Typed Functional Logic Programming

Ernesto Posse Silvia Takahashi

Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes
Apartado Aereo 4976
Bogotá , Colombia,

mposada@impsat.net.co stakahas@uniandes.edu.co

September 9, 1997

Abstract

EPowerFuL is a typed extension of PowerFuL, an untyped lazy functional language extended with the concept of set abstraction to allow a logic style of programming. EPowerFuL adds a type system in the style of ML (maintaining the lazy semantics) with strong typing, user-defined types, data constructors, and pattern matching.

This paper presents EPowerFuL's syntax and semantics. The Extended PowerFuL Abstract Machine, an extension of the PowerFuL Abstract Machine is briefly described.

Keywords: Programming Language design and implementation, Functional Languages, Logic Languages

1 Introduction

There have been attempts to bring functional and logic programming together [BL86]. Most of these approaches have taken the logic paradigm as the starting point [MNRA92]. Others extend a functional language with concepts extracted from logic programming. PowerFuL [SJ92] falls into this category.

The core of PowerFuL is the non-strict $\lambda-$calculus enriched with several usual basic primitive constructs such as conditionals, predicates, etc. This core is extended with the concept of set abstraction. Sets are first-class objects, so they can be passed as arguments to any function, returned as values, and be part of any data structure. EPowerFuL adds a type system in the style of ML (maintaining the lazy semantics) [Mil84] [MTHM97] with strong typing, user-defined types, data constructors, and pattern matching. The type system allows user defined types, and pattern matching as found in contemporary functional programming languages. Arithmetic operations can be made available [Tak94], but as in Prolog, their implementation involves extra-logical operations.

PowerFuL's set abstraction may seem equivalent to the concept of list comprehension found in some functional languages such as Haskell [JP97]. However, PowerFuL uses an interesting optimization technique for generating sets through narrowing, and EPowerFuL allows the use of types as set generators in addition to expressions.

The *PowerFuL Abstract Machine* (or PAM for short), is an abstract operational model for PowerFuL [Tak94]. It is an extension of the *Categorical Abstract Machine* (CAM)[CCM85] [MA86], an operational model for functional languages. It also borrows elements of the *Warren Abstract Machine* (WAM) [War83] EPowerFuL's implementation model is an extension of the PAM, which is able to deal with user defined types.

In this paper we will cover the syntax of EPowerFuL (Section 2), its denotational semantics accompanied with an informal description of the semantics (Section 3), followed by a brief description of the abstract machine, and a description of the compilation (Section 4)

2 Syntax

Now we present the BNF of EPowerFuL. Terminal symbols are in `teletype`, non-terminal symbols are in *italic*.

```
stmt ::=
    define identifier as expr         (* Value definition *)
  | expr
  | type identifier = type_expr       (* Type definition *)

expr ::=
    atomic_expr
  | unary_operator atomic_expr        (* Unary primitives *)
  | expr1 binary_operator expr2       (* Binary primitives *)
  | if expr1 then expr2 else expr3    (* Conditional *)
  | fun pattern_match_list            (* Functional abstraction *)
  | expr1 expr2                       (* Functional application *)
  | let identifier = expr1 in expr2   (* Local scoping *)
  | { expr }                          (* Singleton *)
  | { expr : qualifier_list }         (* Set comprehension *)

atomic_expr ::=
    constant                          (* Literal value *)
  | identifier                        (* Variables *)
  | ( expr_list )                     (* Tuples *)
  | constructor atomic_expr           (* Constructor application *)

constant ::=
    true | false | 'identifier | integer | tphi

unary_operator ::=
    not | - | fst | snd | isphi?

binary_operator ::=
    and | or | = | < | <= | > | >=
  | + | - | * | / | mod
  | U                                 (* Set union *)

expr_list ::=
    expr | expr , expr_list

qualifier_list ::=
    qualifier
  | qualifier , qualifier_list

qualifier ::=
    identifier in set                 (* Set membership *)
  | expr                              (* Guard, condition *)

set ::=
    base_type                         (* bool, atom, or int *)
  | identifier                        (* User defined type *)
  | expr                              (* Any set *)

pattern_match_list ::=
    pattern_match
  | pattern_match | pattern_match_list

pattern_match ::=
    pattern -> expr                   (* Nu abstraction *)

pattern ::=
    constant                          (* Constant pattern *)
  | identifier                        (* Variable pattern *)
  | constructor expr                  (* Constructor pattern *)
  | ( pattern1 , pattern2 )           (* Pair pattern *)
```

constructor ::=
  `identifier`

type_expr ::=
  `identifier`
| *base_type*
| *type_expr* $*$ *type_expr*
| *sum_type*

base_type ::=
  `bool` | `atom` | `int`

sum_type ::=
  constructor_declaration | *constructor_declaration* | *sum_type*

constructor_declaration ::=
  constructor | *constructor type_expr*

3 Semantics

3.1 Informal semantics

The relevant aspect of the language is set abstraction. As in untyped PowerFuL, there are four expressions for sets in EPowerFuL:

1. The constant `phi`: It represents the empty set.
2. The singleton: $\{expr\}$. Its value is the set whose only element is the value of *expr*.
3. The union between sets: $expr_1$ `U` $expr_2$. Its value is the union of the value of $expr_1$ and the value of $expr_2$, which are set expressions.
4. The set comprehension: $\{expr: qualifier\_list\}$. Its value depends on the form of the qualifiers. There are two kinds of qualifiers: membership constraints (*identifier* `in` *set*), and guards (a Boolean expression). The first declares a variable whose value will be taken from the given set. This variable can appear both in *expr*, and the rest of the qualifier list. A guard describes a condition that should be satisfied, and may include variables declared by membership constraints. Thus, if the set expression is of the form $\{expr\ :\ id$ `in` $set, qualifier\_list\}$, it denotes the union of all subsets denoted by clauses of the form $\{expr\ :\ qualifier\_list\}$, where in each such expression, all free occurrences of *id* have been replaced by an element of *set* (where set can be either an expression denoting a set, a base type such as `bool`, `atom` or `int`, or a user-defined type).

The language has non-strict semantics, so it uses lazy evaluation for data structures, function application, and set abstraction.

The domain of values in EPowerFuL is:

$$\mathtt{D} = \mathtt{B} + \mathtt{A} + \mathtt{Z} + \mathtt{D} \times \mathtt{D} + \mathtt{D} \to \mathtt{D} + \mathcal{P}(\mathtt{D}) + \mathtt{T}_1 + \ldots + \mathtt{T}_n$$

Where each $\mathtt{T}_i$ is a user defined type. There are two forms of user-defined types: product types (tuples), and sum types (variants): A type defined as `type t = t1 * t2 * ... * tn` stands for the type of tuples whose i-th value is of type `ti`. A type defined as `type t = C1 t1 | C2 t2 | ... | Cn tn` represents the disjoint union of types `t1`, `t2`, ..., `tn`. Note that each alternative is distinguished by a data constructor.

A simple example of the traditional functional style in EPowerFuL is the following:

```
(*Type definition *)
type list = Nil | Cons int * list ;

(* Empty list?  *)
```

```
define isNil as
  fun Nil -> True
    | Cons(y,l1) -> False

(* Functional list append *)
define append as
  fun Nil -> (fun l -> l)
  |   Cons(x,l1) -> ( fun l2 -> Cons (x, append l1 l2) ;
```

Now an example of the logical style achieved in PowerFuL:

```
(* The set of prefixes of a given list lst *)
{ x:  x in list, y in list, lst = append x y };
```

The same idea could be expressed in a logical language such as Prolog, but in Prolog the interpreter would give us all the solutions independently, while here we have all the solutions grouped in a set which is a first-class value.

Another interesting example is a function that returns the set of all permutations of a given list:

```
(* Membership to a list:  functional style *)
define member as fun elem ->
  fun Nil -> False
    | Cons(y,l1) -> if x = y then true else member elem l1

(* Remove an element of a list:  functional style *)
define remove as fun elem ->
  fun Nil -> nil
    | Cons(y,l1) -> if x = l1 then true else Cons(y, remove elem l1)

(* Permutations a list's elements:  functional logic style *)
define permutations as fun lst ->
  if isNil lst then  Nil
  else
      { Cons(x, y):  x in int, member x lst, y in permutations (remove x lst) }
```

3.2 Denotational Semantics

Semantic function $\mathcal{E}$ applied to an EPowerFuL expression and an environment returns a value in domain D. The following notation is used : ρ is an environment whose signature is ρ : $\texttt{Identifiers} \rightarrow \texttt{D}$. $\rho(id)$ is the value associated with the identifier id in the environment ρ. $\rho[\mathrm{d}/id]$ represents the environment in which id is bound to d. That is: $\rho[d/id] = \lambda x.\text{if } x = id \text{ then } d \text{ else } \rho(x)$. Figure 1 list the semantic equations.

3.2.1 Pattern matching

To handle pattern matching we need a couple of auxiliary functions. Function $\mathcal{P}$ iterates over the patterns in the function definition until it finds one that matches its argument.

$$\mathcal{P}(\text{arg}, [\![]\!], \rho) \;=\; \bot$$
$$\mathcal{P}(\text{arg}, [\![p- > e|rest]\!], \rho) =$$
$$\textbf{Let } r \;=\; \mathcal{M}(arg, [\![p]\!], [\![e]\!], \rho) \texttt{ in } \textbf{If}(\textbf{Left}(r), \textbf{Right}(r), \mathcal{P}(\text{arg}, [\![rest]\!], \rho))$$

Function $\mathcal{P}$ uses function $\mathcal{M}$ which verifies if the argument matches the corresponding pattern, and if so, it evaluates the corresponding expression with the appropriate bindings. It returns a pair whose first element is **True** if the match succeeded and **False** otherwise. Additionally, if it succeeds, the second element is the value of the expression; if not, it is bottom ($\bot$).

We assume we have two functions: **CheckCons** and **RetrCnstr**. The first verifies the presence of a given constructor, and the second removes the constructor from its argument. In section 3.2.3 there is an explanation of these functions.

- Case 1: When the pattern is a constant k:

$\mathcal{E}[\![constant]\!]\rho \;=\; constant$
$\mathcal{E}[\![\texttt{identifier}]\!]\rho \;=\; \rho(\texttt{identifier})$ if $\rho(\texttt{identifier})$ is defined
$\qquad = \;\mathbf{Variable}(\texttt{identifier})$ otherwise
$\mathcal{E}[\![\texttt{C}\ expr]\!]\rho \;=\; \mathbf{CnstrctApp}(\texttt{C}, \mathcal{E}[\![expr]\!]\rho)$
Where C is a data constructor of a sum type.
$\mathcal{E}[\![unaryop\ expr]\!]\rho \;=\; UnaryOp(\mathcal{E}[\![expr]\!]\rho)$
Where $unaryop \in \{\texttt{not}, -, \texttt{fst}, \texttt{snd}, \texttt{isphi?}\}$
and $UnaryOp \in \{\mathbf{Not}, \mathbf{Neg}, \mathbf{Left}, \mathbf{Right}, \mathbf{IsPhi}\}$.
The correspondence between *unaryop* and *UnaryOp* should be obvious.
$\mathcal{E}[\![expr_1\ binaryop\ expr_2]\!]\rho \;=\; BinaryOp(\mathcal{E}[\![expr_1]\!]\rho, \mathcal{E}[\![expr_2]\!]\rho)$
Where $binaryop \in \{\texttt{and}, \texttt{or}, <, <=, >, >=, +, -, *, /, \texttt{mod}, \texttt{U}\}$
and $BinaryOp \in \{\mathbf{And}, \mathbf{Or}, \mathbf{Lt}, \mathbf{Lte}, \mathbf{Gt}, \mathbf{Gte}, \mathbf{Plus}, \mathbf{Minus}, \mathbf{Times}, \mathbf{Div}, \mathbf{Mod}, \mathbf{Union}\}$.
The correspondence between *binaryop* and *BinaryOp* should be obvious
$\mathcal{E}[\![expr_1 \;=\; expr_2]\!]\rho \;=\; \mathbf{DeqD}(\mathcal{T}[\![expr_1]\!], \mathcal{E}[\![expr_1]\!]\rho, \mathcal{E}[\![expr_2]\!]\rho)$
Where T is the function that returns the type of a given expression.
DeqD, is a function defined in section 3.2.2
$\mathcal{E}[\![\texttt{if}\ expr_1\ \texttt{then}\ expr_2\ \texttt{else}\ expr_3)]\!]\rho \;=\; \mathbf{IF}(\mathcal{E}[\![expr1]\!]\rho, \mathcal{E}[\![expr2]\!]\rho, \mathcal{E}[\![expr3]\!]\rho)$
$\mathcal{E}[\![\texttt{fun}\ L]\!]\rho \;=\; \lambda\ d.\mathcal{P}(\texttt{d}, \mathcal{L}, \rho)$
Where L is of the form $p_1 -> e_1 | p_2 -> e_2 | \ldots p_n -> e_n$, in which each p_i is a pattern and each e_i is an expression.
d is a new identifier. $\mathcal{P}$ is the function that defines the semantics of pattern matching and is defined in section 3.2.1
$\mathcal{E}[\![expr_1\ expr_2]\!]\rho \;=\; (\mathcal{E}[\![expr_1]\!]\rho)\mathcal{E}[\![expr_2]\!]\rho$
$\mathcal{E}[\![\texttt{let}\ identifier \;=\; expr_1\ \texttt{in}\ expr_2]\!]\rho \;=\; \mathcal{E}[\![expr_2]\!]\rho[\mathcal{E}[\![expr_1]\!]\rho/identifier]$
$\mathcal{E}[\![\{expr{:}\}]\!]\rho \;=\; \mathbf{Singleton}(\mathcal{E}[\![expr]\!]\rho)$
$\mathcal{E}[\![\{expr{:}condition, qualifierlist\}]\!]\rho \;=\; (\mathbf{IF}_S(\mathcal{E}[\![condition]\!]\rho, \mathcal{E}[\![\{expr{:}qualifierlist\}]\!]\rho, \phi))$
$\mathcal{E}[\![\{expr\ :\ id\ \texttt{in}\ genexpr, qualifierlist\}]\!]\rho \;=\; \mathbf{AppS}(\lambda\ X.\ (\mathcal{E}[\![\{expr\ :\ qualifierlist\}]\!]\rho[X/id]), (\mathcal{E}[\![genexpr]\!]\rho))$
X is a new identifier.
$\mathcal{E}[\![\{expr\ :\ id\ \texttt{in}\ type, qualifierlist\}]\!]\rho \;=\; type(d).(\mathcal{E}[\![\{expr\ :\ qualifierlist\}]\!]\rho[d/id])$
d is a new identifier.

Figure 1: Denotational Equations

$$\mathcal{M}(arg, [\![k]\!], [\![e]\!], \rho) = \mathbf{If}(\mathbf{DeqD}(\mathcal{T}[\![k]\!], arg, k), \mathbf{Pair}(\mathbf{True}, \mathcal{E}[\![e]\!]\rho), \mathbf{Pair}(\mathbf{False}, \bot))$$

- Case 2: When the pattern is a variable x:

$$\mathcal{M}(arg, [\![x]\!], [\![e]\!], \rho) = \mathcal{E}[\![e]\!]\rho[arg/x]$$

- Case 3: When the pattern is the application of a constructor K with argument the pattern p:

$$\mathcal{M}(\text{arg}, [\![K\ \ p]\!], [\![e]\!], \rho) = \mathbf{If}(\mathbf{CheckCons}(K, arg), \mathcal{M}(\mathbf{RetrCnstr}(arg), [\![p]\!], [\![e]\!], \rho), \mathbf{Pair}(\mathbf{False}, \bot))$$

- Case 4: When the pattern is a pair of patterns (p_1,p_2):

$$\begin{array}{l} \mathcal{M}(arg, [\![(p_1, p_2)]\!], [\![e]\!], \rho) = \\ \quad \mathbf{Let}\ a = \mathbf{Left}(arg)\ \mathbf{in} \\ \quad \mathbf{Let}\ b = \mathbf{Right}(arg)\ \mathbf{in} \\ \quad \mathbf{If}(\mathbf{And}(\mathcal{N}(a, p1), \mathcal{N}(b, p2)), \\ \qquad \mathbf{Pair}(\mathbf{True}, (\mathcal{E}[\![\mathbf{fun}\ p_1 -> (\mathbf{fun}\, p_2 -> e)]\!]\rho)\, a\, b), \\ \qquad \mathbf{Pair}(\mathbf{False}, \bot)) \end{array}$$

$$\begin{array}{l} \mathcal{N}(\text{arg}, [\![k]\!]) = \mathbf{DeqD}(\mathcal{T}[\![k]\!], \text{arg}, k) \\ \mathcal{N}(\text{arg}, [\![x]\!]) = \mathbf{True} \\ \mathcal{N}(\text{arg}, [\![K\ \ p]\!]) = \mathbf{If}(\mathbf{CheckCons}(K, \text{arg}), \mathcal{N}(\mathbf{RetrCnstr}(K, \text{arg}), [p], \mathbf{False}) \\ \mathcal{N}(\text{arg}, [\![(p_1, p_2)]\!]) = \mathbf{And}(\mathcal{N}(\mathbf{Left}(\text{arg}), p_1), \mathcal{N}(\mathbf{Right}(\text{arg}), p_2) \end{array}$$

In case 4, $\mathcal{M}$ relies on another semantic function called ($\mathcal{N}$) that also performs pattern matching but without making any bindings or evaluating expressions. After checking the correspondence, $\mathcal{M}$ transforms the pattern matching to evaluate a curried version of the function.

3.2.2 Equality

Function **DeqD** is used for determining equality between values of a same type (a base type or a user-defined type) by performing structural equality between terms. It assumes that type synthesis has been performed, so that the type of its arguments agree, and so that it can compute equality based on the type of its arguments. The first argument is the type (this information should be available, for instance in the expression's node in the abstract syntax tree). We suppose we have a function **BeqB** that tests equality between booleans, **AeqA** between atoms, and **ZeqZ** between integers.

- If the first argument is a base type, equality is tested with the appropriate function:

$$\mathbf{DeqD}(t, x, y) = T\mathbf{eq}T(x, y)$$
$$\texttt{Where } t \in \{\texttt{bool}, \texttt{atom}, \texttt{int}\} \texttt{ and } T \in \{B, A, Z\}$$

- If the first argument is a product type, the test is recursively performed in its constituent parts:

$$\mathbf{DeqD}(t_1 * t_2, x, y) =$$
$$\mathbf{And}(\mathbf{DeqD}(t_1, \mathbf{Left}(x), \mathbf{Left}(y)), \mathbf{DeqD}(t_2, \mathbf{Right}(x), \mathbf{Right}(y)))$$

- If the first argument is a sum type, it must match the constructors and then recursively test the arguments of the constructors:

```
$\mathbf{DeqD}([], x, y) = \perp$
$\mathbf{DeqD}(Ct|L, x, y) =$
  $\mathbf{If}(\mathbf{CheckCons}(C, x),$
    $\mathbf{If}(\mathbf{CheckCons}(C, y), \mathbf{DeqD}(t, \mathbf{RetrCnstr}(C, x), \mathbf{RetrCnstr}(C, y)), \mathbf{False}),$
    $\mathbf{DeqD}(L, x, y))$
```

3.2.3 Primitive Functions

PowerFuL has primitive functions that are not usually found in other languages. Among these there are some primitives that deal with constructors. This is also the case of the set-related primitives: **IsPhi**, $\mathbf{IF}_S$, **AppS**, and The constant **Phi** and functions **Singleton**, **Union**, and *type*(X).*expr* are constructor functions and define new types of values available in the language's domain.

CnstrctApp is a primitive function that represents the application of a constructor of a sum-type. **CheckCons** is a predicate that returns **True** if its second argument (a value) has a constructor, and it is equal to the first argument (a constructor tag). Finally **RetrCnstr** removes the given constructor from its second argument.

Given `T` a user-defined sum-type declared: `type T = `$\texttt{C}_1\ \texttt{T}_1 \mid \ldots\ \texttt{C}_n\ \texttt{T}_n$. We define:

$$\mathbf{ConstructorTag}(\texttt{T}) = \{\texttt{C1}, \texttt{C2}, \ldots, \texttt{Cn}\}$$

$\mathbf{CnstrctApp}: \ \mathrm{ConstructorTag}(T) \times T_i \rightarrow T$
A constructor can be seen as a function of type: $T_i \rightarrow T$.

$\mathbf{CheckCons}: \mathrm{ConstructorTag}(T) \times D \rightarrow B$
$\mathbf{CheckCons}(\mathbf{Ci}, \mathbf{CnstrctApp}(\mathrm{C_i}, \mathrm{e})) = \mathbf{True}$
$\mathbf{CheckCons}(\mathbf{Ci}, \mathbf{CnstrctApp}(\mathrm{K_i}, \mathrm{e})) = \mathbf{False}$ Where $K_i \neq C_i$
$\mathbf{CheckCons}(\mathbf{Ci}, \mathrm{e}) = \perp$ `Where e is not a constructor application`

$\mathbf{RetrCnstr}: \ \mathrm{ConstructorTag}(T) \times D \rightarrow D$
$\mathbf{RetrCnstr}(\mathrm{C_i}, \mathbf{CnstrctApp}(\mathrm{C_i}, \mathrm{e})) = \mathrm{e}$
$\mathbf{RetrCnstr}(\mathrm{C_i}, \mathrm{e}) = \perp$

Phi represents the empty set. **IsPhi** is a predicate that returns **True** if its argument is **Phi**. **Singleton** constructs a one-element set with its argument. **Union**, constructs the union of its arguments (we use ". ∪ ." instead of **Union**(.,.) to compact notation where warranted). $\mathbf{IF}_S$, is defined as **If** except that if its argument is undefined, it returns **Phi** instead of bottom ($\perp$). Function **AppS** distributes a function over all the elements of a set and unions together all results. Its behavior over expressions of the form *type*(X).*expr* is explained in the next section.

$\mathbf{IsPhi} : \mathcal{P}(D) \to B$
$\mathbf{IsPhi}(\mathbf{Phi}) = \mathbf{True}$
$\mathbf{IsPhi}(\mathbf{Singleton}(\mathrm{expr})) = \mathbf{False}$
$\mathbf{IsPhi}(\mathbf{Union}(\mathrm{expr}_1, \mathrm{expr}_2)) = \mathbf{And}(\mathbf{IsPhi}(\mathrm{expr}_1), \mathbf{IsPhi}(\mathrm{expr}_2))$
$\mathbf{IsPhi}(\perp) = \perp$

$\mathbf{IF}_S : B \times \mathcal{P}(D) \times \mathcal{P}(D) \to \mathcal{P}(D)$
$\mathbf{IF}_S(\mathbf{True}, x, y) = x$
$\mathbf{IF}_S(\mathbf{False}, x, y) = y$
$\mathbf{IF}_S(\perp, x, y) = \mathbf{Phi}$

$\mathbf{AppS} : (D \to \mathcal{P}(D)) \times \mathcal{P}(D) \to \mathcal{P}(D)$
$\mathbf{AppS}(\mathrm{f}, \mathbf{Phi}) = \mathbf{Phi}$
$\mathbf{AppS}(\mathrm{f}, \mathbf{Singleton}(\mathbf{expr})) = \mathrm{f}(\mathbf{expr})$
$\mathbf{AppS}(\mathrm{f}, \mathbf{Union}(\mathbf{s1}, \mathbf{s2})) = \mathbf{Union}(\mathbf{AppS}(\mathrm{f}, s1), \mathbf{AppS}(\mathrm{f}, s2))$

3.3 Narrowing

This section summarizes the narrowing techniques used to obtain logic programming capability in [SJ92]. We extend this technique to deal with user defined types.

In $\mathbf{AppS}(\lambda \mathrm{x}.body, generator)$, *body* should be applied to each element in *generator*. When using types instead of expressions, direct enumeration of the elements of a type is theoretically feasible. However, by making this a primitive an interesting optimization can be achieved.

$\mathcal{E}[\![\{expr \ : \ id \in type, qualifierlist\}]\!]\,\rho \ = \ type(d).(\mathcal{E}[\![\{expr : \ qualifierlist\}]\!]$
Where type is either a base type (i.e. `bool`, `atom`, or `int`), or a user defined type.

In this case, instead of generating every element in *type*, a variable (`d`), constrained to take its values from *type*, is used. Thus, `d` becomes an enumeration parameter and is considered a logic variable. An expression of the form

$$t(\mathtt{d}).body$$

can be simplified so that *body* does not have to be evaluated for every element in t, as it would be done in

$$\mathbf{AppS}(\lambda\, \mathtt{d}.body, t).$$

Instead, *body* is evaluated in its parametrisized form. This simplification is known as *narrowing*.

Even with a logic variable, an expression may be simplified. To describe a primitive applied to a logical variable the following notation is used:

$$constraint(u).(\ldots prim(u) \ldots).$$

Here we wish to simplify the primitive *prim* which occurs somewhere in the body and has as argument a logic variable `u`. Consider for instance primitive **Not**, in expression `{e x in bool, ...not x...}`. This translates into

$$\mathbf{bool}(\mathtt{u}).(\ldots \mathbf{Not}(\mathtt{u}) \ldots).$$

By static type-checking we know that `u` is constrained to be in **bool**. Hence, its only possible values are **True** and **False**. Therefore, we can simplify (narrow) and obtain:

$$\mathbf{bool(u)}.(\ldots \mathbf{Not(u)} \ldots) = (\ldots \mathbf{True} \ldots)[\mathbf{False/u}] \ \cup (\ldots \mathbf{False} \ldots)[\mathbf{True/u}]$$

Another example: consider the expression

$$\{\mathtt{e} : \ \mathtt{x\ in\ atom},\ \mathtt{y\ in\ atom}, \ldots \mathtt{x} = \mathtt{y} \ldots\}.$$

This would be ultimately translated into something like

$$\mathbf{atom}(x).\mathbf{atom}(y).(\ldots \mathbf{AeqA}(x, y) \ldots)$$

that is, a primitive applied to two logical variables, and after narrowing would yield:

$$\mathbf{atom}(x).(\ldots\mathbf{True}\ldots)[x/y] \cup \mathbf{atom}(x).\mathbf{atom}(y).(x \neq y).(\ldots\mathbf{False}\ldots).$$

Here we have divided into two possibilities: the equality is either **True** or **False**. If it is **True**, we bind the two variables together and remove a constraint; if it is false, we add a new constraint that indicates the inequality between the two variables. Therefore, for each logic variable we have type constraints and inequality constraints. If **AeqA** is used with only one logical variable instead of two, the result is analogous, but with less constraints:

$$\mathbf{atom}(x).(\ldots\mathbf{AeqA}(x,\mathrm{a})\ldots) = (\ldots\mathbf{True}\ldots)[\mathrm{a}/x] \cup \mathbf{atom}(x).(x \neq \mathrm{a}).(\ldots\mathbf{False}\ldots).$$

We have analogous rules for primitives that can apply to types that can be narrowed. We include narrowing for equality primitives, Boolean primitives, and atom primitives, but we do not allow logic variables for integer primitives, with the exception of **ZeqZ** which is analogous to **AeqA**.

$$\mathbf{bool}(x).\mathbf{bool}(y).(\ldots\mathbf{BeqB}(x,y)\ldots) = \mathbf{bool}(y).(\ldots y\ldots)[\mathbf{True}/x] \cup \mathbf{bool}(y).(\ldots\mathbf{Not}(y)\ldots)[\mathbf{False}/x].$$

The conditional can also be applied to a logic variable:

$$\mathbf{bool}(x).(\ldots\mathbf{IF}(x,e_1,e_2)\ldots) = \mathbf{bool}(y).(\ldots e_1\ldots)[\mathbf{True}/x] \cup \mathbf{bool}(y).(\ldots e_2\ldots)[\mathbf{False}/x].$$

Now, for user defined types we need some special rules:

- If t is a product type of the form $t_1 \times t_2$ then

  $$t(u).(\ldots prim(u)\ldots) = t_1(v).t_2(w).(\ldots prim(u)\ldots)[\mathbf{Pair}(v,w)/u]$$

  Where v and w are new variables. In this case the logic variable is replaced by a pair of new logic variables each constrained to a given element type of the product type.

- If t is a sum type of the form $C_1\ t_1|C_2\ t_2|\ldots|C_n\ t_n$ then

  $$\begin{aligned} &t(u).(\ldots prim(u)\ldots) = \\ &\quad t_1(v_1).(\ldots prim(u)\ldots)[\mathbf{CnstrctApp}(C_1,v_1)/u] \\ &\quad \cup t_2(v_2).(\ldots prim(u)\ldots)[\mathbf{CnstrctApp}(C_2,v_2)/u] \\ &\quad \ldots \\ &\quad \cup t_n(v_n).(\ldots prim(u)\ldots)[\mathbf{CnstrctApp}(C_n,v_n)/u] \end{aligned}$$

  Where $v1$, $v2$, $\ldots$, v_n are new variables. In this case, we divide the simplification among n alternatives each of which assigns the logic variable to an object with a constructor, the argument being a new logic variable constrained to the type associated to the constructor. If the i-th constructor does not have an argument, it is treated as a constructor whose argument is **Nil** and there is no need for a new logical variable:

  $$\begin{aligned} &t(u).(\ldots prim(u)\ldots) = \\ &\quad \ldots \cup (\ldots prim(u)\ldots)[\mathbf{CnstrctApp}(C_i, Nil)/u] \cup \ldots \end{aligned}$$

This leads to the following rules:

- Given type t of the form $t_1 \times t_2$ then

  $$\begin{aligned} &t(u).(\ldots\mathbf{Left}(u)\ldots) = \\ &\quad t_1(v).t_2(w).(\ldots v\ldots)[\mathbf{Pair}(v,w)/u] \\ &t(u).(\ldots\mathbf{Right}(u)\ldots) = \\ &\quad t_1(v).t_2(w).(\ldots w\ldots)[\mathbf{Pair}(v,w)/u] \end{aligned}$$

- Given type t of the form
$$C_1\ t_1|C_2\ t_2|\ldots|C_n\ t_n$$

$$\begin{aligned}
&t(u).(\ldots \mathbf{RetrCnstr}(Ci,u)\ldots) = \\
&\quad t_1(v_1).(\ldots\bot\ldots)[\mathbf{CnstrctApp}(C_1,v_1)/u] \\
&\quad \cup t_2(v_2).(\ldots\bot\ldots)[\mathbf{CnstrctApp}(C_2,v_2)/u] \\
&\quad \ldots \\
&\quad \cup t_i(v_i).(\ldots u\ldots)[\mathbf{CnstrctApp}(C_i,v_i)/u] \\
&\quad \ldots \\
&\quad \cup t_n(v_n).(\ldots\bot\ldots)[\mathbf{CnstrctApp}(C_n,v_n)/u]
\end{aligned}$$

Consider for instance the following type definition and the expression that refers to that type:

```
type t = int*bool ;

{e:  x in t, snd x }
```

The expression's meaning is of the form
$$\mathbf{t(u).(\ldots Right(u)\ldots).}$$

This should narrow as:
$$\mathbf{int}(v).\mathbf{bool}(w).(\ldots w\ldots)[\mathbf{Pair}(v,w)/u].$$

If the primitive was not a pair primitive, the result would be $\bot$.

Another example: if we have:

```
type t = None | Some int;

{e:  x in t, ...x=Some 2 ...}
```

will be translated into something like
$$t(u).(\ldots \mathbf{CheckCons}(\mathit{Some},u)\ldots).$$

This can be simplified as

$$\begin{aligned}
&(\ldots \mathit{False}\ldots)[\mathbf{CnstrctApp(None,Nil)}/u]\cup \\
&\mathbf{int}(v).(\ldots \mathit{True}\ldots)[\mathbf{CnstrctApp(Some},v)/u].
\end{aligned}$$

4 The Abstract Machine

The PAM is based on the Categorical Abstract Machine (CAM) [CCM85] [MA86] and borrows ideas form the Warren Abstract Machine [War83] to implement its logic capabilities. The PAM's design resembles that of the CAMEL [Muc92] which is used implement rule-based functional logic languages. It is important to point out that EPowerFuL can not be implemented directly using CAMEL because this operational model does not support sets as first class objects nor inequality.

The CAM has the following data structures: a stack, a current value and a code. The values it deals with are atoms, pairs, closures for functional abstraction, and closures for dealing with lazy evaluation. The PAM adds two data structures: a heap to store logic variables and a choice point stack to deal with sets. The PAM's state is given by 5-tuple:

$$(\mathtt{Value},\mathtt{stack},\mathtt{choice}-\mathtt{point}-\mathtt{stack},\mathtt{heap})$$

EPowerFuL term representation as well as primitive instructions for each of PowerFuL's primitive functions must be defined. Atoms. integers, and booleans, basic values in CAML-Like's denotational domain, are basic values in the modified CAM. We add a constructor to represent pairs: `Pair(.,.)` explicitly.

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
app_s;C	(clo(V,C1),phi)	S	hp	CS	C	phi	S	hp	CS
app_s;C1	(clo(V,C),single(A))	S	hp	CS	C1	(V,A)	C:S	hp	CS
union;C	(v,u)	S	hp	CS	C	v	S	hp	cp(u,S,C):CS
exe;C	phi	S	hp	cp(u,S1,C1)::CS	C1	u	S1	hp	CS

Figure 2: Transitions for Sets in the PAM

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
$\texttt{cur}_u$(t,K);C	v	S	hp	CS	C	$\texttt{clo}_u$(u,v,K)	S	hp	CS
app1;C	$\texttt{clo}_u$(t,v,K)	S	hp	CS	K	(v,X)	C_1:S	new(t,X,X)@hp	CS
dot(X);C	v	S	hp	CS	C	(_dot,(X,v))	C:S	hp	CS
					Where C_1 = (exe;dot(X);C)				

Figure 3: Some PAM Instructions

A new constructor is added to represent singleton sets: `single(.)`. An expression of the form:

$$expr_1 \cup expr_2$$

can be interpreted as $expr_1$ *or* $expr_2$. Operationally, this is interpreted as follows: when a union between two sets is found, the first subset is chosen as the result of the computation, and information is stored so that the machine can restore it when computation of the first subset has concluded. Each choice point saves the state of the machine at any given moment and is represented with a four-place functor: `cp(Code,Value,Stack,Heap)`. The instruction that implements union introduces choice points. Its description is shown in Figure 2 along with the definition of `app_s`. Instruction `exe` is used to drive computation.

Logic variables are created when implementing objects of the form

$$t(x).Expr.$$

We add a new constructor `LogVar(.)` (at times abbreviated `LV(.)`) to distinguish logic variables from other values. The heap is an array of variables in which the following information is stored: its name, value, type, inequality constraints, and a dependency pointer:

`hp(Name, Type, Value, Constraints, Dependency)`

If a is variable created through narrowing, it depends on the variable that was narrowed to create it; otherwise, it depends on itself. When a primitive cannot be applied because of a free variable, narrowing causes new choice points to be added to the choice point stack and new logic variables may be created.

The symbol table and a type table are active during execution time. The symbol table stores information regarding defined functions. The type table stores information related to user defined types. This information will be used to determine narrowing.

Two different structures are used to represent expressions of the form: $type(X).Exp$. A new kind of closure (called a *constrained closure*) is used to represent unsimplified constrained expressions: $\texttt{clo}_u$`(t,v,C)` where `t` is the type of the variable that is being constrained; `C` is the code which evaluates the constrained expression and `v` is the environment where it should be evaluated. Instruction $\texttt{cur}_u$ is used to build these closures. Variables created through narrowing are part of the value of the "top-most" logic variables, but will not generate constrained closures. When evaluation of a constrained expression is complete, we create a fully computed object of the form `(_dot, (X, value))` where `X` can be bound to a value which may contain newly created logic variables. This structure, in fact, is only used when printing.

In Figure 3, transitions for building constrained closures and for applying constrained closures to newly created logic variables are defined. Notation `new(t,X,D)@hp` is used to represent a heap that is constructed

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
br(C1,C2);C	(true,V)	S	hp	CS	C1	V	S	hp	CS
br(C1,C2);C	(false,V)	S	hp	CS	C2	V	S	hp	CS
br(C1,C2);C	(LV(X),V)	S	hp	CS	C1	V	S	hp[true/X]	cp:CS
					cp=cp(C2,V,S,hp[false/X])				
not;C	true	S	hp	CS	C	false	S	hp	CS
not;C	false	S	hp	CS	C	true	S	hp	CS
not;C	(LV(X),V)	S	hp	CS	C	false	S	hp[true/X]	cp:CS
					cp=cp(C,true,S,hp[false/X])				

Figure 4: Boolean Based PAM Instructions

from heap `hp` by adding a new free logic variable `X` of type `t` and that depends on `Y`. Instruction `dot(X)` also defined in Figure 3 is called before printing a constrained expression

The general schema for narrowing is the following. The variable is assigned one of the possible values, and the machine's data structures are modified accordingly. Additionally, choice points are introduced: one choice point for each other possible value that the logic variable may have and the corresponding changes to the machine's data structures. In fact, we are creating an alternative to which we will return when a failure is encountered.

The basic narrowing scheme for PAM instructions that are applied to Boolean values is shown in Figure 4. Instruction `br(c1,c2)` is used to implement conditionals. Notation `hp[val/X]` is used to represent a heap which is obtained from `hp` by assigning logic variable `X` the value `val`.

Figure 5 shows the transition tables for the implementation of `AeqA`. The interesting case is that in which both arguments are logic variables. If they are the same variable, we replace the value by `true` and continue. If their equality would violate a constraint of any of the two variables, we replace the value by `false` and continue. Finally, if neither of these two cases occur, we bind the two variables together and replace the value by `true` and add a choice point in which an inequality constraint has been added and the value has been set to `false`. Function `comp(X,Y)` receives two unbound logical variables and returns `true` if the two variables are the same variable, `false` if their equality would violate a constraint, and `unknown` otherwise. The following notation is used to indicate changes in the heap: `bind(X,Y,hp)` to indicate that variables `X` and `Y` are bound together and `neq(X,Y,hp)` to indicate that inequality constraints have been added for `X` and `Y` indicating their inequality.

Instruction `isA(a)`, described in Figure 5, is used to check if the current value is the atom `a` If the current value is is a logical variable say `A`, it checks if an inequality constraint would be violated if `A=a`. If so, the current value is replaced by `false`. If not, `A` is assigned the value `a`, the current value is changed to `true`, and a choice point in which with the constraint: `A` $\neq$ `a` and value `false` is added to the choice point stack.

Other narrowing schemes are shown in Figure 6. When applying `fst` to a logic variable `x` of type `(t1*t2)`, two new logic variables: `y` and `z` are added, and `x` is assigned `pair(y,z)`. Both newly created variables depend on `x`. In this case no new choice pint are added. When applying the instruction that implements constraint retraction to a logic variable `x` of type `(C1 t1 | ... | Cn t2)`, We add a choice point for every `Ci` with the corresponding changes to the machine's data structures.

The PAM's correctness was proven in [Tak94].

Compilation consists in translating expressions in PowerFuL's denotational domain into PAM instructions. The following is an excerpt of the translation scheme from PowerFuL denotational syntax to the CAM.

$\mathcal{C}[\![\mathbb{A}_i]\!]_\rho$`=quote(`$A_i$`)`
$\mathcal{C}[\![\lambda x.E]\!]_\rho$`=cur(`$\mathcal{C}[\![E]\!]_{(\rho,x)}$`)`
$\mathcal{C}[\![(Expr_1, Expr_2)]\!]_\rho$`=`
          `push; fre(`$\mathcal{C}[\![Expr_1]\!]_\rho$`); swap;`
          `fre(`$\mathcal{C}[\![Expr_2]\!]_\rho$`); cons`
$\mathcal{C}[\![Expr_1\ Expr_2]\!]_\rho$`= push;` $\mathcal{C}[\![Expr_1]\!]_\rho$`; swap;`
          `fre(`$\mathcal{C}[\![Expr_2]\!]_\rho$`); cons; app`

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
AeqA;C	(a,a_1)	S	hp	CS	IsA(a);C	a_1	S	hp	CS
AeqA;C	(LV(x),a)	S	hp	CS	IsA(a);C	LV(x)	S	hp	CS
AeqA;C	(a,LV(x))	S	hp	CS	IsA(a);C	LV(x)	S	hp	CS
Where a is an atom									
AeqA;C	(LV(y),LV(x))	S	hp	CS	C	true	S	hp_1	cp:CS
if comp(X,Y)=true									
AeqA;C	(LV(y),LV(x))	S	hp	CS	C	false	S	hp_1	cp:CS
if comp(X,Y)=false									
AeqA;C	(LV(y),LV(x))	S	hp	CS	C	true	S	hp_1	cp:CS
if comp(X,Y) = unknown					hp_1=bind(X,Y,hp)				
					cp=cp(C,false,S,neq(hp,X,Y))				

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
IsA(a);C	a_1	S	hp	CS	C	b	S	hp	CS
a_1 is an atom					b = (a= a_1)				
IsA(a);C	LV(X)	S	hp	CS	C	false	S	hp	CS
if comp(X,a) = false									
IsA(a);C	LV(X)	S	hp	CS	C	true	S	hp[a/X]	cp:CS
if comp(X,a) = unknown					cp=cp(C,false,S,neq(X,A,hp))				

Figure 5: Transition Tables for AeqA with Logical Variables

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
fst;C	Pair(u,v)	S	hp	CS	C	u	S	hp	CS
fst;C	LogVar(x)	S	hp	CS	_bot	v	S	hp_1	CS
where x is of type t_1 * t_2									

Where hp_1 =
new(t_1,y, x) @ new(t_2,z, ,x) @ hp[Pair(y,z)/x]

OLD STATE					NEW STATE				
C	V	S	H	CP	C	V	S	H	CP
retrC(K);C	ConsApp(K ,v)	S	hp	CS	C	v	S	hp	CS
retrC(K);C	ConsApp(K_1 ,v)	S	hp	CS	C	_bot	S	hp	CS
where $K_1 \neq$ K									
retrC(K);C	e	S	hp	CS	C	_bot	S	hp	CS
where e is not of the form ConsApp(,)									
retrC(K);C	LogVar(x)	S	hp	CS	C	r1	S	hp1	CS'
where x is of type t= C1 T1 | ... | Cn Tn									

Where CS' = cp(C,r_2,S,hp_2) *::*
...
cp(C,r_n,S,hp_n)
and r_i = *if* K = C_i *then* LogVar(u_i) *else* _bot
and hp_i = new(t_i, u_i, x) @ hp[ConsApp(C_i,u_i)/x]

Figure 6: Narrowing for User-defined types

$\mathcal{C}[\![\mathbf{ConstructorApp}(K, Expr)]\!]_\rho$=
            `fre(`$\mathcal{C}[\![Expr]\!]_\rho$`);`
            `mkCnstr(K)`
$\mathcal{C}[\![\mathbf{Left}(Expr)]\!]_\rho$=$\mathcal{C}[\![Expr]\!]_\rho$`;unf;fst`
$\mathcal{C}[\![\mathbf{RetrCnstr}(K, Expr)]\!]_\rho$=$\mathcal{C}[\![Expr]\!]_\rho$`;unf;retrC(K)`
$\mathcal{C}[\![type(x).Expr]\!]_\rho$=`cur`$_u$`(type,`$\mathcal{C}[\![Expr]\!]_{(\rho,x)}$`;unf)`

5 Conclusions

PowerFuL is a declarative programming language that uses a pure functional language as a starting point. and achieves logic programming capabilities through relative set abstraction. Its approach is interesting in that sets and functions are considered first class objects both syntactically and semantically through the use of domain theory as a unifying semantics.

However interesting from the theoretical point of view, PowerFuL's syntax is not very user-friendly. and it lacks many of the features are provided by modern functional languages. EPowerFuL addresses these drawbacks by adding a type system and changing the syntax so that it resembles modern functional languages while keeping relative set abstraction so the language's logic programming capabilities are maintained. The resulting language is a strongly typed language which is easier to use. Strong typing also simplifies its semantics, for most of type checking is done at compile time instead of at execution time.

The language is in an early stage of development. and the current implementation does not support the more advanced features such as polymorphic types. Its syntax is also somewhat restricted. However, a stack-based implementation model is defined for the language showing its feasibility, and an implementation of this model has been constructed using OCAML.

References

[BL86] M. Bella and G. Levi. The Relation Between Logic and Funcitonal Languages: A Survey. *Journal Logic Programming*, 3:217–236, 1986.

[CCM85] G. Cousineau, P-L. Curien, and M. Mauny. The Categorical Abstract Machine. In J-P Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, LNCS 201, pages 50–64. Springer-Verlag, Nancy, 1985.

[JP97] et. al. J. Peterson. The haskell report-version 1.4. http://haskell.org, 1997.

[MA86] M. Mauny and A.Suarez. Implementing functional languages in the categorical abstract machine. In *ACM Symposium on LISP and Functional Programming*, pages 266–278, Cambridge, 1986.

[Mil84] R. Milner. A proposal for standard ml. In *ACM Symposium on LISP and Functional Programming*. pages 184–197, Austin, 1984.

[MNRA92] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic Programming with Functions and Predicates: The Language Babel. *The Journal of Logic Programming*, 12:191–223, 1992.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Dediniton of Standard ML (Revised)*. MIT Press, Cambridge, 1997.

[Muc92] A. Muck. CAMEL: An extension of the categorical abstract machine to compile functional-logic programs. In M. Bruynooghe and M. Wirsing, editors, *4th international Symposium PLILP 92*, pages 341–354. Lueven, Belgium, 1992.

[SJ92] F.S.K. Silbermann and B. Jayaraman. A domain-theoretic approach to functional and logic programming. *Journal of Functional Programming*, 2(3), 1992.

[Tak94] S. Takahashi. *An Abstract Operational Model for a Functional-Logic Programming Language*. PhD thesis. Tulane University, New Orleans, 1994.

[War83] D. Warren. An abstract instruction set for prolog. Technical Report 309, SRI International, 1983.