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.

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

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 **true**, **false**

numerical *integer*

string *string*

unit **( )** — the record/tuple with no fields

lists **[** . . . **]**

records { . . . }

tuples **(** . . . **)**

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

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

These operators are listed in order of increasing precedence.

unary operations There are five unary operations:

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

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