computer Science. Department of Computer Science, Edinburgh University, March 1989.

[13] M. Tofte. Essentials of Standard ML Modules. Technical Report ECS LFCS 89 73, Department of Computer Science, University of Copenhagen, May 1996.