Extensible records with scoped labels

Daan Leijen

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands
daan@cs.uu.nl
Draft, Revision: 76 , July 23, 2005

Abstract

Records provide a safe and flexible way to construct data structures. We describe a natural approach to typing polymorphic and extensible records that is simple, easy to use in practice, and straightforward to implement. A novel aspect of this work is that records can contain duplicate labels, effectively introducing a form of scoping over the labels. Furthermore, it is a fully orthogonal extension to existing type systems and programming languages. In particular, we show how it can be used conveniently with standard Hindley-Milner, qualified types, and MLF.

1. Introduction

Tuples, or products, group data items together and are a fundamental concept to describe data structures. In ML and Haskell, we can construct a product of three integers as:

$$(7, 7, 1973)$$

Records are tuples where the individual components are labeled. Using curly braces to denote records, we can write the above product more descriptively as:

$$\{day = 7, month = 7, year = 1973\}$$

The record notation is arguably more readable than the plain product. It is also safer as we identify each component explicitly, preventing an accidental switch of the day and month for example.

Even though records are fundamental building blocks of data structures, most programming languages severely restrict their use: labels can not be reused at different types, records must be explicitly declared and are not extensible, etc. This is surprising given the large amount of research that has gone into type systems and compilation methods for records. We believe that the complexity of the proposed systems is one of the most important reasons that they are not yet part of mainstream programming languages. Most systems require non-trivial extensions to a type system that are hard to implement, and, perhaps even more important, that are difficult to explain to the user.

For all systems described in literature, it is assumed that records do not contain duplicate labels. In this article we take a novel view at records where duplicate labels are allowed and retained, effectively introducing a form of scoping over the labels. This leads to a simple and natural system for records that integrates seamlessly with many other type systems. In particular:

- The types are straightforward and basically what a naïve user would expect them to be. The system is easy to use in practice, as the user is not confronted with artificial type system constructs. Of course, all operations are checked and the type system statically prevents access to labels that are absent.
- The records support scoped labels since fields with duplicate labels are allowed and retained. As records are equivalent up to permutation of distinct labels, all basic operations are still well-defined. The concept of scoped labels is useful in its own right and can lead to new applications of records in practice.
- The system is straightforward to implement using a wide range of implementation techniques. For predicative type systems, we can guarantee constant-time field selection.
- The system works with just about any polymorphic type system with minimal effort. We only define a new notion of equality between (mono) types and present an extended unification algorithm. This is all completely independent of a particular set of type rules. We show how it can be used specifically with MLF [15], a higher-ranked, impredicative type system. Building on MLF, we can model a form of first-class modules with records.

The entire system is implemented in the experimental language Morrow [16]. The type system of Morrow is based on MLF, and all the examples in this article, including the first-class modules, are valid Morrow programs.

The work described here builds on numerous other proposals for records, in particular the work of Wand [35], Remy [29], and Gaster and Jones [7]. One can view our work as just a small variation of the previous systems. However, we believe that our design is an important variation, as it leads to a record system with much less complexity. This makes our design more suitable for integration with existing type systems and programming languages.

In the next section we introduce the basic record operations. We explain the type rules and discuss what effect scoped labels have on programs. In Section 4 we show how our system can be used with MLF to encode a form of first-class modules. In Section 5 we discuss how our systems can also supports variant types. We formalize the type rules and inference in section 6 and 7. We conclude with an overview of implementation techniques and related work.

2. Record operations

Following Cardelli and Mitchell [2] we define three primitive operations on records: *selection*, *restriction*, and *extension*. Furthermore, we add the constant $\{\}$ as the empty record.

Extension. We can extend a record r with a label l and value e using the syntax $\{l = e \mid r\}$. For example:

$$origin = \{x = 0 \mid \{y = 0 \mid \{\}\}\}$$

To reduce the number of braces, we abbreviate a series of extensions using comma separated fields, and we leave the extension of

the empty record implicit. The above example can thus be written more conveniently as:

$$origin = \{x = 0, y = 0\}$$

The construction of the record is anonymous: we do not have to declare this record or its fields in advance. Furthermore, extension is polymorphic and not limited to records with a fixed type, but also applies to previously defined records, or records passed as an argument:

$$\begin{array}{ll} origin3 & = \{z = 0 \mid origin\} \\ named\ s\ r & = \{name = s \mid r\} \end{array}$$

Selection. The selection operation $(r.l)$ selects the value of a label l from a record r. For example, we can define a function $distance$ that calculates the distance of a point to the origin:

$$distance\ p = sqrt\ ((p.x * p.x) + (p.y * p.y))$$

In contrast to many programming languages, the $distance$ function works for any record that contains an x and y field of a suitable numeric type. For example, we can use this function on records with a different set of fields:

$$distance\ (named\ \texttt{"2d"}\ origin) + distance\ origin3$$

Restriction. Finally, the restriction operation $(r - l)$ removes a label l from a record r. Using our primitive operations, we can now define the common *update* and *rename* operations:

$$\begin{array}{lll} \{l := x \mid r\} & = & \{l = x \mid r - l\} \quad \text{-- update } l \\ \{l \leftarrow m \mid r\} & = & \{l = r.m \mid r - m\} \quad \text{-- rename } m \text{ to } l \end{array}$$

Here is an example of using update to change the x and y components of a point:

$$move\ p\ dx\ dy = \{x := p.x + dx,\ y := p.y + dy \mid p\}$$

Note that $move$ works on any record containing an x and y field, not just points. Effectively, we use parametric polymorphism to model a limited form of subtyping [2].

2.1 Safe operations

The type system ensures statically that all record operations are safe. In particular, it ensures that record selection and restriction are only applied when the field is actually present. For example, the following expressions are both rejected by the type system:

$$\{x = 1\}.y$$
$$distance\ \{x = 1\}$$

Our type system accepts the extension of a record with a field that is already present, and the following example is accepted:

$$\{x = 1 \mid origin\}$$

We call this *free* extension. Many type systems in the literature require that a record can only be extended with a label that absent, which we call *strict* extension. We believe that strict extension unnecessarily restricts the programs one can write. For example, the function $named$ extends *any* record with a new $name$ field. In a system with strict extension, we need to write two functions: one for records without the label, and one for records that already contain the label. In this particular example this is easy to do, but in general we might want to extend records locally with helper fields. Without free extension, the local extensions would artificially restrict the use of the function.

There are two possible semantics we can give to free extension. If a duplicate label is encountered we can choose to overwrite the previous field with the new field, or we can choose to retain the old field. All previous proposals that allow free extension [35, 29, 1] use the first approach. In those systems, extension is really a mixture of update and extension: if a field is absent, the record is extended. If the field is already present, the previous value is overwritten, after which it is no longer accessible.

We take a novel approach to free extension where the previous fields are always retained, both in the value and in the type. In our system, we clearly separate the concepts of update and extension. To keep selection and restriction well-defined, we need to explicitly define these operations to work on the first matching label in a record. Therefore, we can always unambiguously select a particular label:

$$\begin{array}{ll} \{x = 2, x = True\}.x & \text{-- select the first } x \text{ field} \\ (\{x = 2, x = True\} - x).x & \text{-- select the second } x \text{ field} \end{array}$$

Since previous fields are retained, our record system effectively introduces a form of scoping on labels. This is certainly useful in practice, where we can use scoped labels to model environments with access to previously defined values. For example, suppose we have an environment that includes the current text color:

$$putText\ env\ s = putStr\ (ansiColor\ env.color\ s)$$

We can define a combinator that temporarily changes the output color:

$$warning\ env\ f = f\ \{color = red \mid env\}$$

The function f passed to $warning$ formats its output in a red color. However, it may want to format certain parts of its output in the color of the parent context. Using scoped labels, this is easily arranged: we can remove the first color field from the environment, thereby exposing the previous color field automatically (if present):

$$f\ env = putText\ (env - color)\ \texttt{"parent color"}$$

As we see in the next section, the type of the function f reflects that the environment is required to contain at least two $color$ fields.

Another example of scoped labels occurs when encoding objects as records. Redefined members in a sub-class are simply extensions of the parent class. The scoped labels can now be used to access the overridden members in a parent class.

One can argue that free extension can lead to programming errors where one accidentally extends a record with a duplicate label. However, the type system can always issue a warning if a record with a fixed type contains duplicate labels, which could be attributed to a programmer mistake. This is comparable to a standard shadowed variable warning – and indeed, a warning is more appropriate here than a type error, since a program with duplicate labels can not go wrong!

3. The types of records

We write the type of a record as a sequence of labeled types. To closely reflect the syntax of record values, we enclose record types in curly braces $\{\}$ too:

$$\mathbf{type}\ Point = \{x :: Int, y :: Int\}$$

As we will see during the formal development, it makes sense to talk about a sequence of labeled types as a separate concept. We call such sequence a *row*. Following Gaster and Jones [7], 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

the extension of a row r with a label l and type τ is written as $(\!|\, l :: \tau \mid r \,|\!)$. The full unabbreviated type of a $Point$ is written with rows as:

$$\textbf{type}\ Point = \{(\!|x :: Int \mid (\!|y :: Int \mid (\!|\,|\!)\,|\!)\,|\!)\}$$

Just like record extension, we abbreviate multiple extensions with a comma separated list of fields. Furthermore, we leave out the row brackets if they are directly enclosed by record braces.

3.1 Types of record operations

Using row types, we can now give the type signatures for the basic record operations:

$$\begin{array}{ll} (\_.l) & :: \forall r\alpha.\ \{l :: \alpha \mid r\} \to \alpha \\ (\_ - l) & :: \forall r\alpha.\ \{l :: \alpha \mid r\} \to \{r\} \\ \{l = \_ \mid \_\} & :: \forall r\alpha.\ \alpha \to \{r\} \to \{l :: \alpha \mid r\} \end{array}$$

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 selection operator $(\_.l)$ takes a record that contains a field l of type α, and returns the value of type α. Similarly, the restriction operator $(\_ - l)$ returns the record without the l field. The type of extension is very natural: it takes a value α and any record $\{r\}$, and extends it with a new field $l :: \alpha$. Here is for example the inferred type for $origin$:

$$\begin{array}{l} origin :: \{x :: Int, y :: Int\} \\ origin = \{x = 0, y = 0\} \end{array}$$

The type of selection naturally ensures that a label is present when it is selected. For example, $origin.x$ is well-typed, since the type of the record, $\{x :: Int, y :: Int\}$, is an instance of the type of the expected argument $\{x :: \alpha \mid r\}$ of the selector function $(\_.x)$. Unfortunately, at this point, the type signatures are too strong: the valid expression $origin.y$ is still rejected as $\{x :: Int, y :: Int\}$ is just not an instance of $\{y :: \alpha \mid r\}$.

To accept the above selection, we need a new notion of equality between types where the rows are considered equal *up to permutation of distinct labels*. The new equality relation $(\cong)$ is formalized in Figure 1. The first three rules are standard. Rule $(eq\text{-}trans)$ defines equality as a transitive relation. The last two rules define equality between rows. Rule $(eq\text{-}head)$ defines two rows as equal when their heads and tails are equal. The rule $(eq\text{-}swap)$ is the most interesting: it states that the first two fields of a row can be swapped if (and only if) their labels are different. Together with transitivity $(eq\text{-}trans)$ and row equality $(eq\text{-}head)$, this effectively allows us to swap a field repeatedly to the front of a record, but not past an equal label. With the new notion of equality, we can immediately derive that:

$$\{x :: Int, y :: Int\} \cong \{y :: Int, x :: Int\}$$

The expression $origin.y$ is now well-typed since the isomorphic type $\{y :: Int, x :: Int\}$ is an instance of $\{y :: \alpha \mid r\}$. The new notion of equality is the only addition needed to integrate our notion of records with a specific type system. Since no other concepts are introduced, the types of the primitive operations are basically what a naïve user would expect them to be. The same holds for the inferred types of derived operations such as update and rename:

$$\begin{array}{l} \{l := \_ \mid \_\} :: \forall r\alpha\beta.\ \alpha \to \{l :: \beta \mid r\} \to \{l :: \alpha \mid r\} \\ \{l := x \mid r\} = \ \{l = x \mid r - l\} \end{array}$$

$$\begin{array}{l} \{l \leftarrow m \mid \_\} :: \forall r\alpha.\, \{m :: \alpha \mid r\} \to \{l :: \alpha \mid r\} \\ \{l \leftarrow m \mid r\} = \{l = r.m \mid r - m\} \end{array}$$

$$\begin{array}{ll} (eq\text{-}var) & \alpha \cong \alpha \\ (eq\text{-}const) & c \cong c \\ (eq\text{-}app) & \dfrac{\tau_1 \cong \tau_1' \quad \tau_2 \cong \tau_2'}{\tau_1\ \tau_2 \cong \tau_1'\ \tau_2'} \\ (eq\text{-}trans) & \dfrac{\tau_1 \cong \tau_2 \quad \tau_2 \cong \tau_3}{\tau_1 \cong \tau_3} \\ (eq\text{-}head) & \dfrac{\tau \cong \tau' \quad r \cong s}{(\!|\, l :: \tau \mid r \,|\!) \cong (\!|\, l :: \tau' \mid s \,|\!)} \\ (eq\text{-}swap) & \dfrac{l \neq l'}{(\!|\, l :: \tau, l' :: \tau' \mid r \,|\!) \cong (\!|\, l' :: \tau', l :: \tau \mid r \,|\!)} \end{array}$$

Figure 1. Equality between (mono) types

We see that the type of update is very natural: given a record with an l field of type β, we can assign it a new value of a possibly different type α.

3.2 Scoped labels

As remarked before, the type signature for record extension is free and does not reject duplicate labels. For example, both of the following expressions are well-typed:

$$\begin{array}{l} \{x = 2, x = True\} \ :: \{x :: Int, x :: Bool\} \\ \{x = True, x = 2\} \ :: \{x :: Bool, x :: Int\} \end{array}$$

Note that the types of the two expressions are not equivalent though. Since rule $(eq\text{-}swap)$ only applies to distinct labels, selection and restriction are still well-defined operations. For example, the following expression selects the second field, as signified by the derived type:

$$(\{x = 2, x = True\} - x).x \ \ :: Bool$$

This example shows that it is essential to retain duplicate fields not only in the runtime value, but also in the static type of the record.

3.3 Related designs

In this section we briefly compare our records with closely related systems. The main point that we wish do demonstrate here is how other designs use types that are harder to understand, or lead to complicated implementations. A more complete overview of related work is deferred to Section 9.

Predicates. Gaster and Jones [7] presented an elegant type system for records and variants based on the theory of qualified types [10, 11]. Special *lacks* predicates are used to prevent records from containing duplicate labels. For example, extension is only valid on records that do not yet contain that label, and is thus qualified with a lacks predicate:

$$\{l = \_ \mid \_\} :: \forall r\alpha.\ (r \backslash l) \Rightarrow \alpha \to \{r\} \to \{l :: \alpha \mid r\}$$

As we have argued before, strict extension unnecessarily restrict the programs we can write, and one can view the lacks predicate as an unnecessary part the type. Also, each use of a label leads to a lacks predicate, which in turn can lead to large types that are hard to read. Furthermore, the system relies essentially on a type system that supports qualified types. Adding a fundamental extension like qualified types to an ML-like language just to support records could be hard to justify in practice.

Flags. Remy [29] developed a flexible system of extensible records where flags are used to denote the presence or absence of labels and rows. The type of free extension becomes:

$$\{l = \_ \mid \_\} :: \forall r\varphi\alpha.\ \alpha \to \{l :: \varphi \mid r\} \to \{l :: pre(\alpha) \mid r\}$$

The type variable φ ranges over field types that can be either absent, written as abs, or present with type τ, written as $pre(\tau)$. By being polymorphic in the presence or absence of the field l, the type of extension encodes that a field that is already present is overwritten, while an absent field becomes present. As we can see here, such type would not be obvious to the naïve user; especially since absent labels in the value can be present in the type (with an absent flag abs). We also believe that the semantics of extension should not be a subtle mixture of update and extension. Indeed, a system based on flags can not give a type to our notion of proper free extension since it essentially views records as total functions from labels to values, where duplicate labels are certainly not allowed.

Row extension. Wand [35, 36] was the first to introduce row variables, and described a type system that is very close to our proposal. The types of the basic operations are given as type rules, but these rules have basically the same effect as the type signatures in our system. However, this system also uses an ambiguous interpretation of free extension as a mixture of update and extension. This gives rise to fundamental problems in the type system. Take for example the following expression:

$$\backslash r \to \mathbf{if}\ \mathit{True}\ \mathbf{then}\ \{x = 1 \mid r\}\ \mathbf{else}\ \{x = 2 \mid \{\}\}$$

If extension overwrites previous fields, we can assign two different types to r, namely $\{x :: \mathit{Int}\}$ or the empty record $\{\}$. This is why Wand's original record system proved incomplete [35]. With scoped labels, the choice is unambiguous and r must have type $\{\}$. We show completeness (and soundness) of type inference formally in Section 7.

4. Modules

In this section, we give some examples how our record system can be used to encode (a form of) first-class modules and objects.

4.1 Polymorphic records

Since the type signatures for record operations are so general, we can conveniently package related functions together. Taking an example from Jones [14], we can write a type signature for complex numbers that is parameterized over the implementation type cpx:

$$\begin{array}{l}\mathbf{type}\ \mathit{Complex}\ cpx = \{\mathit{cart} \ :: \mathit{Float} \to \mathit{Float} \to cpx \\ \qquad\qquad\qquad\quad , \mathit{polar} :: \mathit{Float} \to \mathit{Float} \to cpx \\ \qquad\qquad\qquad\quad , re \quad :: cpx \to \mathit{Float} \\ \qquad\qquad\qquad\quad , im \quad :: cpx \to \mathit{Float} \\ \qquad\qquad\qquad\quad ...\}\end{array}$$

Two obvious ways to implement complex numbers are using pairs of floating point numbers with either a cartesian or a polar representation:

$$\begin{array}{rl}\mathit{cartCpx} = \{\mathit{cart}\ x\ y & = (x, y) \\ , \mathit{polar}\ r\ a & = (r * \mathit{cos}\ a, r * \mathit{sin}\ a) \\ , re\ (x, y) & = x \\ ...\}\end{array}$$

$$\begin{array}{rl}\mathit{polarCpx} = \{\mathit{cart}\ x\ y & = (\mathit{sqrt}\ (x * x + y * y), \mathit{atan}_2\ y\ x) \\ , \mathit{polar}\ r\ a & = (r, a) \\ , re\ (r, a) & = r * \mathit{cos}\ a \\ ...\}\end{array}$$

As convenient syntactic sugar, we abbreviate the binding of a functional value $(l = \backslash x_1\ ...\ x_n \to e)$ as $(l\ x_1\ ...\ x_n = e)$. Of course, the types of both implementations are the same, namely $\mathit{Complex}\ (\mathit{Float}, \mathit{Float})$. The ability to treat the type of a complex number representation polymorphically, allows us to create new polymorphic records from any complex number. Take for example the following simple definition of an arithmetic package over elements of type α:

$$\begin{array}{l}\mathbf{type}\ \mathit{Arith}\ \alpha = \{\mathit{plus} :: \alpha \to \alpha \to \alpha \\ \qquad\qquad\qquad\quad , \mathit{zero} :: \alpha\}\end{array}$$

Relying on standard Hindley-Milner polymorphism, we can construct an arithmetic package from any complex number type:

$$\begin{array}{l}\mathit{arithCpx} :: \forall\alpha.\ \mathit{Complex}\ \alpha \to \mathit{Arith}\ \alpha \\ \mathit{arithCpx}\ c = \{\mathit{plus}\ x\ y = c.\mathit{cart}\ (c.re\ x + c.re\ y) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (c.im\ x + c.im\ y) \\ \qquad\qquad , \mathit{zero} \quad\ = c.\mathit{cart}\ 0\ 0\}\end{array}$$

The last definition corresponds closely to a functor definition in SML. However, no special syntax or constructions are necessary and everything is done using standard record operations and normal functions. As noted by Jones [14] the use of the type variable α intuitively corresponds to a sharing specification in SML.

The above examples are relatively simple as the records do not contain polymorphic functions themselves. Things become more interesting when we define a signature for a $\mathit{Prelude}$ package for example:

$$\begin{array}{l}\mathbf{type}\ \mathit{Prelude} = \{\mathit{id} \quad\ :: \forall\alpha.\ \ \alpha \to \alpha \\ \qquad\qquad\qquad\quad , \mathit{const} :: \forall\alpha\beta.\ \alpha \to \beta \to \alpha \\ \qquad\qquad\qquad\quad , \mathit{zip} \quad :: \forall\alpha\beta.\ [\alpha] \to [\beta] \to [(\alpha, \beta)] \\ \qquad\qquad\qquad\quad ...\}\end{array}$$

In this signature, the types of id and $const$ use impredicative higher-rank polymorphism since the quantifiers are nested inside the record structure, and type inference for impredicative rank-n polymorphism is a notoriously hard problem [15, 25, 21].

4.2 Higher-ranked impredicative records

When we move to more complicated type systems, our framework of records proves its value, since it only relies on a new notion of equality between (mono) types and no extra type rules are introduced. This means that it becomes relatively easy to add our system to just about any polymorphic type system. In particular, it integrates seamlessly with MLF, an elegant impredicative higher-ranked type inference system by Le Botlan and Remy [15]. We have a full implementation of this system in the experimental Morrow compiler [16] and all the examples in this article are valid Morrow programs.

The combination of MLF with anonymous polymorphic and extensible records (and variants) leads to a powerful system where fields can have full polymorphic type signatures. For example, we can now implement our previous $\mathit{Prelude}$ example as a first-class structure:

$$\begin{array}{l}\mathit{prelude} :: \mathit{Prelude} \\ \mathit{prelude} = \{\mathit{id}\ x \quad\ = x \\ \qquad\qquad , \mathit{const}\ x\ y = x \\ \qquad\qquad ...\}\end{array}$$

Since Morrow uses the MLF type system, the (higher-ranked) type for $\mathit{prelude}$ is automatically inferred and the type annotation is not necessary. Neither is it necessary to declare the $\mathit{Prelude}$ type; we

can just construct an anonymous record. Indeed, we can use the id member polymorphically without any type declaration:

$$identity :: \forall\alpha.\ \alpha \to \alpha$$
$$identity\ x = prelude.id\ prelude.id\ x$$

In particular, as imposed by MLF, type annotations are only necessary when function arguments of an unknown type are used polymorphically. For example, we can not leave out the type signature if we use an unknown implementation of a prelude passed as an argument:

$$twice :: Prelude \to (Int, Bool)$$
$$twice\ p = (p.id\ 1, p.id\ True)$$

In the above example, the id field of the argument p is used polymorphically at two different types, and thus the type signature is required to reveal that the id field of a $Prelude$ is indeed a polymorphic function.

Apart from abstract types, these examples are very close to the goals of the XHM system, described by Jones [14] as an approach to treat modules as first-class citizens. We believe that our notion of records in combination with higher-order polymorphic MLF is therefore a significant step towards a realistic implementation of polymorphic and extensible first-class modules.

5. Variants

The design that we described lends itself well to also include *variants*. Where records model labeled tuples, variants model a labeled choice among values. It is beyond the scope of this paper to describe the full implications of programming with variants, but we will describe how our basic design can be easily extended with support for variants. We denote variant types by enclosing them in angled brackets ($\langle\rangle$). Here is an example of a variant that models an event:

$$\mathbf{type}\ Event = \langle key :: Char, mouse :: Point\rangle$$

Note that whereas a record includes all fields, the variant describes a choice: an $Event$ is either key event with $Char$ value, or a $mouse$ event with a $Point$ value. As we can see however, the types of records and variants can both share the concept of a row as a sequence of labeled sequence types. Indeed, the unabbreviated type of an $Event$ is:

$$\mathbf{type}\ Event = \langle (\!| key :: Char \mid (\!| mouse :: Point \mid (\!|\,|\!)\,|\!)\rangle$$

Since the rules for type equality in Figure 1 directly work on rows, we do not need to change anything to our basic type system. Just like records, there are three primitive operations on variants: *injection*, *embedding*, and *decomposition*.

$$\langle l = \_\rangle :: \forall\alpha r.\ \alpha \to \langle l :: \alpha \mid r\rangle \quad \text{-- injection}$$
$$\langle l \mid \_\rangle\ :: \forall\alpha r.\ r \to \langle l :: \alpha \mid r\rangle \quad \text{-- embedding}$$
$$(l \in \_\,?\,\_ : \_) \quad \text{-- decomposition}$$
$$:: \forall\alpha\beta r.\ \langle l :: \alpha \mid r\rangle \to (\alpha \to \beta) \to (\langle r\rangle \to \beta) \to \beta$$

The injection operation creates a variant value tagged with a label l. For example, we can construct a key event for the tab key as follows:

$$tab :: \forall r.\ \langle key :: Char \mid r\rangle$$
$$tab = \langle key = \texttt{'\textbackslash t'}\rangle$$

The type signature of tab is polymorphic in the row r and can thus be used in any context where a variant with a key field is expected.

The embedding operation embeds a value in a variant type that also allows for a label l:

$$\langle mouse \mid tab\rangle :: \forall\alpha r.\ \langle mouse :: \alpha, key :: Char, r\rangle$$

In general, this operation is not so useful since injection already constructs polymorphic variants. However, we will see that in combination with duplicate labels this becomes an essential operation for variants.

The decomposition operations allows us to check if a variant is tagged with a particular label l and act accordingly. Here is an example of a function to show an $Event$:

$$showEvent :: Event \to String$$
$$showEvent\ e$$
$$= (key \in e\ ?\ (\backslash c \to showChar\ c)$$
$$: (\backslash e' \to (mouse \in e'\ ?\ (\backslash p \to showPoint\ p)$$
$$: error\ \texttt{"unmatched variant"})))$$

If the variant is tagged with key, the function $(\backslash c \to showChar\ c)$ is applied to its value. If it doesn't match, we continue with a check for the $mouse$ label. Note that the passed argument e' is the same value as e but has a more specific type, namely $\langle mouse :: Point\rangle$. If the variant doesn't match either label, an exception is raised. Of course, programming languages can add extra syntax to make it more convenient to match on variants. For example, in Morrow we can write the above match as:

$$showEvent :: Event \to String$$
$$showEvent\ e = \mathbf{case}\ e\ \{\ key\ c \quad = showChar\ c$$
$$,\ mouse\ p = showPoint\ p\}$$

A curious property of our system is that variants can now contain duplicate tags. However, all operations are still well-defined and can not "go wrong". For example, we can match on the second tag of a variant by doing two decompositions in a row:

$$f :: \forall\alpha\beta.\ \langle l :: \alpha, l :: \beta\rangle \to Int$$
$$f\ v = \mathbf{case}\ v\ \{\ l\ x = 1 \quad \text{-- first } l \text{ tag}$$
$$,\ l\ y = 2\} \quad \text{-- second } l \text{ tag}$$

Even though this may look too liberal, we believe this is not the case because it will take conscious effort to actually construct a variant type with a scoped label. Indeed, the only operation that can do this is embedding. Since it does not serve any other purpose, it can be viewed as a programmers annotation that a scoped label is really required.

A useful application of scoped labels arises when modelling open-ended data types like user-defined exceptions. Suppose we have a function $throw$ that raises an exception as a tagged value:

$$throw :: \forall\alpha r.\ \langle r\rangle \to \alpha$$

Each library can throw its own kinds of exception variants and it is not unlikely that the names for exceptions might clash between different libraries. The embedding operator can be used to prioritize identically named exceptions of different libraries, where the global exception handler matches on duplicate labels.

6. Type rules

In this section, we formalize the concept of rows and define the structure of types. First, we have to make some basic assumptions about the structure of types. This structure is needed as not all types are well-formed. For example, a row type can not extend an integer and a $Maybe$ type needs a parameter:

$$(\!| l = Maybe \mid Int |\!)$$

Following standard techniques [13, 29] we assign *kinds* to types to exclude ill-formed types. The kind language is very simple and given by the following grammar:

$$\begin{array}{lll} \kappa & ::= & * \quad \text{kind of term types} \\ & | & \mathsf{row} \quad \text{kind of row types} \\ & | & \kappa_1 \to \kappa_2 \quad \text{kind of type constructors} \end{array}$$

All terms have types of kind $*$. The arrow kind is used for type constructors like $Maybe$ and function types. Furthermore, the special kind row is the kind of row types. We assume that there is an initial set of type variables $\alpha \in A$ and type constants $c \in C$. Furthermore, the initial set of type constants C should contain at least:

Int	$::: *$	integers
$(\to)$	$::: * \to * \to *$	functions
$(\!\|\|\!)$	$::: \mathsf{row}$	empty row
$(\!\| l = \_ \mid \_ \|\!)$	$::: * \to \mathsf{row} \to \mathsf{row}$	row extension
$\{\_\}$	$::: \mathsf{row} \to *$	record constructor
$\langle\_\rangle$	$::: \mathsf{row} \to *$	variant constructor

For each kind κ, we have a collection of types τ^κ of kind κ described by the following grammar:

$$\begin{array}{lll} \tau^\kappa & ::= & c^\kappa \quad \text{constants} \\ & | & \alpha^\kappa \quad \text{type variables} \\ & | & \tau_1^{\kappa_2 \to \kappa}\ \tau_2^{\kappa_2} \quad \text{type application} \end{array}$$

Note how the above grammar rules for well-kinded types exclude the previous ill-formed type example. The set of type schemes σ is described by quantification of types of kind $*$:

$$\begin{array}{lll} \sigma & ::= & \forall \alpha^\kappa.\, \sigma \quad \text{polymorphic types} \\ & | & \tau^* \quad \text{monotypes} \end{array}$$

Using a simple process of kind inference [13] the kinds of all types can be automatically inferred and no explicit annotations are necessary in practice. In the rest of this article, we therefore leave out most kind annotations when they are apparent from the context. Note that we assume higher-order polymorphism [13, 12] where variables in type expressions can quantify over types of an arbitrary kind. This is necessary since our primitive operations quantify over row kinds. For example, here is the kind annotated type for selection:

$$(\_.l) :: \forall r^{\mathsf{row}} \alpha^*.\, \{l :: \alpha \mid r\} \to \alpha$$

As we remarked before, our framework makes just few assumptions about the actually type rules and can be embedded in any higher-order polymorphic type system. To use our framework with standard Hindley-Milner type rules [9, 20] we need to make the implicit syntactic equality between mono types explicit with our equality relation defined in Figure 1. We do not repeat all the Hindley-Milner type rules here, but just give the application rule as a typical example:

$$(app) \quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \cong \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Exactly the same approach can be used to use our notion of records with qualified types [10] and Haskell. To use our framework with the MLF type rules is even easier as we only need to extend the rule $(eq\text{-}refl)$ of the MLF equality relation on poly types ($\equiv$) to include our notion of equality on mono types ($\cong$):

$$(eq\text{-}refl\text{-}mono) \quad \frac{\tau \cong \tau'}{\tau \equiv \tau'}$$

$$(uni\text{-}const) \quad c \sim c : []$$

$$(uni\text{-}var) \quad \alpha \sim \alpha : []$$

$$(uni\text{-}varl) \quad \frac{\alpha \notin \mathsf{ftv}(\tau)}{\alpha^\kappa \sim \tau^\kappa : [\alpha \mapsto \tau]}$$

$$(uni\text{-}varr) \quad \frac{\alpha \notin \mathsf{ftv}(\tau)}{\tau^\kappa \sim \alpha^\kappa : [\alpha \mapsto \tau]}$$

$$(uni\text{-}app) \quad \frac{\tau_1 \sim \tau_1' : \theta_1 \quad \theta_1\tau_2 \sim \theta_1\tau_2' : \theta_2}{\tau_1\ \tau_2 \sim \tau_1'\ \tau_2' : \theta_2 \circ \theta_1}$$

$$(uni\text{-}row) \quad \frac{s \simeq (\!| l :: \tau' \mid s' |\!) : \theta_1 \quad \mathsf{tail}(r) \notin \mathsf{dom}(\theta_1) \quad \theta_1\tau \sim \theta_1\tau' : \theta_2 \quad \theta_2(\theta_1 r) \sim \theta_2(\theta_1 s') : \theta_3}{(\!| l :: \tau \mid r |\!) \sim s : \theta_3 \circ \theta_2 \circ \theta_1}$$

Figure 2. Unification between (mono) types

$$(row\text{-}head) \quad (\!| l :: \tau \mid r |\!) \simeq (\!| l :: \tau \mid r |\!) : []$$

$$(row\text{-}swap) \quad \frac{l \neq l' \quad r \simeq (\!| l :: \tau \mid r' |\!) : \theta}{(\!| l' :: \tau' \mid r |\!) \simeq (\!| l :: \tau \mid l' :: \tau' \mid r' |\!) : \theta}$$

$$(row\text{-}var) \quad \frac{\mathsf{fresh}(\beta) \quad \mathsf{fresh}(\gamma)}{\alpha \simeq (\!| l :: \gamma \mid \beta |\!) : [\alpha \mapsto (\!| l :: \gamma \mid \beta |\!)]}$$

Figure 3. Isomorphic rows

No change is necessary to the actual type rules of MLF as those are already defined in terms of the standard MLF equality relation on type schemes.

7. Type inference

This section describes how our system supports type inference, where the most general type of an expression is automatically inferred. Central to type inference is the unification algorithm.

7.1 Unification

To support higher-order polymorphism, we use *kind preserving* substitutions in this article. A kind preserving substitution always maps type variables of a certain kind to types of the same kind. Formally, a substitution θ is a *unifier* of two types τ and τ' iff $\theta\tau \cong \theta\tau'$. We call such unifier a most general unifier of these types if every other unifier can be written as the composition $\theta' \circ \theta$, for some substitution θ'. Figure 2 gives an algorithm for calculating unifiers in the presence of rows. We write $\tau \sim \tau' : \theta$ to calculate the (most general) unifier θ for two types τ and τ'.

The first five rules are standard Robinson unification [33], slightly adapted to only return kind-preserving unifications [13]. The last rule $(uni\text{-}row)$ deals with unification of rows. When a row $(\!| l :: \tau \mid r |\!)$ is unified with some row s, we first try to rewrite s in the form $(\!| l :: \tau' \mid s' |\!)$ using the rules for type equality defined in Figure 1. If this succeeds, the unification proceeds by unifying the field types and the tail of the rows.

Figure 3 gives the algorithm for rewriting rows where the expression $r \simeq (\!| l :: \tau \mid s |\!) : \theta$ asserts that r can be rewritten to the form $(\!| l :: \tau \mid s |\!)$ under substitution θ. Note that r and l are input parameters while τ, s, and θ are synthesized. The first two rules correspond to the rules $(eq\text{-}head)$ and $(eq\text{-}swap)$ of type equality in Figure 1. The last rule unifies a row tail that consist of a type variable. Note that this rule introduces fresh type variables which might endanger termination of the algorithm. This is the reason for the side condition in rule $(uni\text{-}row)$: $\mathsf{tail}(r) \notin \mathsf{dom}(\theta_1)$.

If we look closely at the rules in Figure 3 there are only two possible substitutions as the outcome of a row rewrite. When a label l can be found, the substitution will be empty as only the rules $(row\text{-}swap)$ and $(row\text{-}head)$ apply. If a label is not present, the rule $(row\text{-}var)$ applies and a singleton substitution $[\alpha \mapsto (\!|l :: \gamma \mid \beta|\!)]$ is returned, where α is the tail of s. Therefore, the side-condition $\mathsf{tail}(r) \notin \mathsf{dom}(\theta_1)$ prevents us from unifying rows with a common tail but a distinct prefix[1]. Here is an example where unification would not terminate without the side condition:

$$\backslash r \to \mathbf{if}\ \mathit{True}\ \mathbf{then}\ \{x = 2 \mid r\}\ \mathbf{else}\ \{y = 2 \mid r\}$$

During type inference, the rows in both **if** branches are unified:

$$(\!|x :: \mathit{Int} \mid \alpha|\!) \sim (\!|y :: \mathit{Int} \mid \alpha|\!)\ : \theta_3 \circ \theta_2 \circ \theta_1$$

Which implies that $(\!|y :: \mathit{Int} \mid \alpha|\!)$ is rewritten as:

$$(\!|y :: \mathit{Int} \mid \alpha|\!) \simeq (\!|x :: \gamma \mid y :: \mathit{Int} \mid \beta|\!) : \theta_1$$

Where $\theta_1 = [\alpha \mapsto (\!|x :: \gamma \mid \beta|\!)]$. After unification of γ and Int, the unification of the row tails is now similar to the initial situation and thus loops forever:

$$\begin{aligned}&\theta_2(\theta_1\alpha) \sim \theta_2(\theta_1((\!|y :: \mathit{Int} \mid \beta|\!)))\ : \theta_3\\ &\quad =\\ &(\!|x :: \mathit{Int} \mid \beta|\!) \sim (\!|y :: \mathit{Int} \mid \beta|\!)\ : \theta_3\end{aligned}$$

However, with the side condition in place, no such thing will happen since $\mathsf{tail}(r) = \alpha \in \{\alpha\} = \mathsf{dom}(\theta_1)$. Not all record systems described in literature correctly ensure termination of record unification for this class of programs. For example, the unification rules of TREX fail to terminate for this particular example [7].

The reader might be worried that the side condition endangers the soundness or completeness of unification, but such is not the case, as asserted by the following theorems.

Theorem 1. *Unification is sound. If two types unify they are equal under the resulting substitution:* $\tau \sim \tau' : \theta \Rightarrow \theta\tau \cong \theta\tau'$.

Proof. Proved by straightforward induction over the cases of the unification algorithm. A full proof can be found in [17].

Theorem 2. *Unification is complete. If two types are equal under some unifier, unification will succeed and find a most general unifier:* $\theta\tau \cong \theta\tau' \Rightarrow \tau \sim \tau' : \theta_1 \wedge \theta_1 \sqsubseteq \theta$.

Proof. Standard proof of completeness over the structure of types. A constructive proof is given in a separate technical report [17].

As most type inference algorithms reduce to a set of unification constraints, soundness and completeness results carry over directly with the above results for row unification. In particular, the proofs for Hindley-Milner, qualified types [11], and MLF[15] are easily adapted to hold in the presence of row unification.

8. Implementing records

Providing an efficient implementation for extensible and polymorphic records is not entirely straightforward. In this section we discuss several implementation techniques and show in particular how standard compilation techniques can be used to provide constant-time access for label selection.

Association lists. A naïve implementation of records uses a simple association list of label-value pairs. Selection is implemented as a linear search over this list, where the type system ensures that such label is always found. Extension is a constant time operation that adds an element to the head of the list. However, in practice the most common operation by far is label selection, and a linear algorithm would be slow for larger records. This holds especially when using records as modules that can easily contain hundreds of labels.

Labeled vectors. A more efficient representation for label selection is a vector of label-value pairs where the fields are sorted on the label according to some order on the labels. Extension is now a linear operation as the full vector must be copied. This can be made somewhat more efficient by processing multiple extensions together in a single operation. Label selection can now use a binary search to select a particular label and becomes an $O(log(n))$ operation. When labels can be compared efficiently, for example by using a Garrigue's hashing scheme [4], the binary search over the vector can be implemented very efficiently. It is also possible to improve the search time for small and medium sized records by using a partially evaluated header [32], but at the price of a potentially more expensive extension operation.

Labeled vectors + constant folding. Label selection can be divided into two separate operations: looking up the index of the field in the record ($lookup$), and selecting the value using that index ($select$). When the labels of the record are known it is possible to partially evaluate the $lookup$ operation using standard compiler techniques. For example, the expression $\{l = expr\}.l$ could be translated with $lookup$ and $select$ as:

$$\begin{aligned}&\mathbf{let}\ r = \{l = expr\}\\ &\quad i = lookup\ r\ l\\ &\mathbf{in}\ select\ r\ i\end{aligned}$$

Since the type of r is known, the compiler can statically evaluate $lookup\ r\ l$ and replace it by 0, avoiding a binary search at runtime:

$$\mathbf{let}\ r = \{l = expr\}\ \mathbf{in}\ select\ r\ 0$$

This optimization by itself guarantees constant-time label selection for all records with a fixed set of labels. In contrast, if the record type is open, the $lookup$ operation can not be evaluated statically. Here is an example:

$$\begin{aligned}&dist :: \forall r.\, \{x :: \mathit{Int}, y :: \mathit{Int} \mid r\} \to \mathit{Int}\\ &dist\ r = r.x + r.y\end{aligned}$$

However, at the call site, the type of the row variable r is generally known. A compiler can take advantage of this by floating all lookup operations upwards and split the function $dist$ into two functions, a wrapper and a worker:

$$\begin{aligned}&dist\ r &&= dist\_work\ (lookup\ r\ x)\ (lookup\ r\ y)\ r\\ &dist\_work\ i\ j\ r &&= select\ r\ i + select\ r\ j\end{aligned}$$

If the wrapper function $dist$ is inlined at a call site there is a good chance that the type of r is known. In that situation, the offsets for both labels are passed as a runtime parameter giving constant-time access even for record selection on open records. This technique is basically a variant of the worker-wrapper transformation in GHC [24]. An aggressive compiler would also push the lookup operation through extensions in order to float properly. For example:

$$\backslash r \to \{x = 2 \mid r\}.y$$

The compiler would first translate this to:

[1] In practice, this side condition can also be implemented by passing $\mathsf{tail}(r)$ to the $(\simeq)$ function and checking in $(row\text{-}var)$ that $\alpha \neq \mathsf{tail}(r)$

$$
\begin{array}{l}
\backslash r \rightarrow \textbf{let } rx = \mathit{extend}\ x\ 2\ r \\
\qquad\quad\ \ i\ \ = \mathit{lookup}\ rx\ y \\
\qquad \textbf{in } \mathit{select}\ rx\ i
\end{array}
$$

In order to float i upwards, the compiler needs to inline the extension, adjust the lookup operation to work on r, and float it upward:

$$
\begin{array}{l}
\backslash r \rightarrow \textbf{let } i\ \ = \mathit{lookup}\ r\ y + 1 \\
\qquad\quad\ \ rx = \mathit{extend}\ x\ 2\ r \\
\qquad \textbf{in } \mathit{select}\ rx\ i
\end{array}
$$

In this case, the index is adjusted by one as label y is greater than x. No adjustment would be made for labels smaller or equal to y. In general, offsets are adjusted along the lines of the evidence translation of Gaster and Jones [7].

Vectors. One of the most efficient representation for records is a plain vector without labels. This representation is generally used by languages that have no extensible records, like C and ML. Gaster and Jones [7, 6] showed how to this representation can be used with polymorphic extensible records using standard evidence translation in the context of qualified types [10, 11]. We can directly apply this technique by using our system in the context of qualified types. First of all, we add a special *extension* predicate. The predicate $l|r$ asserts that row r is extended with a label l. The types of the basic operations become:

$$
\begin{array}{ll}
(\_.l) & :: \forall r\alpha.\ (l|r) \Rightarrow \{l :: \alpha \mid r\} \rightarrow \alpha \\
(\_ - l) & :: \forall r\alpha.\ (l|r) \Rightarrow \{l :: \alpha \mid r\} \rightarrow \{r\} \\
\{l = \_ \mid \_\} & :: \forall r\alpha.\ (l|r) \Rightarrow \alpha \rightarrow \{r\} \rightarrow \{l :: \alpha \mid r\}
\end{array}
$$

Standard evidence translation turns each predicate $l|r$ into a runtime parameter that corresponds to the offset of l in the extended row r [7]

It may seem that we have sacrificed the simplicity of our system as the type signatures now show an artificial predicate, that is just used for compilation. The crucial observation here is that, in contrast to lacks predicates, extension predicates can always be solved and never lead to an error. This means that the type system can use these predicates under the hood without ever showing them to the user. Explicit type signatures are always interpreted as partial type signatures modulo any extension predicates. The knowledgeable Haskell user may be concerned that we cannot always hide the extension predicate as it might interact with the monomorphism restriction. However, the only way to construct a non-functional value with an unsolved extension predicate is to either use $\bot$, or a phantom type [18]. In both cases, it is safe to resolve the predicate to any offset (including $\bot$). This is a direct consequence of the fact that we can assign an untyped dynamic semantics to our system (in contrast to type classes [22]).

Of the above implementation techniques the plain vector implementation seems the most attractive at first sight: it offers guaranteed constant-time label selection and comes with a straightforward compilation scheme. However, it also comes with some drawbacks inherent to qualified types. Suppose that a polymorphic record is extended with many new fields, for example, when constructing a matrix package from a general math package. This means that for each new label a corresponding offset is passed at runtime which can lead to functions that take tens, or even hundreds, of parameters. The labeled vectors with constant folding shines here as it can use the standard inline heuristics of the compiler to dynamically switch between calculating the offset at runtime or passing it as a parameter. Furthermore, for some operations, it may be convenient to have a runtime representation of the labels available. For example, when showing records, or for dynamic checks when deserializing records from disk. On the theoretical side, it is yet unclear how to use qualified types in an impredicative type system like MLF.

Maybe one of the best ways to implement records combines the best features of both systems: it uses labeled vectors to implement records, together with predicates for the selection and restriction operations. This leads to a simple compilation scheme that gives constant access to labels, but avoids the many runtime parameters for extension. The extension operation is done dynamically but since it is $O(n)$ anyway, we expect that the runtime penalty is negligible. Currently, Morrow uses a labeled vector representation for its records but no predicates since it uses MLF as its core type system.

8.1 Variants

Variants can simply be implemented as a pair of a label and its associated value. The injection operation constructs such pair, and the decomposition operation matches on the label and extracts the value if necessary. A subtle complication arises when duplicate labels occur. Here is an example from Section 5 that illustrates the problem:

$$
\begin{array}{l}
f :: \forall\alpha\beta.\ \langle l :: \alpha, l :: \beta\rangle \rightarrow \mathit{Int} \\
f\ v = \textbf{case } v\ \{\ l\ x = 1 \quad \text{-- first } l \text{ tag} \\
\qquad\qquad\qquad\quad ,\ l\ y = 2\} \quad \text{-- second } l \text{ tag}
\end{array}
$$

If the decomposition operation would just match on the label, it is not able to distinguish between both options as both values are tagged with label l. Therefore, variants must be represented as triple: besides the label and value, it should also include the *nesting level* of a variant. Injection always constructs a variant with a nesting level of zero. The embedding operation now gets a real operational effect as it is only operation that can introduce scoped labels. If the embedding extends a variant with a duplicate label, it will increase the nesting level. The decomposition operation in turn only considers labels as a match when their nesting level is zero, i.e. when it is the first occurrence of that particular label in the type. If the labels match, but the nesting level is unequal to zero, decomposition will decrease the nesting level by one in the "else" branch.

It is also possible to use extension predicates to assign static tags to variants which might lead to a more efficient pattern match implementation. Furthermore, the embedding operation would simply adjust evidence passed at runtime and duplicate labels can be accommodated without the need for explicit nesting levels.

9. Related work

An impressive amount of work has been done on type systems for records and we restrict ourselves to short overview of the most relevant work.

The label selective calculus [5, 3] is a system that labels function parameters. Even though this calculus does not describe records, there are many similarities with our system and the unification algorithm contains a similar side condition to ensure termination.

One of the earliest and most widely used approaches to typing records is subtyping [2, 26]. The type of selection in such system becomes:

$$(\_.l) :: \forall\alpha.\, \forall r \leqslant \{l :: \alpha\}.\ r \rightarrow \alpha$$

That is, we can select label l from any record r that is a subtype of the singleton record $\{l :: \alpha\}$. Unfortunately, the information about the other fields of a record is lost, which makes it hard to describe operations like row extension. Cardelli and Mitchell [2] especially introduce an overriding operator on types to overcome this problem.

Wand [35, 36] was the first to use row variables to capture the subtype relationship between records using standard parametric polymorphism. As we have seen in Section 3.3 not all programs have a principal type in this system. The work of Wand is later refined by Berthomieu and Sagazan [1] where a polynomial unification algorithm is presented. Remy [29] extended the work of Wand with a flexible system of extensible records with principle types. Flags are used to denote the presence or absence of labels and rows.

Ohori [23] was the first to present an efficient compilation method for polymorphic records with constant time label selection, but only for non-extensible rows. Subsequently, Gaster and Jones [7, 6] presented an elegant type sytem for records and variants based on the theory of qualified types [10, 11]. They use strict extension with special *lacks* predicates to prevent duplicate labels. The predicates correspond to runtime label offsets, and standard evidence translation gives a straightforward compilation scheme with constant time label selection.

The primitive operations in our system are based on extension, but many calculi use record concatenation as the primitive operation [8, 30, 31, 36, 34, 27, 19]. Pottier [28] describes a general framework for row based type systems which has an effective type inference algorithm for record concatenation. Unfortunately, the types of those systems are generally hard to use in practice. Here is for example the type of free extension in Pottier's system:

$$\{l = \_ \mid \_\} :: (\{l\} : \partial(pre(\alpha)) \leqslant r_2, (\mathcal{L} \,/\, \{l\}) : r_1 \leqslant r_2) \Rightarrow \alpha \to r_1 \to r_2$$

The first constraint is a singleton filter to indicate that the result contains field l, while the second constraint is a co-singleton filter that indicates that all fields other than l have the same status as in the argument.

10. Conclusion

We believe that polymorphic and extensible records are a flexible and fundamental concept to program with data structures. The complexity of type systems for such records used to prevent widespread adoption in mainstream languages. We presented polymorphic and extensible records based on scoped labels that are safe, convenient, easy to check, and straightforward to compile – every programming language should have them!

Acknowledgments

We would like to thank François Pottier for his feedback and for pointing out the similarities between our system and the label selective calculus.

References

[1] B. Berthomieu and C. de Sagazan. A calculus of tagged types, with applications to process languages. In *TAPSOFT Workshop on Types for Program Analysis*, May 1995.

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

[3] J. P. Furuse and J. Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Kyoto University, Oct. 1995.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] J. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

[10] 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.

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

[12] M. P. Jones. From Hindley-Milner types to first-class structures. In *Proceedings of the Haskell Workshop*, June 1995. Yale University Research Report YALEU/DCS/RR-1075.

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

[14] M. P. Jones. Using parameterized signatures to express modular structure. In *23th ACM Symp. on Principles of Programming Languages (POPL'96)*, pages 68–78. ACM Press, 1996.

[15] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, aug 2003.

[16] D. Leijen. Morrow: a row-oriented programming language. `http://www.cs.uu.nl/~daan/morrow.html`, Jan. 2005.

[17] D. Leijen. Unqualified records and variants: proofs. Technical Report UU-CS-2005-00, Dept. of Computer Science, Universiteit Utrecht, 2005.

[18] 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).

[19] H. Makholm and J. B. Wells. Type inference and principal typings for symmetric record concatenation and mixin modules. Technical Report HW-MACS-TR-0030, School of Mathematical and Computer Sciences, Heriot-Watt University, Mar. 2005.

[20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:248–375, Aug. 1978.

[21] M. Odersky and K. Läufer. Putting type annotations to work. In *23th ACM Symp. on Principles of Programming Languages (POPL'96)*, pages 54–67, Jan. 1996.

[22] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional Programming languages and Computer Architecture*, pages 135–146, 1995.

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

[24] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, Sept. 1998.

[25] S. Peyton-Jones and M. Shields. Practical type inference for arbitrary-rank types. Submitted to the Journal of Functional Programming (JFP), 2004.

[26] 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.

[27] F. Pottier. A 3-part type inference engine. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 320–335. Springer Verlag, Mar. 2000.

[28] F. Pottier. A constraint-based presentation and generalization of rows. In *Eighteenth Annual IEEE Symposium on Logic In Computer Science (LICS'03)*, pages 331–340, Ottawa, Canada, June 2003.

[29] 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.

[30] D. Rémy. Typing record concatenation for free. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[31] D. Rémy. A case study of typechecking with constrained types: Typing record concatenation. Workshop on Advances in Types for Computer Science, Aug. 1995.

[32] D. Rémy. Efficient representation of extensible records. INRIA Rocquencourt, 2001.

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

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

[35] 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.

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