First-class labels for extensible rows

Technical report: UU-CS-2004-051

Daan Leijen

Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
daan@cs.uu.nl

Abstract

This paper describes a type system for extensible records and variants with first-class labels; labels are polymorphic and can be passed as arguments. This increases the expressiveness of conventional record calculi significantly, and we show how we can encode intersection types, closed-world overloading, type case, label selective calculi, and first-class messages. We formally motivate the need for row equality predicates to express type constraints in the presence of polymorphic labels. This naturally leads to an orthogonal treatment of unrestricted row polymorphism that can be used to express first-class patterns.

Based on the theory of qualified types, we present an effective type inference algorithm and efficient compilation method. The type inference algorithm, including the discussed extensions, is fully implemented in the experimental Morrow compiler.

1 Introduction

Records and variants provide a convenient way to construct data types. Furthermore, record calculi can be used as a foundation for features as objects and module systems. Over the years, many aspects of record calculi have been studied, including polymorphic extension, record concatenation, type directed compilation, and even label-less calculi.

The holy grail [6] of polymorphic extensible record calculi are first-class labels, where labels are polymorphic and can be passed as arguments. First-class labels increase the expressiveness of conventional record calculi significantly, and we will show how we can encode many interesting programming idioms, including intersection types, closed-world overloading, type case, label selective calculi, and first-class messages.

One of the earliest works on first-class labels was done by Gaster and Jones [5, 4]. They describe a polymorphic type system for extensible records and variants based on the theory of qualified types. Unfortunately, as we show later in this article, type inference for their proposed extension is incomplete. Sulzmann acknowledges this problem and describes a record calculus with first-class labels as an instance of HM(X) [27, 28, 29], but this calculus is not extensible and lacks a general compilation method. Shields and Meijer introduce an intriguing label-less calculus, λ^{tir}, where records with first-class labels can be encoded with *opaque* types [25, 24]. However, due to general type equality constraints, λ^{tir} is a very complicated system. We wanted to explore an alternative point in the design space with a simpler calculus where labels are explicit; indeed, we can express many of the motivating examples of λ^{tir} without having to work from first principles.

We base our calculus directly on the original record system of Gaster and Jones, with two important technical differences: we introduce *row equality* predicates to express required type constraints in the presence of polymorphic labels, and we move the language of rows into the predicate language to ensure completeness of type inference. Furthermore, our system is the first to naturally describe row polymorphic operations, and it completely avoids the complex unification problems, and resulting restriction to non-empty rows, as described by Gaster [4]. Being able to use unrestricted row polymorphism, we can express first-class extensible patterns and views [30].

We have fully implemented our type inference algorithm, including discussed extensions like row polymorphic operations, in the experimental Morrow compiler [12]. Indeed, Morrow can infer all the types of the examples in this paper. However, at the time of writing, the code generator for Morrow is not yet complete.

In our calculus, row terms are no longer part of the type language but are only present in the predicate language, to ensure completeness of type inference with polymorphic labels. By not folding labels back into the language of types, we also reduce the complexity of improvement with respect to λ^{tir}. Unfortunately, the

improvement algorithm, and thus the test for satisfiability, ambiguity, and entailment, is still exponential. However, just like normal type inference, we believe that the worst case is unlikely to occur in practical programs. This intuition is largely confirmed by practical experience with the Morrow compiler.

In Section 2 we give an overview of conventional extensible records and variants à la Gaster and Jones [5]. In Section 3 we extend this calculus with first-class polymorphic labels, and we discuss many interesting examples in the following section. Section 5 discusses first-class patterns. Section 6 and 7 formally define our calculus, and give the typing rules and an inference algorithm. Section 8 describes simplification and improvement, and discusses complexity issues. We finish with a discussion of related work and the conclusion.

2 Records and variants

There are two dual concepts to describe the structure of data types: products group data items together, while sums describe a choice between alternatives. Here are two examples of both operations:

$$\textbf{type}\ \ \mathit{Point} = \mathit{Int} \times \mathit{Int}$$
$$\textbf{type}\ \ \mathit{Event} = \mathit{Char} + \mathit{Point}$$

In most programming languages, we can name the components of a product and sum with a *label*. A labeled product is called a *record* (or *struct*), while a labeled sum is called a *variant* (or *union*, or *data type*). We describe records with curly braces $\{\}$, and variants with angled brackets $\langle\rangle$:

$$\textbf{type}\ \ \mathit{Point} = \{x :: \mathit{Int}, y :: \mathit{Int}\}$$
$$\textbf{type}\ \ \mathit{Event} = \langle \mathit{key} :: \mathit{Char}, \mathit{mouse} :: \mathit{Point}\rangle$$

We can see that both records and variants are described by a sequence of labeled types, which we call a *row*. In this article, we enclose rows in banana brackets $(\!|\ |\!)$. For convenience, we leave these out when the row is directly enclosed by a record or variant brackets. For example, the unabbreviated type of a *Point* is $\{(\!| x :: \mathit{Int}, y :: \mathit{Int} |\!)\}$.

2.1 Extensible rows

Following Gaster and Jones [5], we consider an *extensible* row calculus where a row is either empty or an extension of a row. The empty row is written as $(\!|\ |\!)$ and an extension of a row r with a label l and type τ is written as $(\!| l :: \tau \mid r |\!)$. Here is an example of a row that can be used to describe coordinates:

$$(\!| x :: \mathit{Int} \mid (\!| y :: \mathit{Int} \mid (\!|\ |\!) |\!) |\!)$$

To reduce the number of brackets, we use the following abbreviations:

$$\begin{aligned}(\!| l_1 :: \tau_1, ..., l_n :: \tau_n |\!) &\equiv (\!| l_1 :: \tau_1 \mid ... \mid (\!| l_n :: \tau_n \mid (\!|\ |\!) ... |\!)\\ (\!| l_1 :: \tau_1, ..., l_n :: \tau_n \mid r |\!) &\equiv (\!| l_1 :: \tau_1 \mid ... \mid (\!| l_n :: \tau_n \mid r\ |\!) ... |\!)\end{aligned}$$

The fields of a row are distinguished by their label and not by their position, and we consider rows equal up to permutation of their fields:

$$(\!| l :: a, m :: b \mid r |\!) = (\!| m :: b, l :: a \mid r |\!)$$

2.2 Lacks contraints

For the purposes of this paper, we restrict ourselves to rows without duplicate labels. Without this constraint, fields can not be addressed unambiguously and, as a result, some programs can not be assigned a principal type [31]. A particularly elegant approach to enforce uniqueness of labels are *lacks* (or *insertion*) constraints [5, 25]. A predicate $(r \backslash l)$ restricts r to rows that do not contain a label l. In general, a row extension $(\!| l :: a \mid r |\!)$ is only valid when the predicate $(r \backslash l)$ holds. For clarity, we always write the lacks constraints explicitly in this paper, but a practical system can normally infer them from row expressions automatically, which simplifies the type signatures a lot.

2.3 Record operations

A record interprets a row as a product of types. The empty record is the only ground value with an empty record type:

$$\{\} :: \{\}$$

Furthermore, there are three basic operations that can be performed on records, namely *selection*, *restriction*, and *extension*:

$$\begin{aligned}(\_.l) \qquad &:: \forall r\, a.\ (r \backslash l) \Rightarrow \{\, l :: a \mid r \,\} \to a\\ (\_ - l) \qquad &:: \forall r\, a.\ (r \backslash l) \Rightarrow \{\, l :: a \mid r \,\} \to \{\, r \,\}\\ \{l = \_ \mid \_\} &:: \forall r\, a.\ (r \backslash l) \Rightarrow a \to \{\, r \,\} \to \{\, l :: a \mid r \,\}\end{aligned}$$

Note that we assume a distfix notation where argument positions are written as "_". Furthermore, we explicitly quantify all types in this paper, but practical systems can normally use implicit quantification. The three basic record operations are not arbitrary but based on the corresponding primitive operations on products in category theory or logic. Note that all type schemes contain a predicate $(r \backslash l)$ to ensure the validity of row extension. For repeated record extension on terms, we apply the same abbreviations as for row extension on types. The basic operations can be used to implement a number of other common operations like *update* and *rename*:

$$\begin{aligned}\{l := x \mid r\} &\equiv \{l = x \mid r - l\}\\ \{l \leftarrow m \mid r\} &\equiv \{m = r.l \mid r - l\}\end{aligned}$$

2.4 Variant operations

A variant interprets a row as a sum of types. Dual to records, the basic operations are based on the corresponding operations on sums in category theory. The primitives consist of the empty variant, *injection*, *embedding*, and *decomposition*:

$$\begin{array}{ll}
\langle\rangle & :: \langle\rangle \\
\langle l = _ \rangle & :: \forall r\, a.\ \ (r\backslash l) \Rightarrow a \ \ \rightarrow \langle l :: a \mid r\rangle \\
\langle l \mid _ \rangle & :: \forall r\, a.\ \ (r\backslash l) \Rightarrow \langle r\rangle \rightarrow \langle l :: a \mid r\rangle \\
(l \in _\ ?\ _ : _) & :: \forall r\, a\, b.\ (r\backslash l) \Rightarrow \langle l :: a \mid r\rangle \rightarrow (a \rightarrow b) \\
 & \qquad\qquad \rightarrow (\langle r\rangle \rightarrow b) \rightarrow b
\end{array}$$

Concrete implementations can provide special pattern matching syntax that use the primitive decomposition operator. Here is a short example to demonstrate basic record selection and a possible pattern matching syntax:

$$\begin{array}{l}
\mathit{showEvent} :: \mathit{Event} \rightarrow \mathit{String} \\
\mathit{showEvent}\ e \\
\quad = \textbf{case}\ e\ \textbf{of} \\
\qquad \langle \mathit{key} = c\rangle \quad \rightarrow \texttt{"key "} +\!\!+ \mathit{showChar}\ c \\
\qquad \langle \mathit{mouse} = p\rangle \rightarrow \texttt{"mouse "} +\!\!+ \mathit{showInt}\ p.x \\
\qquad\qquad\qquad\qquad +\!\!+ \texttt{", "} +\!\!+ \mathit{showInt}\ p.y \\
\mathit{test} = \mathit{showEvent}\ \langle \mathit{key} = \texttt{'a'}\rangle
\end{array}$$

2.5 Implementation

A naïve implementation of records would use a dynamic map from labels to values. Unfortunately, this prevents constant time access to the components of the record. Ohori [18], Gaster and Jones [5], and Shields and Meijer [25] represent records as contiguous blocks of memory where the type of the record is used to calculate the runtime offsets of its fields.

In the presence of polymorphic extensible rows, the offset of a field must sometimes be passed at runtime. This seems rather complicated at first, but, in a shining instance of the Curry-Howard isomorphism, runtime offsets directly correspond to *lacks* predicates. That is, each *lacks* predicate is translated to an implicit runtime parameter that carries the *evidence* for that predicate, namely an offset into the record. In the general treatment of qualified types, this is just an instance of evidence translation [9]. As an example, we consider the following expression:

$$\{\mathit{male} = \mathit{True}, \mathit{age} = 31, \mathit{name} = \texttt{"Daan"}\}.\mathit{male}$$

After evidence translation, the record is represented by a contiguous block of memory (31, *True*, `"Daan"`), where the order is determined by the lexicographical order of the labels. The field is selected by passing evidence to a general selection function:

$$(\lambda \mathit{ev}\ r \rightarrow r[\mathit{ev}])\ 1\ (31, \mathit{True}, \texttt{"Daan"})$$

Note that we write $r[i]$ to select the ith field of a memory block. The offset 1 is the evidence for the predicate $(\!|\mathit{age} :: \mathit{Int}, \mathit{name} :: \mathit{String}|\!)\backslash \mathit{male}$, that resulted from selecting the *male* field. It is resolved to 1 as *male* is larger than *age* but smaller than *name*. Gaster and Jones describe the formal translation from lacks predicates to evidence [5] and we will not repeat that here. Note that common transformations like inlining can be used to optimize this expression. As such, the calculus combines the flexibility of dynamic offset resolution with the efficiency of static offsets.

It is also straightforward to support efficient variant operations. Variants are represented at runtime by tagged values. The evidence for a variant operation is not interpreted as an offset but as the value of its tag, and it enables constant time matching of variant values.

3 First class labels

In the previous section, we have described a conventional set of record and variant operations. Labels are considered part of the syntax of the language and each basic operation was described as a *family* of functions parameterized by the labels. For example, it is not possible to write a general *select* function:

$$\mathit{select}\ r\ l = r.l$$

As it stands, the l parameter is unrelated to the constant l label. The function simply selects the *constant* label l, i.e. its type is:

$$\mathit{select} :: \forall r\, a\, b.\ (r\backslash l) \Rightarrow \{l :: a \mid r\} \rightarrow b \rightarrow a$$

In this article we describe an extension where labels become first-class values that can be passed as arguments. On the type level, we introduce a new type constructor *Lab*, where a value of type $\mathit{Lab}\ l$ is the term representation of a label l. The basic record and variant operations are now primitive functions that take a label as an argument. For example, the types of record selection and restriction become:

$$\begin{array}{ll}
(_._) & :: \forall r\, l\, a.\ (r\backslash l) \Rightarrow \{l :: a \mid r\} \rightarrow \mathit{Lab}\ l \rightarrow a \\
(_ - _) & :: \forall r\, l\, a.\ (r\backslash l) \Rightarrow \{l :: a \mid r\} \rightarrow \mathit{Lab}\ l \rightarrow \{r\}
\end{array}$$

These functions are now polymorphic in the label l. Furthermore, the l parameter to the *Lab* constructor nicely captures the connection between the label value and the label type; without it, we can not express the necessary typing constraints. A lacks predicate is now a binary constraint that takes a row and a label as arguments.

3.1 Label constants

With labels as first-class citizens, we can create functions that use labels passed as arguments. For example, the *zero* function creates a record with two zero values when given two distinct labels:

$$\begin{aligned} &\mathit{zero} :: \forall l\, m.\ (\langle\!| m :: \mathit{Int} |\!\rangle \backslash l) \\ &\qquad \Rightarrow \mathit{Lab}\ l \to \mathit{Lab}\ m \to \{l :: \mathit{Int}, m :: \mathit{Int}\} \\ &\mathit{zero}\ l\ m = \{l = 0, m = 0\} \end{aligned}$$

However, to use a function like *zero*, we need initial label constants. We write a constant label type l as $@l$. On the term level, we introduce a family of label constructors that are also written as $@l$:

$$@l :: \mathit{Lab}\ @l$$

The *zero* function can now be called with specific constant labels:

$$\begin{aligned} &\mathit{origin} :: \{@x :: \mathit{Int}, @y :: \mathit{Int}\} \\ &\mathit{origin} = \mathit{zero}\ @x\ @y \end{aligned}$$

The explicit annotation of label constants quickly becomes a burden as most labels are constant in practice. To simplify notation, we use the following *constantification* rule in Morrow: an identifier that is only used in label positions, is regarded as a constant label. An identifier is in a *label position* when it is used as a label in a row or as a selector. The constantification rule is used for both labels in types and labels in terms. In the following example, the l label is a variable, while the i label is regarded as a constant label:

$$\begin{aligned} &f :: \forall l.\ (\langle\!| i :: \mathit{Int} |\!\rangle \backslash l) \Rightarrow \mathit{Lab}\ l \to \{l :: \mathit{Int}, i :: \mathit{Int}\} \\ &f\ l = \{l = 0, i = 0\} \end{aligned}$$

In Morrow, the quantification is implicit and the lacks predicate is inferred from the record expression. The type of f can therefore also be written as:

$$f :: \mathit{Lab}\ l \to \{l :: \mathit{Int}, i :: \mathit{Int}\}$$

The weakness of constantification is that it is possible to invalidate code by adding a binding with an identifier that previously only occurred in label positions. We have not found this a problem yet in Morrow, but more practical experience is needed. Actually, many times this is an advantage, as we could redefine a label with a simple top-level definition.

3.2 Equality predicates

First class labels seem an innocent extension at first. Indeed the thesis of Gaster [4] describes first-class labels as a straightforward extension of extensible records. However, first-class labels require a new predicate to enjoy completeness of type inference. For example, the following program is not typeable in a naïve extension:

$$f\ r\ l\ m = r.l + r.m$$

The type we could give to f using *lacks* predicates is not as general as we would expect, as it restricts the labels l and m to be distinct:

$$\begin{aligned} &f :: \forall r\, l\, m.\ (\langle\!| m :: \mathit{Int} \mid r |\!\rangle \backslash l, r \backslash m) \\ &\qquad \Rightarrow \{l :: \mathit{Int}, m :: \mathit{Int} \mid r\} \to \mathit{Lab}\ m \to \mathit{Lab}\ l \to \mathit{Int} \end{aligned}$$

Sulzmann acknowledges this problem in his thesis [29] where he describes an instantiation of HM(X) [17] with first-class labels, called REC^l. In this system, only record update and selection are label polymorphic; record construction is still restricted to constant labels. Due to this restriction, *has* predicates suffice to type all operations, and the type of f becomes:

$$\begin{aligned} &f :: \forall r\, l\, m.\ (l :: \mathit{Int} \in r, m :: \mathit{Int} \in r) \\ &\qquad \Rightarrow \{r\} \to \mathit{Lab}\ m \to \mathit{Lab}\ l \to \mathit{Int} \end{aligned}$$

Unfortunately, as discussed by Sulzmann [29], it is not straightforward to extend HM(REC^l) to label polymorphic record extension.

We will instead use *row equality* predicates. The predicate $(r \sim s)$ restricts row r and s to equal rows. The row equality predicate arises naturally as the most general predicate from the row unification rule of Gaster and Jones [5]. This rule is approximately:

$$\frac{(l :: \tau) \in s \qquad r \sim (s - l)}{\langle\!| l :: \tau \mid r |\!\rangle \sim s}$$

As we show later, there is not always a unique unifier between rows in the presence of polymorphic labels, and row equality predicates can be viewed as a delayed unification. As is apparent from the row unification rule, *has* constraints alone lose information; namely $r \sim (s - l)$. This property is sometimes stated as "*has* constraints encode only positive information". This is no problem when only selection and update are label polymorphic, but record extension can not be typed with *has* predicates alone. In an unpublished report by Sulzmann [28], in addition to the *has* predicate, the *distinct* predicate is introduced especially to capture the required type constraints. In contrast, it is easy to express *has* predicates in terms of row equality:

$$(l :: a) \in r \quad \equiv \quad s \backslash l,\ \langle\!| l :: a \mid s |\!\rangle \sim r \quad (\text{fresh } s)$$

Morrow uses the above equivalence to allow *has* predicates as convenient syntactic sugar. However, for clarity, we normally use explicit lacks and row equality predicates in this paper. Using row equality predicates, we can now assign the following principal type to f:

$$\begin{aligned} &f :: \forall r\, s\, t\, l\, m.\ (s \backslash l, t \backslash m, r \sim \langle\!| l :: \mathit{Int} \mid s |\!\rangle, \\ &\qquad\qquad r \sim \langle\!| m :: \mathit{Int} \mid t |\!\rangle) \\ &\quad \Rightarrow \{r\} \to \mathit{Lab}\ l \to \mathit{Lab}\ m \to \mathit{Int} \end{aligned}$$

Note that we can substitute r with either row to get an equivalent type signature:

$$f :: \forall s\, t\, l\, m.\ (s \backslash l, m \backslash t, (\!| l :: Int \mid s |\!) \sim (\!| m :: Int \mid t |\!)) \Rightarrow \{ l :: Int \mid s \} \to Lab\ l \to Lab\ m \to Int$$

4 Expressiveness of first-class labels

This section shows that first-class labels do not only get rid of families of primitive functions, but also increase the expressiveness of our calculus. We assume the definition of a polymorphic label *any*:

$$any :: \forall l.\ Lab\ l$$
$$any = \bot$$

With strict semantics, the *any* label can also be defined with an abstract phantom datatype for labels [13]:

$$\textbf{data}\ Lab\ a = Lab\ String \mid Any$$
$$any :: \forall l.\ Lab\ l$$
$$any = Any$$

Such definition also gives a straightforward implementation for showing and comparing labels.

4.1 Intersection types

Using the polymorphic *any* label, we can select an arbitrary field from a record. This can be used to encode an intersection type. For example:

$$intBool :: \forall r\, l\, a.\ (r \backslash l, (\!| l :: a \mid r |\!) \sim (\!| i :: Int, b :: Bool |\!)) \Rightarrow a$$
$$intBool = \{ i = 1, b = True \}.any$$

The only possible instances of a are *Int* or *Bool*, depending on the context. As such it can be seen equivalent to the type $Int \wedge Bool$ in an intersection type discipline [19, 22]. Here is an example of the use of *intBool*:

$$n = \textbf{if}\ intBool\ \textbf{then}\ intBool + 1\ \textbf{else}\ 0$$

On first sight, it seems that even though the expression type checks, the expression will fail at runtime since *any* is undefined. However, the label values themselves are not used by selection at runtime; only the *evidence* provided by the $(r \backslash l)$ predicate is used to select the proper field. The two occurrences of *intBool* above each unify with a different type, and the predicate $(r \backslash l)$ is resolved to either $(\!| i :: Int |\!) \backslash b$ or $(\!| b :: Bool |\!) \backslash i$, corresponding to the appropiate offset at runtime. After evidence translation, the example looks like:

$$intBool\ ev = (\lambda i\ r\ l \to r[i])\ ev\ (True, 1)\ any$$
$$n = \textbf{if}\ (intBool\ 0)\ \textbf{then}\ (intBool\ 1) + 1\ \textbf{else}\ 0$$

The implicit evidence parameter ensures that the correct value is chosen based on the type context. On the side, we remark that the type of *intBool* can also be written more conveniently as:

$$intBool :: \forall l\, a.\ ((l :: a) \in (\!| i :: Int, b :: Bool |\!)) \Rightarrow a$$

We have not done so yet to illustrate clearly that *lacks* predicates correspond to the runtime evidence. However, in the next sections we will freely use this notation as convenient syntactic sugar.

4.2 Overloading

Using intersection types, we can encode a form of *closed world* type overloading. This view of overloading is adopted by many object-oriented languages, for example in Java. It assumes that all definitions for an overloaded identifier are known, and requires that each definition is declared at a distinct type. As an example, we show how to overload addition in Morrow. The example is adapted from Shields [24] where a similar encoding was presented based on λ^{tir}. First, we declare a record that contains all primitive addition functions:

$$\textbf{type}\ PlusRec = \{\, int :: Int \to Int \to Int,\ float :: Float \to Float \to Float \,\}$$
$$plusRec :: PlusRec$$
$$plusRec = \{\, int = plusInt, float = plusFloat \,\}$$

The $(+)$ operator simply selects one of these functions using *any*:

$$(+) :: \forall l\, a.\ ((l :: a) \in PlusRec)) \Rightarrow a$$
$$(+) = plusRec.any$$

Addition can now be used at different types:

$$ok = (1 + 2, 1.0 + 2.0)$$

Just like the previous example, the evidence passed to $(+)$ selects the appropiate addition function based on the type context. In conventional closed-world overloading, each overloaded definition must be sufficiently monomorphic to resolve the overloading statically, but this is not necessary in our system. We show an example by Shields [24] where we overload the identifier *nList* over functions that return a list with either one or two arguments:

$$nList :: \forall l\, a\, b\, c.\ ((l :: a) \in (\!| one :: b \to [b], two :: c \to c \to [c] |\!)) \Rightarrow a$$
$$nList = \{ one = \lambda x \to [x], two = \lambda x\ y \to [x, y] \}.any$$

Note that we do not experience the simplification limitations described by Shields [24] as we always use explicit labels. Besides *closed-world* overloading, we also have the *open-world* view of overloading, as adopted by Haskell. The type of each definition is required to be an instance of a certain type scheme, but the definitions do not have to be in scope at each point of use.

As suggested by Shields [24], we can encode open-world overloading using implicit parameters [14], but a full discussion is beyond the scope of this paper.

4.3 Type case

Using the same mechanism as with intersection types, we can also encode a form of type case. Given a record of functions, we select any function and apply it to the type case argument:

$$
\begin{aligned}
&\mathit{typecase} :: \forall r\, l\, a\, b.\ (r \backslash l) \Rightarrow a \to \{l :: a \mid r\} \to b \\
&\mathit{typecase}\ x\ r = (r.\mathit{any})\ x
\end{aligned}
$$

Note that this differs from the usual notion of type case, since the result of the operation can also depend on the type of the argument. We shall see in Section 5 how we can use row polymorphic operations to enforce that all functions return the same result type. The function *typecase* does not perform any runtime checking; the evidence for the predicate $r \backslash l$ determines which function is selected, which is resolved at compile-time and passed at run-time as a fixed offset.

4.4 Label selective functions

As shown by Sulzmann [28], we can also encode a label selective calculus along the lines of Garrigue [3]. In a label selective calculus, each function argument is associated with a label. Effectively, the arguments form a record, but we can still use curried function application. As an example, we create a label selective function *lmap* from the standard *map* function:

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to [a] \to [b] \\
&\mathit{map}\ f\ \mathit{xs} = ... \\
&\mathit{lmap} :: \forall l\, m\, a\, b\, c\, d\, e.\ (\!|m :: e|\!) \backslash l, \\
&\qquad (\!|l :: d, m :: e|\!) \sim (\!|\mathit{fun} :: a \to b, \mathit{list} :: [a]|\!) \\
&\qquad \Rightarrow (\mathit{Lab}\ l, d) \to (\mathit{Lab}\ m, e) \to [b] \\
&\mathit{lmap}\ (l, x)\ (m, y) = \mathbf{let}\ r = \{l = x, m = y\} \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{map}\ r.\mathit{fun}\ r.\mathit{list} \\
&\mathit{square}\ \mathit{xs} = \mathit{lmap}\ (\mathit{list}, \mathit{xs})\ (\mathit{fun}, \lambda i \to i * i)
\end{aligned}
$$

We use label polymorphic record extension here and the type of *lmap* cannot be expressed with *has* constraints alone. Sulzmann [28] specifically introduces the *distinct* predicate to capture the necessary type constraints.

4.5 Type selective functions

A *type selective* function determines the rôle of an argument based on its type. In general we can encode functions that take arguments of a different type in any permutation. As an example, we implement a function *permute* that transforms any two argument function into a function that takes its arguments in any order. First, we define a record that holds both variants:

$$
\begin{aligned}
\mathbf{type}\ \mathit{Perm}\ a\ b\ c = \{ &\mathit{normal} :: a \to b \to c, \\
&\mathit{flipped}\ :: b \to a \to c\}
\end{aligned}
$$

Next, we define a function that constructs a permutation record from a two-argument function:

$$
\begin{aligned}
&\mathit{permRec} :: \forall a\, b\, c.\ (a \to b \to c) \to \mathit{Perm}\ a\ b\ c \\
&\mathit{permRec}\ f = \{\mathit{normal} = f, \mathit{flipped} = \lambda x\ y \to f\ y\ x\}
\end{aligned}
$$

Finally, we use *any* to select either function:

$$
\begin{aligned}
&\mathit{permute} :: \forall l\, a\, b\, c\, d.\ ((l :: d) \in \mathit{Perm}\ a\ b\ c) \\
&\qquad \Rightarrow (a \to b \to c) \to d \\
&\mathit{permute}\ f = (\mathit{permRec}\ f).\mathit{any}
\end{aligned}
$$

Using *permute*, a function with two arguments of a different type can take them in any order:

$$
\begin{aligned}
&\mathit{square}\ \mathit{xs} = \mathbf{let}\ \mathit{pmap} = \mathit{permute}\ \mathit{map} \\
&\qquad\qquad\qquad \mathbf{in}\ \mathit{pmap}\ \mathit{xs}\ (\lambda i \to i * i)
\end{aligned}
$$

We can also apply *permute* to a function that takes two arguments of equal type. However, when such function is used, the type inferencer complains about unresolved predicates since the lacks predicate can not be resolved unambigiously. This is a weakness of our approach; deferring unifications may lead to an expressive system, but as a result, errors are also reported late.

4.6 First-class messages

In object-oriented languages, methods are invoked by sending a message. Even though messages are central to object-orientation, they are not first-class citizens in most object-oriented languages, and most languages resort to dynamic type checks in the presence of polymorphic messages. As a result, there has been a lot of research into static type systems for dynamic messages [26, 15, 16]. First-class messages can also be viewed as first-class labels in a record of functions, and we can directly encode the proxy example by Nishimura [16]:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Ftp} = (\!|\mathit{put} :: \mathit{String} \to \mathit{IO}\ (), ...|\!) \\
&\mathit{ftp} \quad :: \{\mathit{Ftp}\} \\
&\mathit{ftp} \quad = \{\mathit{put} = ...\} \\
&\mathit{proxy} \quad :: \forall r\, l\, a.\ ((l :: a) \in r) \\
&\qquad \Rightarrow \{\mathit{new} :: r \to \{\mathit{send} :: \mathit{Lab}\ l \to a\}\} \\
&\mathit{proxy} \quad = \{\mathit{new} = \lambda \mathit{obj} \to \{\mathit{send} = \lambda m \to \mathit{obj}.m\}\} \\
&\mathit{fproxy}\ :: \forall l\, a.\ ((l :: a) \in \mathit{Ftp}) \Rightarrow \{\mathit{send} :: \mathit{Lab}\ l \to a\} \\
&\mathit{fproxy}\ = \mathit{proxy}.\mathit{new}\ \mathit{ftp} \\
&\mathit{submit} :: \mathit{IO}\ () \\
&\mathit{submit} = \mathit{fproxy}.\mathit{send}\ @\mathit{put}\ \texttt{"paper.ps"}
\end{aligned}
$$

Of course, although we can encode first-class messages, we still lack the usual contravariance and subtyping

rules associated with object-oriented languages. In our calculus, the subtype relation between records is always explicit through polymorphic row extension.

5 First-class patterns

Our basic system can be naturally extended with row polymorphic type operations. Using those operations, we can express first-class patterns. Instead of introducing special syntax for destructing variants as shown in Section 2.4, we just use a normal function *case* that takes a record of functions as the pattern. Based on the variant label, the *case* function applies the corresponding function from the record to the variant value. Here is an example in Morrow, where we calculate the sum of a list of integers:

$$\begin{aligned}&\textbf{newtype } \mathit{List}\ a = \mathit{List}\ \langle \mathit{nil} \quad :: \{\,\},\\ &\qquad\qquad\qquad\qquad \mathit{cons} :: \{\mathit{hd} :: a, \mathit{tl} :: \mathit{List}\ a\}\rangle\\ &\mathit{sum} :: \mathit{List}\ \mathit{Int} \to \mathit{Int}\\ &\mathit{sum}\ (\mathit{List}\ \mathit{xs})\\ &\quad = \mathit{case}\ \mathit{xs}\ \{\mathit{nil} \quad = \lambda r \to 0,\\ &\qquad\qquad\quad \mathit{cons} = \lambda r \to r.\mathit{hd} + \mathit{sum}\ r.\mathit{tl}\}\end{aligned}$$

Just like Haskell, Morrow uses a **newtype** declaration to express recursive types. A newtype constructor like *List* only serves as a hint to the type inferencer and has no runtime representation. The pattern that we use to match on the list variant is just a normal record, and patterns become first-class polymorphic and extensible entities. As a result, we can express views on data types [30]. Here is an example of a function *snoc* that gives a reversed view on lists:

$$\begin{aligned}&\mathit{snoc}\ \mathit{xs}\ r = \mathit{case}\ (\mathit{reverse}\ \mathit{xs})\ r\\ &\mathit{last}\ \mathit{xs} \quad = \mathit{snoc}\ \mathit{xs}\ \{\mathit{nil} \quad = \lambda r \to \bot,\\ &\qquad\qquad\qquad\quad \mathit{cons} = \lambda r \to r.\mathit{hd}\}\end{aligned}$$

Note that we can implement the *case* function very efficiently: the *tag* of the variant corresponds directly to the *offset* in the record! Even though patterns are first class, a practical system should probably add syntactic sugar to support nested patterns. Conventional pattern matching also supports default patterns; this is much harder to accomodate in our system as a single default pattern represents a family of fields [2].

Gaster and Jones [5] describe two dual type operations, *to* and *from*, to give a principal type to row polymorphic functions such as *case*. These type operations are built into the type inferencer and are treated as keywords in the language. The *to* operator is recursively defined as:

$$\begin{aligned}&\mathit{to}\ a\ (\!|\,|\!) \qquad\quad = (\!|\,|\!)\\ &\mathit{to}\ a\ (\!|\, l :: b \mid r \,|\!) = (\!|\, l :: b \to a \mid \mathit{to}\ a\ r \,|\!)\end{aligned}$$

This operation can be used to give a principal types to two general functions for decomposing variants and records:

$$\begin{aligned}&\mathit{case} \quad :: \forall r\, a.\ \langle r\rangle \ \to \{\mathit{to}\ a\ r\} \to a\\ &\mathit{recElim} :: \forall r\, a.\ \{r\} \to \langle \mathit{to}\ a\ r\rangle \ \to a\end{aligned}$$

The *to* operator nicely captures that for a *case* on a variant over row r, we need to provide a function for each field in r that transforms the corresponding value into a common type a, i.e. a record over the row $(\mathit{to}\ a\ r)$! Using the dual type operator *from*, with the obvious definition, we can specify the dual operations for constructing records and variants:

$$\begin{aligned}&\mathit{varIntro} :: \forall r\, a.\ a \to \langle \mathit{from}\ a\ r\rangle \ \to \langle r\rangle\\ &\mathit{recIntro} :: \forall r\, a.\ a \to \{\mathit{from}\ a\ r\} \to \{r\}\end{aligned}$$

We are are also investigating the use of other type operators, like *zip* and *join*, but a full discussion is beyond the scope of this article.

Even though Gaster and Jones describe the *to* and *from* type operators, they also mention severe technical difficulties with unification. When unifying two rows $\mathit{to}\ \tau\ r$ and $\mathit{from}\ \tau'\ r'$, the types τ and τ' are not related when r (and r') are empty. To obtain most general unifiers, Gaster must restrict the entire record system to deal with *non-empty* rows only [4]. This complicates the system greatly. For example, special *has* constraints are added to give a principal type to record selection. Furthermore, they lose some of the mathematical elegance as they are no longer able to express the empty record and variant.

Serendipitously, we found that row equality constraints that were introduced to support principal types for first-class labels could also express the necessary type constraints for row polymorphic operations. As we will see during the formal development in Section 6, the language of rows is restricted to predicates only, and the above unification problem simply becomes a deferred predicate: $\mathit{to}\ \tau\ r \sim \mathit{from}\ \tau'\ r'$.

6 Typing rules

This section starts the formal development of our calculus. First the kinds, types and syntax are explained, followed by the type rules and inference. We also discuss issues as improvement, satisfiability, simplification, and subsumption in more detail.

Our formal development is based fundamentally on the theory of qualified types [8, 9] and higher-order polymorphism [11]. The theory of qualified types gives us a general framework for constrained type inference, while higher-order polymorphism allows us to introduce rows and labels as orthogonal language features.

6.1 Kinds

A kind system distinguishes between different kinds of type constructors and ensures well-formedness of types. The set of kinds is defined by the following grammar:

$$\begin{array}{lcll} \kappa & ::= & * & \textit{the kind of value types} \\ & | & \mathsf{row} & \textit{the kind of row types} \\ & | & \mathsf{lab} & \textit{the kind of label types} \\ & | & \kappa \to \kappa' & \textit{the kind of type constructors} \end{array}$$

All terms have types of kind star. The row and lab kinds are used for row and label types and only occur at the type level. The arrow kind is used for type constructors like polymorphic lists.

6.2 Types

We assume an initial set of type variables $\alpha \in \mathcal{A}$ and type constants $c \in \mathcal{C}$, and we assume that the initial set of type constants $\mathcal{C}$ includes at least:

$$\begin{array}{llll} \mathit{Int} & ::: & * & \textit{integers} \\ \to & ::: & * \to * \to * & \textit{function space} \\ \{\_\} & ::: & \mathsf{row} \to * & \textit{record constructor} \\ \langle\_\rangle & ::: & \mathsf{row} \to * & \textit{variant constructor} \\ \mathit{Lab} & ::: & \mathsf{lab} \to * & \textit{label constructor} \end{array}$$

For each kind κ we have a collection of constructors C^κ of kind κ. The set of all constructors is given by the following grammar:

$$\begin{array}{lcll} C^\kappa & ::= & c^\kappa & \textit{type constants} \\ & | & \alpha^\kappa & \textit{type variables} \\ & | & C^{\kappa' \to \kappa}\ C^{\kappa'} & \textit{type application} \\ \\ \tau & ::= & C^* & \textit{(mono) types} \end{array}$$

The set of value types τ is simply the set of constructors of kind star. Although we explicitly annotated all constructors with kinds, this not necessary in practice, as they can be inferred from type expressions [11].

6.3 Rows

Notably absent from the constructors and type constants are row expressions such as row extension; only row variables are part of the constructors. We will see later that row expressions are limited to the set of predicates in order to ensure a sound translation to qualified types. The set of row expressions r is defined as:

$$\begin{array}{lcll} \rho & ::= & \alpha^{\mathsf{row}} & \textit{row variables} \\ l & ::= & c^{\mathsf{lab}} & \textit{label constants} \\ & | & \alpha^{\mathsf{lab}} & \textit{label variables} \\ \\ r & ::= & \rho & \textit{row variable} \\ & | & (\!|\,|\!) & \textit{the empty row} \\ & | & (\!|\, l :: \tau \mid r \,|\!) & \textit{row extension} \end{array}$$

By definition, the kind of rows r is always row. Furthermore, labels become first-class entities since type variables α^{lab} are included in the set of labels. As a notational convenience, we write l_c for constant labels and l_α for label variables.

When we define relations over rows, we denote syntactic label equality by using the same label name. Dually, the relation $(l \neq l')$ holds when two labels are syntactically unequal. We also define a relation $l \neq_c l'$ that only holds when two *constant* labels are distinct.

$$\frac{l \neq l' \qquad l \in \mathcal{C} \qquad l' \in \mathcal{C}}{l \neq_c l'}$$

In constrast to the work of Gaster and Jones [5], we need to define this new notion of inequality, as it is closed under substitution while syntactic inequality is not. This becomes important when proving that our entailment relation is closed under substitution; a necessary requirement in the theory of qualified types [8].

We consider rows equal up to permutation of their fields:

$$(\!|\, l :: \tau, l' :: \tau' \mid r \,|\!) \;=\; (\!|\, l' :: \tau', l :: \tau \mid r \,|\!)$$

The *membership* relation $(l :: \tau) \in r$ defines whether a particular field $(l :: \tau)$ is an element of row r. Gaster [4] defines this relation as:

$$(l :: \tau) \in (\!|\, l :: \tau \mid r \,|\!) \qquad \frac{(l :: \tau) \in r \qquad l \neq_c l'}{(l :: \tau) \in (\!|\, l' :: \tau' \mid r \,|\!)}$$

Although the relation corresponds to our intuitive notion of membership, it is no longer well-defined with respect to row equality. For example, we can derive:

$$(c :: \tau) \in (\!|\, c :: \tau, \alpha :: \tau' \,|\!)$$

but not:

$$(c :: \tau) \in (\!|\, \alpha :: \tau', c :: \tau \,|\!)$$

since label unequality is not defined on label variables, i.e. $(c \neq_c \alpha)$. A solution is to strengthen the membership relation such that the first property no longer holds. First, we define the *lacks* relation as:

$$l \notin (\!|\,|\!) \qquad \frac{l \notin r \qquad l \neq_c l'}{l \notin (\!|\, l' :: \tau \mid r \,|\!)}$$

The membership relation is now strengthened by requiring that a label no longer occurs in the tail:

$$\frac{l \notin r}{(l :: \tau) \in (\!|\, l :: \tau \mid r \,|\!)} \qquad \frac{(l :: \tau) \in r \qquad l \neq_c l'}{(l :: \tau) \in (\!|\, l' :: \tau' \mid r \,|\!)}$$

A straightforward proof by induction on the rows shows that this relation is well-defined with respect to equality on rows. Furthermore, it closed under substitution.

The *restriction* relation $(r - l)$ removes a label l from a row r. Just like membership, we strengthen the relation with respect to Gaster and Jones' original definition to make it well-defined over row equality:

$$\begin{array}{lcll} (\!|l :: \tau \,|\, r|\!) - l & = & r & \text{if } l \notin r \\ (\!|l' :: \tau \,|\, r|\!) - l & = & (\!|l' :: \tau \,|\, r - l|\!) & \text{if } l \neq_c l' \end{array}$$

Gaster proves various properties of these relations [4], including a nice law that relates membership and restriction:

$$(l :: \tau) \in r \quad \Rightarrow \quad r = (\!|l :: \tau \,|\, r - l|\!)$$

6.4 Predicates

Based on the theory of qualified types [9], we will use predicates to express necessary type constraints over row operations. A predicate π is either a *lacks* or a *row equality* predicate:

$$\begin{array}{lcll} \pi & ::= & r\backslash l & \textit{lacks predicate} \\ & | & r \sim r' & \textit{row equality predicate} \end{array}$$

A lacks predicate $(r\backslash l)$ restricts a row r to rows that do not contain a label l. An equality predicate $(r \sim s)$ restricts the rows r and s to be equal (up to permutation).

Entailment relates two finite set of predicates P and Q. A derivation of $P \Vdash Q$ is a proof that when all predicates in the finite set P hold, the predicates Q hold too. We can formalize lacks and row equality predicates using the entailment relation defined in Figure 1. We write $P \Vdash \pi$ as a shorthand for $P \Vdash \{\pi\}$. The first three entailment rules are standard. Note how the rules on *lacks* predicates nicely correspond to our *lacks* relation defined in the previous section. The last three rules handle equality predicates. Rule (*eqHead*) is interesting: two non-empty rows are equal if the first field of the first row is an element of the second, and when the remaining rows without this field are also equal.

To make our calculus an instance of the theory of qualified types, we need to prove that the entailment relation is monotone, transitive, and closed under substitution.

Theorem 1 *The instance relation in Figure 1 satisfies:*

- *Monotonicity:* $Q \subseteq P \Rightarrow P \Vdash Q$.
- *Transitivity:* $P \Vdash Q \wedge Q \Vdash R \Rightarrow P \Vdash R$.
- *Closure:* $P \Vdash Q \Rightarrow SP \Vdash SQ$, *for any substitution* S *mapping type variables to type expressions.*

The first two properties are trivial as we use sets for predicates. The closure property is proved by laborious induction. Furthermore, we can prove that entailment is well-defined with respect to row equality.

(*taut*)	$\dfrac{\pi \in P}{P \Vdash \pi}$
(*lackEmpty*)	$P \Vdash (\!|\,|\!)\backslash l$
(*lackTail*)	$\dfrac{P \Vdash r\backslash l \qquad l \neq_c l'}{P \Vdash (\!|l' : \tau' \,|\, r|\!)\backslash l}$
(*eqEmpty*)	$P \Vdash (\!|\,|\!) \sim (\!|\,|\!)$
(*eqVar*)	$P \Vdash \rho \sim \rho$
(*eqHead*)	$\dfrac{(l : \tau) \in s \qquad P \Vdash r \sim (s - l)}{P \Vdash (\!|l : \tau \,|\, r|\!) \sim s}$

Figure 1: Entailment

In our calculus, row expressions are only part of the predicate language. This means that a practical system should view row expressions in types as syntactic sugar; indeed, we can view any row expression in a type as a row variable that is restricted to be equal to that expression:

$$\ldots\{l :: a\}\ldots \;\equiv\; r \sim (\!|l :: a|\!) \Rightarrow \ldots\{r\}\ldots \quad (\text{fresh } r)$$

6.5 Qualified types

We use the same structure as in the theory of qualified types [8] where we distinguish between *qualified types* φ and *type schemes* σ:

$$\begin{array}{lcll} \varphi & ::= & \tau \mid \pi \Rightarrow \varphi & \textit{qualified type} \\ \sigma & ::= & \varphi \mid \forall\alpha.\sigma & \textit{quantified type or type scheme} \end{array}$$

The free type variables of a type scheme σ are written as $\mathit{ftv}(\sigma)$. We overload this function on other objects in the obvious way. Since the order of the predicates and quantified type variables does not matter, we sometimes use the following abbreviations:

$$\begin{array}{lcl} \forall\alpha_1 \ldots \forall\alpha_n.\varphi & \equiv & \forall\overline{\alpha}.\,\varphi \\ \pi_1 \Rightarrow \ldots \Rightarrow \pi_n \Rightarrow \tau & \equiv & \overline{\pi} \Rightarrow \tau \;\equiv\; P \Rightarrow \tau \end{array}$$

The term language of our calculus is an implicitly typed λ-calculus, extended with constants and *let* bindings:

$$e ::= x \mid k \mid e\ e' \mid \lambda x.e \mid \textbf{let}\ x = e\ \textbf{in}\ e'$$

The constants k should include at least the basic record, variant, and label functions. Each constant k is also assigned an appropiate (closed) type scheme σ_k. An

$$(const) \qquad P \mid A \vdash k : \sigma_k$$

$$(var) \qquad \frac{(x:\sigma) \in A}{P \mid A \vdash x : \sigma}$$

$$(\rightarrow E) \qquad \frac{P \mid A \vdash e : \tau' \rightarrow \tau \qquad P \mid A \vdash e' : \tau'}{P \mid A \vdash e\, e' : \tau}$$

$$(\rightarrow I) \qquad \frac{P \mid A, x : \tau' \vdash e : \tau}{P \mid A \vdash \lambda x.e : \tau' \rightarrow \tau}$$

$$(\Rightarrow E) \qquad \frac{P \mid A \vdash e : \pi \Rightarrow \varphi \qquad P \Vdash \pi}{P \mid A \vdash e : \varphi}$$

$$(\Rightarrow I) \qquad \frac{P \cup \{\pi\} \mid A \vdash e : \varphi}{P \mid A \vdash e : \pi \Rightarrow \varphi}$$

$$(\forall E) \qquad \frac{P \mid A \vdash e : \forall \alpha.\sigma}{P \mid A \vdash e : [\alpha \mapsto \tau]\sigma}$$

$$(\forall I) \qquad \frac{P \mid A \vdash e : \sigma \qquad \alpha \notin \mathit{ftv}(A) \cup \mathit{ftv}(P)}{P \mid A \vdash e : \forall \alpha.\sigma}$$

$$(let) \qquad \frac{P \mid A \vdash e : \sigma \qquad Q \mid A_x, x : \sigma \vdash e' : \tau}{P \cup Q \mid A \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau}$$

Figure 2: Typing rules

assumption A is a finite set of type assignments $x : \sigma$ in which no term variable x occurs more than once.

$$A ::= \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \quad \textit{Typothesis}$$

We write A_x to denote an assumption A where the type assignment for x is removed. We also abbreviate $A \cup \{x : \sigma\}$ as $A, x : \sigma$. Figure 2 shows the standard typing rules for the theory of qualified types extended with the typing rule for constants. A typing judgement $P \mid A \vdash e : \sigma$ asserts that expression e has type σ when the predicates P are satisfied and the types of the free variables are given by the assumption A.

7 Type inference

This section discusses the standard inference algorithm by Jones [8] that calculates principal types with respect to the typing rules of the previous section. Before showing the algorithm, we first look at substitution and unification.

$$C \overset{id}{\sim} C \qquad \frac{\alpha \notin \mathit{ftv}(C)}{\alpha \overset{[\alpha \mapsto C]}{\sim} C} \qquad \frac{\alpha \notin \mathit{ftv}(C)}{C \overset{[\alpha \mapsto C]}{\sim} \alpha}$$

$$\frac{C \overset{U}{\sim} D \qquad UC' \overset{U'}{\sim} UD'}{C\,C' \overset{U'U}{\sim} D\,D'}$$

Figure 3: Kind preserving unification

7.1 Substitution and unification

A substitution is an idempotent map from type variables α to constructors C, and which is the identity function on all but a finite set of type variables. We write the empty substitution as *id*. A substitution that maps a type variable α to a constructor C is written as $[\alpha \mapsto C]$. Finally, we write TS for the composition of substitution T with S. For the purposes of this article, we restrict ourselves to *kind preserving* substitutions where a type variable always maps to a constructor of the same kind.

A substitution S is a *unifier* of two constructors C and C' when $SC = SC'$. We call a substitution S of two constructors C and C' *most general* when every unifier of these constructors can be written as TS for some substitution T. As types are represented by simple Herbrand terms, we can use standard kind-preserving unification [11, 4] as shown in Figure 3. The expression $\tau \overset{U}{\sim} \tau'$ calculates a most general unifier U for types τ and τ'.

Theorem 2 *The unification algorithm in Figure 3 returns a most general unifier when it exists. It fails precisely when no such unifier exists.*

A proof of this result is given by Robinson [23]. With the given unification algorithm, we can directly use the type inference algoritm for qualified types [9] as a type inference algorithm for our calculus. For completeness, we include the inference algorithm in Figure 4. The type inference rules are interpreted as an attribute grammar where each judgement of the form $P \mid S \mid A \vdash^{\mathsf{w}} e : \tau$ is a semantic rule where the assumption A and expression e are inherited, while the predicates P, substitution S, and the type τ are synthesized.

The (let^{w}) rule uses a *generalization* function to quantify a qualified type. The generalization function quantifies over all free variables that are not present in the assumption:

$$\mathit{gen}(A, \varphi) = \forall \overline{\alpha}.\varphi \quad \text{where } \overline{\alpha} = \mathit{ftv}(\varphi) - \mathit{ftv}(A)$$

Jones proves that the given algorithm is both sound and complete with respect to the typing rules in Figure 2 [8].

Theorem 3 *The algorithm in Figure 4 calculates a*

$$(var^{\mathsf{w}}) \quad \frac{(x : \forall \overline{\alpha}.P \Rightarrow \tau) \in A \qquad \mathit{fresh}(\overline{\beta}) \qquad S = [\overline{\alpha \mapsto \beta}]}{SP \mid \epsilon \mid A \vdash^{\mathsf{w}} x : S\tau}$$

$$(\rightarrow E^{\mathsf{w}}) \quad \frac{P \mid S \mid A \vdash^{\mathsf{w}} e : \tau \qquad Q \mid T \mid SA \vdash^{\mathsf{w}} e' : \tau' \qquad T\tau \overset{U}{\sim} \tau' \rightarrow \alpha \qquad \mathit{fresh}(\alpha)}{U(TP \cup Q) \mid UTS \mid A \vdash^{\mathsf{w}} e\, e' : U\alpha}$$

$$(\rightarrow I^{\mathsf{w}}) \quad \frac{P \mid S \mid (A_x, x : \alpha) \vdash^{\mathsf{w}} e : \tau \qquad \mathit{fresh}(\alpha)}{P \mid S \mid A \vdash^{\mathsf{w}} \lambda x.e : S\alpha \rightarrow \tau}$$

$$(let^{\mathsf{w}}) \quad \frac{P \mid S \mid A \vdash^{\mathsf{w}} e : \tau \qquad \sigma = \mathit{gen}(SA, P \Rightarrow \tau) \qquad Q \mid T \mid SA_x, x : \sigma \vdash^{\mathsf{w}} e' : \tau'}{Q \mid TS \mid A \vdash^{\mathsf{w}} \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'}$$

Figure 4: Type inference algorithm W

principal type for a given expression e and assumption A. It fails precisely when no type exists for e under the assumption A.

7.2 Rows as predicates

We made rows part of the predicate language and use standard Robinson unification to unify types. This means that we have deviated quite a bit from the extensible records of Gaster and Jones. In their system, row operations are part of the type language and the unification algorithm is extended to deal with row unification. There are two important technical problems with this approach in the presence of first-class labels.

First of all, we would lose completeness of type inference when row extension is structural (i.e. part of the type language). Take for example the following definition:

$$\begin{array}{l} f :: \forall r\, s\, a\, b.\ (s \backslash x, s \backslash y, r \sim (\!| x :: a, y :: b \mid s |\!)) \\ \qquad \Rightarrow \{r\} \rightarrow (a, b) \\ f\ r = \mathbf{let}\ \mathit{select}\ l = r.l\ \mathbf{in}\ (\mathit{select}\ @x, \mathit{select}\ @y) \end{array}$$

If row extension was part of the type language, the type of f could not be inferred. When inferring the type of $r.l$, the record r would unify with $\{l :: a \mid s\}$. When generalizing the type of the body of *select*, namely $\mathit{Lab}\ l \rightarrow a$, the type variables l and a are free in the type assignment for r in the assumption A. This would prevent generalization of l and a, making *select* monomorphic in the label argument, which eventually leads to rejection of this example.

In our calculus, the row extension is part of the language of predicates. The expression $r.l$ would not unify r with a row, but add a predicate $r \sim (\!| l :: a \mid s |\!)$ instead. The type of *select* can now generalize properly to $\forall s\, l\, a.\ (s \backslash l, r \sim (l :: a \mid s)) \Rightarrow \mathit{Lab}\ l \rightarrow a$, where r is a monomorphic (scoped) type variable.

The second technical problem arises during unification. With constant labels, we can always find a single most general unifier between two rows (if it exists). This makes it possible to extend Robinson unification with row unification. However, with first-class labels there can be many general unifiers. For example:

$$(\!| \alpha :: \mathit{Int} \mid \beta |\!) \ \sim\ (\!| x :: \mathit{Int}, y :: \mathit{Int} |\!)$$

For this expression, there are two unifiers that can not be written in terms of each other, namely:

$$[\alpha \mapsto x, \beta \mapsto (\!| y :: \mathit{Int} |\!)] \quad \text{and} \quad [\alpha \mapsto y, \beta \mapsto (\!| x :: \mathit{Int} |\!)]$$

The lack of unique most general unifiers has profound implications with respect to efficient algorithms for simplifying predicates, and we discuss the unification of rows in more detail in the next section.

8 Simplification and improvement

By restricting row expressions to predicates, we have nicely sidestepped the problem of row unification in our calculus. Furthermore, we can use the standard inference algorithm for qualified types to infer principal types. Unfortunately, while technically correct, the principal types do not take account of satisfiability of predicates. As a result, the inferred types are not always as accurate or simple as we may expect. Jones describes a refined inference algorithm that takes satisfiability into account [10]. The new algorithm adds two new rules: one for *simplification* and one for *improvement* of predicates.

8.1 Simplification

Simplification allows us to replace a set of predicates in a type by another set of equivalent predicates. This can be used for example to discharge constant predicates. For example, the predicate $(\!| x :: \mathit{Int} \mid r |\!) \backslash y$, can be simplified to the equivalent predicate $r \backslash y$, when $x \neq_c y$. We write $P \Leftrightarrow Q$ when P and Q entail each other, i.e. $P \Vdash Q$ and $Q \Vdash P$. The simplification rule is:

$$\frac{P \mid S \mid A \vdash^{\mathsf{w}} e : \tau \qquad P \Leftrightarrow Q}{Q \mid S \mid A \vdash^{\mathsf{w}} e : \tau}$$

The addition of this rule makes the algorithm nondeterministic, but the algorithm is sound as the inferred types are equivalent [10]. In practice, we apply this rule at generalisation. The rule is also used to detect type errors: during simplification, we can detect unsatisfiable constraints and emit an appropriate error message.

The algorithm for simplication in Morrow simplifies constraints according to the entailment rules in Figure 1. Full simplification of *lacks* constraints leads to constant evidence, while partial simplification of *lacks* constraints introduces a coercion term that possibly increases the passed evidence.

8.2 Improvement

The *improvement* rule allows us to apply a substitution to the predicates, as long as the substitution does not change the *satisfiable instances* of a type. For example, the expression $\{x = 2\}.x$ has the following principal type:

$$\forall rsa.\ (s\backslash x,\ r \sim (\!|x :: Int|\!),\ r \sim (\!|x :: a \mid s|\!)) \Rightarrow a$$

Although correct, in a practical system we would rather infer the *principal satisfiable type*. If we would unify the rows, we find an *improving substitution*, namely $[a \mapsto Int, s \mapsto (\!|\,|\!), r \mapsto (\!|x :: Int|\!)]$. When we apply this substitution and use simplification, we get the (expected) principal satisfiable type: *Int*.

We write $\lfloor P \rfloor_{P_0}$ for the set of satisfiable instances of a predicate set P with respect to an initial predicate set P_0. It is defined as:

$$\lfloor P \rfloor_{P_0} \ = \ \{SP \mid S \in Subst,\ P_0 \Vdash SP\}$$

For the purposes of this article, the initial predicate set P_0 is empty, and we will not write the subscript explicitly. We say that a substitution S *improves* P if $\lfloor P \rfloor = \lfloor SP \rfloor$, and when the free variables in S that do not appear in P are 'fresh'. The inference algorithm is extended with the following improvement rule:

$$\frac{P \mid S \mid A \vdash^{\mathsf{w}} e : \tau \qquad T \ improves \ P}{TP \mid TS \mid TA \vdash^{\mathsf{w}} e : T\tau}$$

Jones proves that the inference algorithm is still sound and complete when the simplification and improvement rules are added [10]. However, the completeness of the algorithm with respect to the typing rules is now defined in terms of *principal satisfiable types*. The inference algorithm will not always find the most general type of an expression, but it *will* find a most general type that has satisfiable instances (if it exists).

Note that it is not a requirement to find optimal improvements, even though it is desirable in practice. This provides us with a flexible design space for exploring efficient algorithms for improving row equality predicates. We parameterize our inference algorithm with an improving function *impr* such that *impr*(P) *improves* P for any predicate set P. A trivial instance always returns the identity substitution, *impr*$(P) = id$. In the next subsection, we discuss a more sophisticated algorithm that is used by Morrow and that finds nearly optimal improvements.

$$(\!|\,|\!) \overset{\{id\}}{\sim} (\!|\,|\!) \qquad \frac{\rho \notin ftv(r)}{\rho \overset{\{[\rho \mapsto r]\}}{\sim} r} \qquad \frac{\rho \notin ftv(r)}{r \overset{\{[\rho \mapsto r]\}}{\sim} \rho}$$

$$\frac{(l : \tau) \overset{\Theta}{\in} s \qquad \Theta'' = \{\theta'\theta \mid \theta \in \Theta,\ \theta r \overset{\Theta'}{\sim} \theta s - \theta l,\ \theta' \in \Theta'\}}{(\!|l : \tau \mid r|\!) \overset{\Theta''}{\sim} s}$$

Figure 5: Row unifiers

$$(insVar) \qquad \frac{fresh(\beta) \qquad \alpha \notin ftv(\tau) \qquad \theta = [\alpha \mapsto (\!|l :: \tau \mid \beta|\!)]}{(l :: \tau) \overset{\{\theta\}}{\in} \alpha}$$

$$(insHead) \qquad \frac{(l : \tau) \overset{\Theta}{\sim} (l' : \tau') \qquad (l :: \tau) \overset{\Theta'}{\in} r}{(l :: \tau) \overset{\Theta \cup \Theta'}{\in} (\!|l' : \tau' \mid r|\!)}$$

Figure 6: Inserters

8.3 Row unification

Improvement in Morrow depends essentially on *row unification*. As shown before, rows with label variables do not possess a most general unifier. We define a unification algorithm between rows that derives a set of most general unifiers. Of course, these unifiers are not 'most general' in the usual sense, but only most general for a fixed permutation of row fields.

Figure 5 shows the unification algorithm for rows. The expression $r \overset{\Theta}{\sim} r'$ finds the set Θ of most general unifiers between two rows r and r'. Single substitutions are written as θ. Instead of failing when no rule applies, we assume that row unification returns the empty set $\varnothing$ when no unification can be found. The first three unification rules are standard. The last rule finds all *inserters* of a field in a row, and returns the cartesian product of the unifiers of the inserters with the unifiers of the row tails.

The algorithm for finding inserters is defined in Figure 6 and uses the field unification algorithm shown in Figure 7. The rule (*insVar*) is standard. The rule (*insHead*) takes the union between the unifiers of the head field and the inserters of the tail. Note that field unification returns either a singleton set or an empty set of unifiers.

Using the unification of rows, we can now define an improving function. Figure 8 defines an improving function *impr*(P) that returns an improving substitution θ for the predicates P. We write $P^{\sim}$ for the predicates P restricted to equality predicates, and $P^{\backslash}$ for the restriction to lacks predicates. The function *unis* re-

$$(\mathit{fieldEq}) \qquad \frac{\tau \overset{\theta}{\sim} \tau'}{(l:\tau) \overset{\{\theta\}}{\sim} (l:\tau')}$$

$$(\mathit{fieldR}) \qquad \frac{\theta = [\alpha \mapsto l] \qquad \theta\tau \overset{\theta'}{\sim} \theta\tau'}{(l:\tau) \overset{\{\theta'\theta\}}{\sim} (\alpha:\tau')}$$

$$(\mathit{fieldL}) \qquad \frac{\theta = [\alpha \mapsto l] \qquad \theta\tau \overset{\theta'}{\sim} \theta\tau'}{(\alpha:\tau) \overset{\{\theta'\theta\}}{\sim} (l:\tau')}$$

Figure 7: Field unification

$$(\mathit{impr}) \qquad \frac{\mathit{unis}(P^{\sim}) = \Theta \qquad \mathit{single}(\{\theta \mid \theta \in \Theta,\ P_0 \Vdash \theta P^{\setminus}\}) = \theta'}{\mathit{impr}(P) = \theta'}$$

$$(\mathit{unis1}) \qquad \mathit{unis}(\varnothing) = \{\mathit{id}\}$$

$$(\mathit{unis2}) \qquad \frac{r \overset{\Theta}{\sim} r' \qquad \Theta'' = \{\theta'\theta \mid \theta \in \Theta,\ \mathit{unis}(\theta P) = \Theta',\ \theta' \in \Theta'\}}{\mathit{unis}(\{r \sim r'\} \cup P) = \Theta''}$$

Figure 8: Improvement

turns the cartesian product of row equality unifications. In the rule (*impr*), the set of all row equality unifiers Θ is further restricted to those unifiers that satisfy the lacks constraints.

The function *single* reduces the resulting set of unifiers to a single improving substitution. When the unifier set is empty, the predicates are not satisfiable and an error message is emitted. When the unifier set consists of a single unifier, we can unambigiously improve the predicate set. A possible definition of *single* therefore only succeeds when the choice is unambigious; $\mathit{single}(\{\theta\}) = \theta$.

If more than one unifier is returned, there may still be improving substitutions. In Morrow, we calculate the greatest common unifier of all the substitutions: $\mathit{single}(\Theta) = \mathit{gcu}(\Theta)$. In order to calculate the greatest common unifier efficiently, we need to be careful to construct only normalized substitutions during unification. For example, we take the order on type variables into account to only produce variable substitutions of the form $[\alpha \mapsto \beta]$ instead of $[\beta \mapsto \alpha]$; However, a full discussion is beyond the scope of this paper.

8.4 Complexity of improvement

The previous section developed an algorithm to calculate improving substitutions in the presence of row equality predicates. Unfortunately, the algorithm is also exponential in the number of polymorphic labels! This should not come as a surprise: Sulzmann shows that even for an weaker system of first-class labels, satisfiability is NP-complete [29]. The worst case of our algorithm is formed by a row of n polymorphic labels with polymorphic types that is unified with a row of constant labels. Each of the n polymorphic labels can unify with any of the n constant labels, and we find $n!$ total unifiers.

However, we feel that there are some mitigating factors. First of all, even though it is desirable to find 'optimal' improvements, this is not a requirement of the calculus. A practical implementation could resort to a more naïve improvement algorithm if the optimal one takes too long. Secondly, in practice, the algorithm can be made more efficient by doing unambigious insertions first. Morrow first transforms all the equality predicates to inserters and performs substitutions until there are no more unambigious choices, before resorting to a brute-force enumeration of unifiers. Finally, just like in normal Hindley-Milner type inference, practical experience shows that the worst case does not seem to appear in normal programs. Even when it occurs, the number of labels tends to be low enough to be able to solve the improvement fast. However, experience with larger programs is necessary to validate this claim.

9 Related work

Subtyping is one of the earliest approaches to record type systems [1, 20]. Predicates include a subtype relation on rows, such that a row r is a subrow of r' if r includes all fields of row r'. The type of selection is:

$$\forall r\, a.\ (r \leqslant \{l :: a\}) \Rightarrow r \to a$$

Unfortunately, with this approach information about the other fields of a row is lost, which makes it hard to describe operations like row extension. Cardelli and Mitchell [1] especially introduce an overriding operator on types to overcome this problem. Wand [31, 32] was the first to introduce row variables to capture row subtyping explicitly through parametric polymorphism. In his system, the type of selection is:

$$\forall r\, a.\ \{l :: a \mid r\} \to a$$

The calculus does not impose constraints on record extension though, and as a result, not all programs have a principal type [31]. Remy [21] extended the work of Wand with constrained record operations. His calculus contains flags that denote the presence or absence of certain fields. The type of selection becomes:

$$\forall r\, a.\ \{l :: \mathit{pre}(a) \mid r\} \to a$$

The system of Remy does not lead directly to an efficient or simple compilation method though. Ohori [18] was the first to present a polymorphic record system that had a simple and efficient compilation scheme, but it can not handle extensible row operations.

Gaster and Jones [5, 4] presented a polymorphic type system for extensible records and variants that was based on the theory of qualified types [8, 9]. This system has a simple and efficient compilation method and is implemented as the TREX extension to Hugs.

Type rules for record concatenation have been proposed too [7, 32], where Sulzmann presents an effective type inference method [27]. However, no efficient compilation method for concatenation has been published.

First-class labels

Gaster describes first-class labels in his thesis [4] but does not introduce predicates beyond lacks predicates. As a result, not all programs can be assigned a (principal) type. Sulzmann [29, 28] introduces first-class labels as an instance of HM(X) [17] and his system enjoys principal types as it uses *has* constraints (instead of just *lacks* constraints). Unfortunately, it is not obvious how to extend this system with polymorphic row extension, as it would prevent resolution of equality predicates [29]. This is one of the main reasons to base our work on the more general theory of qualified types [9] instead, which gives us a flexible design space with respect to improvement of equality predicates. Another reason is that a system based on HM(X) has no general compilation method, whereas we can use standard evidence translation.

Shields and Meijer introduce an extremely expressive row calculus, called λ^{tir}, where types themselves are used as indices in the rows [25, 24]. The λ^{tir} calculus is strictly more general than our system; our calculus can be encoded in λ^{tir} using *opaque* newtypes to express first-class labels. However, to avoid the exponential behaviour of our improvement algorithm, λ^{tir} uses a restricted polynomial improvement algorithm. Due to the lack of labels, this is more or less essential for λ^{tir} as there are much more ambigious unifications in practical programs. As a result of restricted improvement, λ^{tir} can not always infer that a given type signature entails an inferred type.

Our work is strongly motivated by λ^{tir}, where we wanted to develop a simpler calculus based on the theory of qualified types. By using explicit labels that are not part of the type language, we explored a useful alternative point in the design space, where we can express many of the motivating examples of λ^{tir} while using standard theory.

10 Conclusion

We have described a sound and complete system of polymorphic extensible records and variants with first-class labels. We have shown how first-class labels increase expressiveness significantly, and are able to express intersection types, closed-world overloading, and type selective functions. The type inference algorithm is fully implemented in Morrow [12], including discussed extensions such as row polymorphic operations.

In the future, we expect to try the inferencer on larger programs. In particular, we would like to investigate the use of higher-rank types to express first-class modulesand we would like to experiment with first-class patterns to encode views [30].

Acknowledgements

I would like to thank Bastiaan Heeren, Mark Shields, and Frank Atanassow for their valuable comments on the draft paper. Furthermore, I thank Andres Löh, Erik Meijer, Martijn Schrage, and Arjan van IJzendoorn for their remarks and interesting discussions about the design of Morrow.

References

[1] L. Cardelli and J. Mitchell. Operations on records. *Journal of Mathematical Structures in Computer Science*, 1(1):3–48, Mar. 1991.

[2] J. Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *Proceedings of the JSSST Workshop on Programming and Programming Languages*, Mar. 2004.

[3] J. Garrigue and H. Aït-Kaci. The typed polymorphic label selective calculus. In *21th ACM Symp. on Principles of Programming Languages (POPL'94)*, pages 35–47, Portland, OR, Jan. 1994.

[4] B. R. Gaster. *Records, Variants, and Qualified Types.* PhD thesis, Dept. of Computer Science, University of Nottingham, July 1998.

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

[6] T. Gilliam and T. Jones. Monty Python and the Holy Grail. By G. Chapman and J. Cleese, 1975.

[7] R. Harper and B. C. Pierce. A record calculus based on symmetric concatenation. In *18th ACM Symp. on Principles of Programming Languages (POPL'91)*, pages 131–142, Jan. 1991.

[8] M. P. Jones. A theory of qualified types. In *4th. European Symposium on Programming (ESOP'92)*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, Feb. 1992.

[9] M. P. Jones. *Qualified types in Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[10] M. P. Jones. Simplifying and improving qualified types. Technical Report YALEU/DCS/RR-1040, Dept. of Computer Science, Yale University, 1994.

[11] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, Jan. 1995.

[12] D. Leijen. Morrow: a row-oriented programming language. `http://www.cs.uu.nl/~daan/morrow.html`, July 2004.

[13] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct. 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[14] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: dynamic scoping with static types. In *27th ACM Symp. on Principles of Programming Languages (POPL'00)*, pages 108–118, Boston, Massachussets, Jan. 2000.

[15] M. Müller and S. Nishimura. Type inference for first-class messages with feature constraints. *International Journal of Foundations of Computer Science*, 11(1):29– 63, 2000.

[16] S. Nishimura. Static typing for dynamic messages. In *25th ACM Symp. on Principles of Programming Languages (POPL'98)*, pages 266–278, 1998.

[17] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

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

[19] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, Apr. 1997.

[20] B. C. Pierce and D. N. Turner. Simple type theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994.

[21] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[22] J. C. Reynolds. Design of the programming language Forsythe. In P. O'Hearn and R. Tennent, editors, *ALGOL-like languages*, pages 173–233. Birkhäuser, 1997.

[23] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan. 1965.

[24] M. Shields. *Static Types for Dynamic Documents*. PhD thesis, Oregon Graduate Institute, Feb. 2001.

[25] M. Shields and E. Meijer. Type indexed rows. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 261–275, London, England, Jan. 2001.

[26] P. Shroff and S. F. Smith. Type inference for first-class messages with match-functions. In *Foundations of Object-Oriented Languages*, Jan. 2004.

[27] M. Sulzmann. Designing record systems. Technical Report YALEU/DCS/RR-1128, Dept. of Computer Science, Yale University, Apr. 1997.

[28] M. Sulzmann. Type systems for records revisited. Unpublished report, June 1998.

[29] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Dept. of Computer Science, Yale University, May 2000.

[30] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14th ACM Symp. on Principles of Programming Languages (POPL'87)*, pages 307–313. ACM Press, 1987.

[31] M. Wand. Complete type inference for simple objects. In *Proceedings of the 2nd. IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in LICS'88, page 132.

[32] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.