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

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 [14], 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 [15]. 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 [26], Remy [23], 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. We formalize the type rules and inference in section 5 and 6. 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:

$$origin3 \quad = \{z = 0 \mid origin\}$$
$$named\ s\ r = \{name = s \mid r\}$$

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:

$$\{l := x \mid r\} \ = \ \{l = x \mid r - l\} \quad \text{-- update } l$$
$$\{l \leftarrow m \mid r\} \ = \ \{l = r.m \mid r - m\} \quad \text{-- rename } m \text{ to } l$$

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 [26, 23, 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 another 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:

$\{x = 2, x = True\}.x$ -- select the first *x* field
$(\{x = 2, x = True\} - x).x$ -- select the second *x* field

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:

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

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 $origin$:

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

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:

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

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

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.

$$(eq\text{-}var) \qquad \alpha \cong \alpha$$

$$(eq\text{-}const) \qquad c \cong c$$

$$(eq\text{-}app) \qquad \frac{\tau_1 \cong \tau_1' \quad \tau_2 \cong \tau_2'}{\tau_1\ \tau_2 \cong \tau_1'\ \tau_2'}$$

$$(eq\text{-}trans) \qquad \frac{\tau_1 \cong \tau_2 \quad \tau_2 \cong \tau_3}{\tau_1 \cong \tau_3}$$

$$(eq\text{-}head) \qquad \frac{\tau \cong \tau' \quad r \cong s}{(\!|l :: \tau \mid r|\!) \cong (\!|l :: \tau' \mid s|\!)}$$

$$(eq\text{-}swap) \qquad \frac{l \neq l'}{(\!|l :: \tau, l' :: \tau' \mid r|\!) \cong (\!|l' :: \tau', l :: \tau \mid r|\!)}$$

FIGURE 1. Equality between (mono) types

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:

$$\{x = 2, x = \mathit{True}\} :: \{x :: \mathit{Int}, x :: \mathit{Bool}\}$$
$$\{x = \mathit{True}, x = 2\} :: \{x :: \mathit{Bool}, x :: \mathit{Int}\}$$

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:

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

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.

4 HIGHER-RANKED IMPREDICATIVE RECORDS

Since the type signatures for record operations are so general, we can conveniently package related functions together. Together with a mechanism for local type declarations, we can view these packages as a form of first-class modules. However, records should be able to contain polymorphic values in order to encode more complicated modules. Take for example the following type signature for a Monad module:

$$\mathbf{type}\ \mathit{Monad}\ m = \{\mathit{unit} :: \forall\alpha.\quad \alpha \to m\ \alpha$$

$$
\begin{array}{l}
\quad , bind :: \forall \alpha\beta.\ m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta \\
\quad \}
\end{array}
$$

In this signature, the monad implementation type m is polymorphic over the monad record. Furthermore, the types of $unit$ and $bind$ members are itself polymorphic, and thus use impredicative higher-rank polymorphism since the quantifiers are nested inside the record structure. Unfortunately, type inference for impredicative rank-n polymorphism is a notoriously hard problem [14, 21, 18].

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 [14]. We have a full implementation of this system in the experimental Morrow compiler [15] 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 for example give an implementation of an identity monad:

$$
\begin{array}{l}
\mathbf{newtype}\ Id\ \alpha = Id\ \alpha \\
idm :: Monad\ Id \\
idm = \{unit\ x \qquad\quad = Id\ x \\
\qquad , bind\ (Id\ x)\ f = f\ x \\
\qquad \}
\end{array}
$$

Since Morrow uses the MLF type system, the (higher-ranked) type for idm is automatically inferred and the type annotation is not necessary. Neither is it necessary to declare the $Monad$ type; we can just construct an anonymous record. Indeed, we can use the $unit$ member polymorphically without any type declaration:

$$
\begin{array}{l}
twice :: \forall \alpha.\ \alpha \to Id\ (Id\ \alpha) \\
twice\ x = idm.unit\ (idm.unit\ x)
\end{array}
$$

Apart from abstract types, these examples are very close to the goals of the XHM system, described by Jones [13] 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 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 [12, 23] we assign *kinds* to types to exclude ill-formed types. The kind language is very simple and given by the following grammar:

κ	$::=$	$*$	kind of term types
	$\mid$	row	kind of row types
	$\mid$	$\kappa_1 \to \kappa_2$	kind of type constructors

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:

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:

τ^κ	$::=$	c^κ	constants
	$\mid$	α^κ	type variables
	$\mid$	$\tau_1^{\kappa_2 \to \kappa}\ \tau_2^{\kappa_2}$	type application

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 $*$:

σ	$::=$	$\forall \alpha^\kappa.\ \sigma$	polymorphic types
	$\mid$	τ^*	monotypes

Using a simple process of kind inference [12] 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 [12, 11] 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 [8, 17] 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 [9] 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 ($\cong$):

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

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.

6 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.

6.1 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 [25], slightly adapted to only return kind-preserving unifications [12]. 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

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

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

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

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

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

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

FIGURE 2. Unification between (mono) types

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

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

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

FIGURE 3. Isomorphic rows

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:

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

[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 $\alpha \neq \mathsf{tail}(r)$

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

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

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

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

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:

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

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 [16].

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 [16].

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 [10], and MLF[14] are easily adapted to hold in the presence of row unification.

7 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 becomes a linear operation, and this is probably too inefficient for most practical applications.

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. 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 quite efficiently. It is also possible to improve the search time for small and medium sized records by using a partially evaluated header [24], 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. The expression $\{l = expr\}.l$ is translated with $lookup$ and $select$ as:

$$\textbf{let } r = \{l = expr\}; i = lookup\ r\ l\ \textbf{in } select\ r\ i$$

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:

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

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

$$dist :: \forall r.\{x :: Int, y :: Int \mid r\} \rightarrow Int$$
$$dist\ r = r.x + r.y$$

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:

$$dist\ r = dist\_work\ (lookup\ r\ x)\ (lookup\ r\ y)\ r$$
$$dist\_work\ i\ j\ r = select\ r\ i + select\ r\ j$$

If the wrapper function $dist$ is inlined at each call site since it is a cheap function and 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 [20]. An aggressive compiler also pushes the lookup operation through extensions in order to float properly, where 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. Gaster and Jones [7, 6] showed how to this representation can be used with polymorphic extensible records using standard evidence translation of qualified types [9, 10]. We can apply this technique to our system in the context of qualified types by adding an *extension* predicate $l|r$, that asserts that row r is extended with a label l. For example, the type of selection becomes:

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

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.

8 RELATED WORK

An impressive amount of work has been done on type systems for records and we necessarily 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, 22]. The type of selection in such system becomes:

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

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] introduce an overriding operator on types to overcome this problem.

Wand [26, 27] was the first to use row variables to capture the subtype relationship between records using standard parametric polymorphism. However, since his notion of free extension can overwrite previous labels, not all programs have a principal type. The work of Wand is later refined by Berthomieu and Sagazan [1] where a polynomial unification algorithm is presented. Remy [23] extended the work of Wand with a flexible system of extensible records with principle types, where flags are used to denote the presence or absence of labels and rows. For example, 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 abs or present $pre(\tau)$ with a type τ. The type of Remy's extension encodes that a field that

is already present is overwritten, while an absent field becomes present. This is a very flexible system, but the resulting types can be somewhat confusing since absent labels in the value can be present in the type (with an absent flag abs).

Ohori [19] 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 [9, 10]. They use strict extension with special *lacks* predicates to prevent duplicate labels. For example, extension is qualified with a lacks predicate:

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

The predicates correspond to runtime label offsets, and standard evidence translation gives a straightforward compilation scheme with constant time label selection. A drawback is that this system relies on a type system that supports qualified types, and that each use of a label leads to a lacks predicate, which in turn can lead to large types that are hard to read or expensive to implement.

9 CONCLUSION

We believe that polymorphic and extensible records are a flexible and fundamental concept to program with data structures, but that the complexity of type systems for such records prevents 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!

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

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*, 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 1041, Kyoto University, Oct. 1995.

[4] J. Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *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, University of Nottingham, 1996.

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

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

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

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

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

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

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

[26] 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.

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