Extensible Programming with First-Class Cases

Matthias Blume Umut Acar Wonseok Chae

Toyota Technological Institute at Chicago

{blume,umut,wchae}@tti-c.org

Abstract

We present language mechanisms for polymorphic, extensible records and their exact dual, polymorphic sums with extensible first-class cases. These features make it possible to easily extend existing code with new cases. In fact, such extensions do not require any changes to code that adheres to a particular programming style. Using that style, individual extensions can be written independently and later be composed to form larger components. These language mechanisms provide a solution to the expression problem.

We study the proposed mechanisms in the context of an implicitly typed, purely functional language PolyR. We give a type system for the language and provide rules for a 2-phase transformation—first into an explicitly typed λ-calculus with record polymorphism, and finally to efficient index-passing code. The transformations eliminate sums and cases by taking advantage of the duality with records.

We implement a version of PolyR extended with imperative features and pattern matching—we call this language **MLPolyR**. Programs in **MLPolyR** require no type annotations—the implementation employs a reconstruction algorithm to infer all types. The compiler generates machine code (currently for PowerPC) and optimizes the representation of sums by eliminating closures generated by the dual construction.

1. Introduction

In this paper we present a language, called **MLPolyR**, with polymorphic extensible records and their duals: polymorphic sums with a mechanism for adding new cases to existing code handling such sums. The type system of our language is a straightforward application of row polymorphism and has an efficient type reconstruction algorithm (essentially a variant of the well-known algorithm W [19]) which infers principal types. The key technical insight to type reconstruction with principal types in the presence of polymorphic records and their duals is to include explicit sum- and case-types in the type system and eliminate them after type inference is complete.

Our polymorphic sums provide a notion of *type refinement* similar to data sorts of Freeman and Pfenning [9] and give rise to a simple programming pattern facilitating *composable extensions*. Composable extensions can be used as a principled approach to solving the well-known *expression problem* described by Wadler [30]. There have been many attempts at solving the expression problem, most of them in an object-oriented context [28, 22, 3, 16, 27, 6, 8, 33, 18, 4, 29, 34]. Garrigue shows an approach based on polymorphic variants which is very similar but somewhat less general [11].

As in any dual construction, the introduction form of the primal corresponds to the elimination form of the dual. Thus, elimination forms of sums (usually **match** or **case**) correspond to introduction forms of records. Record extension, which is supported by **MLPolyR**, is one such introduction form. To key idea for extensibility is to also provide the dual, namely a mechanism for extending *cases*. Cases in **MLPolyR** are first-class values and not merely a syntactic form.[1] With cases being first-class and extensible, one can use the usual mechanisms of functional abstraction in a style of programming that facilitates composable extensions.

To understand the underlying mechanism, it is instructive to first look at an example:

Composable record extension and its dual

In **MLPolyR** we write `{ a = 1, ... = r }` to create a new record which extends record r with a new field `a`. Since records are first-class values, we can abstract over the record being extended and obtain a function `add_a` that extends any argument record (as long as it does not already contain `a`) with a field `a`. Such a function can be thought of as the "difference" between its result and its argument:

```
fun add_a r = { a = 1, ... = r }
```

Here the difference consists of a field labeled `a` of type `int` and value `1`. The type of function `add_a` is inferred as:[2]

```
val add_a: { β } → { a: int, β }
```

We can write similar functions `add_b` and `add_c` of types $\{\beta\}\to\{$`b:bool,`$\beta\}$ and $\{\beta\}\to\{$`c:string,`$\beta\}$ which add fields `b` and `c` respectively:

```
fun add_b r = { b = true, ... = r }
fun add_c r = { c = "hello", ... = r }
```

We can then "add up" record differences represented by `add_a`, `add_b`, `add_c` by composing these functions:

```
fun add_ab r = add_a (add_b r)
fun add_bc r = add_b (add_c r)
```

The inferred types are:

```
val add_ab: { β } → { a: int, b: bool, β }
val add_bc: { β } → { b: bool, c: string, β }
```

Finally, we can create actual records by "adding" differences to the empty record:

```
val a = add_a {}
val ab = add_ab {}
val bc = add_bc {}
```

When translated to the dual, extensibility of records becomes extensibility of code. Here is a function representing the difference

ICFP '06 September 2006, Portland, Oregon.

[1] In Standard ML [20], the corresponding syntactic form is known as *match*.

[2] We omit the universal quantifier and the kind of the row variable β.

between two code fragments, one of which can handle case `‘A` while the other, represented by the argument c, cannot:

```
fun add_A c = cases ‘A () => print "A"
              default: c
```

Note that function `add_A` corresponds to `add_a` of the dual. The type inferred for `add_A` is:

```
val add_A: (<β> ↪ ()) → (<A of (), β> ↪ ())
```

Here a type $\langle\rho\rangle \hookrightarrow \tau$ denotes the type of first-class cases where $\langle\rho\rangle$ is the sum type that is being handled and τ is the result. One can think of $\hookrightarrow$ as an alternative function arrow whose elimination form will be discussed below. Examples for functions `add_B` and `add_C` (corresponding to `add_b` and `add_c` in the dual) are:

```
fun add_B c = cases ‘B () => print "B"
              default: c
fun add_C c = cases ‘C () => print "C"
              default: c
```

As in the dual, we can now compose difference functions to obtain larger differences:

```
fun add_AB c = add_A (add_B c)
fun add_BC c = add_B (add_C c)
```

By applying a difference to the empty case **nocases** we obtain case values:

```
val case_A = add_A nocases
val case_AB = add_AB nocases
val case_BC = add_BC nocases
```

These values can be used in a **match** form. The **match** construct is the elimination form for the case arrow $\hookrightarrow$. The following expression will cause `"B"` to be printed:

```
match ‘B () with case_BC
```

The previous examples demonstrate how functional record extension in the primal corresponds to code extension in the dual. This forms the basis for our proposed solution to the expression problem. Section 2 discusses extensible CPS conversion as a realistic example.

2. Case study: CPS conversion

Let us consider some type `exp` with the following four constructors, modeling the untyped λ-calculus:[3]

```
con ‘Con : int → exp
con ‘Var : int → exp
con ‘Lam : ([int], exp) → exp
con ‘App : (exp, [exp]) → exp
```

Terms whose outermost constructor is `‘Con`, `‘Var` or `‘Lam` are called *syntactic values*. A commonly considered refinement of the `exp` type restricts `‘App` to only take such syntactic values. This is known as the CPS-restriction, since assuming a particular evaluation strategy (e.g., call-by-value, left-to-right) every unrestricted term can be converted into an “equivalent” CPS-term by making use of Continuation-Passing Style.

Following Appel [1], the conversion can be performed in essentially linear time using an approach that could be called “continuation-builder passing style” where the converter function `cvt` (see Figure 1) receives the expression e to be converted and k_b, the *continuation builder*, which represents the context in which e appeared within the original expression. Once the converter comes up with a syntactic value v for the result of e, k_b is invoked on this v in order to produce the CPS expression representing e's original context.

[3] This is a Scheme-like calculus [21] with multi-parameter functions. Variables `‘Var` x are represented by small integers x.

```
fun kv2kb k_v = λv.‘App(k_v,[v])
fun kb2kv k_b = withfresh(λx_r.‘Lam([x_r],k_b(‘Var x_r)))
fun cvt_app(e,e⃗,k_v) = let
   fun lc([],k_b) = k_b([])
     | lc(e::e⃗,k_b) = pc(e,e⃗,λ(v,v⃗).k_b(v::v⃗))
   and pc(e,e⃗,k_b) = cvt(e,λv.lc(e⃗,λv⃗.k_b(v,v⃗)))
in pc(e,e⃗,λ(v,v⃗).‘App(v,k_v::v⃗)) end
and cvt_lam(x⃗,e) = withfresh (λx_k.
   ‘Lam(x_k::x⃗,cvt(e,kv2kb(‘Var x_k))))
and cvt(e,k_b) = match e with
   cases ‘Con i ⇒ k_b(‘Con i)
      | ‘Var x ⇒ k_b(‘Var x)
      | ‘Lam(x⃗,e)⇒ k_b(cvt_lam(x⃗,e))
      | ‘App(e,e⃗)⇒ cvt_app(e,e⃗,kb2kv(k_b))
fun convert e = cvt_lam([],e)
```

Figure 1. A simple CPS-converter. When converting `‘App`, the continuation builder k_b must be turned into a syntactic value k_v that can be passed as an additional argument; this k_v is an new `‘Lam` with a single parameter x_r and a body obtained by applying k_b to x_r (see function `kb2kv`). Conversely, to convert a `‘Lam` an extra formal argument x_k representing the continuation is added. The corresponding k_b simply constructs an `‘App` of x_k to whatever argument v is given to k_b (see function `kv2kb`). For clarity, the code for `‘App` and `‘Lam` has been separated out into functions `cvt_app` and `cvt_lam`; `cvt_app` uses helper functions `lc` and `pc` to recursively convert the operator e and all operands $\vec{e}$.

2.1 Variants, polymorphism, subtyping, and refinement

As explained in the introduction, our language **MLPolyR** has polymorphic sum types in the style of Ocaml.[4] The type system is based on Rémy-style *row polymorphism*, handles equi-recursive types, and can infer principle types for all language constructs. For function `convert` in Figure 1, the compiler calculates the following type:

```
val convert:
 ∀α.∀ξ : {App}, ζ : {Con,Lam,Var}.
   (ε as <App of (ε, [ε]), Con of α,
          Lam of ([int], ε), Var of int>) →
   (ν as <Con of α,
          Lam of ([int], <App of (ν, [ν]), ξ>),
          Var of int, ζ>)
```

Here ϵ is a recursive sum type, indicated by keyword **as** and a type row enclosed in `< ... >`; ϵ can clearly be recognized as the type of lambda expressions.[5] Similarly, type ν is the type of syntactic values. Function `convert` is polymorphic in α, the type of values carried by `‘Con`. Finally, ξ and ζ are row type variables constrained to a particular *kind*. The kind is a set of labels that must be *absent* in any instantiation. Like in Standard ML, the type printer in our compiler does not ever print universal quantifiers for ordinary type variables, and it suppresses them for row variables whenever the kind can be inferred from how the variable is used. In this case, since ξ appears in a row with `‘App` and ζ appears in a row with `‘Con`, `‘Lam`, and `‘Var`, kind information can be left implicit. In fact, even the identity of the variables is irrelevant since there is only one occurrence of each. Therefore, the actual type expression printed by the **MLPolyR** compiler is the following:

[4] In fact, even our syntax for constructors is inspired by Ocaml's choice.

[5] We use Haskell notation $[t]$ for **MLPolyR**'s built-in list type.

```
fun cvt_app(cvt,e,ē,k_v) = ... as before ...
fun cvt_lam(cvt,x⃗,e) = ... as before ...

fun cvt_c(cvt,k_b) = cases ... same cases as before ...

fun mkConvert (c,e) =
    let fun cvt(e,k_b) = match e with c(cvt,k_b)
    in cvt_lam(cvt,[],e) end

fun convert e = mkConvert(cvt_c,e)
```

Figure 2. Preparing for extensibility. Explicitly open-code recursion and separate the cases from the scrutinee in the **match**-construct.

```
val convert:
  (ϵ as <App of (ϵ, [ϵ]), Con of α,
         Lam of ([int], ϵ), Var of int>) →
  (ν as <Con of α,
         Lam of ([int], <App of (ν, [ν]), ...>),
         Var of int, ...>)
```

A remarkable fact about this inferred type is its precision. Even though we had in mind only one restriction, namely that no `‘App` can be directly inside another `‘App`, the inference engine noticed that the converter enforces another invariant: all bodies of `‘Lam` are instances of `‘App`. Nevertheless, this extra precision is not harmful. The result type is polymorphic and can be instantiated to the one we may have had in mind:

```
(ν as <Con of α,
       Lam of ([int], (ϵ as <App of (ν, [ν]), Con of α,
                               Lam of ([int], ϵ),
                               Var of int>)),
       Var of int>)
```

In other words, although any occurrence of `‘Lam` in the output will in fact have been applied to a value constructed with `‘App`, one can send this output into a context that is prepared to also handle other cases for `‘Lam`. In fact, the output type is flexible enough to be instantiated to the original unrestricted expression type. This means that we could, for example, compose the `convert` function with itself:

```
fun convert_twice e = convert (convert e)
```

Doing so may be of limited use, but more importantly, existing utility routines such as pretty-printers, evaluators, and so forth that work on unrestricted expression can also be composed with function `convert`.

Conversely, if the code for `convert` contained a bug that can cause its output to violate the intended invariant, then this fact would also be visible in the type. Composition with code that *expects* the invariant will then fail to type-check.

2.2 Preparing for extensibility

As we have seen, row polymorphism represents a form of type refinement for the output of function `convert`. The input type, however, is rigid. This means that we cannot apply `convert` to a value that potentially contains constructors other than `‘Con`, `‘Var`, `‘Lam`, and `‘App`. Clearly, the type system is doing the right thing here, since the code itself is in no way prepared to handle anything but those four cases.

To make the code extensible, we need a way of adding new cases, i.e., new language constructs that are handled. Since these new constructs should be allowed to appear anywhere within a given input expression, we also must open up the recursion.

```
fun Let(x,e_1,e_2) = ‘App(‘Lam([x],e_2),[e_1])

fun cvti_c(cvt,k_b) =
    cases ‘If(e_c,e_t,e_e)⇒withfresh(λx_k.
       Let(x_k,bk2kv k_b,cvt(e_c,λv_c.
          let val k'_b=kv2kb(‘Var x_k)
          in ‘If(v_c,cvt(e_t,k'_b),cvt(e_e,k'_b)) end)))
    default: cvt_c(cvt,k_b)

fun converti e = mkConvert(cvti_c,e)
```

Figure 3. Extending the CPS converter to handle `‘If`.

```
fun cvt_lcc(cvt,x_c,e,x_k) =
   withfresh(λx_d.withfresh(λx_r.
      Let(x_c,‘Lam([x_d,x_r],‘App(‘Var x_k,[‘Var x_r]),
         cvt(e,kv2kb(‘Var x_k))))))

fun cvtc_c(cvt,k_b) =
   cases ‘LetCC(x_c,e)⇒
      withfresh(λx_k.Let(x_k,kb2kv(k_b),
         cvt_lcc(cvt,x_c,e,x_k)))
   default: cvt_c(cvt,k_b)
fun convertc e = mkConvert(cvtc_c,e)
```

Figure 4. Extending the CPS converter to handle `‘LetCC`.

As explained earlier, in **MLPolyR**, the cases of a **match**-expression handling a sum type $\langle\rho\rangle$ and returning a value of type τ are represented by first-class values of type $\langle\rho\rangle \hookrightarrow \tau$, and these values are extensible in the same sense in which records can be extended with new fields. Thus, to prepare the code for future extensions we separate the cases from the scrutinee and parameterizing them by closing over their free variables. By letting one of these free variables be the recursive instance of function `cvt` itself we straightforwardly achieve open recursion. Finally, we put the mechanism that closes the recursion into its own reusable routine `mkConvert` and then use it to recover the original function `convert` by applying `mkConvert` to `cvt_c` (see Figure 2).

2.3 Extending the input language

With this preparation in place, it is now very easy to extend the converter to handle new language constructs. For example, the code in Figure 3 introduces a conditional `‘If` which can appear in the input, and, if it does, will also appear in the output. The key construct that makes this work is the **cases** form with a **default:** clause. Here, a single new case (`‘If`) is handled, and the default explicitly refers to the original set of four cases represented by `cvt_c`. A new converter `converti`, now handling five cases including `‘If`, is obtained by closing the recursion using the same function `mkConvert` as before.

Another example for an extension is the addition of `‘LetCC` to the input language. `‘LetCC` is a binding construct which introduces a variable that, within its scope, refers to an “escape procedure” representing the current continuation.[6] In CPS-converted code, the current continuation is always directly available as a value, meaning that `‘LetCC` can be supported without need for a new language construct in the output language. As a result, the extension shown in Figure 4 extends the input language only.

[6] `‘LetCC(`x`,`e`)` is the same as Scheme’s `(call/cc (lambda (`x` `e`))` [21].

```
fun cvti_c other_c (cvt,k_b) =
    cases ‘If(e_c,e_t,e_e)⇒ ... as before ...
    default: other_c(cvt,k_b)
fun cvtc_c other_c (cvt,k_b) =
    cases ‘LetCC(x_c,e)⇒ ... as before ...
    default: other_c(cvt,k_b)

fun converti e = mkConvert(cvti_c cvt_c,e)
fun convertc e = mkConvert(cvtc_c cvt_c,e)
fun convertci e = mkConvert(cvtc_c(cvti_c cvt_c),e)
```

Figure 5. Extensions parameterized by what is being extended.

$$
\begin{array}{lcl}
\tau & ::= & \alpha \mid \mathbf{int} \mid \tau_1 \to \tau_2 \mid \{\rho\} \mid \langle\rho\rangle \mid \alpha\ \mathbf{as}\ \langle\rho\rangle \mid \langle\rho\rangle \hookrightarrow \tau \\
\rho & ::= & \beta \mid \cdot \mid l : \tau, \rho \\
\kappa & ::= & \{l_1, \ldots, l_k\} \\
\sigma & ::= & \forall(\alpha_1, \ldots, \alpha_m).\ \forall(\beta_1 : \kappa_1, \ldots, \beta_n : \kappa_n).\tau \\
v & ::= & n \mid \mathbf{fun}\ f\ x = e \mid l\ v \mid \{\ l_i = v_i\ \}_{i=1}^{k} \mid \{\ l_i\ x_i \Rightarrow e_i\ \}_{i=1}^{k} \\
e & ::= & n \mid \mathbf{fun}\ f\ x = e \mid x \mid l\ e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\
 & & \{\ l_i = e_i\ \}_{i=1}^{k} \mid e_1 \otimes \{l = e_2\} \mid e \oslash l \mid \\
 & & \{\ l_i\ x_i \Rightarrow e_i\ \}_{i=1}^{k} \mid e_1 \oplus \{\ l\ x \Rightarrow e_2\ \} \mid e \ominus l \mid \\
 & & e.l \mid \mathbf{match}\ e_1 \mathbf{with}\ e_2
\end{array}
$$

Figure 6. The abstract syntax of PolyR.

2.4 Linearly composable extensions

One major weakness of the two extensions (`‘If` and `‘LetCC`) shown so far is that they are not orthogonal since each of them explicitly extends the *original* converter rather than another, potentially already extended version. But with the mechanisms shown, this deficiency can be overcome quite easily by parameterizing the extension over what is being extended. The resulting pattern is shown in Figure 5. Notice how the two extensions have become “slices” that are *composable* by “layering” or “stacking.”

One can take the idea of extension composition to the extreme by using a programming style (or “pattern”) where every case is written individually as a slice in the above sense. Given k such slices, one could then easily generate a converter for any of the 2^k possible input languages simply by layering the corresponding subset of slices. To support this idea, **MLPolyR** provides the syntactic form **nocases** of type $\langle\rangle \hookrightarrow \alpha$ for any α. This form represents “no functionality” and can be used as the “base” upon which to layer.

3. The PolyR Language

This section describes a language, called PolyR, with polymorphic, extensible sums, records, and first-class cases. To compile PolyR, we first translate it into a version of System F, called $\mathsf{F_R}$ (Section 3.2), that has support for records only. For brevity, we give the static semantics for PolyR and the translation to $\mathsf{F_R}$ together (Section 3.3). Section 3.5 describes the translation from $\mathsf{F_R}$ into an untyped lambda calculus.

3.1 Abstract Syntax

Figure 6 shows the abstract syntax for the language PolyR. The meta-variable x (and variants) range over variables and the meta-variable l (and variants) ranges over an unspecified set of labels. The meta variables α and β (and variants) range over type and row-type variables respectively. The variables, labels, type variables, and row-type variables are mutually disjoint sets.

$$
\begin{array}{lcl}
\bar\tau & ::= & \alpha \mid \mathbf{int} \mid \bar\tau_1 \to \bar\tau_2 \mid \{\bar\rho\} \mid \alpha\ \mathbf{as}\ \bar\tau \mid \\
 & & \forall(\alpha_1, \ldots, \alpha_m).\ \forall(\beta_1 : \kappa_1, \ldots, \beta_n : \kappa_n).\bar\tau \\
\bar\rho & ::= & \beta \mid \beta \rightarrowtail \bar\tau \mid \cdot \mid l : \bar\tau, \bar\rho \\
\kappa & ::= & \{l_1, \ldots, l_k\} \\
\bar v & ::= & n \mid \mathbf{fun}\ f\ x : \bar\tau = \bar e \mid \{\ l_i = \bar v_i\ \}_{i=1}^{k} \mid \\
 & & \Lambda(\alpha_1, \ldots, \alpha_m).\Lambda(\beta_1{::}\kappa_1, \ldots, \beta_n{::}\kappa_n).\bar e \\
\bar e & ::= & x \mid n \mid \mathbf{fun}\ f\ x : \bar\tau = \bar e \mid \mathbf{let}\ x : \bar\tau = \bar e_1\ \mathbf{in}\ \bar e_2 \\
 & & \Lambda(\alpha_1, \ldots, \alpha_m).\Lambda(\beta_1{::}\kappa_1, \ldots, \beta_n{::}\kappa_n).\bar e \mid \\
 & & \bar e_1\ \bar e_2 \mid \bar e[\bar\tau_1, \ldots, \bar\tau_m][\bar\rho_1, \ldots, \bar\rho_n] \mid \\
 & & \{\ l_i = \bar e_i\ \}_{i=1}^{k} \mid \bar e.l \mid \bar e_1 \otimes \{l = \bar e_2\} \mid \bar e \oslash l
\end{array}
$$

Figure 7. The abstract syntax of $\mathsf{F_R}$.

The types of the language are separated into (ordinary) types (denoted by τ and variants), *row-types* (denoted by ρ and variants), and type schemes (denoted by σ and variants). The types consists of type variables α, the base type `int`, function types, record types ($\{\rho\}$), sum types ($\langle\rho\rangle$), recursive sum types (α **as** $\langle\rho\rangle$), and case types $\rho \hookrightarrow \tau$. We denote the set of free type variables of a type τ by $\mathrm{FTV}(\tau)$ and that of a typing context Γ by $\mathrm{FTV}(\Gamma)$. Row-types consist of a row(-type) variables β, and possibly empty sequence of label and type matchings. The set of free row type variables of a type τ is denoted by $\mathrm{FRV}(\tau)$. For that of a typing context Γ we write $\mathrm{FRV}(\Gamma)$. The recursive sum α **as** $\langle\rho\rangle$ specifies a sum type where the type variable α can recursively occur in the definition of ρ.

Type schemes rely on *kinds*, denoted by κ (and variants), defined as sets of labels. The kinds are associated with row variables and define the labels that a row variable must not contain. *Type schemes*, denoted by σ (and variants), are defined as

$$\sigma ::= \forall(\alpha_1, \ldots, \alpha_m).\ \forall(\beta_1 : \kappa_1, \ldots, \beta_n : \kappa_n).\tau, \text{where}$$

the variables $\{\alpha_1, \ldots, \alpha_m\}$ are free type variables of τ and $\{\beta_1, \ldots, \beta_m\}$ are free row variables of τ, and $\{\kappa_1, \ldots, \kappa_n\}$ are the kinds of the free row variables.

The expressions consist of values, variables, data type constructors ($l\ e$), applications, let bindings, record expressions, case expressions, and cases. Record expressions consists of record constructors, $\{\ l_i = v_i\ \}_{i=1}^{k}$, (or equivalently $\{l_1 = v_1, \ldots, l_k = v_k\}$) record extension $e_1 \otimes \{l = e_2\}$, record subtraction $e_1 \oslash l$, and record selection $e.l$. Case expressions are symmetric to records and consists of case constructors $\{\ l_i\ x_i \Rightarrow e_i\ \}_{i=1}^{k}$ (or equivalently $\{\ l_1\ x_1 \Rightarrow e_1, \ldots, l_k\ x_k \Rightarrow e_k\ \}$), case extension $e_1 \oplus \{\ l\ x \Rightarrow e_2\ \}$, and case subtraction $e \ominus l$. A match expression $\mathbf{match}\ e_1 \mathbf{with}\ e_2$ matches e_1 to the expressions e_2 whose value must be a case. Values consist of numbers, named functions, records where each field is a value, and cases.

3.2 System F

A PolyR expression can be translated into a variant of System F, called $\mathsf{F_R}$, that has records and named functions. Figure 7 shows the syntax for the $\mathsf{F_R}$ language. For the rest of the paper, we use the term “System F” to refer to the $\mathsf{F_R}$ language. The language can be derived from PolyR by excluding sum types, case types, operations on sum types and cases, adding type abstraction, and type application. To distinguish between PolyR types and expression from $\mathsf{F_R}$ types and expressions, the $\mathsf{F_R}$ expressions and types are written with an over bar, e.g., $\bar e, \bar v, \bar\tau$.

The types of $\mathsf{F_R}$ consists of type variables, `int` type, function types, sum types, sums, recursive types, and polymorphic types. Record types are defined in terms of row types denoted by $\bar\rho$ (and variants) that consist of sequences of labeled types that can either

$$\frac{}{\beta \blacktriangleright \beta} \qquad \frac{}{\boldsymbol{.} \blacktriangleright \boldsymbol{.}} \qquad \frac{\tau \blacktriangleright \bar{\tau} \quad \rho \blacktriangleright \bar{\rho}}{l:\tau,\rho \blacktriangleright l:\bar{\tau},\bar{\rho}}$$

$$\frac{}{\beta,\bar{\tau} \blacktriangleright \beta \rightarrowtail \bar{\tau}} \qquad \frac{}{\boldsymbol{.},\bar{\tau} \blacktriangleright \boldsymbol{.}} \qquad \frac{\tau_1 \blacktriangleright \bar{\tau}_1 \quad \rho,\bar{\tau}_2 \blacktriangleright \bar{\rho}}{(l:\tau_1,\rho),\bar{\tau}_2 \blacktriangleright l:\bar{\tau}_1 \to \bar{\tau}_2,\bar{\rho}}$$

$$\frac{}{\alpha \blacktriangleright \alpha} \qquad \frac{}{\mathbf{int} \blacktriangleright \mathbf{int}} \qquad \frac{\tau_1 \blacktriangleright \bar{\tau}_1 \quad \tau_2 \blacktriangleright \bar{\tau}_2}{\tau_1 \to \tau_2 \blacktriangleright \bar{\tau}_1 \to \bar{\tau}_2}$$

$$\frac{\rho \blacktriangleright \bar{\rho}}{\{\rho\} \blacktriangleright \{\bar{\rho}\}} \qquad \frac{\rho,\alpha \blacktriangleright \bar{\rho}}{\langle\rho\rangle \blacktriangleright \forall\alpha.\,\{\bar{\rho}\} \to \alpha}$$

$$\frac{\langle\rho\rangle \blacktriangleright \bar{\tau}}{\alpha\ \mathbf{as}\ \langle\rho\rangle \blacktriangleright \alpha\ \mathbf{as}\ \bar{\tau}} \qquad \frac{\tau \blacktriangleright \bar{\tau} \quad \rho,\bar{\tau} \blacktriangleright \bar{\rho}}{\langle\rho\rangle \hookrightarrow \tau \blacktriangleright \{\bar{\rho}\}}$$

Figure 8. The translation of rows (top) and types (bottom) of PolyR to $\mathsf{F_R}$.

$$\frac{}{\Delta \vdash \boldsymbol{.} \setminus \kappa} \qquad \frac{\kappa \subseteq \Delta(\beta)}{\Delta \vdash \beta \setminus \kappa} \qquad \frac{\Delta \vdash \rho \setminus \kappa \quad l \notin \kappa}{\Delta \vdash (l:\tau,\rho) \setminus \kappa}$$

$$\frac{}{\Delta \vdash \alpha\ \mathbf{ok}} \qquad \frac{}{\Delta \vdash \mathbf{int}\ \mathbf{ok}} \qquad \frac{\Delta \vdash \tau_2\ \mathbf{ok} \quad \Delta \vdash \tau_1\ \mathbf{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \mathbf{ok}}$$

$$\frac{\Delta \vdash \rho\ \mathbf{ok}}{\Delta \vdash \{\rho\}\ \mathbf{ok}} \qquad \frac{\Delta \vdash \rho\ \mathbf{ok}}{\Delta \vdash \langle\rho\rangle\ \mathbf{ok}}$$

$$\frac{\Delta \vdash \rho\ \mathbf{ok}}{\Delta \vdash \alpha\ \mathbf{as}\ \langle\rho\rangle\ \mathbf{ok}} \qquad \frac{\Delta \vdash \rho\ \mathbf{ok} \quad \Delta \vdash \tau\ \mathbf{ok}}{\Delta \vdash \langle\rho\rangle \hookrightarrow \tau\ \mathbf{ok}}$$

$$\frac{}{\Delta \vdash \beta\ \mathbf{ok}} \qquad \frac{}{\Delta \vdash \boldsymbol{.}\ \mathbf{ok}}$$

$$\frac{\Delta \vdash \tau\ \mathbf{ok} \quad \Delta \vdash \rho \setminus \{l\} \quad \Delta \vdash \rho\ \mathbf{ok}}{\Delta \vdash l:\tau,\rho\ \mathbf{ok}}$$

Figure 9. The *lacks* relations, and well-formed types and row-types (from top to bottom in that order) .

end with an empty row $\boldsymbol{.}$, a row variable β, or a row arrow $\beta \rightarrowtail \bar{\tau}$. The key difference between the row types of the PolyR language and $\mathsf{F_R}$ language is the inclusion of the *row-arrow* $\beta \rightarrowtail \bar{\tau}$. Row arrows are critical to represent sums and cases in terms of records.

The expressions of the language consists of variables, numbers, functions, type abstractions $(\Lambda(\alpha_1,\ldots,\alpha_m).\Lambda(\beta_1{::}\kappa_1,\ldots,\beta_n{::}\kappa_n).\bar{e})$, applications, let expressions, type application $(\bar{e}[\bar{\tau}_1,\ldots,\bar{\tau}_m][\bar{\rho}_1,\ldots,\bar{\rho}_n])$, and record expressions. The values consist of numbers, functions, type abstractions, and records where each field is a value.

Throughout the paper, we omit the bindings for type variables and row-type variables when they are empty in type abstractions and type applications. For examples, we may write $\forall\beta : \kappa.\bar{\tau}$ or $\bar{e}[\bar{\rho}/\beta]$ when no type variables are quantified.

3.3 Static Semantics and Translation to System F

We present the static semantics of PolyR and simultaneously show the translation of PolyR to System F.

Figure 8 shows the translation for row arrows and the translation of types of the PolyR language to those of $\mathsf{F_R}$. Row-types are translated either directly, written as $\rho \blacktriangleright \bar{\rho}$, or in the context of a type $\bar{\tau}$, written as $\rho,\bar{\tau} \blacktriangleright \bar{\rho}$. The translation $\rho \blacktriangleright \bar{\rho}$ translates ρ pointwise by translating each field. The translation $\rho,\bar{\tau}_2 \blacktriangleright \bar{\rho}$ translates each field of type τ_1 into the type $\bar{\tau}_1 \to \bar{\tau}_2$, where $\bar{\tau}_1$ is obtained by translating τ_1, and the row-type variable β into $\beta \rightarrowtail \bar{\tau}_2$.

The types of PolyR are translated into System F by translating sum types and case types into records (Figure 8). The most interesting rules concern the translation of sum types $\langle\rho\rangle$ and cases. Sum types are translated into record types where each field is a function from a member of the sum type to a universally quantified type variable α. More precisely, the sum type $\langle\rho\rangle$ is translated by first translating the row type ρ into $\bar{\rho}$ under a type variable α and then generalizing the function type $\{\bar{\rho}\} \to \alpha$. For example, the sum type $\langle l_1 : \mathbf{int}, l_2 : \mathbf{int} \to \mathbf{int}\rangle$ is translated into the type $\forall\alpha.\,\{l_1 : \mathbf{int} \to \alpha, l_2 : (\mathbf{int} \to \mathbf{int}) \to \alpha\} \to \alpha$.

The typing judgments for PolyR (Figure 10) are non-deterministic. Therefore, care must be taken to not introduce ill-formed types when "guessing" the types of functions, i.e., when creating an instance of a polymorphic type, and when constructing bigger row-types from existing row-types. Figure 9 defines the notion of well-formed types and row-types. The definitions rely on a *lacks* relation between rows and sets of labels. We say that a row ρ *lacks* a set of labels κ under the kinding context Δ, denoted $\Delta \vdash \rho \setminus \kappa$, if ρ does not contain any of the labels from κ; if ρ contains a row-type variable β, then β cannot contain the labels from κ. We say that a row-type ρ is well formed under Δ, denoted $\Delta \vdash \rho\ \mathbf{ok}$ if ρ consists of distinct labels and lacks the labels specified by the kinding environment. We say that a type τ is well-formed under some kinding context Δ, denoted $\Delta \vdash \tau\ \mathbf{ok}$, if all row-(sub)types of τ are well formed under Δ.

Figure 10 shows the typing rules for PolyR and their translation to System F. The judgments take place under a kinding context Δ and the typing context Γ. The kinding context maps row variables to kinds—the kind of a row variable is the set of labels that the variable is known not to contain. The typing context maps (ordinary) variables to type schemes. The judgments take the form $\Delta,\Gamma \vdash e : \tau \blacktriangleright \bar{e} : \bar{\tau}$ and state that, under the kinding context Δ and the typing context Γ, the PolyR expression e has type τ and translates to the $\mathsf{F_R}$ expression $\bar{e}$ with $\bar{\tau}$. The following lemma states that the translation preserves the types of terms with respect to the translation. The proof of this lemma is omitted here.

Lemma 1
If $\Delta,\Gamma \vdash e : \tau \blacktriangleright \bar{e} : \bar{\tau}$*, then* $\tau \blacktriangleright \bar{\tau}$.

The most interesting judgments are those that introduce and eliminate polymorphism (the `let/val` and the `var` judgments), and those that operate on sums, records, and cases.

The PolyR language supports ML-style polymorphism (let polymorphism). The type checking of a let expression depends on whether the expression whose value is being bound is a syntactic value or not. If the expression is of the form $\mathbf{let}\ x = v_1\ \mathbf{in}\ e_2$, then the type of the value $\bar{\tau}_1$ is generalized over all free type variables and row-type variables; the generalization requires constructing a kind for each row type (let/val judgment). If the expression is of the form $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$, where e_1 is not a value, then the type of e_1 is not generalized (let/non-val judgment). There are two motivations behind differentiating between syntactic values and non-values: 1) it ensures that the transformation to System F preserves non-termination semantics of the program, and 2) it makes it easier to extend the language with side effects (e.g., references). When used, a variable with a polymorphic type is instantiated to a non-polymorphic type by selecting types and row types for its polymorphic variables (var rule). An instantiation is translated into System F as a type application.

$$\frac{\begin{array}{c}\Gamma(x) = \forall\alpha_1 \ldots \alpha_m.\forall\beta_1 :: \kappa_1 \ldots \beta_n :: \kappa_n.\tau' \\ \forall i.\Delta \vdash \tau_i \ \mathbf{ok} \qquad \forall j.(\Delta \vdash \rho_j \ \mathbf{ok}\ ) \wedge (\Delta \vdash \rho_j \setminus \kappa_j) \\ \tau = \tau'[\tau_i/\alpha_i, \rho_j/\beta_j]_{i=1\ldots m, j=1\ldots n} \\ \tau_i \blacktriangleright \bar{\tau}_i \qquad \rho_i \blacktriangleright \bar{\rho}_i \qquad \tau \blacktriangleright \bar{\tau}\end{array}}{\Delta;\Gamma \vdash x : \tau \quad \blacktriangleright \quad x[\bar{\tau}_1,\ldots,\bar{\tau}_m][\bar{\rho}_1,\ldots,\bar{\rho}_n] : \bar{\tau}}\ \textbf{(var)} \qquad \frac{}{\Delta;\Gamma \vdash n : \mathbf{int} \blacktriangleright n : \mathbf{int}}\ \textbf{(int)}$$

$$\frac{\Delta;\Gamma, f : \tau_2 \to \tau, x : \tau_2 \vdash e : \tau \blacktriangleright \bar{e} : \bar{\tau} \quad \Delta \vdash \tau_2 \ \mathbf{ok} \quad \tau_2 \blacktriangleright \bar{\tau}_2}{\Delta;\Gamma \vdash \mathbf{fun}\ f\ x = e : \tau_2 \to \tau \blacktriangleright \mathbf{fun}\ f\ x = \bar{e} : \bar{\tau}_2 \to \bar{\tau}}\ \textbf{(fun)}$$

$$\frac{\Delta;\Gamma \vdash e : \tau \blacktriangleright \bar{e} : \bar{\tau} \quad \Delta \vdash (l : \tau, \rho)\ \mathbf{ok} \quad (l : \tau, \rho), \alpha \blacktriangleright \bar{\rho}}{\Delta;\Gamma \vdash l\ e : \langle l : \tau, \rho\rangle \blacktriangleright (\mathbf{let}\ x_v : \bar{\tau} = \bar{e}\ \mathbf{in}\ \Lambda\alpha.\mathbf{fun}\ \_\ x_r = x_r.l\ x_v) : \forall\alpha.\{\bar{\rho}\} \to \alpha}\ \textbf{(data const.)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau \blacktriangleright \bar{e}_1 : \bar{\tau}_2 \to \bar{\tau} \quad \Delta;\Gamma \vdash e_2 : \tau_2 \blacktriangleright \bar{e}_2 : \bar{\tau}_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau \blacktriangleright \bar{e}_1\ \bar{e}_2 : \bar{\tau}}\ \textbf{(app)}$$

$$\frac{\begin{array}{c}\vec{\alpha} = \alpha_1,\ldots,\alpha_m = \mathrm{FTV}(\tau_1) \setminus \mathrm{FTV}(\Gamma) \quad \beta_1,\ldots,\beta_n = \mathrm{FRV}(\tau_1) \setminus \mathrm{FRV}(\Gamma) \quad \overrightarrow{\beta :: \kappa} = \beta_1 :: \kappa_1,\ldots,\beta_n :: \kappa_n \\ \Delta, \overrightarrow{\beta :: \kappa};\Gamma \vdash e_1 : \tau_1 \blacktriangleright \bar{e}_1 : \bar{\tau}_1 \quad \Delta;\Gamma, x : \forall\vec{\alpha}.\forall\overrightarrow{\beta :: \kappa}.\tau_1 \vdash e_2 : \tau_2 \blacktriangleright \bar{e}_2 : \bar{\tau}_2 \quad e_1 \text{is a syntactic value}\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \blacktriangleright \mathbf{let}\ x : \forall\vec{\alpha}.\forall\overrightarrow{\beta :: \kappa}.\bar{\tau}_1 = \Lambda\vec{\alpha}.\Lambda\overrightarrow{\beta :: \kappa}.\bar{e}_1\ \mathbf{in}\ \bar{e}_2 : \bar{\tau}_2}\ \textbf{(let/val)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \blacktriangleright \bar{e}_1 : \bar{\tau}_1 \quad \Delta;\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \blacktriangleright \bar{e}_2 : \bar{\tau}_2 \quad e_1 \text{is not a syntactic value}}{\Delta;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \blacktriangleright \mathbf{let}\ x : \bar{\tau}_1 = \bar{e}_1\ \mathbf{in}\ \bar{e}_2 : \bar{\tau}_2}\ \textbf{(let/non-val)}$$

$$\frac{\Delta;\Gamma \vdash e : \langle\rho[\alpha\ \mathbf{as}\ \langle\rho\rangle/\alpha]\rangle \blacktriangleright \bar{e} : \bar{\tau}[\alpha\ \mathbf{as}\ \bar{\tau}/\alpha]}{\Delta;\Gamma \vdash e : \alpha\ \mathbf{as}\ \langle\rho\rangle \blacktriangleright \bar{e} : \alpha\ \mathbf{as}\ \bar{\tau}}\ \textbf{(roll)} \qquad \frac{\Delta;\Gamma \vdash e : \alpha\ \mathbf{as}\ \langle\rho\rangle \blacktriangleright \bar{e} : \alpha\ \mathbf{as}\ \bar{\tau}}{\Delta;\Gamma \vdash e : \langle\rho[\alpha\ \mathbf{as}\ \langle\rho\rangle/\alpha]\rangle \blacktriangleright \bar{e} : \bar{\tau}[\alpha\ \mathbf{as}\ \bar{\tau}/\alpha]}\ \textbf{(unroll)}$$

$$\frac{\begin{array}{c}\forall i.\Delta;\Gamma \vdash e_i : \tau_i \blacktriangleright \bar{e}_i : \bar{\tau}_i \\ \Delta \vdash l_1,\ldots,l_k\ \mathbf{ok}\end{array}}{\begin{array}{lll}\Delta;\Gamma & \vdash & \{\, l_i = e_i \,\}_{i=1}^k : \{\, l_i : \tau_i \,\}_{i=1}^k \\ & \blacktriangleright & \{\, l_i = \bar{e}_i \,\}_{i=1}^k : \{\, l_i : \bar{\tau}_i \,\}_{i=1}^k\end{array}}\ \textbf{(r)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \{\rho\} \quad \blacktriangleright \quad \bar{e}_1 : \{\bar{\rho}\} \\ \Delta \vdash \rho \setminus \{l\} \\ \Delta;\Gamma \vdash e_2 : \tau_2 \quad \blacktriangleright \quad \bar{e}_2 : \bar{\tau}_2\end{array}}{\begin{array}{lll}\Delta;\Gamma & \vdash & e_1 \otimes \{l = e_2\} : \{l : \tau_2, \rho\} \\ & \blacktriangleright & \bar{e}_1 \otimes \{l = \bar{e}_2\} : \{l : \bar{\tau}_2, \bar{\rho}\}\end{array}}\ \textbf{(r/ext)}$$

$$\frac{\Delta;\Gamma \vdash e : \{l : \tau, \rho\} \blacktriangleright \bar{e} : \{l : \bar{\tau}, \bar{\rho}\}}{\Delta;\Gamma \vdash e \oslash l : \{\rho\} \blacktriangleright \bar{e} \oslash l : \{\bar{\rho}\}}\ \textbf{(r/sub)}$$

$$\frac{\Delta;\Gamma \vdash e : \{l : \tau, \rho\} \blacktriangleright \bar{e} : \{l : \bar{\tau}, \bar{\rho}\}}{\Delta;\Gamma \vdash e.l : \tau \blacktriangleright \bar{e}.l : \bar{\tau}}\ \textbf{(select)}$$

$$\frac{\begin{array}{c}\forall i.(\Delta;\Gamma \vdash \tau_i\ \mathbf{ok}\ ) \wedge (\Delta;\Gamma, x_i : \tau_i \vdash e_i : \tau \blacktriangleright \bar{e}_i : \bar{\tau}) \\ \Delta \vdash l_1,\ldots,l_k\ \mathbf{ok} \qquad \forall i.(\tau_i \blacktriangleright \bar{\tau}_i)\end{array}}{\begin{array}{lll}\Delta;\Gamma & \vdash & \{\, l_i\ x_i \Rightarrow e_i \,\}_{i=1}^k : \langle\, l_i : \tau_i \,\rangle_{i=1}^k \hookrightarrow \tau \\ & \blacktriangleright & \{\, l_i = \mathbf{fun}\ \_\ x_i : \bar{\tau}_i = \bar{e}_i \,\}_{i=1}^k : \{\, l_i : \bar{\tau}_i \to \bar{\tau} \,\}_{i=1}^k\end{array}}\ \textbf{(c)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \langle\rho\rangle \hookrightarrow \tau \quad \blacktriangleright \quad \bar{e}_1 : \{\bar{\rho}\} \\ \Delta \vdash \rho \setminus \{l\} \qquad \Delta \vdash \tau_1\ \mathbf{ok} \qquad \tau_1 \blacktriangleright \bar{\tau}_1 \\ \Delta;\Gamma, x : \tau_1 \vdash e_2 : \tau \quad \blacktriangleright \quad \bar{e}_2 : \bar{\tau}\end{array}}{\begin{array}{lll}\Delta;\Gamma & \vdash & e_1 \oplus \{\, l\ x \Rightarrow e_2 \,\} : \langle l : \tau_1, \rho\rangle \hookrightarrow \tau \\ & \blacktriangleright & \bar{e}_1 \otimes \{l = \mathbf{fun}\ \_\ x : \bar{\tau}_1 = \bar{e}_2\} : \{l : \bar{\tau}_1 \to \bar{\tau}, \bar{\rho}\}\end{array}}\ \textbf{(c/ext)}$$

$$\frac{\Delta;\Gamma \vdash e : \langle l : \tau_1, \rho\rangle \hookrightarrow \tau \blacktriangleright \bar{e} : \{l : \bar{\tau}_1 \to \bar{\tau}, \bar{\rho}\}}{\Delta;\Gamma \vdash e \ominus l : \langle\rho\rangle \hookrightarrow \tau \blacktriangleright \bar{e} \oslash l : \{\bar{\rho}\}}\ \textbf{(c/sub)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \langle\rho\rangle \blacktriangleright \bar{e}_1 : \forall\alpha.(\{\bar{\rho}_\alpha\} \to \alpha) \\ \Delta;\Gamma \vdash e_2 : \langle\rho\rangle \hookrightarrow \tau \blacktriangleright \bar{e}_2 : \{\bar{\rho}_\tau\}\end{array}}{\Delta;\Gamma \vdash \mathbf{match}\ e_1 \mathbf{with}\ e_2 : \tau \blacktriangleright \bar{e}_1[\bar{\tau}]\ \bar{e}_2 : \bar{\tau}}\ \textbf{(match)}$$

Figure 10. The static semantics and translation for basic terms (top), and records and cases.

The bottom box in Figure 10 shows the typing rules records (left) and cases (right). The judgments are arranged to bring out the symmetry between these rules.

A record constructor is assigned the record type that maps the labels and to the types of the corresponding fields as long as the labels are distinct. The type of a record extension $e_1 \otimes \{l = e_2\}$ is a record type that extends the type $\{\rho\}$ of the record e_1 with the label l under the condition that l is not included in the record type. The type of a record subtraction $\bar{e} \oslash l$ is a record where label l is excluded under the condition that $\bar{e}$ contains l. The type of a record selection is the type of the field l being selected, under the condition that the record expression contains the field with label l. Since the $\mathsf{F_R}$ language includes the record expressions included in PolyR, all record expressions are translated into the $\mathsf{F_R}$ language directly.

A case constructor is assigned a case type that identifies the result type τ of the bodies (e_i's) and maps each label l_i to its domain type τ_i. A case is translated into a record of functions, one for each label l_i, whose argument type is equal to the domain type $\bar{\tau}_i$ of l_i and whose body is the body of the case e_i. A case extension extends the type of a case with a new branch. A case extension is translated to a record extension. A case subtraction takes out the specified branch from a case type and translates it into a record subtraction. A match expression $\mathbf{match}\ e_1 \mathbf{with}\ e_2$ is well typed if the domain type of e_2 is the same as the type e_1. Since data constructors are transformed into functions that select the appropriate function from their argument and apply their value to that function, a match expression is compiled into a function application. Since translated sum expressions have polymorphic type, this requires instantiating the function type first. We note that

$$
\begin{array}{rcl}
t & ::= & n \mid x \mid t_1 + t_2 \mid t_1 - t_2 \mid \mathbf{len}(t) \mid \\
 & & \mathbf{fun}\ f\ x = t \mid t_1\ t_2 \mid \langle\ s_i\ \rangle_{i=1}^{n} \mid t.t \mid \\
 & & \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
s & ::= & t \mid (t, t, t) \\
v & ::= & n \mid \mathbf{fun}\ x\ t_1 = t_2 \mid
\end{array}
$$

Figure 11. The abstract syntax for LRec.

the symmetry to a selection is indirect (through the translation of data constructors).

3.4 Dynamic Semantics

The dynamic semantics of PolyR is mostly standard. The full semantics is given in Appendix A. Although the PolyR language is purely functional, the dynamic semantics is written with the same implicit threading of state in mind that is also used by the Definition of Standard ML [20]. This removes all non-determinism by enforcing an evaluation order. The primary motivation for this is to enable a precise specification of the transformation of PolyR into an untyped lambda calculus (Section 3.5) without altering the execution order. A secondary motivation is to ensure that the transformation would be consistent with imperative features, if the languages are extended with them.

3.5 Translation to Untyped Lambda Calculus

We describe the translation of System F expressions (Section 3.2) into an untyped language, called LRec. The LRec language extends the untyped lambda calculus with (n-ary) tuples and named functions; Figure 11 shows the abstract syntax for LRec. The terms of the language, denoted by t (and variants), consist of numbers n, variables x, the operations plus and minus, the $\mathbf{len}(t)$ for determining the number fields in a tuple t, named functions, function application, and introduction and eliminations forms for tuples. The introduction form for uples, $\langle\ s_i\ \rangle_{i=1}^{n}$, specifies a sequence of slices from which the tuple is being constructed. The elimination form for tuples is selection (projection), written $t_1.t_2$, that projects out the field with index t_2 from the tuple t_1. The terms include a let expression (as syntactic sugar for application). A *slice*, denoted by s (and variants), is either a term, or a triple of terms (t_1, t_2, t_2), where the t_1 is a record and t_2 and t_3 are numbers. A slice (t_1, t_2, t_3) specifies consecutive fields of the record t_1 between the indices of t_2 (including) and t_3 (excluding).

Figure 12 shows the dynamic semantics for LRec. We enforce an order on evaluation by assuming that the premises are evaluated from left to right and top to bottom (in that order). The semantics is largely standard. The only interesting judgments concern evaluation of slices and construction of tuples. Slices evaluate to a sequence of values selected by the specified indices (if any). Tuple selection projects out the specified field with the specified index from the tuple. Since tuples can be implemented as arrays, selection can be implemented in constant time. Thus, if records can be transformed into tuples and record selection can be transformed into tuple selection, record operations can be implemented in constant time. The computation of the indices is the key component of the translation from System F to LRec.

Figure 13 shows the translation from System F (the $\mathsf{F_R}$ language) into the LRec language. The translation takes place in the context of an *index context*, denoted by Δ that maps row variables to sets consisting of label and term pairs. More precisely, for a row variable β, $\Delta(\beta) = \{(l_1, t_1), \ldots, (l_k, t_k)\}$, where t_i is the term that will aid in computing the index for l_i in a record. We write $\Delta(\beta)(l)$ for the index (term) of l for β, i.e., if $(l, t) \in \Delta(\beta)$, then $\Delta(\beta)(l) = t$. Given Δ, the kind of a row variable β, denoted $\kappa(\Delta, \beta)$ can be recovered by projecting out the labels. More precisely $\kappa(\Delta, \beta)$ is defined as $\kappa(\Delta, \beta) = \{l \mid (l, t) \in \Delta(\beta)\}$.

$$
\frac{}{v \Downarrow v}\ \textbf{(val)}
$$

$$
\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 + t_2 \Downarrow n_1 + n_2}\ \textbf{(plus)}
\qquad
\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 - t_2 \Downarrow n_1 - n_2}\ \textbf{(minus)}
$$

$$
\frac{\begin{array}{c} t_1 \Downarrow \mathbf{fun}\ f\ x = t_1' \qquad t_2 \Downarrow v_2 \\ t_1'[\mathbf{fun}\ f\ x = t_1'/f_1, v_2/x] \Downarrow v \end{array}}{t_1\ t_2 \Downarrow v}\ \textbf{(app)}
$$

$$
\frac{t_1 \Downarrow v_1 \quad t_2[v_1/x] \Downarrow v}{\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \Downarrow v}\ \textbf{(let)}
\qquad
\frac{t \Downarrow \langle v_0, \ldots, v_{n-1} \rangle}{\mathbf{len}(t) \Downarrow n}\ \textbf{(length)}
$$

$$
\frac{t_1 \Downarrow \langle v_0, \ldots, v_i, \ldots, v_{n-1} \rangle \quad t_2 \Downarrow i \quad 0 \le i < n}{t_1.t_2 \Downarrow v_i}\ \textbf{(select)}
$$

$$
\frac{s_1 \Downarrow_s v_{1,0}, \ldots, v_{1,k_1-1} \quad \ldots \quad s_n \Downarrow_s v_{n,0}, \ldots, v_{n,k_n-1}}{\langle s_1, \ldots, s_n \rangle \Downarrow \langle v_{1,0}, \ldots, v_{1,k_1-1}, \ldots, v_{n,0}, \ldots, v_{n,k_n-1} \rangle}\ \textbf{(tuple)}
$$

$$
\frac{t \Downarrow v}{t \Downarrow_s v}\ \textbf{(slice/singleton)}
$$

$$
\frac{\begin{array}{c} t_1 \Downarrow \langle v_0, \ldots, v_i, \ldots, v_j, \ldots, v_{n-1} \rangle \\ t_2 \Downarrow i \qquad t_3 \Downarrow j \qquad 0 \le i \le j \le n \end{array}}{(t_1, t_2, t_3) \Downarrow_s v_i, \ldots, v_{j-1}}\ \textbf{(slice/sequence)}
$$

Figure 12. The dynamic semantics for LRec.

The translation of numbers, variables, functions, applications, and let expressions are straightforward. A record is translated into a tuple of slices, each of which is obtained by translating the label expressions. The slices are sorted based on the corresponding labels. Since sorting can re-arrange the ordering of the fields, the transformation first evaluates the fields in their original order by binding them to variables and then constructs the tuple using these variables.

A record selection is translated by computing the index for the label being projected based on the type of the record. To compute indices for record labels, the translation relies on two operations. Given a set of labels κ and a label l, define the *position* of l in κ, denoted $\mathrm{pos}(l, \kappa)$, as the number of labels of l that are less than l in the total order defined on the labels. Formally, $\mathrm{pos}(l, \kappa) = |\{l' \mid l' \in \kappa \wedge l' <_l l\}|$, where $<_l$ denotes the ordering relation on the labels. For a given row ρ, define the $\mathrm{labels}(\rho)$, to be the pair consisting of the set of variables of ρ and the remainder row, which is either empty or a row variable. More precisely:

$$
\begin{array}{rcl}
\mathrm{labels}(\{l_1, \ldots, l_k, \cdot\}) & = & (\{l_1, \ldots, l_k\}, \cdot) \\
\mathrm{labels}(\{l_1, \ldots, l_k, \beta\}) & = & (\{l_1, \ldots, l_k\}, \beta) \\
\mathrm{labels}(\{l_1, \ldots, l_k, \beta \rightarrowtail \tau\}) & = & (\{l_1, \ldots, l_k\}, \beta)
\end{array}
$$

Notice that we treat $\beta \rightarrowtail \tau$ just like plain β, taking advantage of the fact that $(\beta \rightarrowtail \tau) \setminus l$ if and only if $\beta \setminus l$.

Let τ be some record type, and let $(L, \rho) = \mathrm{labels}(\tau)$. We compute the *index* of a label l in τ, denoted $\mathrm{indexOf}(\Delta, l, (L, \rho))$, as follows:

$$
\begin{array}{rcl}
\mathrm{indexOf}(\Delta, l, (L, \cdot)) & = & \mathrm{pos}(l, L) \\
\mathrm{indexOf}(\Delta, l, (L, \beta)) & = & \Delta(\beta)(l) - \mathrm{pos}(l, \kappa(\Delta, \beta) \setminus L)
\end{array}
$$

To compute the indices for labels, the translation requires access to the System F types of certain expressions. We denote the type of an expressions e by $\mathrm{typeOf}(e)$.

The record extension $e_1 \ \otimes\ \{l = e_2\}$ is translated by first finding the index of l in the tuple corresponding to e_1, and then splitting the tuple into two slices at that index, and creating a tuple that consists of the these two slices along with a slice consisting

$$\frac{}{\Delta \vdash n \rhd n}\ \textbf{(int)} \qquad \frac{}{\Delta \vdash x \rhd x}\ \textbf{(var)}$$

$$\frac{\Delta \vdash e \rhd t}{\Delta \vdash \mathbf{fun}\ f\ x : \tau = e \rhd \mathbf{fun}\ f\ x = t}\ \textbf{(fun)}$$

$$\frac{\Delta \vdash e_1 \rhd t_1 \quad \Delta \vdash e_2 \rhd t_2}{\Delta \vdash e_1\ e_2 \rhd t_1\ t_2}\ \textbf{(app)}$$

$$\frac{\Delta \vdash e_1 \rhd t_1 \quad \Delta \vdash e_2 \rhd t_2}{\Delta \vdash \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 \rhd \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2}\ \textbf{(let)}$$

$$\frac{\Delta \vdash e \rhd t \quad t' = \mathrm{indexOf}(\Delta, l, \mathrm{labels}(\mathrm{typeOf}(e)))}{\Delta \vdash e.l \rhd t.t'}\ \textbf{(select)}$$

$$\frac{\begin{array}{c}\forall i, j. i < j \Rightarrow l_{\#(i)} <_l l_{\#(j)} \\ \{l_{\#(1)}, \ldots, l_{\#(n)}\} = \{l_1, \ldots, l_n\} \\ \forall i.\, (\Delta \vdash e_i \rhd t_i)\end{array}}{\begin{array}{lll}\Delta & \vdash & \{\ l_i = e_i\ \}_{i=1}^{n} \\ & \rhd & \mathbf{let}\ x_1 = t_1\ \mathbf{in} \ldots \mathbf{let}\ x_n = t_n\ \mathbf{in}\ \langle\ x_{\#(i)}\ \rangle_{i=1}^{n}\end{array}}\ \textbf{(r)}$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 \rhd t_1 \qquad \Delta \vdash e_2 \rhd t_2 \\ t_0 = \mathrm{indexOf}(\Delta, l, \mathrm{labels}(\mathrm{typeOf}(e_1)))\end{array}}{\begin{array}{lll}\Delta & \vdash & e_1 \otimes \{l = e_2\} \\ & \rhd & \mathbf{let}\ x = t_1\ \mathbf{in}\ \langle (x, 0, t_0), t_2, (x, t_0, \mathbf{len}(x)) \rangle\end{array}}\ \textbf{(r/ext)}$$

$$\frac{\begin{array}{c}\Delta \vdash e \rhd t \\ t_0 = \mathrm{indexOf}(\Delta, l, \mathrm{labels}(\mathrm{typeOf}(e)))\end{array}}{\begin{array}{lll}\Delta & \vdash & e \oslash l \\ & \rhd & \mathbf{let}\ x = t\ \mathbf{in}\ \langle (x, 0, t_0), (x, t_0 + 1, \mathbf{len}(x)) \rangle\end{array}}\ \textbf{(r/sub)}$$

$$\frac{\begin{array}{c}\Delta, \ldots, \beta_i{::}\{(l_i^1, x_i^1), \ldots, (l_i^{m_i}, x_i^{m_i})\}, \ldots \vdash \bar{e} \rhd t \\ \forall i.\, 1 \le i \le n.\, \kappa_i = \{l_i^1, \ldots, l_i^{m_i}\}\end{array}}{\begin{array}{lll}\Delta & \vdash & \Lambda(\alpha_1, \alpha_k).\Lambda\,(\beta_1{::}\kappa_1, \ldots, \beta_n{::}\kappa_n).\bar{e} \\ & \rhd & \lambda x_1^1 \ldots \lambda x_1^{m_1} \ldots \lambda x_n^1 \ldots \lambda x_n^{m_n}.\, t\end{array}}\ \textbf{(ty/abs)}$$

$$\frac{\begin{array}{c}\Delta \vdash e \rhd t \\ \mathrm{typeOf}(e) = \forall\,(\beta_1{::}\kappa_1 \ldots \beta_n{::}\kappa_n).\, \tau \\ \forall i.\, 1 \le i \le n.\, \kappa_i = \{l_i^1, \ldots, l_i^{m_i}\} \\ \forall i \in \{1, \ldots, n\} \forall j \in \{1, \ldots, m_i\}. \\ t_i^j = \mathrm{indexOf}(\Delta, l_i^j, (L_j \cup \kappa_i, \rho_i')) \\ \text{where } (L_i, \rho_i') = \mathrm{labels}(\{\rho_i\})\end{array}}{\begin{array}{lll}\Delta & \vdash & e[\tau_1, \ldots, \tau_k][\rho_1, \ldots, \rho_n] \\ & \rhd & t\ t_1^1 \ldots t_1^{m_1}\ \ldots\ t_n^1 \ldots t_n^{m_n}\end{array}}\ \textbf{(ty/app)}$$

Figure 13. The translation from the $\mathsf{F_R}$ into the LRec language.

of the new field. Similarly, record subtraction splits the tuple for the record immediately before and immediately after the label being subtracted into two slices and creates a tuple from these slices. Type abstractions are translated into functions by creating an argument x_i^j for each label l_i^j in the kind κ_i of the β_i. Note that abstractions of ordinary type variables (α_i's) are simply dropped. Type applications are transformed into function applications by generating "evidence" for each substituted row-type variable. As with type abstractions, substitutions into ordinary type variables are dropped. Evidence generation requires computing the indices of each label $l_i^j \in \kappa_i$ in any record type that extends $\{\rho_i\}$ by adding fields for every such l_i^j.

4. Implementation

The compiler for **MLPolyR** is written in Standard ML. It compiles to relatively simple, yet reasonably efficient PowerPC assembly code. The output can be assembled and executed under Mac OS X.

4.1 Basic language features

As currently implemented, the **MLPolyR** language takes a small subset of the Standard ML core language and extends it with the following features:

- Ohori-style record polymorphism
- polymorphic functional record extension and polymorphic functional record trimming (dropping of fields via "row capture" patterns)
- inferred row-polymorphic sum types and equi-recursive types
- extensible first-class cases
- mutable record fields

4.2 Compiler Phases

The compiler is structured in a fairly traditional way and consists of the following phases:

lexer lexical analysis, tokenization

parser LALR(1) parser, generating abstract syntax trees (AST)

elaborator perform type reconstruction and generation of annotated abstract syntax (Absyn)

translate generate index-passing LRec code

anf-convert convert LRec code into A-normal form [7]

flatten flatten arguments, eliminating most record- and tuple arguments by passing fields separately (i.e., in individual registers)

uncurry eliminate of most curried functions

anf-optimize constant folding, simple constant- and value propagation, elimination of useless bindings, short-circuit selection from known tuples, inline tiny functions, some arithmetic expression simplification; execution of this pass is repeated and interleaved with other optimization phases (e.g., flatten and uncurry)

closure convert to first-order code by closure conversion

clusters separate closure-converted blocks into clusters of blocks; each cluster roughly corresponds to a single C function but may have multiple entry points

treeify re-grow larger expression trees to make tree-tiling instruction selection more useful

traceschedule arrange basic blocks to minimize unconditional jumps

cg instruction selection by tree-tiling (maximum-munch algorithm)

regalloc graph-coloring register allocation

emit generate assembly code

4.3 Type-checking and translation

Type reconstruction is performed by a variant of the classic algorithm W [19], augmented to handle Rémy-style row polymorphism and equi-recursive types. Resembling the corresponding parts of other compilers (e.g., SML/NJ [2]), the process of type checking and translation is divided into two phases: *elaboration* and *translation*.

The elaboration phase takes an abstract syntax tree and annotates it with type information, using an imperative-style unification algorithm as a subroutine. It permits equi-recursive types as long as type-level recursion goes though at least one sum type[7] by selectively turning the occurs check off. To avoid looping, the implemen-

[7] This is a pragmatic implementation decision based on experience with fully general equi-recursive types that seems to indicate that most of the

tation of unification variables employs a union-find data structure that is used to efficiently detect cycles. To enable the translation phase to properly insert type abstractions and type applications, the elaborator leaves *poly-row information* consisting of row type variables and label sets in the annotated syntax tree.

The translation phase combines generation of System F-code and the transformation to index-passing LRec-code into a single step. This means that in the current compiler there is no manifestation of the System F language.

4.4 Implementation of extensible polymorphic records

Indexing. Our implementation of polymorphic record indexing is essentially equivalent to that of Ohori's SML# [24]. Values that are polymorphic in some row variable turn into functions taking integer indices as arguments. The index calculation is given by the $\mathrm{indexOf}(\cdot,\cdot,\cdot)$ function in section 3.5. In many cases, row-polymorphic values are themselves functions, which means that the index-passing transformation creates curried functions. In most cases, such currying is later eliminated by general-purpose uncurrying and argument-flattening passes within the optimizer.

Slices. In SML#, the only polymorphic record operation is field access. For this, it suffices to have a field selection operation where the index is given by a variable instead of a constant. In **MLPolyR**, however, due to the presence of functional record extension and row capture, the compiler must be able to generate code for constructing new records whose shape is not fully known at compile time. This is expressed by the "scatter-gather" feature of tuple construction in LRec, where the values for fields may be given as slices of other tuples.

The compiler attempts a number of optimizations on slices. In particular, if—after constant propagation and similar transformations—the endpoints of a slice become known to be constants, the slice is replaced with a sequence of individual values. Still, in the general case there will be slices that cannot be optimized away. In this case the instruction selection phase will emit code for copying slices. Using the features of the PowerPC and the memory allocation architecture used by the **MLPolyR** runtime system, the inner loop in such code is quite compact and consists of only three instructions.

Unit type. The empty record type is known as the singleton type denoted `()`.[8] The compiler normally represents the only value of this type by the scalar constant 0. However, with row capture it is possible that at runtime an empty record is created without statically knowing this to be the case. In this situation the program will actually allocate an empty record on the heap, which is supported by our garbage collector. The representation of the empty record does not matter since by soundness of the type system no program will attempt to access any field within such a value. There can be slices taken from the empty record, but those slices must necessarily be empty themselves, so no actual runtime access will take place.

Record length. In section 3.5, the LRec language came with a primitive $\mathbf{len}(\cdot)$ for obtaining the number of fields in a tuple. While length information is indeed present in the GC header of each tuple, getting access to it is potentially expensive since it incurs memory traffic. In the actual implementation, length information is passed as an additional index to a "virtual" *end-of-tuple* field. For this purpose, the type system implemented in the compiler uses slightly more complicated kinds: instead of plain sets of labels, a kind is a label set together with a boolean flag. The flag indicates whether or not length information is required for a given row variable.

time when such a type is inferred it was not actually intended by the programmer [17].

[8] In Standard ML this type is known as **unit**.

One disadvantage of this approach is that the boolean flag truly becomes part of the user-visible type. This might not be seen as a big problem, since in our compiler all types are fully inferred anyway. Still, even in our very small language the flag does show up in type error messages, which are often complicated enough already. A more complete language that allows for type annotations and comes with an ML-style module system, the programmer would have to worry about this detail when writing types and module signatures. A possible workaround would be to "clamp" the value of the flag to true, implying that we always pass length information whether it is needed it or not. Of course, this trick does have some runtime cost.

Record expressions and record patterns. In its concrete syntax, **MLPolyR** establishes a high degree of symmetry between record expressions and record patterns. In particular, *row capture* patterns generalize Standard ML's ellipsis notation. For example, one can define a function `f` as follows:

```
fun f { name, age, ... = other } = e
```

Any argument to `f` must be a record containing at least fields labeled `name` and `age`, but potentially more. Within the body e, the variables `name` and `age` are bound to the values of these fields, and `other` will be bound to a record value that contains all *other* fields except `name` and `age` that were present in the argument value. In essence, this notation combines selection and functional removal of fields.

Conversely, functional record extension is written using a record expression involving an ellipsis:

```
val fred = { name = "Fred", age = 29,
             ... = fred's_other_info }
```

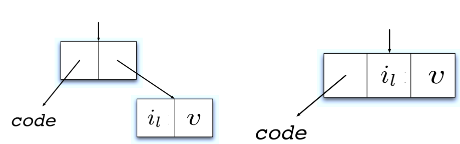
Functional record *update*, i.e., the replacement of existing fields with new fields, can be synthesized from row capture and record extension. The **MLPolyR** language provides special syntax for record update, but it's meaning can be explained as a derived form.

4.5 Implementation of sums

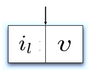
Section 3.3 shows how sums and first-class cases are completely eliminated and represented by corresponding record constructs using the well-known dual construction:

- Cases become records of functions.
- Sum values (aka "variants") lv become functions that take cases c (in form of function records) as arguments, select the function $c.l$ corresponding to label l, and invoke it with the constructor's argument v.

This encoding is elegant and has the advantage of not needing any new runtime machinery; everything is handled by the mechanisms that implement polymorphic extensible records. However, the encoding is also inefficient, both in space consumption and in performance. The variant lv becomes $\mathbf{fun}\ \_\ c = c.i_l v$ where i_l is the index corresponding to label l. Such a function value would normally be represented by a closure consisting of a code pointer and a record of the free variables, here i_l and v, in other words, at least three distinct values. Two possible ways of implementing this closure can be depicted as follows:



We obtain a less space-consuming and faster representation by observing that the code is the same for *every* element of *every* sum type! Since the compiler also knows precisely where this code is invoked, namely at call sites generated by translating **match** expressions where it can easily be inlined, the code pointer does not need to be represented at all. This leaves us with a representation of the variant as a pair consisting just of i_l and v:



But that is precisely the "traditional" representation of tagged unions, i_l playing the role of the tag. Space is saved by the elimination of the code pointer and possibly the second indirection. The time savings are due to the inlining of the code, since general function call overhead, the memory access for obtaining the entry address, and the need for an indirect jump dominate the cost of the naive implementation.

This optimization is implemented quite conveniently as part of our translation phase. The fact that, as has been noted above, we skip System F has practical benefits here. Normally, when generating plain System F code, we would lose information on which of the closures correspond to sum values, and which applications correspond to **match**. This information would either have to be recovered by some flow analysis or preserved using ad-hoc annotation on System F terms.

4.6 Coherence

Incoherence manifests itself in the translation phase as a non-generalized and uninstantiated type variable. Since the transformation discards ordinary type variables, the lack of coherence only matters when it involves row types. Here is a concrete example for how this might happen:

```
fun loop() = loop()
val x = (loop()).a
```

The type of `loop` is inferred to be $\forall\alpha.() \rightarrow \alpha$. The typing rule for field selection can pick an arbitrary instantiation for α as long as it is a record containing a field `a`. But the underspecified shape of the instantiation determines the index for accessing `a`! Notice, however, that `loop()` does not produce a record value. In fact, it will never return at all, so the index for `a` does not matter at runtime. In the elaboration/translation algorithm, this situation manifests itself as an uninstantiated (unification-) row type variable.

The phenomenon of coherence (or rather: the lack thereof) is well-known and has been studied in the context of the translation of ML into an explicitly typed calculus (a variant of System F) by Ohori [23]. It was later rediscovered in the context of Haskell's type class mechanism [15]. Like in SML#, we can take advantage of what amounts to a parametricity result for ML, namely that *closed* programs are, in fact, coherent.[9] Intuitively, whenever incoherence occurs, the actual choice of type will not matter at runtime because the code in question will never get executed. Our compiler (like Ohori's) picks arbitrary index values for labels that belong to uninstantiated (unification-) row type variables.

4.7 Runtime system

The runtime system, written in C, implements a simple two-space copying garbage collector [13] and provides basic facilities for input and output.

Data representation and memory management. For the tracing garbage collector to be able to reliably distinguish between pointers and integers, we employ the usual tagging trick. Integers are 31-bit 2's-complement numbers. An integer value i is represented internally as a 2's-complement 32-bit quantity of value $2i$. This makes all integers even, with their least significant bits cleared. Heap pointers, on the other hand, are represented as odd 32-bit values. In effect, instead of pointing to the beginning of a word-aligned heap object, they point to the object's second byte. Generated load- and store-instructions account for this skew by using an accordingly adjusted displacement value. With this representation trick, the most common arithmetic operations (addition and subtraction) can be implemented as single instructions as usual; they do not need to manipulate tag bits. The same is true for most loads and stores.

Allocation- and limit pointers are stored in registers, and taking advantage of the PowerPC's `stwu` instruction we can allocate one memory word in a single instruction.[10] As mentioned before, the code for copying a slice out of an existing record into a newly allocated one uses an inner loop of only three instructions (`lwzu, stwu, bdnz`), but there is also a four-instruction preamble (`addi, srwi., mtctr, beq`) that loads the *count register* and bypasses the loop when the count is initially zero.

The `String` module. Our language does not yet have a module system, but as long as only values but no types are involved, one can use records as a poor-man's substitute. The runtime system exports a special record bound to the global variable `String` which contains routines for manipulating string values, for converting from and to strings, and for performing very basic input-output operations. This record is allocated using C code and does not reside within the **MLPolyR** heap.

4.8 Mutable record fields

Our type system supports mutable fields in records. Type reconstruction still works since corresponding operations on mutable and immutable fields are syntactically distinguishable. Records with mutable fields have identity, and allocation of such records is a side-effecting operation.

In hindsight it appears that it would have been better to instead distinguish between two kinds of records: those that are guaranteed to be immutable, and those that *may* contain mutable fields. Mutability interacts in some undesirable ways with row polymorphism. For example, we cannot say that the right-hand side in the following **let**-binding is a syntactic value and, therefore, its type cannot be generalized:

```
let val r = { a = foo, ... = bar }
```

Whether or not the allocation of this record expands the store depends on the type of `bar`. Ignoring the problem with the value restriction, in the general case the compiler is unable to perform certain optimizations such as, e.g., common subexpression elimination for code like this:

```
let val r1 = { a = foo, ... = bar }
    val r2 = { a = foo, ... = bar }
```

Therefore, with our current design, the mere existence of the mutable fields feature in the language incurs certain penalties, both in terms of the static semantics and in terms of runtime efficiency, even if that feature is never used.

Since we prefer a pay-as-you go scheme where features incur penalties only when they are actually being used, we plan to go back to immutable general records in the style of Standard ML and support mutable fields separately.

[9] The same argument does not work for Haskell, because due to type classes Haskell's polymorphism is not parametric.

[10] The cost of the heap limit check is amortized over multiple allocations within a basic block.

5. Related work

Record calculi and the study of record polymorphism have a long history [31, 32, 26, 5, 25]. Ohori shows that polymorphic records can be compiled very efficiently, using an index-passing transformation based on a kinded type system for records [25]. He also points out the duality between records and sums and suggests that the same index-passing techniques can be adopted to implement polymorphic sums. Rémy gives a more general type system capable of expressing linearly *extensible* polymorphic records. Rémy's calculus employs row polymorphism and has an efficient type reconstruction algorithm that infers principal types [26]. Jones and Peyton Jones describe an implementation of extensible records based on the same ideas for Haskell [14].

Gaster and Jones attempt a direct encoding of the dual construction for sum types within Haskell's type system [12]. The encoding requires type system features absent from most languages, in particular higher-order polymorphism and a type constructor which roughly corresponds to the row arrow $\rightarrowtail$ in our System F. Type inference in such a system seems difficult, and, indeed, Gaster and Jones report that they had to impose an ad-hoc restriction to obtain most general unifiers. Their restriction is to disallow empty rows, meaning that they could not type our **nocases** construct.

Garrigue implements a version of polymorphic sum types in OCaml. His approach does not take advantage of the duality between sums and records but instead provides a form of extensibility based on so-called *variant dispatching* [10, 11]. As Zenger and Odersky point out [33], variant dispatching requires wrapper functions to forward control to existing code. This is a consequence of the fact that in Garrigue's system, extensions need to know what they are extending. As a result, extensions cannot be composed directly.

It should be noted that a suitably modified typing rule for a **match** expression with a default case could actually be used to give Garrigue's implementation the same power of extensibility that we provide in **MLPolyR**. Consider the following example:

```
fun g y = ...
fun f x =
    match x with
          `A () => print "A"
        | y => g y
```

Here the types of x and y should be related sums that share a common row, the only difference being the presence of the `` `A`` constructor in x's type and its absence in y's type. The typing rule for this could be:

$$\boxed{\frac{\Gamma \vdash e_1 : \langle l : \tau_l, \rho\rangle \quad \Gamma, x : \tau_l \vdash e_2 : \tau \quad \Gamma, y : \langle\rho\rangle \vdash e_3 : \tau}{\Gamma \vdash \mathbf{match}\ e_1\ \mathbf{with}\ lx \Rightarrow e_2 \mid y \Rightarrow e_3 : \tau}}$$

This approach does not require the alternative function arrow $\hookrightarrow$ for cases but uses the ordinary function arrow in its place. The main advantage of having the case arrow $\hookrightarrow$ in the type system and statically distinguishing cases from other functions lies in the fact that this makes it very easy to use different runtime representations for the two. In particular, we can represent **MLPolyR** cases as records of functions. These records represent jump tables. In Garrigue's implementation, however, case analysis for polymorphic variants proceeds by direct comparisons of constructor names (significantly sped up via hashing). Thus, his implementation technique essentially corresponds to our semantics of PolyR. In this setting, extending functions by extra cases can be implemented by simple chaining of conditionals.

References

[1] A. W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag.

[3] F. Bourdoncle and S. Merz. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, Paris, France, 15–17 1997.

[4] K. Bruce. Some challenging typing issues in object-oriented languages. *In Electronic notes in Theoretical Computer Science, volume 82(8)*, 2003.

[5] L. Cardelli and J. C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 295–350. The MIT Press, Cambridge, MA, 1994.

[6] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, volume 34(1) of *SIGPLAN*, pages 94–104, New York, NY, June 1999. ACM.

[7] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *1993 Conference on Programming Language Design and Implementation.*, pages 21–25, June 1993.

[8] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Department of Computer Science, Rice University, 1999.

[9] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.

[10] J. Garrigue. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML*, 1998.

[11] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Nov. 2000.

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

[13] K. Gowani and M. Blume. Writing a garbage collector for the MLPolyR compiler, July 2005. Final report on independent study project in the CSPP program.

[14] M. Jones and S. Jones. Lightweight extensible records for haskell, 1999.

[15] M. P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA, 1993.

[16] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-Use. In *European Conference on Object-Oriented Programming*, 1998.

[17] X. Leroy. [caml-list] cyclic types. http://caml.inria.fr/pub/ml-archives/caml-list/, Jan. 2005.

[18] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 110–122, New York, NY, USA, 2002. ACM Press.

[19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 13(3):348–375, 1978.

[20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[21] I. N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P.

Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, J. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[22] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[23] A. Ohori. A simple semantics for ml polymorphism. In *FPCA*, pages 281–292, 1989.

[24] A. Ohori. A compilation method for ml-style polymorphic record calculi. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165, New York, NY, USA, 1992. ACM Press.

[25] A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995.

[26] D. Rémy. Type inference for records in a natural extension of ML. Research Report 1431, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, may 1991.

[27] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Objects Systems*, 4(1):27–50, 1998.

[28] J. Reppy and J. Riecke. Simple objects for Standard ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 171–180, Philadelphia, Pennsylvania, 21–24 1996.

[29] M. Torgersen. The expression problem revisited: Four new solutions using generics. In M. Odersky, editor, *ECOOP 2004—Object-Oriented Programming, 18th European Conference, Oslo, Norway, Proceedings*, volume 3086 of *LNCS*, pages 123–143, New York, NY, July 2004. Springer-Verlag.

[30] P. Wadler. The expression problem. Email to the Java Genericity mailing list, Dec. 1998.

[31] M. Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science, Ithaca, NY*, June 1987.

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

[33] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 241–252, New York, NY, USA, 2001. ACM Press.

[34] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, Long Beach, California, 2005. ACM.

A. Dynamic Semantics for PolyR

Figure 14 shows the dynamic semantics for the PolyR language.

$$\frac{}{v \Downarrow v}\ \textbf{(val)} \qquad \frac{\begin{array}{c} e_1 \Downarrow \mathbf{fun}\ f\ x = e_1' \qquad e_2 \Downarrow v_2 \\ e_1'[\mathbf{fun}\ f\ x = e_1'/f, v_2/x] \Downarrow v \end{array}}{e_1\ e_2 \Downarrow v}\ \textbf{(app)}$$

$$\frac{e \Downarrow v}{l\ e \Downarrow l\ v}\ \textbf{(data const.)} \qquad \frac{e_1 \Downarrow v_1 \quad e_2[v_1/x] \Downarrow v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow v}\ \textbf{(let)}$$

$$\frac{e_1 \Downarrow v_1 \ldots e_k \Downarrow v_k}{\{\ l_i = e_i\ \}_{i=1}^{k} \Downarrow \{\ l_i = v_i\ \}_{i=1}^{k}}\ \textbf{(r)}$$

$$\frac{e_1 \Downarrow \{\ l_i = v_i'\ \}_{i=1}^{k} \quad e_2 \Downarrow v_2}{e_1\ \otimes\ \{l = e_2\} \Downarrow \{l_1 = v_1', \ldots, l_k = v_k', l = v_2\}}\ \textbf{(r/ext)}$$

$$\frac{e_1 \Downarrow \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_k = v_k\}}{\begin{array}{l} e_1\ \oslash\ l_i \quad \Downarrow \quad \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, \\ \qquad\qquad\qquad l_{i+1} = v_{i+1}, \ldots, l_k = v_k\} \end{array}}\ \textbf{(r/sub)}$$

$$\frac{e \Downarrow \{l_1 = v_1, \ldots, l_i = v_i, \ldots, l_k = v_k\}}{e.l_i \Downarrow v_i}\ \textbf{(select)}$$

$$\frac{e_1 \Downarrow \{\ l_i\ x_i \Rightarrow e_i'\ \}_{i=1}^{k}}{\begin{array}{l} e_1\ \oplus\ \{\ l\ x \Rightarrow e_2\ \} \quad \Downarrow \quad \{l_1\ x_1 \Rightarrow e_1', \ldots, \\ \qquad\qquad\qquad\qquad l_k\ x_k \Rightarrow e_k', l\ x \Rightarrow e_2\} \end{array}}\ \textbf{(c/ext)}$$

$$\frac{e \Downarrow \{\ l_1\ x_1 \Rightarrow e_1', \ldots, l_i\ x_i \Rightarrow e_i', \ldots, l_k\ x_k \Rightarrow e_k'\ \}}{\begin{array}{l} e\ \ominus\ l_i \quad \Downarrow \quad \{l_1\ x_1 \Rightarrow e_1', \ldots, l_{i-1}\ x_{i-1} \Rightarrow e_{i-1}', \\ \qquad\qquad\quad l_{i+1}\ x_{i+1} \Rightarrow e_{i+1}', \ldots, l_k\ x_k \Rightarrow e_k'\} \end{array}}\ \textbf{(c/sub)}$$

$$\frac{\begin{array}{c} e_1 \Downarrow l_i\ v \\ e_2 \Downarrow \{\ l_1\ x_1 \Rightarrow e_1', \ldots, l_i\ x_i \Rightarrow e_i', \ldots, l_k\ x_k \Rightarrow e_k'\ \} \\ e_i'[v/x_i] \Downarrow v' \end{array}}{\mathbf{match}\ e_1 \mathbf{with}\ e_2 \Downarrow v'}\ \textbf{(match)}$$

Figure 14. The dynamic semantics for PolyR.