A Polymorphic Type System for Extensible Records and Variants

Benedict R. Gaster and Mark P. Jones
Department of Computer Science, University of Nottingham,
University Park, Nottingham NG7 2RD, England.
{brg,mpj}@cs.nott.ac.uk

Abstract

Records and variants provide flexible ways to construct datatypes, but the restrictions imposed by practical type systems can prevent them from being used in flexible ways. These limitations are often the result of concerns about efficiency or type inference, or of the difficulty in providing accurate types for key operations.

This paper describes a new type system that remedies these problems: it supports extensible records and variants, with a full complement of polymorphic operations on each; and it offers an effective type inference algorithm and a simple compilation method. It is a practical system that can be understood and implemented as a natural extension of languages like Standard ML and Haskell. In an area that has already received a great deal of attention from the research community, the type system described here is the first to combine all of these features in a practical framework.

One important aspect of this work is the emphasis that it places on the use of rows in the construction of record and variant types. As a result, with only small extensions of the core type system, we are able to introduce new, powerful operations on records using features such as row polymorphism and first-class labels.

1 Introduction

Products and sums play fundamental roles in the construction of datatypes: products describe grouping of data items, while sums express choices between alternatives. For example, we might represent a date as a product, with three integer components specifying the day, month, and year:

\[ Date = \text{Int} \times \text{Int} \times \text{Int}. \]

For a simple example of a sum, consider the type of input events to a window system, with one alternative indicating that a character has been entered on the keyboard, and another indicating a mouse click at a particular point on the screen:

\[ Event = \text{Char} + \text{Point}. \]

These definitions are adequate, but they are not particularly easy to work with in practice. For example, it is easy to confuse datatype components when they are accessed by their position within a product or sum, and programs written in this way can be hard to maintain.

To avoid these problems, many programming languages allow the components of products, and the alternatives of sums, to be identified using names drawn from some given set of labels. Labelled products are more commonly known as records or structs, while labelled sums are perhaps better known as variants or unions. For example, the Date and Event datatypes described above might be defined more attractively as:

\[ Date = \text{Rec} \{ \text{day:}\text{Int}, \text{month:}\text{Int}, \text{year:}\text{Int} \} \]
\[ Event = \text{Var} \{ \text{key:}\text{Char}, \text{mouse:}\text{Point} \}. \]

This notation captures a common feature in the construction of record and variant types, using rows of the form \(\{l_1:\tau_1, \ldots, l_n:\tau_n\}\) to describe mappings that associate a type \(\tau_i\) with each of the (distinct) labels \(l_i\). Record types are obtained by preceding rows with the symbol \text{Rec}. Variant types are constructed using \text{Var}. For example, if \(r = \{l_1:\tau_1, \ldots, l_n:\tau_n\}\) and \(e_1, \ldots, e_n\) have types \(\tau_1, \ldots, \tau_n\), respectively, then we can form a record \(\langle l_1 = e_1, \ldots, l_n = e_n \rangle\) of type \text{Rec} \(r\), or distinct variants, \(\langle l_1 = e_1 \rangle, \ldots, \langle l_n = e_n \rangle\), each of type \text{Var} \(r\). Thus, \((\text{day} = 25, \text{month} = 12, \text{year} = 1996)\) and \((\text{key} = 'a')\) represent values of type \text{Date} and \text{Event}, respectively.

1.1 Polymorphism and extensibility

Unfortunately, practical languages are often less flexible in the operations that they provide to manipulate records and variants. For example, many languages—from C to Standard ML (SML)—will only allow the programmer to select the \(l\) component, \(r.l\), from a record \(r\)
if the type of \( r \) is uniquely determined at compile-time\(^1\). These languages do not support polymorphic operations on records—such as a general selector function \((\_l)\) that will extract a value from any record that has an \( l \) field. A further weakness in many of these languages is that they provide no real support for extensibility; there are no general operators for adding a field, removing a field, renaming a field, or replacing a field (possibly with a value of a different type) in a record value.

There have been many previous attempts to design type systems for records and variants that support polymorphism and extensibility. Such work is important, not just in its own right, but also in its application to the study of object-oriented or database programming languages where these facilities seem particularly useful. We will summarize the key features of some of these earlier systems here before describing our own proposal.

Subtyping: Subtyping is one of the most widely used techniques for building type systems for records and variants \([3, 5, 4, 21]\). We can define a subtyping relation by specifying that a row \( r_1 \) is a subrow of \( r_2 \), written \( r_1 \leq r_2 \), if \( r_1 \) contains all the fields of \( r_2 \), and possibly more. The intuition here is that, for example, a record of type \( \text{Rec} \ r_1 \) could be used in any context where a value of type \( \text{Rec} \ r_2 \) is expected, and conversely, that a variant of type \( \text{Var} \ r_2 \) can be substituted in any context where a value of type \( \text{Var} \ r_1 \) is required. In particular, the selection operator \((\_l)\) has type:

\[
\forall \alpha.\forall r. \text{Rec} \{ l : \alpha \} \rightarrow \alpha.
\]

This operation is implemented, at least conceptually, by coercing from \( \text{Rec} \ r \) to a known type—the singleton \( \text{Rec} \{ l : \alpha \} \)—and then extracting the required field. One weakness of this approach is that information about the other fields in the record is lost, so it is harder to describe operations like record extension. For example, observing that bounded quantification is not by itself sufficient, Cardelli and Mitchell \([5]\) used an overriding operator on types to overcome this problem.

Row extension: Motivated by studies of object-oriented programming, Wand \([26]\) introduced the concept of row variables to allow incremental construction of record and variant types. For example, a record of type \( \text{Rec} \{ l : \tau \mid r \} \) has all of the fields of a record of type \( \text{Rec} \ r \), together with a field \( l \) of type \( \tau \). Wand did not discuss compilation, but his approach supports both polymorphism and extensibility. For example, the selection operator \((\_l)\) has type:

\[
\forall \alpha.\forall r. \text{Rec} \{ l : \alpha \mid r \} \rightarrow \alpha.
\]

However, the operations and types in Wand’s system are unchecked; for example, extending a row with an \( l \) field may either add a completely new field, or replace an existing field labelled with \( l \). As a result, some programs do not have principal types \([27]\).

Flags: Rémy has developed a flexible treatment for extensible records and variants in a natural extension of ML \([23]\). A key feature of his system is the use of flags to encode both positive and negative information—that is, to indicate which fields must be present, and which must be absent. Again, a concept of row variables is used to deal with other fields whose presence or absence is not significant in a particular situation. For example, the selection operator has type:

\[
\forall \alpha.\forall r. \text{Rec} \{ l : \text{pre}(\alpha) \mid r \} \rightarrow \alpha,
\]

where \( \text{pre}(\alpha) \) is a flag indicating the presence of a field of type \( \alpha \), and \( r \) is a row variable representing the rest of the record. Intuitively, records in Rémy’s system are represented by tuples with a slot for each possible label. The type system prevents access to undefined components, but does not lead directly to a simple and efficient implementation.

Predicates: Harper and Pierce \([8, 7]\) studied type systems for extensible records using predicates on types to capture information about presence or absence of fields, and to restrict attention to checked operations. For example, writing \( r_1 \# r_2 \) for the assertion that the rows \( r_1 \) and \( r_2 \) have disjoint sets of labels, the selection operator operator \((\_l)\) has type:

\[
\forall \alpha.\forall r. (r \# \{ l : \alpha \}) \Rightarrow \text{Rec} \ (r \parallel \{ l : \alpha \}) \rightarrow \alpha,
\]

where \( r_1 \parallel r_2 \) is the row obtained by merging \( r_1 \) and \( r_2 \), and is only defined if \( r_1 \# r_2 \). Harper and Pierce’s work does not deal with variants, type inference, or compilation, and does not provide an operational interpretation of predicates. However, their approach to record typing was one of the motivating examples in Jones’ work on qualified types \([11]\) where a general framework for type inference and compilation was developed, including a type system for records as a special case. One of the achievements of the present paper is to refine and extend that work to a practical system, avoiding problems such as the lack of most general unifiers in Jones’ full system—the result of including record restriction in the type language.

\(^1\)In implementations using boxed representations for values, only the set of labels in \( r \) is needed; the actual component types are not required.
Kinds: Ohori [18] described a type system that extends SML with polymorphic operations on both records and variants. Significantly, Ohori also presented a simple and effective compilation method: input programs are translated into a target language that adds extra parameters to specify field offsets. In fact, the end result is much the same as that suggested by Jones’ work on qualified types, even though the two approaches were developed independently. But Ohori’s work differs substantially from other systems in its use of a kind system; this allows variables ranging over record types to be annotated with a specification of the fields that they are expected to contain. For example, the selection operator operator \( \ell \) has type:
\[
\forall \alpha. \forall r \| l : \alpha \|. \text{Rec } r \rightarrow \alpha.
\]

The main limitation of Ohori’s type system is its lack of support for extensibility.

1.2 This paper

The type system described in this paper combines many of the ideas that have been used in previous work into a practical type system for implicitly typed languages like SML [16] and Haskell [20]. In particular, it supports polymorphism and extensibility, records and variants, type inference, and compilation. The type system is an application of qualified types, extended to deal with a general concept of rows. Positive information about the fields in a given row is captured in the type language using row extension, while negative information is reflected by the use of predicates.

The most obvious benefit of this approach is that we can adapt results and properties from the general framework of qualified types—such as the type inference algorithm and the compilation method—without having to go back to first principles. The result is a considerable simplification of both the overall presentation and of specific proofs. Another important advantage of this approach is that it guarantees compatibility with other applications of qualified types. For example, our type system can be used—and indeed, has already been used in our prototype implementation—in conjunction with the type class mechanisms of Haskell.

The remaining sections of this paper are as follows. Section 2 provides a general overview of our new type system, with a more detailed formal presentation in Section 3. This is followed by discussions of type inference in Section 4 and compilation in Section 5. With some small extensions to the core system, Section 6 shows how our framework can be used to support some new, powerful operations on records and variants using row polymorphism and labels as first-class values. Finally, Section 7 concludes with pointers to further work.

2 Overview

Both record and variant types are defined in terms of rows, and these are constructed by extension, starting from the empty row, \( \{ \} \). It is convenient to introduce abbreviations for rows obtained in this way:
\[
\{ l_1 : \tau_1, \ldots, l_n : \tau_n | r \} = \{ l_1 : \tau_1 | \ldots | l_n : \tau_n | r \} \ldots
\]
\[
\{ l_1 : \tau_1, \ldots, l_n : \tau_n \} = \{ l_1 : \tau_1, \ldots, l_n : \tau_n | \{ \} \}.
\]

Note, however, that we treat records, and hence record or variant types, as equals if they include the same fields, regardless of the order in which those fields are listed.

2.1 Basic operations

Intuitively, a record of type \( \text{Rec } \{ l : \alpha | r \} \) is like a pair whose first component is a value of type \( \alpha \), and whose second component is a record of type \( \text{Rec } r \). This motivates our choice of basic operations on records, which correspond directly to the two projections and the pairing constructor for products in category theory or logic. There is, however, one complication: we do not allow repeated uses of any label within a particular row, so the expression \( \{ l : \alpha | r \} \) is only valid if \( l \) does not appear in \( r \). This is reflected by prefixing each of the types below with a predicate \( (r \setminus l) \), pronounced “\( r \) lacks \( l \)”, that restricts instantiation of \( r \) to rows without an \( l \) field:

- Selection: to extract the value of a field \( l \):
  \[
  (\ell \cdot l) :: (r \setminus l) \Rightarrow \text{Rec } \{ l : \alpha | r \} \rightarrow \alpha.
  \]

- Restriction: to remove a field labelled \( l \):
  \[
  (\ell \cdot l) :: (r \setminus l) \Rightarrow \text{Rec } \{ l : \alpha | r \} \rightarrow \text{Rec } r.
  \]

- Extension: to add a field \( l \) to an existing record:
  \[
  (l \cdot \ell) :: (r \setminus l) \Rightarrow \alpha \rightarrow \text{Rec } r \rightarrow \text{Rec } \{ l : \alpha | r \}.
  \]

We can use these basic operations to implement a number of additional operators, including:

- Update/replace: to update the value in a particular field, possibly with a value of a different type:
  \[
  (l := \ell \cdot l) :: (r \setminus l) \Rightarrow \alpha \rightarrow \text{Rec } \{ l : \beta | r \} \rightarrow \text{Rec } \{ l : \alpha | r \}
  \]
  \[
  (l := x | r) = (l = x | r - l)
  \]

\(^2\)To simplify the notation, we assume implicit universal quantification over free type variables. For example, the full type of the selection operator in \( \forall r. \forall \alpha. (r \setminus l) \Rightarrow \text{Rec } \{ l : \alpha | r \} \rightarrow \alpha. \)
Renaming: to change the label attached to a particular field:

\[ [l \leftarrow m] \quad :: \quad (r \setminus l, r \setminus m) \Rightarrow \text{Rec} \{ l : \alpha | r \} \rightarrow \text{Rec} \{ m : \alpha | r \} \]

\[ r[l \leftarrow m] = (m = r.l | r - l) \]

The empty record, (), plays an important role as the only proper value of type \( \text{Rec} \{ \} \). Again, it is convenient to introduce abbreviations for the construction of record values by repeated extension:

\[ (l_1 = e_1, \ldots, l_n = e_n | r) = (l_1 = e_1 | \ldots (l_n = e_n | r) \ldots) \]

\[ (l_1 = e_1, \ldots, l_n = e_n) = (l_1 = e_1, \ldots, l_n = e_n | ()) \]

We can specify the basic operations on variants in a similar way. Again, they correspond closely to the standard operations on sums in category theory or logic:

- **Injection:** to tag a value with the label \( l \):
  \[
  \langle l = \_ \rangle :: (r \setminus l) \Rightarrow \alpha \rightarrow \text{Var} \{ l : \alpha | r \}.
  \]

- **Embedding:** to embed a value in a variant type that also allows for a new label, \( l \):
  \[
  \langle l \_ \_ \_ \rangle :: (r \setminus l) \Rightarrow \text{Var} r \rightarrow \text{Var} \{ l : \alpha | r \}.
  \]

- **Decomposition:** to act on the value in a variant, according to whether or not it is labelled with \( l \):
  \[
  (l \in \_?) :: (r \setminus l) \Rightarrow \text{Var} \{ l : \alpha | r \} \rightarrow (\alpha \rightarrow \beta) \rightarrow (\text{Var} r \rightarrow \beta) \rightarrow \beta.
  \]

The empty variant, \( \_ \), is the only value of type \( \text{Var} \{ \} \). More sophisticated language constructs, for example, pattern matching facilities, or record update, are easily described using the operations listed here. In addition, we expect that practical implementations will use, but not display predicates implied by the context in which they appear. For example, all of the types above include a row \( \{ l : \alpha | r \} \) that is only valid if \( r \setminus l \); so displaying this predicate is, in some sense, redundant. However, as we will see in the next section, this predicate plays a central role in the implementation of the basic operations.

### 2.2 Implementation details

Our next task is to explain how the data structures and operations described above can be implemented. We will focus on the treatment of records and, in particular, the implementation of selection, \( (\_\_l) \), which is probably the most frequently used basic operation. A naive approach would be to represent a record by an association list, pairing labels with values. This would allow simple implementations for each of the basic operations, with the type system providing a guarantee that the search for any given labelled field would not fail. A major disadvantage is that it does not allow constant time access to record components.

To avoid these problems, we will assume instead that a record value is represented by a contiguous block of memory that contains a value for each individual field. To select a particular component \( r.l \) from a record \( r \), we need to know the offset of the \( l \) field in the block of memory representing \( r \). Languages without polymorphic selection will usually only allow an expression of the form \( r.l \) if the offset value, and hence the structure or even the full type of \( r \), is known at compile-time.

However, it is not actually necessary to know the position of every field at compile-time; instead, we can treat unknown offsets as implicit parameters whose values will be supplied at run-time when the full types of the records concerned are known. This is essentially the compilation method that was used by Ohori [18], and also suggested, independently, by Jones [11]. If we forget about typing issues for a moment and assume that records are implemented as arrays or tuples, then the \( (\_l) \) operator can be implemented by a function \( \lambda i.\lambda r.r[i] \), using the extra parameter \( i \) to supply the offset of \( l \) in \( r \). For example, the expression:

\[
(day = 25, month = 12, year = 1996),
\]

\[ (\lambda i.\lambda l.r[i]) 0 (25, 12, 1996) \]

which evaluates to 25, as expected. Of course, there are run-time overheads in calculating and passing offset values as extra parameters. However, an attractive feature of our system is that these costs are only incurred when the extra flexibility of polymorphic selection is required.

Each predicate \( (r \setminus l) \) in the type of a function signals the need for an extra run-time parameter to specify the offset at which a field labelled \( l \) would be inserted into a record of type \( \text{Rec} r \). Obviously, the same offset can also be used to locate or remove the \( l \) field from a record of type \( \text{Rec} \{ l : \alpha | r \} \), or treated as ordinal numbers to access and tag values in a variant. So, this one extra piece of information is all that we need to implement the basic operations.

Operations like record extension and restriction will, in general, be implemented by copying. Optimizations can be used to combine multiple extensions or restrictions of records, avoiding unnecessary allocation and initialization of intermediate values. For example, a compiler can generate code that will allocate and initialize the storage for a record \( (x = 1, y = 2, z = 3) \) in
a single step, rather than a sequence of three individual allocations and extensions as a naive interpretation might suggest.

The typechecker gathers and simplifies the predicates generated by each use of an operator on records or variants. For example, if \( today \) is a value of type \( Date \), then an expression like \( today.month \) will generate a single constraint, \( \{ day : Int, year : Int \} \_\_month \). Predicates like this, involving rows whose structure is known at compile-time, are easily discharged by calculating the appropriate offset value. Obviously, a compiler can use this information to produce efficient code by inlining and specializing the selector function, \( (_\_month) \).

Predicates that are not discharged within a section of code will, instead, be reflected in the type assigned to it. For example, there is nothing in the following definition to indicate the full type of \( d \):

\[
newYear d = d.day = 1 \land d.month = 1,
\]

so the inferred type will be:

\[
(r\_day, r\_month) \Rightarrow
\text{Rec } \{ \text{day : Int, month : Int} | r \} \rightarrow \text{Bool}.
\]

We would not expect this definition to have been accepted at all by a compiler for SML which requires the set of labels in a record to be uniquely determined by 'program context'. But the meaning of this phrase is defined only loosely by an informal note in the definition of SML [16]. Now, with the ideas used in this paper, there is a way to make this precise: a definition is only acceptable in SML if the inferred type does not contain any predicates. For programs written with these restrictions, a language based on our type system should offer the same levels of performance as SML.

It is possible that our more general treatment of record operations could result in compiled programs that are littered with unwanted offset parameters; experience with our prototype implementations will help to substantiate or dismiss such concerns. In any case, there are simple steps that can be taken to avoid such problems. For example, a compiler might reject any definition with an inferred type containing predicates, unless an explicit type signature has been given to signal the programmer’s acceptance. This is closely related to the *monomorphism restriction* in Haskell [20] and to proposals for a *value restriction* in SML [29, 13].

## 3 Formal presentation

This section provides a formal presentation of our type system, based on two particular ingredients:

- The theory of qualified types [11], which provides a general framework for describing restricted polymorphism and overloading. In the current application, we use constraints to capture assumptions about the occurrences of labels within rows.
- A higher-order version of the Hindley-Milner type system [9, 15, 6], originally introduced in the study of *constructor classes* [12]. Amongst other things, this provides a simple way to introduce the new constructs for rows, records, and variants without the need for special, ad-hoc syntax.

We split the presentation into sections: kinds (Section 3.1), types and constructors (Section 3.2), predicates (Section 3.3), and typing rules (Section 3.4).

### 3.1 Kinds

One of the most important aspects of the work described here is the use of a *kind* system to distinguish between different kinds of type constructor. Formally, the set of kinds is specified by the following grammar:

\[
\kappa ::= * \quad \text{the kind of all types} \\
| \text{row} \quad \text{the kind of all rows} \\
| \kappa_1 \rightarrow \kappa_2 \quad \text{function kinds}
\]

Intuitively, the kind \( \kappa_1 \rightarrow \kappa_2 \) represents constructors that take something of kind \( \kappa_1 \) and return something of kind \( \kappa_2 \). The *row* kind is new to the system presented here and was not part of the type system used in the development of constructor classes.

### 3.2 Types and constructors

For each kind \( \kappa \), we have a collection of constructors \( C^\kappa \) (including variables \( \alpha^\kappa \)) of kind \( \kappa \):

\[
C^\kappa ::= \chi^\kappa \quad \text{constants} \\
| \alpha^\kappa \quad \text{variables} \\
| C^{\kappa_1 \rightarrow \kappa_2} \quad \text{applications}
\]

\[
\tau ::= C^* \quad \text{types}
\]

The usual collection of types, represented here by the symbol \( \tau \), is just the constructors of kind \(*\). For the purposes of this paper, we assume that the set of constant constructors includes at least the following, writing \( \chi: \kappa \rightarrow \kappa \) to indicate the kind \( \kappa \) associated with each constant \( \chi \):

\[
\rightarrow :: * \rightarrow * \rightarrow * \quad \text{function space} \\
\{ \}
:: \text{row} \quad \text{empty row} \\
\{ l:: \}
:: * \rightarrow \text{row} \rightarrow \text{row} \quad \text{extension, for each } l \\
\text{Rec}
:: \text{row} \rightarrow * \quad \text{record construction} \\
\text{Var}
:: \text{row} \rightarrow * \quad \text{variant construction}
\]

For example:

- The result of applying the function space constructor \( \rightarrow \) to two types \( \tau \) and \( \tau' \) is the type of functions
from \( \tau \) to \( \tau' \), and is written as \( \tau \rightarrow \tau' \) in more conventional notation.

- The result of applying the \( \text{Rec} \) constant to the empty row \( \emptyset \) of kind \( \text{row} \) is the type \( \text{Rec} \emptyset \) of kind \( * \).

- The result of applying an extension constructor \( \{ l : \tau \} \) to a type \( \tau \) and a row \( r \) is a row, usually written as \( \{ l : \tau | r \} \), obtained by extending \( r \) with a field labelled \( l \) of type \( \tau \). Note that we include an extension constructor for each different label \( l \). To avoid problems later, we will also need to prohibit partial application of extension constructors.

The kind system is used to ensure that type expressions are well-formed. While it is sometimes convenient to annotate individual constructors with their kinds, there is no need in practice for a programmer to supply these annotations. Instead, they can be calculated automatically using a simple kind inference process [12].

We consider two rows to be equivalent if they include the same fields, regardless of the order in which they are listed. This is described formally by the equation:

\[
\langle l : \tau, l' : \tau' | r \rangle = \langle l' : \tau', l : \tau | r \rangle,
\]

and extends in the obvious way to an equality on arbitrary constructors.

For the purposes of later sections, we define a membership relation, \( (l : \tau) \in r \), to describe when a particular field \( (l : \tau) \) appears in a row \( r \):

\[
(l : \tau) \in \{ l : \tau | r \} \if\quad (l : \tau) \notin \{ l' : \tau' | r \} \if\quad (l \neq l')
\]

and a restriction operation, \( r - l \), that returns the row obtained from \( r \) by deleting the field labelled \( l \):

\[
\{ l : \tau | r \} - l = r
\]
\[
\{ l' : \tau | r \} - l = \{ l' : \tau | r - l \}.
\]

It is easy to prove that these operations are well-defined with respect to the equality on constructors, and to confirm intuitions about their interpretation by showing that, if \( (l : \tau) \in r \), then \( r = \{ l : \tau | r - l \} \).

### 3.3 Predicates

The syntax for rows allows examples like \( \{ l : \tau, l : \tau' \} \) where a single label appears in more than one field. While this might be useful in some applications, it is not appropriate for work with records or variants where we allow at most one field with any given label. Clearly, some additional mechanisms are needed to enable us to specify that a type of the form \( \text{Rec} \{ l : \tau | r \} \), for example, is only valid if the row \( r \) does not contain a field labelled with \( l \).

One way to achieve this is to use a more sophisticated kind system, with sets of labels, \( L \), as kinds instead of the single row kind. For example, rows with field labels \( l_1, \ldots, l_n \) can be represented by the kind \( L = \{ l_1, \ldots, l_n \} \); this is essentially the approach adopted by Ohori [18]. Unfortunately, this becomes much more complicated if we try to extend it to deal with extensible rows. In particular, we would need to assign whole families of kinds, indexed by label sets, \( L \), to some of the constructor constants introduced in the previous section:

\[
\begin{align*}
\{ l : \tau \} &::= * \rightarrow L \rightarrow (L \cup \{ l \}) \quad l \notin L \\
\text{Rec, \ Var} &::= L \rightarrow * 
\end{align*}
\]

The alternative that we adopt in this paper is based on the theory of qualified types [11], using predicates to capture any side conditions that are required to ensure that a given type expression is valid. In fact, only a single form of predicate is needed for this purpose:

\[
\pi ::= C^\text{row} \setminus l \ \text{predicates}
\]

Intuitively, the predicate \( r \setminus l \) can be read as an assertion that the row \( r \) does not contain an \( l \) field. More precisely, we explain the meaning of predicates using the entailment relation defined in Figure 1. A deriv-
The term language is just core-ML, an implicitly
typed λ-calculus, extended with constants and a let
construct, and described by the following grammar:
\[ E ::= x \mid c \mid EF \mid \lambda x.E \mid \text{let } x = E \text{ in } F \]
We assume that the set of constants \( c \) includes the oper-
ations and values required for manipulating records
and variants as described in Section 2, and that each
constant \( c \) is assigned a closed type scheme, \( \sigma_c \).

The typing rules are presented in Figure 2. A judge-
ment of the form \( P \vdash A : \sigma \) represents an assertion
that, if the predicates in \( P \) hold, then the term \( E \) has
type \( \sigma \), using assumptions in \( A \) to provide types for free
variables. These are just the standard rules for qualified

\[
\begin{align*}
(const) & \quad P | A \vdash c : \sigma_c \\
(var) & \quad (x : \sigma) \in A \quad P | A \vdash x : \sigma \\
(\rightarrow E) & \quad P | A \vdash E : \tau' \rightarrow \tau \quad P | A \vdash F : \tau' \quad P | A \vdash EF : \tau \\
(\rightarrow I) & \quad P | A \vdash \lambda x.E : \tau' \rightarrow \tau \\
(\Rightarrow E) & \quad P | A \vdash E : \pi \Rightarrow \rho \quad P \parallel \pi \\
(\Rightarrow I) & \quad P | A \vdash E : \rho \\
(\forall E) & \quad P | A \vdash E : \forall \alpha.\sigma \\
(\forall I) & \quad P | A \vdash E : \forall \alpha.\sigma \\
(let) & \quad P | A \vdash E : \sigma \quad Q | A, x : \sigma \vdash F : \tau \\
& \quad P, Q | A \vdash (\text{let } x = E \text{ in } F) : \tau
\end{align*}
\]

Figure 2: Typing rules.

4 Type Inference

This section provides a formal presentation of a type in-
ference algorithm for our system. The most important
feature is the introduction of inserters in Section 4.1 to
account for non-trivial equalities between row expres-
sions during unification.

4.1 Unification and insertion

Unification is a standard tool in type inference, and is
used, for example, to ensure that the formal and actual
parameters of a function have the same type. Formally,
a substitution\(^3\) \( S \) is a unifier of constructors \( C, C' \in C^\kappa \)
if \( SC = SC' \), and is a most general unifier of \( C \) and \( C' \)
if every unifier of these two constructors can be written
in the form \( RS \), for some substitution \( R \).

The rules in Figure 3 provide an algorithm for cal-

\[
\begin{align*}
(id) & \quad C^{id}C \\
(bind) & \quad \alpha^{[C/\alpha]}C \quad \alpha \notin TV(C) \\
(apply) & \quad C^{U}C' \quad UD^{U}UD' \\
& \quad \frac{CD^{U}U}{CD \sim C'D'} \\
(row) & \quad (l : \tau)\in \tau' \quad I_{U}^{\tau}(Ir' - l) \\
& \quad \frac{\{l : \tau | r\}^{U} \sim r'}
\end{align*}
\]

Figure 3: Kind-preserving unification.

\(^3\)For the purposes of this paper, we restrict our attention to kind-

preserving substitutions; that is, to substitutions that map variables
\( \alpha \in C^\kappa \) to constructors of the corresponding kind, \( \kappa \).
choose the substitution \( S \) so that it will ‘insert’ the missing fields into the two rows \( r' \) and \( r \), respectively. In this particular case, assuming that \( r, r' \not\in TV(\tau, \tau') \), then we can choose:

\[
S = \frac{\{l' : \tau'|r''\}}{r}, \frac{\{l : \tau|r''\}}{r'}
\]

where \( r'' \) is a new type variable.

More generally, we will say that a substitution \( S \) is an inserter of \( (l : \tau) \) into \( r \in C_{row} \) if \( (l : S\tau) \in Sr \). \( S \) is a most general inserter of \( (l : \tau) \) into \( r \) if every such inserter can be written in the form \( RS \), for some substitution \( R \). The rules in Figure 4 define an algorithm for calculating inserters, writing \( (l : \tau) \) for the assertion that \( I \) is an inserter of \( (l : \tau) \) into \( r \in C_{row} \). Note that

\[
\begin{align*}
(\text{inVar}) & \quad (l : \tau) \notin r, \ l \not\in l' \quad \frac{(l : \tau) \in r}{(l : \tau) \in l' : \tau | r} \\
(\text{inTail}) & \quad \frac{(l : \tau) \in r}{(l : \tau) \in l : \tau | r} \\
(\text{inHead}) & \quad \frac{\tau \sim \tau'}{(l : \tau) \in \{l : \tau | r\} \leadsto (l : \tau) \in \{l' : \tau | r\}}
\end{align*}
\]

Figure 4: Kind-preserving insertion.

the last rule here, \((\text{inHead})\), makes use of the unification algorithm in Figure 3, so the two algorithms are mutually recursive. The important properties of the two algorithms—both soundness and completeness—are captured in the following result:

**Theorem 1** The unification (insertion) algorithm defined by the rules in Figure 3 (Figure 4) calculates most-general unifiers (inserters) whenever they exist. The algorithm fails precisely when no unifier (inserter) exists.

### 4.2 A type inference algorithm

Given the unification algorithm described in the previous section, we can use the type inference algorithm for qualified types [11] as a type inference algorithm for the type system presented in this paper. For completeness, we include a definition of the algorithm using the rules in Figure 5. Following Rémy [23], these rules can be understood as an attribute grammar; in each typing judgement \( P \mid TA \vdash^W E : \tau \), the type assignment \( A \) and the term \( E \) are inherited attributes, while the predicate assignment \( P \), type \( \tau \), and substitution \( T \) are synthesized. The \((\text{let})^W\) rule uses an auxiliary function to calculate the generalization of a qualified type \( \rho \) with respect to a type assignment \( A \). This is defined as:

\[ Gen(A, \rho) = \forall \alpha_i. \rho, \text{ where } \{\alpha_i\} = TV(\rho) \setminus TV(A). \]

The type inference algorithm is both sound and complete with respect to the original typing rules.

**Theorem 2** The algorithm described by the rules in Figure 5 can be used to calculate a principal type for a given term \( E \) under assumptions \( A \). The algorithm fails precisely when there is no typing for \( E \) under \( A \).

### 5 Compilation

Previously, we have described informally how programs involving operations on records and variants can be compiled and executed using a language that adds extra parameters to supply appropriate offsets. This section shows how this process can be formalized, including the calculation of offset values.

#### 5.1 Compilation by translation

In the general treatment of qualified types [11], programs are compiled by translating them into a language that adds extra parameters to supply evidence for predicates appearing in the types of the values concerned. The whole process can be described by extending the typing rules to use judgements of the form:

\[ P \mid A \vdash E \leadsto E : \sigma \]
which include both the original source term $E$ and a possible translation, $E'$. A further change here is the switch from predicate sets to *predicate assignments*; the symbol $P$ used above represents a set of pairs $(v : \pi)$ in which no variable $v$ appears twice. Each variable $v$ corresponds to an extra parameter that will be added during compilation; $v$ can be used whenever evidence for the corresponding predicate $\pi$ is required in $E'$.

In the current setting, predicates are expressions of the form $(r \backslash l)$ whose evidence is the offset in $r$ at which a field labelled $l$ would be inserted. The calculation of evidence is described by the rules in Figure 6, which are direct extensions of the earlier rules for predicate entailment that were given in Figure 1. Intuitively, a $l$ is a direct extension of the earlier rules for predicate sets that include both the original source term $E$ and a label $\pi$ which should be inserted in a row $\{l' : \pi \backslash r\}$.

This tells us that we need to supply suitable evidence $e$ in the translation of any program whose type is qualified by a predicate $\pi$. The second rule is analogous to function abstraction, and allows us to move constraints from the predicate assignment $P$ into the inferred type:

$$P \cup \{v : \pi\} \vdash E \sim E' : \rho$$

$$P \vdash A \vdash E \sim \lambda v. E' : \pi \Rightarrow \rho$$

These two rules are direct extensions of $(\Rightarrow E)$ and $(\Rightarrow I)$ in Figure 2, and combined with simple extensions of the other rules there, we can construct a translation for any term in the source calculus.

### 6 Extensions

The type system that we have described in the previous sections offers a flexible, but fairly conventional set of operations on records and variants. In this section we consider two extensions to the original system. Section 6.1 describes a generalization of the record and variant operations to include, amongst others, first-class extensible case statements. Section 6.2 shows how the system can be modified to allow labels to be treated as first-class values.

#### 6.1 Row polymorphism

Working with a general notion of rows has provided us with an elegant way to deal with the common structure in record and variant types. However, we have not seen any compelling examples in the previous sections where it was essential to consider rows separately from records and variants; we could have just defined completely independent sets of record and variant types.

However, there are some applications in which the ability to separate rows from records and variants offers significant benefits. To illustrate this, consider again the basic operations that were discussed in Section 2. For example, if we generalize the rules in category theory or logic for decomposing a sum to deal with $n$-ary sums, then we obtain the following rule:

$$A_1 \rightarrow C \ldots\ldots A_n \rightarrow C$$

In terms of records and variants, this rule provides a method for decomposing a variant—represented by the sum $A_1 + \ldots + A_n$ in the conclusion—using a record of functions—represented by the hypotheses $A_i \rightarrow C$. This suggests a general operation for variant elimination:

$$plusElim :: \forall \alpha. \forall r. Rec\ (to\ \alpha\ r) \rightarrow Var\ r \rightarrow \alpha.$$
The \( \text{to } \tau \ r \) construct used here is defined as follows:

\[
\text{to } \tau \ \{ \} = \{ \}
\]

\[
\text{to } \tau \ \{ l : \tau' | r \} = \{ l : \tau' \to \tau \text{ to } \tau \ r \}.
\]

This behaves like a particular kind of *map* operation on rows, replacing each component type \( \tau' \) in \( r \) with a type of the form \( \tau' \to \tau \).

For an example of where such an operation may prove useful, consider the type of integer lists that can be obtained as a fixpoint of the following functor [14]:

\[
data L l = L (\text{Var} \ \{ \text{nil} : \text{Rec} \ \{ \},\
\text{cons} : \text{Rec} \ \{ tl : l, \text{hd} : \text{Int} \} \} )
\]

The sum of a list of integers can be calculated using a general catamorphism:

\[
cata (\lambda (L v).\text{plusElim} (\text{nil} = \lambda ().0,
\text{cons} = \lambda r. r. \text{hd} + r. tl) v).
\]

From this example, it is clear that \text{plusElim} is an alternative to the \text{case} construct of languages like Haskell and SML. However, unlike these languages, it is not a builtin part of the syntax; instead, it allows us to treat \text{case} constructs as first-class, extensible values.

In a similar way, we can adapt the other rules for constructing sums, and for decomposing or constructing products, to functions involving records and variants. The full set of operations are specified by the following type signatures:

\[
\begin{align*}
\text{plusIntro} &:: \forall \alpha. \forall r. \text{Rec} \ (\text{to } \alpha \ r) \to \text{Var} \ r \to \alpha \\
\text{prodIntro} &:: \forall \alpha. \forall r. \text{Var} \ (\text{from } \alpha \ r) \to \alpha \to \text{Var} \ r
\end{align*}
\]

Given our earlier representations for records and variants, it is easy to implement each of these operations as builtin primitives.

One technical difficulty that we face with this approach is in extending the treatment of unification to deal with uses of the \text{from} and \text{to} constructs. This turns out to be straightforward, except for potential complications caused by the presence of empty rows. For example, in unifying two rows \( \text{to } t \ r \) and \( \text{to } t' \ r' \), we cannot simply unify \( t \) with \( t' \); if \( r \), and hence \( r' \), is empty, then there is no direct relationship between \( t \) and \( t' \). More precisely, to obtain most general unifiers for \text{from} and \text{to} constructs, we need to restrict ourselves to work with non-empty rows. There are several alternatives to consider here. For example, the obvious approach would be to banish the empty row from our original type system. A more satisfactory solution might be to adopt a kind system that distinguishes between empty and non-empty rows. We leave further investigation of this topic to future work.

### 6.2 First-class labels

In previous sections of the paper, we have considered the labels used to refer to fields as part of the basic syntax of our language. As a result, we had to describe selection from a record using a family of functions:

\[
(\_\_l) :: (r \\backslash l) \Rightarrow \text{Rec} \ \{ l : \alpha | r \} \to \alpha
\]

with one function for each choice of label \( l \). A more attractive approach might be to allow primitive operations on records and variants to be parameterized over labels. We can extend the type system described in previous sections to accomplish this, treating selection, for example, as a single function of type:

\[
(\_\_l) :: (r \\backslash l) \Rightarrow \text{Rec} \ \{ l : \alpha | r \} \to \text{Label} \ l \to \alpha
\]

This requires an extension of the kind system in Section 3.1 with a new kind \text{lab}, and also a new type constant \text{Label} of kind \text{lab} \to \ast. Intuitively, each type of the form \text{Label} \ l contains a unique label value. The \( l \) parameter is important because it establishes a connection between types and label values; a nullary \text{Label} type would not have provided any way for us to express the necessary typing constraints. We can also dispense with the family of extension constructors defined in Section 3.2, replacing them with a single constructor constant:

\[
\{ \_\_\} :: \text{lab} \to \ast \to \text{row} \to \text{row}.
\]

Finally, we need to generalize the lacks predicate from Section 3.3 to a two place relation \( r' \\backslash t \) which takes both a row \( r \) and a label \( l \) of kind \text{lab}. This can be defined in much the same way as before, and has the same interpretation as an offset value in the underlying implementation.

We can generalize the other basic operations on records and variants in a similar way. For example, the expression \( \lambda x. \lambda y. (y = 2, x = 3) \), which involves two uses of extension, will be assigned the type:

\[
\{ \_\_\} \ y = \Rightarrow \\
\text{Label} \ x \to \text{Label} \ y \to \text{Rec} \ \{ x : \text{Int}, y : \text{Int} \}.
\]

To our knowledge, this is the first work—in either implicitly or explicitly typed record and variant calculi—to consider a type system in which labels can be treated as first-class values. This increases the expressiveness of our system quite dramatically, and we already have some interesting applications for these new features.
However, it remains to see how well this will work in practice, and what its implications for language design might be; we leave these topics to future work.

7 Conclusion

We have described a flexible, polymorphic type system for extensible records and variants with an effective type inference algorithm and compilation method. Prototype implementations have been written in SML and Haskell, and a implementation of records has been added to Hugs, an implementation of Haskell 1.3. Our experience to date shows that these implementations work well in practice. There are a number of areas for further work:

Object systems. Another potential application for rows would be in a type system for object-oriented programming. For example, a constructor $\text{Obj}$ of kind $\text{row} \rightarrow *$ might be used to describe objects with a given row of methods [17, 22, 1, 10, 21, 2].

More sophisticated basic operations. Our type system does not support record append [28, 24] or the database join that was one of the original motivations for Ohori’s work [19]. One approach would be to adopt the ideas of Harper and Pierce [7], choosing an appropriate form of evidence for the $(r_1 \# r_2)$ predicates described in Section 1.1.

A new approach to datatypes. Languages like SML and Haskell already provide facilities for defining new datatypes and these overlap to some extent with the mechanisms described in this paper. Unifying and combining these different approaches would obviously be useful, with a long term goal of developing a general framework for extensible datatypes.

Acknowledgements

This work was supported in part by an EPSRC studentship 9530 6293. We would also like to thank our colleagues, Colin Taylor and Graham Hutton, in the functional programming group at Nottingham, for the valuable contributions that they have made to the work described in this paper.

References


