MLPolyR

A (mini-)ML with type inference, polymorphic records, and functional record update

by Matthias Blume
for CMSC 22620/23620, Spring 2005
Department of Computer Science, University of Chicago
and
Toyota Technological Institute at Chicago

1 Syntax

Figure 1 shows the syntax of MLPolyR in EBNF. Notice that this grammar, as written, is highly ambiguous.

2 Type inference, record polymorphism, and functional record update

The language has no type declarations or type annotations; all types are inferred by a Hindley-Milner-style type inference engine that is part of the type checker.

Unlike Standard ML, the language incorporates Ohori-style *record polymorphism*. This means that, for example, the following function `addab` can subsequently be applied to any record argument as long as it has immutable integer fields `a` and `b`:

```
fun addab r = r.a + r.b
in
   addab { a = 5, b = 7, c = "hello" } -
   addab { b = 23, a = 0 } *
   addab { z = 1, a = 22, y = 15, b = ~1, x = 4 }
```

Record fields can be mutable. Field expressions for mutable fields use **:=** instead of = between label and expression. Access to the contents of mutable fields is done using **!** in place of the usual selection operator for immutable fields (**.**).

The language offers two functional record update operators: **with** and **where**. The former strictly extends a given record (the left-hand side expression) with fields which had not been present, while the latter strictly replaces existing fields with new fields.

$$
\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{functions}^{*}\ \mathit{mainfun} \\
\mathit{functions} & \rightarrow & \underline{\textbf{fun}}\ \mathit{fundecl}\ (\underline{\textbf{and}}\ \mathit{fundecl})^{*} \\
\mathit{fundecl} & \rightarrow & \mathit{name}\ \mathit{formals}\ \underline{=}\ \mathit{exp} \\
\mathit{formals} & \rightarrow & \mathit{varpat}\ \mid \underline{(}\ (\mathit{varpat}\ (,\ \mathit{varpat})^{*})_{\mathrm{opt}}\ \underline{)} \\
\mathit{varpat} & \rightarrow & \_\ \mid \mathit{name} \\
\mathit{exp} & \rightarrow & \underline{\textbf{let}}\ \mathit{varpat}\ \underline{=}\ \mathit{exp}\ \underline{\textbf{in}}\ \mathit{exp} \\
 & \mid & \mathit{functions}\ \underline{\textbf{in}}\ \mathit{exp} \\
 & \mid & \underline{\textbf{if}}\ \mathit{exp}\ \underline{\textbf{then}}\ \mathit{exp}\ \underline{\textbf{else}}\ \mathit{exp} \\
 & \mid & \underline{\textbf{case}}\ \mathit{exp}\ \underline{\textbf{of}}\ \mathit{match} \\
 & \mid & \mathit{exp}\ \underline{\textbf{with}}\ \mathit{recordexp} \\
 & \mid & \mathit{exp}\ \underline{\textbf{where}}\ \mathit{recordexp} \\
 & \mid & \mathit{exp}\ \mathit{binconn}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \mathit{exp} \\
 & \mid & \mathit{unaryop}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \textbf{.}\ \mathit{label} \\
 & \mid & \mathit{exp}\ \underline{!}\ \mathit{label} \\
 & \mid & \mathit{exp}\ \underline{!}\ \mathit{label}\ \underline{:=}\ \mathit{exp} \\
 & \mid & \mathit{name} \\
 & \mid & \underline{\textbf{true}}\ \mid \underline{\textbf{false}}\ \mid \mathit{integer}\ \mid \mathit{string} \\
 & \mid & \underline{(}\ \underline{)} \\
 & \mid & \underline{(}\ \mathit{exp}\ \underline{)} \\
 & \mid & \underline{(}\ \mathit{exp}\ \textbf{,}\ \mathit{exp}\ (\textbf{,}\ \mathit{exp})^{*}\ \underline{)} \\
 & \mid & \underline{(}\ \mathit{exp}\ \textbf{;}\ \mathit{exp}\ (\textbf{;}\ \mathit{exp})^{*}\ \underline{)} \\
 & \mid & \underline{[}\ (\mathit{exp}\ (\textbf{,}\ \mathit{exp})^{*})_{\mathrm{opt}}\ \underline{]} \\
 & \mid & \mathit{recordexp} \\
\mathit{match} & \rightarrow & \mathit{nilcase}\ \underline{\mid}\ \mathit{conscase} \\
 & \mid & \mathit{conscase}\ \underline{\mid}\ \mathit{nilcase} \\
\mathit{nilcase} & \rightarrow & \underline{[\ ]}\ \underline{=>}\ \mathit{exp} \\
\mathit{conscase} & \rightarrow & \mathit{varpat}\ \underline{::}\ \mathit{varpat}\ \underline{=>}\ \mathit{exp} \\
\mathit{binconn} & \rightarrow & \mathit{boolconn}\ \mid \mathit{cmpop}\ \mid \mathit{arithop}\ \mid \underline{::} \\
\mathit{boolconn} & \rightarrow & \underline{\textbf{andalso}}\ \mid \underline{\textbf{orelse}} \\
\mathit{cmpop} & \rightarrow & \underline{==}\ \mid \underline{>}\ \mid \underline{>=}\ \mid \underline{<>}\ \mid \underline{<}\ \mid \underline{<=} \\
\mathit{arithop} & \rightarrow & \underline{+}\ \mid \underline{-}\ \mid \underline{*}\ \mid \underline{/}\ \mid \underline{\%} \\
\mathit{unaryop} & \rightarrow & \underline{-}\ \mid \underline{\textbf{isnull}}\ \mid \underline{\textbf{hd}}\ \mid \underline{\textbf{tl}}\ \mid \underline{\textbf{not}} \\
\mathit{label} & \rightarrow & \mathit{name}\ \mid \mathit{integer} \\
\mathit{recordexp} & \rightarrow & \underline{\{}\ (\mathit{fieldexp}\ (\textbf{,}\mathit{fieldexp})^{*})_{\mathrm{opt}}\ \underline{\}} \\
\mathit{fieldexp} & \rightarrow & \mathit{name}\ \underline{=}\ \mathit{exp}\ \mid \mathit{name}\ \underline{:=}\ \mathit{exp} \\
\mathit{mainfun} & \rightarrow & \underline{\textbf{fun}}\ \textbf{main}\ \underline{(}\ \mathit{varpat}\ \textbf{,}\ \mathit{varpat}\ \underline{)}\ \underline{=}\ \mathit{exp} \\
\mathit{name} & \rightarrow & \ldots \\
\mathit{integer} & \rightarrow & \ldots \\
\mathit{string} & \rightarrow & \textbf{"}\ldots\textbf{"}
\end{array}
$$

Figure 1: Syntax of **MLPolyR**

Either form of functional record update creates a brand-new record. There is no sharing between fields of the old and the new record; mutable fields are "cloned."

Record labels can be small positive integers. A *tuple* is a special case of a record where the labels form an initial segment of the positive integers. The tuple syntax $(e_1, \ldots, e_k)$ is equivalent to the record syntax $\{1 = e_1, \ldots, k = e_k\}$.

The **let**-bound variables and **fun**-defined functions are given polymorphic types within the respective body expression (after **in**). The *value restriction* that you perhaps are familiar with from Standard ML applies: a **let**-bound variable's type is not generalized if the right-hand side of the binding is not a *syntactic value*. Similarly, row types (record types where not all fields are known) are generalized only if the record type in question has not been involved in any functional record update (**with** or **where**). The following code will not pass the type checker since `augmentc` is not polymorphic in `r`'s row type:

```
fun augmentc (r, x) = r with { c = x }
in (augmentc ({ a = 1 }, 8), augmentc ({ b = 2 }, 9))
```

However, the function still *is* polymorphic in `x`, which means that this code is ok:

```
fun augmentc (r, x) = r with { c = x }
in (augmentc ({ a = 1 }, 8), augmentc ({ a = 2 }, "a string"))
```

3 Expressions

literal data MLPolyR programs can use the following constants:

boolean **<u>true</u>**, **<u>false</u>**

numerical *integer*

string *string*

unit <u>( )</u> — the record/tuple with no fields

lists <u>[</u> ... <u>]</u>

records <u>{</u> ... <u>}</u>

tuples <u>(</u> ... <u>)</u>

identifiers Identifiers (*name*) in MLPolyR name values, not locations. They are not mutable. The only form of assignment is update of mutable record fields.

binary operations In general, binary operations have the form $e_1 \otimes e_2$ where $\otimes$ is one of:

- *short-circuiting logical or:* **<u>orelse</u>** — boolean arguments and results
- *short-circuiting logical and:* **<u>andalso</u>** — boolean arguments and results
- *comparisons:* <u>==</u> <u><></u> <u><</u> <u>></u> <u><=</u> <u>>=</u> — integer operands, boolean result

- *list cons:* <u>`::`</u> — element and list operands, list result
- *addition and subtraction:* <u>`+`</u> <u>`-`</u> — integer operands and result
- *multiplication and division:* <u>`*`</u> <u>`/`</u> <u>`%`</u> — integer operands and result

These operators are listed in order of increasing precedence.

unary operations There are five unary operations:

- *boolean negation:* <u>**not**</u> e — boolean argument, boolean result
- *arithmetic negation:* <u>`-`</u> e — integer argument, integer result
- *empty list test:* <u>**isnull**</u> e — list argument, boolean result
- *list head:* <u>**hd**</u> e — list argument, element result
- *list tail:* <u>**tl**</u> e — list argument, list result

conditional expression An <u>**if**</u> expression evaluates its boolean condition and depending on the outcome proceeds to evaluate either the <u>**then**</u> branch or the <u>**else**</u> branch. The expression that is not needed does not get evaluated. Notice that e_1 <u>**andalso**</u> e_2 and e_1 <u>**orelse**</u> e_2 are equivalent to <u>**if**</u> e_1 <u>**then**</u> e_2 <u>**else**</u> <u>**false**</u> and <u>**if**</u> e_1 <u>**then**</u> <u>**true**</u> <u>**else**</u> e_2, respectively.

4 Built-in functions

MLPolyR programs are compiled in a global environment containing a binding for the following record value:

```
val String : { toInt     : string -> int,
               fromInt   : int -> string,
               inputLine : () -> string,
               size      : string -> int,
               output    : string -> (),
               sub       : string * int -> int,
               concat    : string list -> string,
               substring : string * int * int -> string,
               compare   : string * string -> int }
```

The elements of this record can be used to perform simple I/O tasks and string manipulation.