Adding Records to Haskell

John Peterson and Alastair Reid
Department of Computer Science, Yale University,
P.O. Box 208285, New Haven, CT 06520, USA.
Electronic mail: {peterson-john,reid-alastair}@cs.yale.edu

October 4, 1998

Abstract

The Haskell programming language has a very simple yet elegant view of data structures. Unfortunately, this minimalist approach to data structures, in particular record-like structures, presents serious software engineering problems. We have implemented an extension to standard Haskell which provides record-like structures in addition to ordinary algebraic data types. Our extension provides named fields in data structures, default field values, field update functions, detection of uninitialized slots and multiple inheritance. Our major design goal was to supply as much functionality as possible without changing any of the basic components of the Haskell language (in particular, we avoided further complication of the type system). The purpose of this paper is not to advocate this specific extension to Haskell, but to examine the basic engineering issues associated with records; describe our experiences with the implementation and use of one particular proposal; and consider alternative approaches (some of which have been used in other languages).

1 Introduction

The Haskell language [2] includes only the most basic support for a fundamental programming language feature: the record type. In the most general sense of the term, a record simply groups a heterogeneous collection of objects into a single value. There are many different manifestations of record-like features in programming languages, including tuples, structures, and objects. While the algebraic data types found in Haskell have the necessary functionality to build record data structures, Haskell lacks many desirable features found in other languages for dealing with complex data objects.

This paper describes our implementation of record types in the Yale Haskell system. The purpose of this experiment is not to advocate any specific implementation of records in Haskell, but to fully explore one possible approach to this problem and to gain practical experience with the problem of integrating records with the Haskell programming style. After presenting our implementation, we compare our system of records to those found in other languages and discuss alternatives to our design.

Before proceeding, we will clarify our terminology. We use the term *record* in only the most general sense. The components of records are *fields*. Within the context of our specific proposal, we use the terms *structure* and *slot* to denote our particular implementation of records and fields (respectively).

The issues of concern here are not so much in the fundamental language semantics, but instead are matters of engineering. From a software engineering standpoint, the record structures provided by a programming language benefit from the following properties:

- Expandability. Adding a new field to a record should not require modification of code which references old fields. It should be possible to new fields silently without changing existing code.
- Reusability. A record should be able to include (inherit) other records; operations which apply to the included records should also apply the including record.
- Efficiency. Basic record operations must be extremely efficient; there must be no hidden performance costs.
- Privacy. The program must be able to hide the internal details of a record.

Along with these engineering issues, we have one further goal: to keep our system as much in the spirit of standard Haskell as possible.

The basic features of our proposal are:

- The semantics are entirely defined via a translation to standard Haskell. No modifications are required in the Haskell type system.
- Slots may be accessed via pattern matching or by function application.
- Slots may be (functionally) updated.
- Default values may be provided for slots.
- Uninitialized slots can be detected by the programmer.
- Special syntax is used for creating, updating, coercing, etc. This avoids generating new names for these operations (as is done in Common Lisp [6]).
- Explicit declarations are required for all record types. This avoids the efficiency and type-inference problems associated with more general record types and produces more accurate messages when type errors occur.
- Structures may be polymorphic.
- Multiple inheritance is allowed. Inheritance is implemented using Haskell's type class mechanism; structure operations and user-defined functions are overloaded to allow them to apply to any structures defining appropriate fields. Coercion functions are provided to move up and down the inheritance graph.

2 Data structuring in standard Haskell

Before presenting our proposal, we explore what can be done in standard Haskell. This both illustrates the need for improvement and provides the basis for describing the semantics of our proposal.

For example the following datatype which is used to represent named entities within the Yale Haskell compiler.[1]

```
data Definition =
     MkDef
       String         -- name
       String         -- module in which it is defined
       String         -- unit in which it is defined
       Bool           -- is it exported?
       Bool           -- is it a PreludeCore symbol?
       Bool           -- is it a Prelude symbol?
       Bool           -- is it created by an interface?
       Bool           -- is it ``made up'' by the compiler?
       (Maybe SrcLoc) -- where it was defined.
```

This datatype is hard to use reliably. There are several fields of the same type — the type system is not able to detect simple errors such as accidentally swapping the fourth and fifth fields. Such problems are very difficult to spot when fields are identified only by their position with respect to a constructor. It is also hard to maintain: adding an extra field to this definition requires changes to every use of the constructor `MkDef` (i.e. taking `Definition`s apart in patterns and constructing `Definition`s in expressions.)

The usual solution to the problem of reliably handling many fields is to define "access functions" for updating and selecting each field of the record. For this example, we must define 18 different access functions — one to extract each slot and one to update each slot:

```
getName, getModule, getUnit :: Definition -> String
getName   (MkDef nm _ _ _ _ _ _ _ _)   = nm
getModule (MkDef _ mod _ _ _ _ _ _)  = mod
getUnit   (MkDef _ _ unit _ _ _ _ _) = unit
...

setName, setModule, setUnit :: String -> Definition -> Definition
setName nm (MkDef _ mod unit isEx isCore isPrel isIface isInternal loc)
  = (MkDef nm mod unit isEx isCore isPrel isIface isInternal loc)
setModule mod (MkDef nm _ unit isEx isCore isPrel isIface isInternal loc)
  = (MkDef nm mod unit isEx isCore isPrel isIface isInternal loc)
setUnit unit (MkDef nm mod _ isEx isCore isPrel isIface isInternal loc)
  = (MkDef nm mod unit isEx isCore isPrel isIface isInternal loc)
...
```

[1] The Yale Haskell compiler is written in Lisp; this example is obtained by translating from Lisp to Haskell. Similar examples occur in the Glasgow Haskell compiler — which *is* written in Haskell.

Using these access functions instead of referencing the constructor `MkDef` directly results in more readable code and simplifies the task of adding new fields to a record. However, the reader will appreciate that creation of these access functions is a somewhat tedious and error-prone task.

A further problem with this approach is that it is no longer possible to use pattern matching to extract components of records. This makes programs more verbose.

3 Syntactic Support for Records

The core of our proposal is to provide special syntax for defining structure types, accessing slots, and initializing structures. The semantics of our proposal is defined as a translation into code like that given in the previous section.

The additions to Haskell syntax rules (appendix B of [2]) are as follows:

3.1 Structure declarations

topdecl	→	`structure` [`~`] *simple* `where` { *structbody* [`;`] } [`deriving` `(` *tyclses* `)`]
simple	→	*tycon tyvar*$_1$ ... *tyvar*$_k$
structbody	→	*structsigns* [`;` *valdefs*]
structsigns	→	*structsign*$_1$ `;` ... `;` *structsign*$_n$
structsign	→	*vars* `::` [*context* `=>`] *type*

Using this syntax, the datatype and access functions in section 2 can be more concisely defined by

```
structure ~Definition where
  name, moduleName, unit        :: String
  isExported, isCore, isPrelude :: Bool
  fromInterface, isInternalDef  :: Bool
  definedIn                     :: Maybe SourceLoc
```

(The "twiddle" is related to inheritance and is described in section 4.1.)

The selector functions have exactly the same name as the slot they extract; for example, the following function prints the original name of a definition:

```
showDefName :: Definition -> ShowS
showDefName d
   = showString (moduleName d) . showChar '.' . showString (name d)
```

Translation: The declaration

```
structure ~S t1 ... tk where
  v1 :: u1; ... ; vm :: um
  vi1 = init1; ... ; vin = initn
```

is equivalent to the type declaration and function declarations

```
data S t1 ... tk = MkS u1 ... um
v1 :: S t1 ... tk -> u1
v1 (MkS x1 ... xm) = x1
⋮
vm :: S t1 ... tk -> um
vm (MkS x1 ... xm) = xm
```

(The meaning of the default values (v_{i1} `=` $init_1$`;` ... `;` v_{in} `=` $init_{in}$) is defined below; the meaning of omitting the "twiddle" (`~`) is defined in section 4.1.)

Note: Although we define the semantics of our system by translation into standard Haskell, the constructor MkS used in this translation is not made directly available to the programmer.

3.2 Pattern Matching

$apat \rightarrow$ `(` $spat_1$ `,` ... `,` $spat_n$ `)` (structure pattern, $n \geq 1$)
$spat \rightarrow var$ `=` pat

An alternative way of extracting slots is by pattern matching. Structure patterns consist of a list of pairs of slot-names and patterns. For example, the function `showDefName` could be written as:

```
showDefName :: Definition -> ShowS
showDefName (moduleName = m, name = nm)
  = showString m . showChar '.' . showString nm
```

The order in which slot names are listed does not matter. Pattern matching proceeds left to right as in all other patterns.

Translation: The expression `case` e_0 `of (`$s1$ `=` p_1`,` ... sn `=` p_n`) ->` e`; _ ->` e' , for slot names $s1$, ... sn, is equivalent to:

```
let { y = e' } in
case e0 of { MkS x1 ... xk ->
case xs1 of { p1 -> ... case xsn of { pn -> e ; _ -> y } ...
              _ -> y
}}
```

where y, $x_1 \ldots x_k$ are new variables and x_s is the value of the slot named s.

3.3 Updates

aexp	→	`(` *var* `=)`	(*update section*)
	|	`(` upd_1 `,` ... `,` upd_n `)`	(*update function*, $n \geq 1$)
upd	→	*var* `=` *exp*	

For each slot $s :: t$ of a structure S, the update section `(`s`=)` is a function of type $t \to S \to S$ which copies the structure updating the value of the slot s. The value to be placed in the slot can be placed inside the parenthesis, as in `(name = "foo")`. More than one slot can be updated at once, as in `(name = ".", moduleName = "Prelude")`.

Translation: Given the structure declaration
`structure ~`S $t_1 \ldots t_k$ `where`
v_1 `::` u_1`;` ... `;` v_m `::` u_m
v_{i1} `=` $init_1$`;` ... `;` v_{in} `=` $init_n$
the notation (`(`v_i `=)`) is equivalent to:

`(\ x (`MkS x_1 ... x_i ... x_m`) -> (`MkS x_1 ... x ... x_m`))`

and the notation (`(`v_{i1} `=` e_1`,` ... v_{in} `=` e_n`)`) is equivalent to:

`(\` s `->` `(`v_{i1}`=)` e_1 `(` ... `(`v_{in}`=)` e_n s ... `))`

The order in which slot names are listed does not matter; but it is a static error to use the same slot name more than once.

3.4 Structure Creation

There is no special syntax for structure creation. The structure name is used as a modified data constructor: instead of being applied to the component values, this constructor applies an update function to an initial value constructed from the defaults specified in the structure declaration.

For example, the function `mkCoreDef` creates a `PreludeCore` definition. The list of slot names and values is an update function as defined in the previous section and `Definition` is a function which applies the update function to an "empty" structure in which each slot is undefined (there are no default slot values declared in this example).

```
mkCoreDef :: String -> SourceLoc -> Definition
mkCoreDef nm src = Definition (
                     name = nm,
                     moduleName = "PreludeCore",
                     isExported = True,
                     isCore = True,
                     isPrelude = True,
                     fromInterface = False,
                     definedIn = src
                     )
```

The syntax for declaring structures allows default values to be specified for some of the slots. A straightforward approach would require the default value of each slot to have the same type as the slot. For example, one might add the following default values to the structure declaration.

```
...
isExported = False
isCore = False
isPrelude = False
fromInterface = False
definedIn = Nothing
```

However, by making the default a function mapping the structure being defined onto a slot value it becomes possible for default values to depend on the values of other slots — particularly those of explicitly-initialized slots. For example, the values of the slots `isCore` and `isPrelude` can be made to depend on the value of the `moduleName` slot.

```
...
isExported self = False
isCore (moduleName = mod) = mod == "PreludeCore"
isPrelude (moduleName = mod) = take 7 mod == "Prelude"
fromInterface self = False
definedIn self = Nothing
```

The implementation of this style of default argument is somewhat subtle: we use a recursion to allow explicitly initialized slots to override default values and to allow default values to depend on other slots in the same structure.

Translation:
For a structure type constructor S, the occurrence of the type constructor S in an expression is equivalent to the function

$\backslash init$ `->` `let` s `=` $init$ `((`v_1 `=` $init_1$ s`, ...,` v_n `=` $init_n$ s`)` `(`MkS $\bot \ldots \bot$`)` `in` s`)`

where the default values for variables $v_1, \ldots, v_n$ are $init_1, \ldots, init_n$ (respectively).

It is a static error to provide more than one default value for a slot. Uninitialized slots with no default are bound to error calls.

Strictness annotations in data type definitions cause problems with initialization: an uninitialized structure slot would immediately cause a program error. Our solution is that strict slots must have a default value and that default value should have the same type as the slot (rather than being a function whose argument is the structure being created). This constant is used instead of $\bot$ in the above translation.

3.5 Uninitialized Slots

Slots which have no default value may remain uninitialized by structure creation. While accessing such slots results in a runtime error, it is sometimes useful to test whether a slot is initialized without actually referencing its value. It is, of course, possible to avoid this by adopting the convention that every slot must have a default value. On the other hand, by allowing uninitialized slots to be detectable, a robust derived `Text` instance for structures can simply skip over uninitialized slots instead of crashing when attempting to access such a slot.

The changes to the translation are straightforward but tedious: the datatype

`data` $S\ t_1 \ldots t_k$ `=` $MkS\ u_1\ \ldots\ u_m$

is changed to

`data` $S\ t_1 \ldots t_k$ `=` MkS `(Maybe` u_1`)` `...` `(Maybe` u_m`)`

The definition of selector functions and update sections are modified to accommodate this change

v_i `(`$MkS\ x_1$ `...` `(Just` x_i`)` `...` x_m`)` `=` x_i
`(`v_i `=)` `=` `\` `x` `(`$MkS\ x_1$ `...` x_i `...` x_m`)` `->` `(`$S\ x_1$ `...` `(Just` x`)` `...` x_m`)`

and (in the absence of an explicit default) the default value of every slot is changed from $\perp$ to `Nothing`.

Translation: Given the structure declaration
`structure ~`$S\ t_1 \ldots t_k$ `where`
v_1 `::` u_1`;` `...` `;` v_m `::` u_m
v_{i1} `=` $init_1$`;` `...` `;` v_{in} `=` $init_{in}$
the notation `(=` v_i`)` is equivalent to:
`(\` `(`$MkS\ x_1$ `...` x_i `...` x_m`)` `->`
`case` x_i `of` `{` `Just _ -> True; Nothing -> False` `})`

This translation is rather inefficient — imposing an overhead on creation, selection and updates. Fortunately, it is easy to detect undefined slots without an explicit `Maybe` datatype in the representation. To produce meaningful error messages, each potentially undefined slot is already associated with a particular error thunk. Instead of wrapping the slot value up in the `Maybe` data type, the definedness check simply compares the slot value with the associated error thunk using pointer equality.

4 Adding Inheritance

It is possible to extend this translation further to allow a structure to inherit slots from other structures. For example, one might define variables which are just like definitions but

provide additional slots to store the type, signature, fixity, definition, etc of the variable. We extend the syntax slightly to specify which structures slots are being inherited from.

topdecl → `structure` $tycon_1, \ldots, tycon_n$ `=>` [`~`] *tycon* `where`
{ *structbody* [`;`] } [`deriving` `(` *tyclses* `)`] $(n \geq 1)$

For example, to define a type `Variable` which inherits slots from the type `Definition`, we write:

```
structure Definition => Variable where
  varType :: Signature
  varSignature :: Maybe Signature
  fixity :: Fixity
  definition :: Expression
```

The major change required to make this work is that the functions to select slots and update structures must be overloaded [9]. That is, instead of translating a structure definition into just a datatype and a collection of slot selection and update functions, structure definitions are translated into a type class with selection and update functions as methods, a new datatype and an instance of the datatype for that class. We use the same name for the type its corresponding class — this would normally be a syntax error since Haskell does not allow types and classes to share names.

For example, the definition of the structure `Definition` must be changed to define a type class (also called `Definition`) with methods

```
name, moduleName, unit :: Definition a => a -> String
...
(name=), (moduleName=), (unit=) :: Definition a => String -> a -> a
...
```

The old definition of the access functions is used to define an instance of the class `Definition` at the type `Definition`.

Similarly, the definition of the type `Variable` is used to define a type class `Variable`, and a data type `Variable` which is an instance of both `Definition` and `Variable`.

A structure may be either narrowed to a contained structure or widened to a containing structure. Widening is accomplished by adding undefined slots to the value. For a structure type `S`, the function `(-> S)` narrows a value from any type which includes `S` onto `S` and the function `(S ->)` widens a value of type `S` into any type containing `S`. The types of these operators are:

```
(-> S) :: S a => a -> S
(S ->) :: S a => S -> a
```

Translation:

```
instance S S' where
...
    (-> S) (MkS' x_1 ... x_n) = MkS n_1 ... n_i
    (S ->) (MkS x_1 ... x_m) = MkS' w_1 ... w_j
```

where n_i is the x in the corresponding slot and w_i is the corresponding x when the slot is part of S or $\perp$ otherwise.

For simplicity, widening does not invoke the defaulting mechanism to fill the new slots added by widening.

The most difficult change is in pattern matching. Since we do not know the exact type of the structure, the translation given in section 3.2 is no longer valid. The translation we implemented is:

Translation: The expression `case` e_0 `of (`$s1$ `=` p_1`, ...` sn `=` p_n`) -> `e`; _ ->` e' , for slot names $s1$, ... sn, is equivalent to:

```
let { x = e_0; y = e' } in
case s1 y of { p_1 -> ... case sn x of { p_n -> e ; _ -> y } ...
             _ -> y }
```

where x, y, $x_1 \ldots x_k$ are new variables and x_s is the value of the slot named s.

This translation has the drawback that it may occasionally cause a space leak if any p_i is irrefutable. The problem is exactly that reported by Wadler [7]: slot extraction is only performed when the value of the slot is actually required; not when the pattern matching occurs. This can cause the entire structure to be retained when only one slot is required.

The following alternative translation would eliminate this space leak, but *may* make overloaded pattern matching more expensive. (This translation is for single inheritance. Extending it to handle multiple inheritance is straightforward but tedious.)

Alternative translation: If e_0 has type $S'\ \alpha \Rightarrow \alpha$, and S' has slots $s1$, ... sn, the expression `case` e_0 `of (`$s1$ `=` p_1`, ...` sn `=` p_n`) -> `e`; _ ->` e' , is equivalent to:

```
let { x = e_0; y = e' } in
case (->S') x of { MkS' x_1 ... x_k ->
case x_s1 of { p_1 -> ... case x_sn of { p_n -> e ; _ -> y } ...
             _ -> y
}}
```

where x, y, $x_1 \ldots x_k$ are new variables and x_s is the value of the slot named s.

4.1 Avoiding Inheritance

Inheritance is a powerful tool but its use presents two problems:

1. Since inheritance is implemented with the class system, using inheritance involves the same overhead that overloading functions entails. This overhead consists of both the instances needed to define an operation over a set of data types and the extra level of indirection needed to call overloaded functions. While the execution time overhead can be eliminated using type signatures to eliminate overloading, this is very burdensome for the programmer.

2. Inheritance also may prevent early detection of some errors. For example, given two structures

   ```
   structure S1 where a1, b1  :: Int
   structure S2 where a2, b2  :: Int

   f (a1 = x, b2 = y) = x + y
   ```

   The definition of `f` is almost certainly incorrect since its argument must contain slots from two different structure types. However, this does not cause a type error since a third structure may later be declared (perhaps in a separately-compiled module) which includes both `S1` and `S2`.

   On the other hand, a type error does occur if we try to apply `f` to an argument of type `S1` (which is probably what the programmer intended to do.)

   If `S1` had not been overloaded, this error would have been caught when `f` was declared. (Providing the type signature `f :: S1 -> Int` would also have caught this error.)

We thus make inheritance optional: a structure declaration may indicate that the declared structure will not be inherited by any other structure. This is accomplished using a `~` in front of the structure name in the declaration:

```
structure S1 => ~S2 where s :: Int
```

The `~` prevents `S2` from being used as a class and allows any use of the slot `s` to precisely determine the typing of an update or pattern.

4.2 Multiple Inheritance and Defaulting

The Haskell type class system allows a class to have multiple superclasses. Since structures are translated into type classes, our translation naturally allows *multiple inheritance:* a structure is allowed to inherit slots from any set of other structures.

In the type class system, defaults can only apply to methods directly associated with a class, not those inherited from superclasses. This avoids ambiguity over which default to apply

when the same method is inherited via several routes (e.g. the standard class `Integral` inherits `Ord` via both the `Ix` class and the `Real` class).

We have chosen to relax this rule for structures. Structure declarations may define default methods for inherited slots. The following rule is used to avoid ambiguity:

> If a structure inherits a slot `s`, it may either define a new default for `s` or use the default associated with the first structure in the list of included structures containing `s`.

4.3 The Polymorphic Inheritance Problem

The reader may have noticed that the syntax for structure declarations does not allow both polymorphism and inheritance. This is to avoid the following limitation of Haskell's type system.

The declarations generated by:

```
structure S1 a where
 s1 :: a

structure S2 b where
 s2 :: b

structure S1 a, S2 b => S3 a b
```

would be:

```
data S1 a = MkS1 a
data S2 b = MkS2 b
data S3 a b = MkS3 a b

class S1 s where
  s1 :: s a -> a

class S2 s where
  s2 :: s b -> b

-- instances for S1, S2 omitted
instance S1 (S3 b) where
  s1 (MkS3 x _) = x

instance S2 (S3 a) where
  s2 (MkS3 _ x) = x
```

This "class declaration" is not legal Haskell since the type variable `s` must be instantiated with a type constructor rather than a type. This may appear to be legal using constructor classes, but the instance declaration for `S1` will still not work.

For now we simply prohibit the inheritance of polymorphic structures but allow polymorphic structures and unrestricted inheritance of non-polymorphic structures. It remains to be seen whether this is excessively restricting in real programs.

5 Alternatives and Related Work

Our system is an experiment, not a finished product. Having the experience of carrying an implementation all the way through and using it on a number of real applications, including the Yale debugger and a prototype GUI system, we can assess our design and consider alternatives.

5.1 The Namespace Issue

Our placement of slot names into the value namespace is a significant difference from languages such as C, Pascal, or ML. Using a separate namespace for each structure in the manner of C is not possible, however, because this depends on a bottom-up style of type inference which determines which type of structure is involved before resolving field names.

In practice, we have found that placing selector functions in the value namespace makes it almost essential to use long field names. For example, the structure `Point` defined by

```
structure ~Point where x, y :: Int
```

introduces two top-level function names `x` and `y` which the programmer is likely to want to use for other purposes. We adopted the convention of using the structure name as a prefix for the field name. For example, we would normally choose slot names `pointX` and `pointY` instead of `x` and `y`.

This problem could be reduced by providing special syntax for selector functions — avoiding the need to place selector functions in the value namespace— but this would not completely avoid the problem: all slot names would still be in the same namespace.

A more radical solution is used in ML which allows "labels" to be shared among different records. These labels do not carry typings in the same way the slot names do. Instead, they simply attach names to tuple components. Implementing records using shared labels would require significant changes to the syntax and further complicate the type system.

5.2 Default values

In our experience, some sort of defaulting mechanism is essential. This allows new fields to be inserted into a structure without changing all references to the associated constructor. Although not often used, the expressiveness of mutually recursive slot initialization can be very useful and seems to be more in the Haskell spirit than restricting defaults values to constants or imposing some sort of evaluation order on the default computation.

5.3 Uninitialized Slots

Though easy to implement, the ability to detect uninitialized structure slots is somewhat dubious. To date, our only use of this feature has been to allow the derived `Text` instances for structures to skip over uninitialized slots.

The need to detect uninitialized slots could be eliminated by making it impossible to leave a slot uninitialized. This could be done by changing the syntax of structure creation to require a list of slot names and values (rather than allowing any expression of the right type). It would then be possible for the compiler to check that every slot had either a default value or an explicitly provided value. A similar restriction is imposed in ML [3], where it is required by the combination of strict semantics and type safety.

5.4 Pattern Matching

Pattern matching in Haskell lacks the extensibility of other language features. It would certainly be better to add a general purpose mechanism flexible enough to define structure pattern matching than to add structure pattern matching as a special case, as in our implementation. Sadly, Wadler's "views" [8, 1] would not be flexible enough to handle this case.

In practice, we found that we didn't use pattern matching very much, preferring to use selector functions to extract slots at the place where they are needed rather than at the head of a function. This may be caused by a number of factors: our familiarity with this style of programming from other languages that support records; our use of long field names (section 5.1); the fact that structure pattern matching is generally not connected with control flow; or our use of structures in big, complicated programs that solve real problems instead of in highly polished classroom examples.

5.5 Allowing Polymorphic Inheritance

There appears to be a simple extension to constructor classes which would allow polymorphic inheritance. The problem with constructor classes is that only those types which are curried applications of a type constructor are available. Thus, for a type `T a b`, constructor classes can make use of `T`, `T a`, and `T a b` as types. Expanding the implicit currying, these types are `\a b -> T a b`, `\b -> T a b`, and `T a b`. Unfortunately, polymorphic inheritance requires a type such as `\a -> T a b`. We conjecture that adding a limited lambda to the type language is possible: this lambda is needed only to permute the arguments to the type constructors.

5.6 Syntax Issues

Using similar syntax for update functions (which are functions) and structure patterns (which match data values) is somewhat irregular. In hindsight, it would be possible to drop the parenthesis in single update functions and to drop multiple update functions. Where one currently writes update functions such as `(moduleName = m, name = nm)`, one would instead write `(moduleName = m . name = nm)`.

Our use of special syntax such as `(s=)`, `(=s)`, `(-> S)` and `(S ->)` is somewhat contorted. An alternative would be to indulge in name mangling (deriving one name from another) as in Common Lisp. (For example, the function `setFoo` would be used to alter the values of slot `foo`.) However, no other Haskell feature uses name mangling so we hesitate to add this.

5.7 Record Types

An entirely different system can be constructed using labeled records and subtype inference [5, 4]. The advantage of such as system would be that structure declarations would be unnecessary. While type systems have been proposed featuring subtyping based on extensible records, these have two disadvantages: these require a fundamental change to the Haskell type system and it may be difficult to generate efficient record operations using these systems.

5.8 Generalizing to Arbitrary Datatypes

The structures considered in this proposal are just syntactic sugar for tuples; but, Haskell's datatypes allow one to define a "sum of tuples". It would be straightforward to adapt the inheritance-free translation in section 3 to allow one to define field names for arbitrary datatypes. For example, given the datatype:

```
data Expr = Lambda (arg :: Var) (body :: Expr)
          | App (fun :: Expr) (arg :: Expr)
          | Var (v :: Var)
```

one could use pattern matching such as:

```
eval env (Lambda (arg = v, body = e)) = \x. eval ((v,x):env) e
eval env (App (fun = f, arg = a))     = (eval env f) (eval env a)
eval env (Var (v = x))                = lookup env x
```

5.9 Object-Oriented Programming

The ability to inherit structure slots is a step toward a more object-oriented programming paradigm. However, when we used our structure system in a GUI system in an object-oriented style, a number of deficiencies became obvious.

First, the classes defined for structures contain only slot accessing functions. To add other class methods (as with C++ virtual functions), we were forced to add an extra class for each structure type. That is, for a structure `S` (which defines a class `S`), we added the class `S => S'` to hold methods associated with structures inheriting from `S`. This was very unsatisfactory — it would be much nicer to be extend structure definitions to directly include these methods.

Dynamic binding, which would allow methods (dictionaries) to be attached directly to data values, is not available in Haskell without some sort of existential typing. This makes non-homogeneous lists impossible in standard Haskell.

The coercion functions were very useful — these allow objects to be moved up or down the class hierarchy so as to dispatch methods associated with other types.

A more general object-oriented extension to Haskell would eliminate the need for slot inheritance at the structure level. Provided any extra overhead could be eliminated by the compiler, such an extension may be preferable to using the inheritance mechanism described here.

5.10 Code Generation

We have found that three factors significantly affect the quality of the generated code:

1. Inlining selection and update functions eliminates a function call and allows further optimizations to be performed. Inlining the initialization function avoids constructing and destructuring many partial records.

2. Using pattern matching on function arguments produces code that is both more efficient and less likely to leak space than if selector functions are used. The reason is simple: pattern matching is performed when the function is called whereas selection functions are only executed when the selected value is evaluated. Exactly the same difference occurs if programmers use pattern matching on lists instead of `head` and `tail`.

3. Avoiding overloading (whether by shunning inheritance or by providing explicit type signatures) eliminates dictionary lookups and allows selection and update functions to be inlined.

   Restricting ourselves to single inheritance would allow a more efficient implementation of inheritance: inherited slots could be placed at the same offset from the start of a structure as in their parents allowing exactly the same code sequence to be used for selecting a slot — no matter what it's type. This optimisation would eliminate the need to pass dictionaries around; greatly improving performance.

By choosing the best options (inline structure operations, use pattern matching and avoid overloading), we are able to generate exactly the same code as if no abstraction mechanisms had been used.

6 Conclusions

Our experience of being able to name fields has been entirely positive — we feel that it significantly improves the readability and maintainability of our programs. Having an elegant notation for updates is also very useful. Programs using these features are easier to maintain and the code is very readable.

The best way to deal with inheritance is not yet known. A more advanced object-oriented extension to Haskell may provide the same capabilities we have implemented. Simplifying to a single inheritance style would eliminate the performance problems introduced through the use of the class system.

Much of the implementation baggage could be eliminated by removing non-constant defaults and inheritance. This would make structure creation trivial: an update is applied to a structure containing the constant defaults. No class or instance declarations would be generated by structures; only data declarations. No support functions would be required — all structure operations could be expanded inline. Such a stripped-down system would address many, but not all, of the engineering issues described earlier. At a minimum, such a system should be considered for Haskell 1.3.

Acknowledgments

We are grateful to Warren Burton, Mark Jones, and Randy Hudson for their comments on an early design of this system. Sandra Loosemore and others in the Yale Haskell group also provided valuable assistance.

References

[1] FW Burton and RD Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.

[2] P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain, and J Peterson. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27, May 1992.

[3] R Milner, M Tofte, and R Harper. *The definition of Standard ML.* MIT Press, 1990.

[4] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Principles of Programming Languages*, pages 154–165. ACM, January 1992.

[5] D Rémy. Typechecking records in a natural extension of ML. In *Principles of Programming Languages*, pages 242–249. ACM, January 1989.

[6] GL Steele. *Common Lisp — The Language.* Digital Press, 2nd edition, 1994.

[7] PL Wadler. Fixing a space leak with a garbage collector. *Software — Practice and Experience*, 17(9):595–608, 1987.

[8] PL Wadler. Views — a way for pattern matching to cohabit with data abstraction. Technical Report 34, Programming Methodology Group, Chalmers University, Sweden, March 1987.

[9] PL Wadler and S Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages.* ACM, January 1989.