Lightweight Extensible Records for Haskell

Mark P. Jones
Oregon Graduate Institute
mpj@cse.ogi.edu

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

September 6, 1999

Abstract

Early versions of Haskell provied only a *positional* notation to build and take apart user-defined datatypes. This positional notation is awkward and error-prone when dealing with datatypes that have more than a couple of components, so later versions of Haskell introduced a mechanism for *labeled fields* that allows components to be set and extracted by *name*. While this has been useful in practice, it also has several significant problems; for example, no field name can be used in more than one datatype.

In this paper, we present a concrete proposal for replacing the labeled-field mechanisms of Haskell 98 with a more flexible system of records that avoids the problems mentioned above. With a theoretical foundation in the earlier work of Gaster and Jones, our system offers lightweight, extensible records and a complement of polymorphic operations for manipulating them. On a more concrete level, our proposal is a direct descendent of the Trex implementation ("typed records with extensibility") in Hugs, but freed from the constraints of that setting, where compatibility with Haskell 98 was a major concern.

1 Introduction

Records are one of the basic building blocks for data structures. Like a simple tuple or product, a record gathers together some fixed number of components, each of a potentially different type, into a single unit of data. Unlike a tuple however, the individual components are accessed by *name* rather than *position*. This is particularly important in languages where there is no support for pattern matching, or in cases where the number of different components would make pattern matching unwieldy and error-prone.

Haskell allows programmers to define a wide range of algebraic datatypes, but early versions of the language (up to and including Haskell 1.2) did not provide any support for records. As Haskell became an increasingly attractive platform for general purpose program development, the need for some form of records became more pressing: it is quite common to find data structures with many components in real-world applications, and it is very awkward to deal with such datatypes in a language that supports only the traditional constructor and pattern matching syntax for algebraic datatypes. Motivated by such practical concerns, the mechanisms for defining algebraic datatypes were extended in Haskell 1.3 (and carried over into Haskell 1.4 and Haskell 98) to support *labeled fields*, which allows the components of a datatype to be accessed by name. In essence, the syntax of Haskell was extended so that the definition of a particular algebraic datatype could include not just the names of its constructors, as in earlier versions of the language, but also the names for its selectors. We review the Haskell 98 record system in Section 2.

The Haskell 98 system has the merit of being explicable by translation into a simpler positional notation, and has no impact at all on the type system. However, this simplicity comes at the price of expressiveness, as we discuss in Section 2.1. This paper presents a concrete proposal for an alternative record system, designed to replace the current one (Section 3).

Our proposal is based closely on the extensible records of Gaster and Jones [4]. An implementation of their system has been available for a couple of years, in the form of the "Trex" extension to Hugs, so quite a bit of experience has accumulated of the advantages and shortcomings of the system. Trex is designed to be compatible with Haskell 98, and that in turn leads to some notational clumsiness. This proposal instead takes a fresh look at the whole language.

The resulting design is incompatible with Haskell 98 in minor but pervasive ways; most Haskell 98 programs would require modification. However, we regard this as a price worth paying in exchange for a coherent overall design. We review the differences between our proposal and Trex, Gaster's design, and other system in Section 4.

The paper contains essentially no new technical material; the design issues are mainly notational. So why have we written it? Firstly, there seems to be a consensus that some kind of full-blown record system is desirable, but debate is hampered by the lack of a concrete proposal to serve as a basis for discussion. We hope that this paper may serve that role. Second, it is not easy to see what the *disadvantages* of a new feature might be until a concrete design is presented. Our system does have disadvantages, as we discuss in Section 5. We hope that articulating these problems may spark off some new ideas.

2 Haskell 98 records: datatypes with labeled fields

The Haskell 98 record system simply provides syntactic sugar for what could otherwise be written using ordinary, positional, algebraic data types declarations. For example, to a first approximation, the following definition:

```
data List a = Nil
            | Cons {head :: a, tail :: List a}
```

can be thought of as an abbreviation for separate datatype and selector definitions:

```
data List a = Nil              -- datatype
            | Cons a (List a)

head (Cons x xs) = x           -- selectors
tail (Cons x xs) = xs
```

In fact, Haskell takes things a step further by introducing special syntax for construction and update operations. These allow programmers to build and modify data structures using only the names of components, and without having to worry about the order in which they are stored. For example, we can build a list with just one element in it using either of the constructions `Cons{head=value, tail=Nil}` or `Cons{tail=Nil, head=value}`, and we can truncate any non-empty list `xs` to obtain a list with just one element using the update expression `xs{tail=Nil}`.

The following definition illustrates another convenient feature of Haskell's record notation:

```
data FileSystem
  = File {name :: String, size :: Int, bytes :: [Byte]}
  | Folder {name :: String, contents :: [FileSystem]}
```

Values of this datatype represent a standard hierarchical file system with files contained in potentially nested folders. Notice that `name` is used as a field name in both branches; as a result, the selector function `name` that is generated from this definition can be applied to both `File` and `Folder` objects without requiring the programmer to treat the two alternatives differently each time the name of a `FileSystem` value is required.

The result of these extensions is a record-like facility, layered on top of Haskell's existing mechanisms for defining algebraic datatypes. From a theoretical perspective, of course, these features are just a form of syntactic sugar, and do nothing to make the language any more expressive. In practice, however, they have significant advantages, greatly simplifying the task of programming with data structures that have many components. Moreover, the resulting programs are also more robust because the code for selectors, update, and construction is generated directly by the compiler, automatically taking account of any changes to the datatype when fields or constructors are added, deleted or reordered.

2.1 Shortcomings of Haskell 98 records

Unfortunately, there are also some problems with the Haskell 98 approach:

- Record types are not lightweight; record values can only be used once a suitable algebraic data type has been defined. By contrast, the standard Haskell environment automatically provides lightweight tuple types—records without labels—of all sizes.
- Field names have top-level scope, and cannot be used in more than one datatype. This is a serious problem because it prevents the direct use of a datatype with labeled fields in any context where the field names are already in scope, possibly as field names in a different datatype. The only ways to work around such conflicts are to rely on tedious and error-prone renaming of field names, or by using (module) qualified names.
- Within a single datatype definition, we can only use any given field name in multiple constructors if the same types appear in each case. In the definition of the `FileSystem` datatype above, for example, it was necessary to to use different names to distinguish between the contents (i.e., `bytes`) of a `File`, and the `contents` of a `Folder`.
- There is no way to add or remove fields from a data structure once it has been constructed; records are not extensible. Each record type stands by itself, unrelated to any other record type; functions over records are not polymorphic over extensions of that record.

In the remaining sections of this paper, we present a concrete proposal for replacing the labeled field mechanisms of Haskell with a more flexible system of records that avoids the problems described above. There seems little point in retaining the current labeled field mechanisms of Haskell in the presence of our proposed extensions, as this would unnecessarily complicate the language, and duplicate functionality. Firm theoretical foundations for our system are provided by the earlier work of Gaster and Jones [5], with a type system based on the theory of qualified types [8]. This integrates smoothly with the rest of the Haskell type system, and supports lightweight, extensible records with a complement of polymorphic operations for manipulating them. On a more concrete level, our proposal is inspired by practical experience with the Trex implementation ("typed records with extensibility") in current versions of the Hugs interpreter, but freed from the constraints of that setting, where compatibility with Haskell 98 was a major concern.

3 The proposed design

This section provides an informal overview of our proposal for adding a more flexible system of extensible records to Haskell. It covers all of the key features, and sketches out our proposed syntax. Some aspects of our proposal are illustrated using extracts from a session with an interpreter for a Haskell dialect that supports our extensions. The interpreter prompts for an input expression using a single `?` character, and then displays the result of evaluating that expression on the next line. At the time of writing, we have not actually built such an interpreter. However, based on our experience implementing and using the Trex system in Hugs, we are confident that our proposals are feasible, practical, and useful.

We begin by describing the basic syntax for constructing and selecting from record values (Section 3.1), and for representing record types (Section 3.3). But for a few (mostly minor) differences in syntax, these aspects our proposal are almost indistinguishable from the lightweight records of Standard ML (SML). Turning to issues that are specific to Haskell, we show that record types can be included as instances of the standard classes for equality, `Eq`, and display, `Show` (Section 3.4). One key feature of our proposal, which clearly distinguishes it from the existing mechanisms in languages like SML and Haskell 98, is the support for *extensibility* (Section 3.5). By allowing record extension to be used in pattern matching, we also provide a simple way to select or remove specific record components. A combination of extension and removal can be used to update or rename individual field in a record. In practice, we believe that these operations are useful enough to warrant a special syntax (Section 3.6). A second key feature of the underlying type system is *row polymorphism* (Section 3.7), and this leads us to introduce a general syntax for rows (Section 3.8). Finally, we turn to a collection of additional features that are less essential, but that look very attractive (Section 3.10).

3.1 Construction and Selection

In essence, records are just collections of values, each of which is associated with a particular label. For example:

```
{a = True, b = "Hello", c = 12::Int}
```

is a record with three components: an `a` field, containing a boolean value, a `b` field containing a string, and a `c` field containing the number `12`. The order in which the fields are listed is not significant, so the same record value could also be written as:

```
{c = 12::Int, a = True, b = "Hello"}
```

These examples show simple ways to construct record values. We can also inspect the values held in a record using the traditional dot notation, where an expression of the form `r.l` simply returns the value of the `l` component in the record `r`. For example:

```
? {a = True, b = "Hello", c = 12::Int}.a
True
? let f r = r.b in f {a = True, b = "Hello"}
"Hello"
?
```

In all previous versions of Haskell, the '`.`' character has been used to represent function composition. It has also been used in more recent versions of Haskell in the syntax for qualified names. The first of these is clearly incompatible with our proposal, as it would allow a second reading of `r.l` as the composition of `r` with `l`. To avoid this conflict, we propose adopting a different symbol for function composition; we believe that `#` would be a good choice, but debate on that is beyond the scope of this paper. Our use of the dot notation is, however, entirely compatible with the syntax for qualified names, and with proposals for a structured module namespace in the style of Java packages. Another appealing consequence of this design is that it gives a single and consistent reading to the '`.`' character as selection, be it from a record or a module. On a practical level, this shows up in minor, but pleasing ways. For example, we can remove the rather ad-hoc restriction in the Haskell 98 syntax for qualified name that currently prohibits the use of spaces in a qualified name like `Prelude.map` that might otherwise have been confused with compositions like `Just . f`.

Repeated selections can be used to extract values from record-valued components of other records. As usual, "`.`" associates to the left, so that `r.l.k` is equivalent to `(r.l).k`:

```
? {a = True, b = {x="Hello"}, c = 12::Int}.b.x
"Hello"
?
```

3.2 Pattern matching

Record values can also be inspected by using pattern matching, with a syntax that mirrors the notation used for constructing a record:

```
? (\{a=x, c=y, b=_} -> (y,x))
  {a=True, b="Hello", c=12::Int}
(12,True)
?
```

The order of fields in a record *pattern* (unlike a record expression) is significant because it determines the order—from left to right—in which they are matched. Consider the following two examples:

```
? [x | {a=[x],b=True} <- [{b=undefined,a=[]},
                           {a=[2],b=True}]]
[2]

? [ x | {b=True, a=[x]} <- [{b=undefined, a=[]},
                             {a=[2],b=True}]]
Error: {program uses the undefined value}
?
```

In the first example, the attempt to match the pattern `{a=[x], b=True}` against the record `{b=undefined, a=[]}` fails because field `a` is matched first and `[x]` does not match the empty list; but matching the same pattern against `{a=[2],b=True}` succeeds, binding `x` to 2. Swapping the order of the fields in the pattern to `{b=True, a=[x]}` forces matching to start with the `b` component. But the first element in the list of records used above has `undefined` in its `b` component, so now the evaluation produces a run-time error message.

3.3 Record types

Like all other values in Haskell, records have types, and these are written in the form `{r}`, where `r` represents a 'row' that associates labels with types. For example, the record:

```
{c = 12::Int, a = True, b = "Hello"}
```

has type:

```
{a::Bool, b::[Char], c::Int}
```

This tells us, unsurprisingly, that the record has three components: an `a` field containing a `Bool`, a `b` field containing a `String`, and a `c` field of type `Int`. As with record values themselves, the order of the components in a row is not significant, and so the previous type can also be written as:

```
{b::String, c::Int, a::Bool}
```

In the special case when the row is empty, we obtain the empty record type {}, whose only value (other than $\perp$) is the empty record, also written as {}.

Of course, the type of a record must be an accurate reflection of the fields that appear in the corresponding value. The following example produces an error because the specified type does not list all of the fields in the record value:

```
? {a=True, b="Hello", c=12} :: {b::String, c::Int}

ERROR: Type error in type signature expression
*** term           : {a=True, b="Hello", c=12}
*** type           : {a::Bool, b::[Char], c::a}
*** does not match : {b::String, c::Int}
*** because        : field mismatch

?
```

Notice that our system does not allow the kind of subtyping on record values that would permit a record like `{a=True, b="Hello", c=12}` to be treated implicitly as having type `{b::String, c::Int}`, simply by 'forgetting' about the `a` field. Finding an elegant and tractable way to support this kind of implicit coercion in a way that integrates properly with other aspects of the Haskell type system remains an interesting problem for future research. However, as we shall see in Section 3.7, our use of row polymorphism offers many of the benefits of subtyping.

3.4 Overloaded operations on records

Record types are automatically included in the standard `Eq` and `Show` classes of Haskell, provided that the types of each field in the records concerned are themselves instances of the appropriate class. Our interpreter uses these functions to allow comparison and display of record values in the following examples:

```
? {a=True, b="Hello"} == {b="Hello", a=True}
True
? {a = True, b = "Hello", c = 12::Int}
{a=True, b="Hello", c=12}
? {c = 12::Int, a = True, b = "Hello"}
{a=True, b="Hello", c=12}
?
```

Note that these operations always process record fields according the dictionary ordering of their labels. The fact that the fields appear in a specific (but, frankly, arbitrary) order is very important; the results of the `(==)` operator and the `show` function must be uniquely determined by their input, and not by the way in which that input is written. The records used in the last two lines of the example have exactly the same value, and so we expect exactly the same output for each. The difference in behavior between the following two examples is also a consequence of this:

```
? {a=0, b="Hello"} == {b=undefined, a=1}
False
? {b=0, a="Hello"} == {a=undefined, b=1}
Error: {program uses the undefined value}
?
```

In the first case, the equality test returns `False` because the two values differ in their first component, labeled as `a`. In the second case, where the labels have been switched, the equality test begins with an attempt to compare the string `"Hello"` with an undefined value, resulting in an error.

Arguably, records should automatically be instances of the classes `Ord`, `Ix`, `Bounded`, and `Read`, on the grounds that these are the classes (beyond `Eq` and `Show`) of which tuples are automatically instances. The difficulty is that the order of the fields matters even more for these four than they do for the former two. There is no difficulty in principle — fields can be lexically ordered — but the arbitrary nature of this ordering is apparent in more than just the strictness of the class methods.

3.5 Extension

An important property of our system is that the same label name can appear in many different record types, and potentially with a different type in each case. However, all of the examples that we have seen so far deal with records of some fixed shape, where the set of labels and the type of values associated with each one are fixed, and there is no apparent relationship between records of different type. In fact, all record values and record types in our system are built up incrementally, starting from an empty record and extending it with additional fields, one at a time. This is what it means for records to be *extensible*.

In the simplest case, any given record `r` can be extended with a new field labeled `l`, provided that `r` does not already include an `l` field. For example, we can construct the record `{a=True, b="Hello"}` by extending `{a = True}` with a field `b="Hello"`:

```
? {{a=True} | b = "Hello"}
{a=True, b="Hello"}
?
```

Note that we write the record value that is being extended first, followed by a '`|`' character, and then by a list of the fields that are to be added. Another way to construct exactly the same result is by extending `{b = "Hello"}` with a field `a=True`:

```
? {{b = "Hello"} | a = True}
{a=True, b="Hello"}
?
```

It is often convenient to add more than one field at a time, as shown in the following example:

```
? {{b1="World"} | a=True, b="Hello", c=12::Int}
{a=True, b="Hello", b1="World", c=12}
?
```

On the other hand, a record cannot be extended with a field of the same name, even if it has a different type. The following example illustrates this:

```
? let r = {c=12::Int} in {r | c=True}
ERROR: {c::Int} already includes a "c" field
```

```
?
```

Much the same syntax can be used in patterns to decompose record values:

```
? (\{r | b=bval} -> (bval,r)) {a=True, b="Hello"}
("Hello",{a=True})
?
```

Notice that we can match, not just against individual components of a record value, but also against the portion of the record that is left after the explicitly named fields have been removed. In previous examples, we saw how a record could be extended with new fields. As this example shows, we can use pattern matching to do the reverse operation of removing fields from a record.

3.6 Update

It is often useful to update a record by changing the values associated with some of its fields. Update operations like this can be coded by hand, using pattern matching to remove the appropriate fields, and then extending the resulting record with the new values. However, it seems much more attractive to provide special syntax for these operations, using `:=` instead of `=` to distinguish update from extension. Slightly more formally, a record expression:

```
{r | x := e}
```

is treated as an abbreviation for the following update:

```
case r of {s | x=_} -> {s | x=e}
```

Providing a special syntax makes updates easier for programmers to code and also makes them easier for a compiler to recognize, which can often permit a more efficient implementation that avoids building the intermediate record `s`. For further convenience, we allow updates to be freely mixed with record extension in expressions like the following:

```
{r | x=2, y:=True}
```

Unlike the extension syntax, however, it does not seem sensible to allow the use of update syntax in a record pattern.

3.7 Row polymorphism

We can also use pattern matching to understand how selector functions are handled. For example, evaluating an expression of the form `r.l` is much like passing `r` as an argument to the function:

```
(\{_|l=value} -> value)
```

This function is polymorphic in the sense that it can be used with *any* record containing an `l` field, regardless of the type associated with that particular component, or of any other fields that the record might contain:

```
? (\{_|l=value} -> value) {l=True, b="Hello")
True
? (\{_|l=value} -> value)
  {name="Record", age=2, l="None")
"None"
?
```

To see how this works, we need to look at the type of this function, which can be inferred automatically as:

```
(r\x) => {r | x::a} -> a
```

There are two important pieces of notation here that deserve further explanation:

- `{r | l::a}` is the type of a record with an `l` component of type `a`. The *row variable* `r` represents the rest of the row; that is, it represents any other fields in the record apart from `l`. This syntax for record type extension mirrors the syntax that we have already seen in the examples above for record value extension. We discuss rows further in Section 3.8.
- The constraint `r\l` tells us that the type on the right of the `=>` symbol is only valid if "`r` lacks `l`," that is, if `r` is a row that does not contain an `l` field. If you are already familiar with Haskell type classes, then you may like to think of `\l` as a kind of class constraint, written with postfix syntax, whose instances are precisely the rows without an `l` field.

For example, if we apply our selector function to a record `{l=True,b="Hello"}` of type `{b::String, l::Bool}`, then we instantiate the variables `a` and `r` in the type above to `Bool` and `(b::String)`, respectively.

The row constraints that we see here can also occur in the type of any function that operates on record values if the types of those records are not fully determined at compile-time. For example, given the following definition:

```
average r = (r.x + r.y) / 2
```

our interpreter would infer a principal type of the form:

```
average :: (Fractional a, r\y, r\x)
        => {r | y::a, x::a} -> a
```

However, any of the following, more specific types could be specified in a type declaration for the `average` function:

```
average  :: (Fractional a) => {x::a, y::a} -> a
average  :: (r\x, r\y)
         => {r | x::Double, y::Double} -> Double
average  :: {x::Double, y::Double} -> Double
average  :: {x::Double, y::Double, z::Bool} -> Double
```

Each of these is an instance of the principal type given above.

These examples show an important difference between the system of records described here, and the record facilities provided by SML. In particular, SML prohibits definitions that involve records for which the complete set of fields cannot be determined at compile-time. So, the SML equivalent of the `average` function described above would be rejected because there is no way to determine if the record `r` will have any fields other than `x` or `y`. SML programmers usually avoid such problems by giving a type annotation that completely specifies the structure of the record. Of course, if a definition is limited in this way, then it also less useful.

With the expected implementation for our type system, as described in Section 3.9, there is an advantage to knowing the full type of a record at compile-time because it will allow the compiler to generate more efficient code. However, unlike SML, the type system also offers the flexibility of polymorphism and extensibility over records if that is needed.

3.8 Rows

To deal more formally with record types, we extend the kind system of Haskell with a new kind, *row*: in a record type of the form {expr}, the expression expr must have kind *row*. Types of kind *row* are written using essentially the same notation that we use for records, but enclosed in parentheses rather than braces. For example:

- The empty row is written as `()`, and the empty record type `{}` is really just a convenient abbreviation for `{()}`. Note that this is a change from Haskell 98, where the symbol `()` is used to denote the unit type and its only proper (i.e., non bottom) value. With our proposal, the empty record, `{}` of type `{}`, can be used in place of a special unit value.
- Non-empty rows are formed by extension. For example, `(r|x::Int)` is the row obtained from row `r` by extending it with an `x` field of type `Int`. Multiple fields can be specified using comma-separated lists. For example, `(r|x::Int, y::Bool)` is a shorthand for `((r|x::Int)|y::Int)`. Another shorthand allows us to write extensions of the empty row as a comma-separated list of fields. For example, `(x::Int, y::Bool)` is an abbreviation for `(()|x::Int, y::Bool)`, which is in turn just an abbreviation for `((()|x::Int)|y::Bool)`. In all cases, we allow the outermost pair of parentheses to be omitted when a row expression appears inside a pair of braces. For example, `{(x::Int)}` can be abbreviated to `{x::Int}`.
- Row variables (i.e., type variables of kind *row*) represent unknown rows. As in Haskell, the kinds of all type variables are inferred automatically by the compiler using a simplified form of type inference.

Row expressions can be used anywhere that a type constructor of kind *row* is required, including the right hand side of a `type` definition, or the parameters of any programmer defined class or datatype constructor. For example, the following definitions introduce a type synonym, `Point`, of kind *row*, and then extend this to define a second type synonym `ColoredPoint` that adds an extra `Color` field:

```
-- Point :: row
type Point        = (x::Int, y::Int)

-- ColoredPoint :: row
type ColoredPoint = (Point | c::Color)
```

As the comments indicate, `Point` and `ColoredPoint` have kind *row*. (Haskell `type` declarations can already introduce type constructors of kinds other than $*$.)

This style of definition allows us to build up row types (and hence record types) in a style akin to single inheritance. It does not, however, support multiple inheritance. For example, the following definition of `ColoredPoint` is ill-formed because our proposal requires a field list to the right of a `|`, and does not permit arbitrary row expressions.

```
type Point           = (x::Int, y::Int)
type Coloring        = (c::Color)
type BadColoredPoint = (Point | Coloring) -- NO!
```

While we can model single inheritance, this style does not make it possible to define polymorphic functions. To illustrate this point, consider the following example:

```
move :: Int -> Int -> {Point} -> {Point}
move a b p = {p | x:=a, y:=b}
```

The function `move` works fine on values of type `{Point}` but it is type-incorrect to apply it to a value of type `{ColoredPoint}`.

However, it is easy to obtain a `move` that is applicable to points of all varieties by defining the types a little differently:

```
-- Point :: row -> row
type Point r = (r | x::Int, y::Int)

-- Colored :: row -> row
type Colored r = (r | c::Color)

-- ColoredPoint :: row -> row
type ColoredPoint r = Point (Colored r)
```

Notice that it is entirely legitimate for the type synonym `Point` to abstract over a row variable, so that `Point` itself has kind *row*→*row*. Of course, the original definitions for each of these rows are just extensions of the empty row `()`. For example, with these definitions, we can write the type of a record `{x=0, y=0, c=Red}` as `{ColoredPoint()}`. Now we can define `move` thus:

```
move :: (r\x, r\y) => Int -> Int
                   -> {Point r} -> {Point r}
move a b p = {p | x:=a, y:=b}
```

The type neatly expresses that `move` works on any "subclass" (i.e., substitution instance) of `{Point r}`; any `{ColoredPoint s}` will do, for example. It also expresses that `move` returns a `Point` of the same variety as it is given as its argument, a well-known problem in many object systems (e.g., Cardelli and Mitchell [2, Section 2.6] discuss the "update problem" at some length). The observation that row polymorphism deals with this problem is not new [1].

Even though we have constructed `ColoredPoint` in a "sequential" way, row composition is commutative. For example, the following definition of `ColoredPoint` is entirely equivalent—to see this, just expand out the synonyms:

```
-- ColoredPoint :: row -> row
type ColoredPoint r = Colored (Point r)
```

We can also use variables of kind *row* as the parameters of user-defined datatypes, thus:

```
data T r = MkT {r | x :: Bool}
```

According to the normal rules for kind inference, `T` will be treated as a type constructor of kind *row* $\rightarrow *$, but it is clear that this kind is inaccurate; it does not seem sensible to allow `T` to be applied to *any* row argument, only to those that do not have an `x` field. Our kind system is not expressive enough to capture this restriction directly, but it can be reflected by including a constraint `r\x` in the type of `T`. From a practical perspective, this is a minor issue; without any further restrictions, the type system will allow us to use types like `T (x::Int)` without flagging any errors, but it will not allow us to construct any values of that type, apart from $\perp$. However, although it makes no real difference, it seems more consistent with other aspects of Haskell to require the definition of datatypes like `T` to reflect any constraints that are needed to ensure that their component types are well-formed. For example, we can correct the previous definition of `T` by inserting a `r\x` constraint, as follows:

```
data (r\x) => T r = MkT {r | x :: Bool}
```

(Haskell old-timers who remember the `Eval` class, may also recall that similar constraints were once required on datatypes that used strictness annotations.)

3.9 Implementation

A major merit of Gaster and Jones's record system is that it smoothly builds on Haskell's type class mechanism. This analogy applies to the implementation as well. The details are covered elsewhere [5] but the basic idea is simple enough.

Each "lacks" constraint in a function's type gives rise to an extra argument passed to that function that constitutes "evidence" that the constraint is satisfied. In particular, evidence that `r` lacks a field `l` is given by passing the offset at which `l` would be stored in the record `{r}` extended by `l`.

The size of the entire record is also required when performing record operations. This can be obtained either from the record itself, or by augmenting "evidence" to be a pair of the offset and record size.

As usual with overloading, much more efficient code can be obtained by specialisation. In the case of records, specialisation "bakes into" the code the size of the record and the offsets of its fields.

3.10 Additional Features

In this section, we collect together some small, but potentially useful ideas for further extensions of our core proposal.

3.10.1 Presentation of inferred types

One comment that some experienced users of the Trex system have made is that user-written type signatures become unreasonably large. For example, consider the type signature for `move` in Section 3.8:

```
move :: (r\x, r\y) => Int -> Int -> {Point r} -> {Point r}
move a b p = {p | x:=a, y:=b}
```

The `Point` synonym allowed us not to enumerate (twice) the details of a `Point`, but the context `(r\x, r\y)` must enumerate the fields that `r` must lack, otherwise the type is ill-formed. This is annoying, because, if we expand the type synonym, it is absolutely manifest that `r` must lack fields `x` and `y`:

```
move :: (r\x, r\y) => Int -> Int
     -> {r | x::Int, y::Int}
     -> {r | x::Int, y::Int}
```

Not only is it annoying, but it is also non-modular: adding a field to `Point` will force a change to the type signature of `move`, even if `move`'s code does not change at all. In practice, these annoyances are enough to cause programmers to omit type signatures altogether on functions with complex types — arguably just the functions for which a type signature would be most informative.

Thus motivated, an obvious suggestion is to permit constraints in a type signature to be omitted if they could be inferred directly from the rest of the signature. This is akin to the omission of explicit universal quantification. Haskell already lets us write `f::a->a`, when we really mean `f::forall a.a->a`. The "`forall a`" is inferred. In a similar way, we propose that a similar inference process adds "lacks" constraints to a type signature, based solely on the rest of the type signature (after expanding type synonyms).

Note that this is a matter of presentation only, and the actual types used inside the system do not change. Constraints of this form cannot always be omitted from the user type signature, as illustrated in the following (pathological) example:

```
g   :: (r\l) => {r} -> Bool
g x = {x | l=True}.l
```

As a slightly more realistic example where constraints cannot be omitted, consider the following datatype of trees, which allows each node to be annotated with some additional information `info`:

```
data Tree info a b
  = Leaf {info | value :: a}
  | Fork {info | left  :: Tree info a b,
                 value :: b,
                 right :: Tree info a b}
```

In an application where there are many references to the height of a tree, we might choose to add height information to each node, and hence avoid repeated unnecessary repeated computation:

```
addHeight (Leaf i) = Leaf {i | height=0}
addHeight (Fork i)
   = Fork {i | left := l, right := r,
               height = 1 + max (height l)
                                (height r) }
     where l = addHeight i.left
           r = addHeight i.right

height (Leaf i) = i.height
height (Fork i) = i.height
```

Careful examination of this code shows that the type of `addHeight` is:

```
(info\height, info\left, info\right)
  => Tree info a b -> Tree (info | height::Int) a b
```

Note here that only the first of the three constraints, `info\height`, can be inferred from the body of the type, and hence the remaining two constraints cannot be omitted. In our experience, such examples are quite uncommon, and we believe that many top-level type signatures could omit their "lacks" constraints, so this facility is likely to be very attractive in practice.

3.10.2 Tuples as values

As in Standard ML, records can be used as the underlying representation for tuples; all that we need to do is pick out canonical names for each position in a tuple. For example, if we write `field1` for the label of the first field in a tuple, `field2` for the second, and so on, then a tuple value like `(True,12)` is just a convenient shorthand for `{field1=True, field2=12}`, and its type `(Bool,Int)` is just a shorthand for `{field1::Bool, field2::Int}`. The advantages of merging currently separate mechanisms for records and tuples are clear, as it can remove redundancy in both the language and its implementations. In addition, it offers a more expressive treatment of tuples because it allows us to define functions like:

```
fst  :: (r\field1) => {r|field1::a} -> a
fst r = r.field1
```

that can extract the first component from any tuple value; in current versions of Haskell, the `fst` function is restricted to pairs, and different versions must be defined for each different size of tuple.

The exact choice of names for the fields of a tuple is a matter for debate. For example, it would even be possible (though not necessarily desirable) to use the unadorned integers themselves — e.g. `x.2`, `{r | 3=True}`.

3.10.3 Constructors as field names

Programmers often use algebraic data types to define *sums*; for example:

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
```

In such cases it is common to define projection functions:

```
just :: Maybe a -> a
just (Just x) = x

left :: Either a b -> a
left (Left x) = x

right :: Either a b -> b
right (Right y) = y
```

An obvious notational convenience would be to re-use the "dot" notation, an allow a constructor to be used after the dot to indicate a projection. That is, `m.Just` would be equivalent to `just m`, and `e.Left` would be equivalent to `left e`, and so on. This would often avoid the need to define the projection functions explicitly.

This notation is particularly convenient for Haskell 98 `newtype` declarations, which allow one to declare a new data type isomorphic to an old one. For example:

```
newtype Age = Age Int
```

Here `Age` is isomorphic to `Int`. The "constructor" `Age` has the usual type

```
Age :: Int -> Age
```

and allows one to convert an `Int` into an `Age`. (We put "constructor" in quotes, because it is implemented by the identity function, and has no run-time cost.) The reverse coercion is less convenient but, if the constructor could be used as a projection function, one could write `a.Age` to coerce a value `a::Age` to an `Int`.

The proposal here is entirely syntactic: to use the same "dot" notation for sum projections as well as for record field selection. The two are syntactically distinguishable because constructors begin with an upper-case letter, whereas fields do not.

3.10.4 Punning

It is often convenient to store the value associated with a particular record field in a variable of the same name. Motivated by practical experience with records in the Standard ML community, it is useful to allow a form of *punning* in the syntax for both record expressions and patterns. This allows a field specification of the form `var` is treated as an abbreviation for a field binding of the form `var=var`, and is referred to as a *pun* because of the way that it uses a single name in two different ways. For example, `(\{x,y,z} -> x + y + z)` is a function whose definition uses punning to sum the components of a record. Punning permits definitions such as:

```
f :: {x::Int, y::Int} -> {x::Int, y::Int}
f {x,y} = {x=y-1, y=x+1}
```

Here, in the expressions `y-1` and `x+2`, the variables `x` and `y` are bound to the fields of the same name in `f`'s argument.

Punning was also supported in some versions of Haskell prior to Haskell 98, but was removed because of concerns that it was not well behaved under renaming of bound variables. For example, in the definition

```
f :: Int -> Int
f x = x+1
```

one can rename both occurrences of "x" to "y". But in the definition:

```
f :: {x::Int} -> Int
f {x} = x+1
```

one cannot perform such a renaming, because `x` is serving as a record label. In fact punning is perfectly well-behaved under these circumstances, provided one remembers that it is simply an abbreviation, which may need to be expanded before completing the task of renaming:

```
f :: {x::Int} -> Int
f {x=x} = x+1
```

Now one can rename as follows:

```
f :: {x::Int} -> Int
f {x=y} = y+1
```

Anecdotal experience from the Standard ML community suggests that the merits of punning greatly exceed the disadvantages.

3.10.5 Renaming

It is easy to extend the set of supported operations on records to include a renaming facility, that allows us to change the name associated with a particular field in a record. This notation can be defined more formally in terms of a simple translation:

```
{r | x->y}  =  case r of {s | x=t} -> {s | y=t}
```

However, as in the case of update (Section 3.6), use of this notation makes it easier for programmers to use renaming, and easier for a compiler to implement it.

3.10.6 Kind signatures for type constructors

Earlier in this paper we used comments to indicate the kinds of type constructors in examples like:

```
  -- Point :: row -> row
  type Point r = (r | x::Int, y::Int)
```

The clear implication is that Haskell should provide a way to give kind signatures for type constructors, perhaps simply by permitting the above commented kind signature. Such kind signatures are syntactically distinguishable from value type signatures, because type constructors begin with an upper case letter. Another alternative would be to allow kind annotations of the form:

```
  type Point (r::row) :: row = (r | x::Int, y::Int)
```

Annotations like this would also be useful elsewhere, such as in `data` of `class` declarations.

The need for explicit kind information is not restricted to extensible records. In this very workshop proceedings, Hughes writes [7]:

```
  data Set cxt a = Set [a] | Unused (cxt a -> ())
```

The *only* reason for the `Unused` constructor which, as its name implies, is never used again, is to force `cxt` to have kind $* \to *$. It would be far better to say:

```
  Set :: (*->*) -> * -> *
  data Set cxt a = Set [a]
```

3.10.7 Topics for further work

Our proposal does not support all of the operations on records that have been discussed in the literature. Examples of this include:

- Record concatenation. This allows the fields of two distinct records to be merged to form a single record. Several researchers have studied this operator, or closely related variants. For example, Wand [14] used it as a way to describe multiple inheritance in object-oriented languages, and Rémy [13] described a technique for typing a form of record concatenation for 'free' in any language supporting record extension.
- Unchecked operations. These are variations of the operations on records that we have already seen that place slightly fewer restrictions on the types of their input parameters. For example, an unchecked extension operator guarantees that the specified field will appear in its result with the corresponding value, regardless of whether there was a field of the same name in the input record. With the checked operators that we have presented in this paper, the programmer must distinguish between the two possible cases using extension or update, as appropriate. Unchecked operations are supported, for example, in Rémy's type system for records in a natural extension of ML [12].
- First-class labels. This allows labels to be used and manipulated as program values, with a number of potentially useful applications. A prototype implementation was developed by Gaster [4], but there are still some details to be worked out.

It is not yet clear whether our proposal could be extended to accommodate these operations, and we believe that each of them would make interesting topics for future work.

4 Comparison with other systems

In this section we provide brief comparisons of our proposal with the facilities for defining an using records in other systems. We focus here on practical implementations, and refer interested readers to the work of Gaster and Jones [5] for comparisons of the underlying theory with other more theoretical proposals.

4.1 Comparison with Standard ML

The system of records in Standard ML was one of the original inspirations for this proposal, but of course our system also supports extensibility, update, and polymorphic operations over records. This last point shows up in Standard ML when we try to use a record in a situation where its corresponding set of field labels cannot be determined at compile-time, and resulting in a compile-time error.

4.2 Comparison with SML#

Based on his earlier work on type systems for records [11], Atsushi Ohori built a version of the Standard ML interpreter known as SML#, which extends the SML record system with support for update and for polymorphic operations over records[1]. Ohori's system does not provide the separation between rows and records that our proposal offers (Section 3.8), nor is it clear how records would interact with type classes, but it would be wrong to criticize SML# on the basis of these omissions, because they are much more relevant in the context of Haskell, with its advanced kind and class system, than they are in SML. Thus, apart from differences in syntax, the main advantage of our system over Ohori's is the support that it provides for extensibility.

[1]Further information about SML# is available on the World Wide Web at `http://www.kurims.kyoto-u.ac.jp/~ohori/smlsharp.html`.

4.3 Comparison with Trex

The proposal presented in this paper is closely related to the Trex implementation in current releases of Hugs 98 [10]. The only real differences are in the choice of notation:

- In record types, Trex uses `Rec r` where this proposal uses `{r}`. The latter choice could not be used with Trex because of the conflict with the syntax for labeled fields in Haskell 98.
- In record values, Trex uses `(...)` where this proposal uses `{...}`. The latter could not be used in Trex because it conflicts with the update syntax for datatypes with labeled fields in Haskell 98. For example, in Haskell 98, an expression of the form `e{x=12}` is treated as an update of value `e` with an `x` field of `12`. For the current proposal, we would expect to treat this expression as the application of a function `e` to a record `{x=12}` with just one field.
- Trex uses `(x::a | r)` where this proposal uses `(r | x::a)`. We deviate from Trex because it can be easy for the trailing "`| r`" to become lost when it follows a large block of field definitions. (In a similar way, Haskell puts guards at the beginning of an equation defining a function, rather than at the end as some languages do.) This choice is ultimately a matter of taste — we have not found any compelling technical reason to justify the use of one of these over the other.
- Like SML, Trex uses `#l` to name the selector for a field `l`; this proposal uses dot notation for field selection, and the function `#l` must be written as `(\r -> r.l)`. Dot notation could not be used in Trex because it conflicts with the use of `.` for function composition in Haskell.
- Trex does not support the update notation; update is one of several features that appeared in the original work on Trex that were not implemented in the Hugs prototype.
- Trex uses `EmptyRow` where this proposal uses `()`; the latter could not be used in Trex because it conflicts with the notation used for the unit type in Haskell 98.
- Trex does not use punning (Section 3.10.4) because of a conflict with the syntax for tuples: an expression like `(x,y)` could be read in two different ways, either as a tuple, or as an abbreviation for the record `(x=x, y=y)`.

In short, the current proposal differs in only small ways from Trex, and most of the changes were made possible only by liberating ourselves from any need to retain compatibility with the syntax of Haskell 98.

4.4 Comparison with Gaster's proposal

Our proposal is quite similar to that of [3]. Most notably, we both adopt the idea of using "`.`" for record selection.

We have gone further than Gaster by abandoning Haskell 98's current record system altogether, using "`()`" for the empty row instead of the unit tuple, providing a syntax for record updates (Section 3.6), and using constructors as selectors (Section 3.10.3). We have also elaborated a little more on the implications of row polymorphism. But the two proposals clearly share a common foundation.

5 Shortcomings of our proposal

One of the main reasons to turn a general idea into a concrete design is to highlight difficulties that deserve further attention.

5.1 Where Haskell 98 does better

We began this paper by describing some of the weaknesses of the labeled field mechanism in Haskell 98, and using those to motivate the key features of this proposal. In this section, therefore, we focus on areas where the Haskell 98 approach sometimes offers advantages over our proposal.

- *The double-lifting problem.* Part of the price that we pay for having lightweight records is that it becomes more expensive to embed a record in a datatype. For example, with our proposal, the definition of the following datatype introduces two levels of lifting:

  ```
  data P = MkP {x::Double, y::Double}
  ```

  In semantic terms, this means that the datatype `P` contains both $\perp$ and `MkP` $\perp$ as distinct elements. In implementation terms, it means that an attempt to access the `x` or `y` coordinates will require a double indirection. In comparison, the same definition in Haskell 98 introduces only one level of lifting. The same behavior can be recovered in our proposal by adding a strictness annotation to the record component of the datatype.

  ```
  data P = MkP !{x::Double, y::Double}
  ```

- *The unpacking problem.* Even if we use the second definition of the `P` datatype, it will only help to avoid two levels of indirection when we construct a value of type `P`; we still need a two stage process to extract a value from a datatype. For example, to extract the `x`, we must first remove the outer `MkP` constructor to expose the record from which the required `x` field can be obtained.

  This provides an additional incentive to adopt the use of constructors as field selectors (Section 3.10.3). This would allow the selection of the `x` component from a value `p` of type `P` to be written more succinctly as `p.MkP.x`.

  This approach does not help us to deal with examples like the `FileSystem` datatype from Section 1, where the same field name appears in multiple branches of an algebraic datatype. In Haskell 98, the compiler takes care of generating an appropriate definition for the selector function. With our proposal, this must be coded by hand:

  ```
  name            :: FileSystem -> String
  name (File r)   = r.name
  name (Folder r) = r.name
  ```

- *Strictness annotations.* Haskell 98 allows individual components of a datatype to be marked with strictness annotations, as in the following example:

  ```
  data P = MkP {x :: Double, y :: !Double}
  ```

  The proposal described in this paper does not allow this because record types are lightweight, not declared. An advantage is that the same labels can be used in different types. The disadvantage here is that there is no way to attach any special meaning, in this case a strictness annotation, to any particular label. One way to overcome this restriction would be to use lexically distinct sets of field labels to distinguish between strict and non-strict components. Alternatively, we could introduce strict versions of the extension and update operators. The problem with this approach is that strict evaluation will only be used when the programmer remembers to insert the required annotations.

The impact of these problems will depend, to a large extent on the way that records are used within algebraic datatypes.

5.2 Polymorphism

Although it is not part of the Haskell 98 standard, both Hugs and GHC allow the components of an algebraic datatype to be assigned polymorphic types. A standard example of this might be to define a concrete representation for monads as values of the following type:

```
data Mon m
  = MkMon {unit :: forall a. a -> m a,
           bind :: forall a, b. m a -> (a -> m b) -> m b}
```

To support the use of this datatype, the `MkMon` constructor, and, to a lesser degree, the `unit` and `bind` selectors are given a special status in the type checker, and are used to propagate explicit typing information to places where values of the datatype are used. (Type inference would not be possible without such information.) With our proposal, this special status is lost: all records are constructed in the same way, and all fields are selected in the same way. For example, the function to extend a record with a `unit` field is just:

```
(\r u -> {r | unit=u})
  :: (r\unit) => {r} -> a -> {r|unit::a}
```

The type variables `r` and `a` here range over monotypes (of kind *row* and $*$, respectively), and there is nothing to hint that a polymorphic value for `unit` should be expected.

Intuitively, it seems clear that we should still be able to propagate programmer-supplied type information to the places where it is needed, but the mechanisms that we need to support this are rather different from the mechanisms that are used to support the current Hugs and GHC extensions illustrated above. One promising approach, previously used in work with *parameterized signatures* [9], is to use so-called “has” predicates for records instead of the “lacks” predicates used here. These “has” predicates provide a looser coupling between records and their component types, which delays the need to resolve them to a point where more explicit typing information is likely to be available. However, it is not immediately obvious how we can integrate this approach with the main proposals in this paper, which rely instead on “lacks” predicates.

Another possibility is to provide a special typing rule for the syntactic composition of constructor application and record construction, effectively recovering the rule for constructors used by GHC and Hugs. GHC and Hugs's constructor-application rule is already restricted to the case where the constructor is applied to enough arguments to saturate all its universally-quantified arguments (e.g., `map MkMon xs` is rejected); requiring the record construction to be syntactically visible is arguably no worse.

5.3 Instances

In Section 3.4 we propose that records are automatically instances of certain built-in classes (`Eq`, `Show`, etc), and no others. Like any user-defined type, programmers may want to make a record type an instance of other classes, or to provide their own instance decalaration for the built-in classes. It is possible to define such instances for records whose shape (i.e., set of field names) is fixed. For example, we could define tuples as instances of standard Haskell classes in this manner:

```
instance (Ord a , Ord b)
    => Ord {field1::a, field2::a} where ...
```

However, some care is required to deal with instances for record types involving extension. To illustrate this point, consider the following collection of instance declarations:

```
instance C {r|x::Int}  where ... -- OK
instance C {r|x::Bool} where ...

instance D {r|x::a} where ...   -- INSTANCES
instance D {r|y::b} where ...   -- OVERLAP!
```

The first pair of instances for class `C` are acceptable, but the second pair will be rejected because they overlap. For example, these declarations provide two distinct, and potentially ambiguous ways for us to demonstrate that a type like `{x::Int,y::Bool}` is an instance of `D`. An overlap like this would not be a problem if we could be sure that both options gave the same final result, but there is no obvious way to guarantee this.

The trouble is that declarations like those for class `D` seem necessary for modular user-defined instances of record types. For example, imagine trying to declare `Eq` instances for a record. One might be led to say:

```
instance (Eq a, Eq {r}) => Eq {r | x::a} where
  {r1 | x=x1} == (r2 | x=x2}
       = x1 == x2 && {r1} == {r2}
```

But we need one such instance declaration for each distinct field label, which leads to declarations just like those for `D` above. The ambiguity in this case boils down to defining the order in which fields are compared. A way out of this impasse is an obvious piece of further work.

Acknowledgements

We would like to acknowledge our debt to John Hughes, David Espinosa, and the Haskell workshop referees, for their constructive feedback on this paper.

References

[1] L. Cardelli. Extensible records in a pure calculus of sub-typing. In Gunter and Mitchell [6], pages 373–426.

[2] L. Cardelli and J. Mitchell. Operations on records. In Gunter and Mitchell [6], pages 295–350.

[3] B. Gaster. Polymorphic extensible records for Haskell. In J. Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[4] B. Gaster. *Records, variants, and qualified types.* PhD thesis, Department of Computer Science, University of Nottingham, 1998.

[5] B. Gaster and M. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov. 1996.

[6] C. Gunter and J. Mitchell, editors. *Theoretical aspects of object-oriented programming.* MIT Press, 1994.

[7] R. Hughes. Restricted data types in Haskell. In E. Meijer, editor, *Haskell workshop*, Paris, Sept. 1999.

[8] M. Jones. A theory of qualified types. In *European Symposium on Programming (ESOP'92)*, number 582 in Lecture Notes in Computer Science, Rennes, France, Feb. 1992. Springer Verlag.

[9] M. Jones. Using parameterized signatures to express modular structure. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 68–78. ACM, St Petersburg Beach, Florida, Jan. 1996.

[10] M. Jones and J. Peterson. Hugs 98 user manual. Technical report, Oregon Graduate Institute, May 1999.

[11] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov. 1995.

[12] D. Rémy. Type inference for records in a natural extension of ML. In Gunter and Mitchell [6].

[13] D. Rémy. Typing record concatenation for free. In Gunter and Mitchell [6].

[14] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.