Non-strict higher-order functional programming in Prolog (DRAFT)

Sven Eric Panitz

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail:panitz@informatik.uni-frankfurt.de
Tel: +49 (69)-798-22890 Fax: +49 (69)-798-28919

Abstract. We implement a non-strict interpreter for a higher-order functional core language in Prolog. The interpreter itself consists of only a few Prolog clauses.

1 Introduction

1.1 Non-strict functional programming

Core-fp

1.2 Logic programming

2 The Language

We define core-fp, a simple polymorphic typed functional core language. It closely resembles the language used in [PJL91b] which is the source language for our implementation of abstract reduction [Sch95] and which has been used in [SSPS95]. The reader who is familiar with [SSPS95] may skip this section without any harm.

2.1 Expressions and Supercombinator Definitions

The type system for core-fp consists of given basic types including *Num*, function types and a finite number of algebraic type names. Polymorphic types are allowed. The following constants are in core-fp:

- Numerical constants.
- Built-in functions like $+, -, *, /$.
- For every algebraic type A there exist finitely many constructor constants $\mathbf{con}_{A,1}, \ldots, \mathbf{con}_{A,n_i}$, such that the constructor $\mathbf{con}_{A,j}$ has a type of the form
  $\tau_{j_1} \to \ldots \tau_{j_k} \to A$.

- one case constant $\mathbf{case}_A$ for every algebraic type name, which takes $n+1$ arguments, where n is the number of construtor constants for A. The first argument is the expression to be cased. The other arguments of $\mathbf{case}_A$ are functions taking m_k arguments for $1 \leq k \leq n$, where m_k is the arity of the kth constructor of A.
- supercombinators that may be defined in a recursive way (see below).

An example of an algebraic type is *Bool* with the constants **True**, **False** and the case-constant **if**. Another example is $(List\ \alpha)$ with the constants **Nil**, **Cons** and the case-constant $\mathbf{case}_{List}$, which requires three arguments: the list expression, the expression for the **Nil**-case and a function taking the head and tail as arguments in the non-nil case.

2.2 Core-fp Expressions

core-fp expressions are composed in the standard way, i.e., expressions may be variables x, constants c, and applications $(d\ e)$, where we require that core-fp expressions are well-typed. We assume that application is left associating, i.e., $e_1\ e_2\ e_3$ is equivalent to $(e_1\ e_2)\ e_3$.
Recursions are defined via supercombinators. An expressions e with n free variables $x_1, \ldots, x_n$ may be defined as a supercombinator: $sc\ x_1\ \ldots\ x_n = e$. If e has type σ then sc is treated as constant with type $\tau_1 \to \ldots \tau_n \to \sigma$.

2.3 Operational Semantics

The operational semantics consists of the δ-reductions. If $sc\ x_1\ \ldots\ x_n = e$ is a supercombinator definition, then $sc\ t_1\ \ldots\ t_n \to_\delta e[x_1/t_1, \ldots, x_n/t_n]$.
$\delta-$rules for built-in primitive functions are the usual ones, i.e., applications of built-in functions to arguments can only be evaluated, if all arguments are basic values. Furthermore, they are assumed to be strict in all arguments.
For every case constant of an algebraic type there is a $\delta-$rule of the following form:

$$\mathbf{case}_{A_i}\ (\mathbf{con}_j\ t_1 \ldots t_m)\ e_1 \ldots e_n \to_\delta (e_j\ t_1 \ldots t_m)$$

An expression is in *head normal form* (HNF), if it is in one of the following forms:

- $c\ e_1\ \ldots\ e_n$, where c is a constructor
- $c\ e_1\ \ldots\ e_n$, where c is a built-in function, a supercombinator or a **case** constant that requires more than n arguments.

3 Implementation

3.1 General reduction setting

First of all we will define a predicate which denotes de reduction of an expression to some of its reducts. We assume a predicate, which denotes a one-step reduction in normal-order of an expression. An expression can be reduced by a sequence of fintely many normal-order reductions.

```
red(X,Y):-step(X,Y).
red(X,Z):-step(X,Y),red(Y,Z).
```

3.2 Function definitions

The first thing we have to do, is to encode the supercombinator definitions into Prolog clauses. For representing core-fp expressions in Prolog we will use Prolog terms. Since in core-fp it is possible to apply a function unsaturated, i.e. to less arguments than its actual arity, we have to use the usual translation of expressions into terms, which are built with the single function symbol `ap` of arity 2 for function application. Constants in the Prolog terms are used to denote supercombinator names, constructor names and names for the case functions.
Thus the translation function τ of core-fp expressions into Prolog terms can be recursively defined as:

- $\tau(x) = X$, for variables x.
- $\tau(c) = c$, for supercombinator, constructor or case c.
- $\tau((e_1\ e_2)) = ap(\tau(e_1), \tau(e_2))$.

For supercombinator definitions of the form
 `sc` $x_1 \ldots x_n$ = e
we define the Prolog clause which denotes this definition by:
 `def(ap(ap(ap`$\ldots$`ap(sc,`x_1`),`$x_2 \ldots x_n$`),`$\tau(e)$.
As an example consider the well-known combinator functions `S`, `K`, `I`:

```
i x = x
k x y = x
s x y z = (x z) (y z)
```

These we denote in Prolog by the following three clauses.

```
def(ap(i,X),X).
def(ap(ap(k,X),Y),X).
def(ap(ap(ap(s,X),Y),Z),ap(ap(X,Z),ap(Y,Z))).
```

Now we can simply state, how a single reduction step will look like:

```
step(X,Y):-def(X,Y).
```

Note that such a reduction step can only reduce the top-level redex, which is of course a normal-order redex.

3.3 Algebraic data types

An algebraic data type consists of a set of n constructors and of a *case*-function of arity $(n+1)$. Since there are no reduction rules for constructors, we do not need to introduce any clauses, which denote reduction for constructors. Constructors are simply further constants in Prolog terms.
What we have to do in Prolog is to define, how an application of *case* is being reduced. Consider as the best-known example lists. The reduction rule for the *case*-function on lists is as follows:

– **caselist** (`Nil`) e_1 $e_2 \rightarrow_\delta e_1$

– **caselist** (`Cons` a as) e_1 $e_2 \ldots \rightarrow_\delta (e_2\ a\ as)$

Theses reduction steps can directly be translated into Prolog clauses:

```
step(ap(ap(ap(caselist,nil),E1),E2),E1).
step(ap(ap(ap(caselist,ap(ap(cons,X),XS)),E1),E2),ap(ap(E2,X),XS)).
```

These two clauses cannot describe the complete way how to reduce an application of a *case*-function. We have to denote, what has to be done, when the first argument of the case function is not in HNF. Then we have to reduce this argument to HNF first, i.e. *case* is strict in its first argument. This can be stated as a Prolog clause explicitely:

```
step(ap(ap(ap(caselist,E),E1),E2),ap(ap(ap(caselist,En),E1),E2))
 :-step(E,En).
```

Since we do not have any reduction rule for expressions which are in HNF, we have ensured that we only make normal-order reductions. A reduction underneath the *case*-function will stop when a HNF is reached.

3.4 Reduction to HNF

With the things done so far we can actually reduce expressions in normal-order. If we want to reduce an expression e then we can simply put the query: `red(`e`,RES).`. The Prolog interpreter will then produce all solutions of expressions e' for the variable `RES` such that e can be reduced to e'.

Example 1. We consider the example of the standard function `append` for list concatenation. In its Prolog definition this is:

```
def(ap(ap(append,XS),YS),ap(ap(ap(caselist,XS),YS),ap(append2,YS))).
def(ap(ap(ap(append2,YS),X),XS),ap(ap(cons,X),ap(ap(append,XS),YS))).
```

If we are interested in the reduction of the following expression:

```
append (append (Cons a nil) nil) (append (Cons b nil) nil)
```

We make the following query:

```
red(ap(ap(append,
          ap(ap(append,ap(ap(cons,a),nil)),nil)),
          ap(ap(append,ap(ap(cons,b),nil)),nil)),RES).
```

The Prolog interpreter will then produce the following sequence of solutions:

```
RES =
 ap(ap(ap(caselist,
     ap(ap(append, ap(ap(cons, a), nil)), nil)),
     ap(ap(append, ap(ap(cons, b), nil)), nil)),
     ap(append2, ap(ap(append, ap(ap(cons, b), nil)), nil)))

More? (;)

RES =
 ap(ap(ap(caselist,
      ap(ap(ap(caselist,
                 ap(ap(cons, a), nil)),
                 nil), ap(append2, nil))),
      ap(ap(append, ap(ap(cons, b), nil)), nil)),
      ap(append2, ap(ap(append, ap(ap(cons, b), nil)), nil)))

More? (;)

RES =
 ap(ap(ap(caselist,
      ap(ap(ap(append2, nil), a), nil)),
      ap(ap(append, ap(ap(cons, b), nil)), nil)),
      ap(append2, ap(ap(append, ap(ap(cons, b), nil)), nil)))
     More? (;)

RES =
 ap(ap(ap(caselist,
      ap(ap(cons, a), ap(ap(append, nil), nil))),
      ap(ap(append, ap(ap(cons, b), nil)), nil)),
      ap(append2, ap(ap(append, ap(ap(cons, b), nil)), nil)))

     More? (;)

RES =
 ap(ap(ap(append2, ap(ap(append, ap(ap(cons, b), nil)), nil)), a),
          ap(ap(append, nil), nil))

     More? (;)

RES =
 ap(ap(cons, a), ap(ap(append, ap(ap(append, nil), nil)),
    ap(ap(append, ap(ap(cons, b), nil)), nil)))

     More? (;)

no (more) solution.
```

As can be seen: the sequence of solutions is just the normal-order reduction sequence, until a HNF has been reduced.

If we are not interested in all reducts of an expression but only in the HNF this expression yields, we have to select the solution, which is a HNF. Therefore we need a further predicate, which denotes that an expression is in HNF. For list expressions this is rather easy:

```
hnf(nil).
hnf(ap(ap(cons,X),Y)).
```

Furthermore we have to specify how an expression of function type in HNF may look like. These are unsaturated applications of functions or constructors:

```
hnf(cons).
hnf(ap(cons,X)).
hnf(append).
hnf(ap(append,X)).
hnf(append2).
hnf(ap(append2,X)).
```

Since we are usually not interested in the HNF of a function we will neglect unsaturated applications as HNFs in the future. To state our query for a HNF of an expression, we will add the predicate `whathnf` with the following clauses:

```
whathnf(X,X):-hnf(X).
whathnf(X,Y):-red(X,Y),hnf(Y).
```

Now we can ask for an HNF of some arbitrary expression:

```
whathnf(ap(ap(append,
              ap(ap(append,ap(ap(cons,a),nil)),nil)),
              ap(ap(append,ap(ap(cons,b),nil)),nil)),RES).

RES =
 ap(ap(cons, a),
      ap(ap(append, ap(ap(append, nil), nil)),
                ap(ap(append, ap(ap(cons, b), nil)), nil)))

More? (;)
no (more) solution.
```

3.5 Basic values and built-in functions

We have not treated the built-in functions of core-fp yet. In order to incorporate built-in arithmetic expression in our interpreter, we will use built-in predicates of Prolog, which are dealing with arithmetic expressions[1]. To modell integers we will use the same method as some implementations of Haskell do: the data-type `Int` is an algebraic data-type consisting of exactely one constructor `Box` of arity 1. The argument of `Box` is an integer constant. This technique is known as *boxing* [PJL91a]. We have a further constructor, which can form a HNF:

[1] This means that the following Prolog-code cannot be interpreted with the Prolog interpreter written in Haskell which is part of the *hugs* distribution.

```
hnf(box(X)).
```

Built-in arithmetic functions can directely calculate the result on *unboxed* arguments. We add the following clauses, which define reduction steps for built-in functions:

```
step(box(X)+box(Y),box(Z)):- +(X,Y,Z).
step(box(X)-box(Y),box(Z)):- -(X,Y,Z).
step(box(X)*box(Y),box(Z)):- *(X,Y,Z).
step(box(X)/box(Y),box(Z)):- /(X,Y,Z).
```

Built-in arithmetic functions are strict in both arguments. The functions operating on unboxed values require their arguments to be evaluated. If there is some argument of a built-in function that is not in HNF, we have to reduce this argument first. We denote this by the following clauses:

```
step(box(X)+Y,box(X)+YN) :-step(Y,YN).
step(X+Y,XN+Y) :-step(X,XN).

step(box(X)-Y,box(X)-YN) :-step(Y,YN).
step(X-Y,XN-Y) :-step(X,XN).

step(box(X)*Y,box(X)*YN) :-step(Y,YN).
step(X*Y,XN*Y) :-step(X,XN).

step(box(X)/Y,box(X)/YN) :-step(Y,YN).
step(X/Y,XN/Y) :-step(X,XN).
```

Finally we want to be able to make equality tests on arithmetic expression. The result will be of the algebraic type which denotes boolean values:

```
step(ap(ap(ap(casebool,true),E1),E2),E1).
step(ap(ap(ap(casebool,false),E1),E2),E2).
step(ap(ap(ap(casebool,E),E1),E2),ap(ap(ap(casebool,En),E1),E2))
  :-step(E,En).

hnf(false).
hnf(true).
```

The strict equality function for integer expressions can thus be implemented by:

```
step(eq(box(X),box(Y)),true)      :-X =:= Y.
step(eq(box(X),box(Y)),false)     :-X =\= Y.
step(eq(box(X),Y),eq(box(X),YN)) :- step(Y,YN).
step(eq(X,Y),eq(XN,Y))            :- step(X,XN).
```

Now the operational semantics of core-fp are completely implemented. We can use our interpreter for some experiments.

Example 2. Let us extract some element from an infinite list. We define the standard library functions `from`, `last` and `take` on lists:

```
def(ap(from,I),ap(ap(cons,I),ap(from,I+box(1)))).

def(ap(last,XS),ap(ap(ap(caselist,XS),bot),last2)).
def(ap(ap(last2,X),XS),ap(ap(ap(caselist,XS),X),last3)).
def(ap(ap(last3,X),XS),ap(last,ap(ap(cons,X),XS))).

def(ap(ap(take,N),XS),
    ap(ap(ap(casebool,(eq(N,box(0)))),nil),ap(ap(take2,XS),N))).
def(ap(ap(take2,XS),N),
    ap(ap(ap(caselist,XS),nil),ap(take3,N-box(1)))).
def(ap(ap(ap(take3,N),X),XS),ap(ap(cons,X),ap(ap(take,N),XS))).
```

Our interpreter is now able to answer the following query:

```
whathnf(ap(last,ap(ap(take1,box(100)),ap(from,box(1)))),RES).

RES = box(100)
More? (;)

no (more) solution.
```

3.6 Is it lazy

We can define a simple test, to see, if the interpreter we developed so far is not only non-strict in its reduction strategy, but *lazy* as well. In a lazy reduction common subexpression are not duplicated but only stored once in the heap. This avoids duplication of work. We can check with a simple test query, whether our interpreter is lazy:

```
def(ap(square,X),X * X).

red(ap(square,ap(square,box(5))),RES).

RES = ap(square, box(5)) * ap(square, box(5))
More? (;)

RES = box(5) * box(5) * ap(square, box(5))
More? (;)

RES = box(25) * ap(square, box(5))
More? (;)

RES = box(25) * (box(5) * box(5))
More? (;)
```

```
RES = box(25) * box(25)
More? (;)

RES = box(625)
More? (;)

no (more) solution.
```

As can be clearly seen: the interpreter is not lazy. The redex `ap(square, box(5))` becomes duplicated and is reduced twice.

Example 3. To get a feeling of what an effect the lacking of lazyness may have, consider the `take` example again. This time we avoid the duplication of work by using a strict version of the function `take`. This means that the argument will be reduced to HNF before it is duplicated:

```
def(ap(ap(takestr,box(N)),XS),ap(ap(take,box(N)),XS)).
def(ap(ap(takestr,N),XS),ap(ap(takestr,NN),XS)):-step(N,NN).
def(ap(ap(take,N),XS),
    ap(ap(ap(casebool,(eq(N,box(0)))),nil),ap(ap(take2,XS),N))).
def(ap(ap(take2,XS),N),
    ap(ap(ap(caselist,XS),nil),ap(take3,N-box(1)))).
def(ap(ap(ap(take3,N),X),XS),ap(ap(cons,X),ap(ap(takestr,N),XS))).
```

With this modification we are able to run applications of `take` to larger numbers in our interpreter:

```
whathnf(ap(last,ap(ap(take,box(1000)),ap(from,box(1)))),RES).

RES = box(1000)     More? (;)

no (more) solution.
```

4 Conclusion

We have implemented an interpreter for non-strict evaluation of a higher-order functional language with built-in arithmetic functions. The interpreter needs almost no clauses for its run-time system, but uses the Prolog mechanisms for unification and garbage collection. It is simply invoced by a formulation of the core-fp program in Prolog clauses. The interpreter can be used for giving a stepwise execution of an expression or to calculate directly its HNF.
We believe the interpreter can be used as a good tool to demonstrate some basics of non-strict functional languages:

- Expressions can be reduced stepwise. This can be useful in teaching functional programming.

- The nature of how higher-order programs are coded into first-order terms by way of the application function `ap` can be experienced practically.
- The notions of a HNF can be simply explained.
- Even rather technical issues can be demonstrated with the interpreter, e.g. unboxed values can be treated as first-class citizens and functions can be split into worker and wrapper functions.
- The difference between strict and non-strict functions becomes rather clear. A non-strict reduction is formulated by a single literal, a strict reduction has some antecedence in its clause. The effect of adding strictness can be observed.
- The interpreter itself is a precise and formal definition of the operational semantics of core-fp.

References

[PJL91a] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 636–666. Springer, 1991.

[PJL91b] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall International, London, 1991.

[Sch95] Marko Schütz. The $G^{\#}$-machine: Efficient strictness analysis in Haskell. Technical Report 1/95, Johann Wolfgang Goethe-Universität, Fachbereich Informatik, January 1995.

[SSPS95] M. Schmidt-Schauß, S.E. Panitz, and M. Schütz. Strictness analysis by abstract reduction using a tableau calculus. In Alan Mycroft, editor, *Static Analysis Symposium '95*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer, 1995.

This article was processed using the LaTeX macro package with LLNCS style