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.

\[
\text{stmt ::=}
\begin{align*}
& \text{define identifier as expr} \quad \text{(* Value definition *)} \\
& \text{expr} \\
& \text{type identifier = type_expr} \quad \text{(* Type definition *)}
\end{align*}
\]

\[
\text{expr ::=}
\begin{align*}
& \text{atomic_expr} \\
& \text{unary_operator atomic_expr} \quad \text{(* Unary primitives *)} \\
& \text{expr1 binary_operator expr2} \quad \text{(* Binary primitives *)} \\
& \text{if expr1 then expr2 else expr3} \quad \text{(* Conditional *)} \\
& \text{fun pattern_match_list} \quad \text{(* Functional abstraction *)} \\
& \text{expr1 expr2} \quad \text{(* Functional application *)} \\
& \text{let identifier = expr1 in expr2} \quad \text{(* Local scoping *)} \\
& \{ expr \} \quad \text{(* Singleton *)} \\
& \{ expr : qualifier_list \} \quad \text{(* Set comprehension *)}
\end{align*}
\]

\[
\text{atomic_expr ::=}
\begin{align*}
& \text{constant} \quad \text{(* Literal value *)} \\
& \text{identifier} \quad \text{(* Variables *)} \\
& \text{( expr_list )} \quad \text{(* Tuples *)} \\
& \text{constructor atomic_expr} \quad \text{(* Constructor application *)}
\end{align*}
\]

\[
\text{constant ::=}
\begin{align*}
& \text{true | false | 'identifier | integer | tphi}
\end{align*}
\]

\[
\text{unary_operator ::=}
\begin{align*}
& \text{not | - | fst | snd | isphi?}
\end{align*}
\]

\[
\text{binary_operator ::=}
\begin{align*}
& \text{and | or | = | < | <= | > | >=} \\
& \text{| + | - | * | / | mod} \quad \text{(* Set union *)}
\end{align*}
\]

\[
\text{expr_list ::=}
\begin{align*}
& \text{expr | expr , expr_list}
\end{align*}
\]

\[
\text{qualifier_list ::=}
\begin{align*}
& \text{qualifier} \\
& \text{qualifier , qualifier_list}
\end{align*}
\]

\[
\text{qualifier ::=}
\begin{align*}
& \text{identifier in set} \quad \text{(* Set membership *)} \\
& \text{expr} \quad \text{(* Guard, condition *)}
\end{align*}
\]

\[
\text{set ::=}
\begin{align*}
& \text{base_type} \quad \text{(* bool, atom, or int *)} \\
& \text{identifier} \quad \text{(* User defined type *)} \\
& \text{expr} \quad \text{(* Any set *)}
\end{align*}
\]

\[
\text{pattern_match_list ::=}
\begin{align*}
& \text{pattern_match} \\
& \text{pattern_match | pattern_match_list}
\end{align*}
\]

\[
\text{pattern_match ::=}
\begin{align*}
& \text{pattern -> expr} \quad \text{(* Nu abstraction *)}
\end{align*}
\]

\[
\text{pattern ::=}
\begin{align*}
& \text{constant} \quad \text{(* Constant pattern *)} \\
& \text{identifier} \quad \text{(* Variable pattern *)} \\
& \text{constructor expr} \quad \text{(* Constructor pattern *)} \\
& \text{( pattern1 , pattern2 )} \quad \text{(* Pair pattern *)}
\end{align*}
\]
\begin{verbatim}
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
\end{verbatim}

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: \( expr_1 \cup expr_2 \). 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:

\[ D = B + A + Z + D \times D + D \rightarrow D + P(D) + T_1 + \ldots + T_n \]

Where each \( T_i \) is a user-defined type. There are two forms of user-defined types: product types (tuples), and sum types (variants): A type defined as \( type \ t = t_1 * t_2 * \ldots * t_n \) stands for the type of tuples whose i-th value is of type \( t_i \). A type defined as \( type \ t = C_1 t_1 | C_2 t_2 | \ldots | C_n t_n \) represents the disjoint union of types \( t_1, t_2, \ldots, t_n \). Note that each alternative is distinguished by a data constructor.

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

\begin{verbatim}
(* Type definition *)
type list = Nil | Cons int * list;

(* Empty list? *)
\end{verbatim}
define isNil as
  fun Nil -> True
  | Cons(y, l1) -> False

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

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 = y 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 $E$ applied to an EPowerFuL expression and an environment returns a value in domain $D$. The following notation is used: $p$ is an environment whose signature is $\rho : \text{Identifiers} \rightarrow D$. $\rho(id)$ is the value associated with the identifier $id$ in the environment $\rho$. $\rho[d/id]$ 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 lists the semantic equations.

3.2.1 Pattern matching

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

$P(\text{arg}, [], \rho) = \bot$

$P(\text{arg}, [p => e|\text{rest}], \rho) =$

Let $r = \text{M(\text{arg}, [p|e|\text{rest}], \rho)}$ in($\text{Left}(r)$, $\text{Right}(r)$, $P(\text{arg}, [\text{rest}], \rho)$)

Function $P$ uses function $\text{M}$ 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 $\text{True}$ if the match succeeded and $\text{False}$ 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 $k$: 632
\[ \rho = \text{constant} \]
\[ \rho(\text{identifier}) = \text{constant} \]
\[ \text{where } \text{C is a data constructor of a sum type.} \]
\[ \text{variables } \text{C} \text{C} \text{p} = \text{C(nexpr) p} \]
\[ \text{if } \text{CnstructApp(C, |expr| p) is defined} \]
\[ \text{otherwise } \text{Variable}(\text{identifier}) \]
\[ \text{E[ConstrctApp(C, |expr| p) = CnstrctApp(C, |expr| p) vVhere } \]
\[ \text{C is a data constructor of a sum type.} \]
\[ \text{E[UnaryOp(e |expr| p) = UnaryOp(E[ |expr| p) vVhere } \]
\[ \text{unaryop } \text{E} \{\text{not, -}, \text{fst, snd, isphi?}\} \]
\[ \text{and UnaryOp } \text{E} \{\text{Not, Neg, Left, Right, IsPhi}\}. \]
\[ \text{The correspondence between unaryop and UnaryOp should be obvious.} \]
\[ \text{E[BinaryOp(e |expr| p, e |expr| p) = BinaryOp(E[ |expr| p, E[ |expr| p) vVhere } \]
\[ \text{binaryop } \text{E} \{\text{and, or, <, <=, >, >=, +, -, *}, /, \text{mod, U}\} \]
\[ \text{and BinaryOp } \text{E} \{\text{And, Or, Lt, Lte, Gt, Gte, Plus, Minus, Times, Div, Mod, Union}\}. \]
\[ \text{The correspondence between binaryop and BinaryOp should be obvious} \]
\[ \text{E[DeqD(T} |expr| , E[ |expr| p, E[ |expr| p) = DeqD(T[ |expr| , E[ |expr| p, E[ |expr| p) vVhere } \]
\[ \text{T} \text{is the function that returns the type of a given expression.} \]
\[ \text{DeqD, is a function defined in section 3.2.2} \]
\[ \text{E[If |expr| then |expr| else |expr| p = IF(E[ |expr| p, E[ |expr| p, E[ |expr| p) vVhere } \]
\[ \text{L is of the form } p_1 > e_1 | p_2 > e_2 | \ldots | p_n > e_n, \text{in which each } p_i \text{ is a pattern and each } e_i \text{ is an expression.} \]
\[ \text{d is a new identifier. P is the function that defines the semantics of pattern matching and is defined in section 3.2.1} \]

\[ \text{Figure 1: Denotational Equations} \]

\[ \text{M(arg, [k], [e], p) =} \]
\[ \text{If(DeqD(T[k], arg, k), Pair(True, E[e] p), Pair(False, 1))} \]

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

\[ \text{M(arg, [x], [e], p) = E[e] p[ arg/x]} \]

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

\[ \text{M(arg, [K p], [e], p) = If(CheckCons(K, arg), M(Retrxnstr(arg), [p], [e], p), Pair(False, 1))} \]

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

\[ \text{M(arg, [(p_1, p_2)], [e], p) =} \]
\[ \text{Let } a = \text{Left}(arg) \text{ in} \]
\[ \text{Let } b = \text{Right}(arg) \text{ in} \]
\[ \text{If(And(N(a, p_1), N(b, p_2)),} \]
\[ \text{Pair(True, (E[fun p_1 > (fun p_2 > e)] p) ab),} \]
\[ \text{Pair(False, 1))} \]
\[ \text{N(arg, [k]) = DeqD(T[k], arg, k) } \]
\[ \text{N(arg, [e]) = True} \]
\[ \text{N(arg, [K p]) = If(CheckCons(K, arg), N(Retrxnstr(K, arg), [p], False)} \]
\[ \text{N(arg, [(p_1, p_2)]) = And(N(Left(arg), p_1), N(Right(arg), p_2))} \]

In case 4, \( M \) relies on another semantic function called \( N \) that also performs pattern matching but without making any bindings or evaluating expressions. After checking the correspondence, \( M \) 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:

\[
\text{DeqD}(t, x, y) = \text{TeqT}(x, y)
\]

where \( t \in \{ \text{bool}, \text{atom}, \text{int} \} \) and \( T \in \{ B, A, Z \} \)

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

\[
\text{DeqD}(t_1 \times t_2, x, y) =
\]

\[
\text{And(DeqD}(t_1, \text{Left}(x), \text{Left}(y)), \text{DeqD}(t_2, \text{Right}(x), \text{Right}(y)))
\]

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

\[
\text{DeqD}([], x, y) = \bot
\]

\[
\text{DeqD}(\text{C}[t], x, y) =
\]

\[
\text{If}(\text{CheckCons}(C, x),
\text{If}(\text{CheckCons}(C, y), \text{DeqD}(t, \text{RetrCnstr}(C, x), \text{RetrCnstr}(C, y)), \text{False}, \text{DeqD}(L, x, y))
\]

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, IFs, 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: \( \text{type } T = C_1 T_1 | ... | C_n T_n \). We define:

\[
\text{ConstructorTag}(T) = \{ C_1, C_2, ..., C_n \}
\]

\[
\text{CnstrctApp} : \text{ConstructorTag}(T) \times T_i \rightarrow T
\]

A constructor can be seen as a function of type: \( T_i \rightarrow T \).

CheckCons : ConstructorTag(T) \times D \rightarrow B

CheckCons(C, CnstrctApp(Ci, e)) = True

CheckCons(C, CnstrctApp(Ki, e)) = False \hspace{1em} \text{where } K_i \neq C_i

CheckCons(C, e) = \bot \hspace{1em} \text{where } e \text{ is not a constructor application}

RetrCnstr : ConstructorTag(T) \times D \rightarrow D

RetrCnstr(C, CnstrctApp(Ci, e)) = e

RetrCnstr(C, e) = \bot

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 "\( \cup \)" instead of Union(",") to compact notation where warranted). IFs, is defined as If except that if its argument is undefined, it returns Phi instead of bottom (\( \bot \)). 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.
\[ \text{IsPhi} : \mathcal{P}(D) \rightarrow B \]
\[ \text{IsPhi}(\Phi) = \text{True} \]
\[ \text{IsPhi}(\text{Singleton}(\text{expr})) = \text{False} \]
\[ \text{IsPhi}(\text{Union}(\text{expr}_1, \text{expr}_2)) = \text{And}(\text{IsPhi}(\text{expr}_1), \text{IsPhi}(\text{expr}_2)) \]
\[ \text{IsPhi}(\bot) = \bot \]

\[ \text{IFs} : \mathcal{B} \times \mathcal{P}(D) \times \mathcal{P}(D) \rightarrow \mathcal{P}(D) \]
\[ \text{IFs}(\text{True}, x, y) = x \]
\[ \text{IFs}(\text{False}, x, y) = y \]
\[ \text{IFs}(\bot, x, y) = \Phi \]

\[ \text{AppS} : (D \rightarrow \mathcal{P}(D)) \times \mathcal{P}(D) \rightarrow \mathcal{P}(D) \]
\[ \text{AppS}(f, \Phi) = \Phi \]
\[ \text{AppS}(f, \text{Singleton}(\text{expr})) = f(\text{expr}) \]
\[ \text{AppS}(f, \text{Union}(\text{sl}, \text{s2})) = \text{Union}(\text{AppS}(f, \text{sl}), \text{AppS}(f, \text{s2})) \]

### 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 \text{AppS}(\lambda x.\text{body}, \text{generator}), \text{body} should be applied to each element in \text{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.

\[ \varepsilon[[\{\text{expr} : \text{id} \in \text{type.\ qualifierlist}\}]]_{\rho} = \text{type}(\text{d}); \varepsilon[[\{\text{expr} : \text{qualifierlist}\}]]_{\rho} \]

Where type is either a base type (i.e. \text{bool}, \text{atom}, or \text{int}), or a user defined type.

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

\[ t(d)\text{.body} \]

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

\[ \text{AppS}(\lambda d.\text{body}, t). \]

Instead, \text{body} is evaluated in its parametrized form. This simplification is known as \textit{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:

\[ \text{constraint}(u).\ldots\text{prim}(u)\ldots. \]

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

\[ \text{bool}(u).\ldots\text{Not}(u)\ldots. \]

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

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

Another example: consider the expression

\[ \{e : x \in \text{atom}, y \in \text{atom}, \ldots x = y\ldots\}. \]

This would be ultimately translated into something like

\[ \text{atom}(x)\text{.atom}(y).\ldots\text{AeqA}(x, y)\ldots\]
that is, a primitive applied to two logical variables, and after narrowing would yield:

\[
\text{atom}(x).(...) [x/y] \cup \text{atom}(x).\text{atom}(y).(x \neq y).(...) [z/y].
\]

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:

\[
\text{atom}(x).(...) \text{AeqA}(x, a) = (...) \text{True}/a \cup \text{atom}(x). (x \neq a). (...) \text{False}/a.
\]

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.

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

The conditional can also be applied to a logic variable:

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

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

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

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 \( C_1 t_1 | C_2 t_2 | ... | C_n t_n \) then

\[
t(u).(...) \text{prim}(u) ... = \\
t_1(v_1).(...) \text{prim}(u) ... \text{CnstrctApp}(C_1, v_1)/u \\
\cup ... \cup t_n(v_n).(...) \text{prim}(u) ... \text{CnstrctApp}(C_n, v_n)/u
\]

Where \( v_1, v_2, ..., v_n \) 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:

\[
t(u).(...) \text{prim}(u) ... = \\
... \cup (...) \text{prim}(u) ... \text{CnstrctApp}(C_i, \text{Nil})/u \cup ...
\]

This leads to the following rules:

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

\[
t(u).(...) \text{Left}(u) ... = \\
t_1(v).t_2(w).(...) \text{Pair}(v, w)/u
\]

\[
t(u).(...) \text{Right}(u) ... = \\
t_1(v).t_2(w).(...) \text{Pair}(v, w)/u
\]
• Given type \( t \) of the form

\[
C_1 \ t_1 | C_2 \ t_2 | \ldots | C_n \ t_n
\]

\[
t(u). (\ldots \text{RetrCnstr}(C_i, u) \ldots ) = \\
t_1(v_1). (\ldots | \text{CnstrctApp}(C_1, v_1)/u) \\
\cup t_2(v_2). (\ldots | \text{CnstrctApp}(C_2, v_2)/u) \\
\ldots \\
\cup t_i(v_i). (\ldots | \text{CnstrctApp}(C_i, v_i)/u) \\
\ldots \\
\cup t_n(v_n). (\ldots | \text{CnstrctApp}(C_n, v_n)/u)
\]

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

```haskell
type t = int•bool;
{e: x in t, snd x }
```

The expression's meaning is of the form

\[
t(u). (\ldots \text{Right}(u) \ldots ).
\]

This should narrow as:

\[
\text{int}(v). \text{bool}(w). (\ldots u \ldots )| \text{Pair}(v, w)/u).
\]

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

Another example: if we have:

```haskell
type t = None | Some int;
{e: x in t, x=Some 2}
```

will be translated into something like

\[
t(u). (\ldots \text{CheckCons}(\text{Some}, u) \ldots ).
\]

This can be simplified as

\[
(\ldots \text{False} \ldots )| \text{CnstrctApp}(\text{None}, \text{Nil})/\cup \\
\text{int}(v). (\ldots \text{True} \ldots )| \text{CnstrctApp}(\text{Some}, v)/\cup.
\]

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:

\[
\text{(Value, stack, choice – point – stack, heap)}
\]

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 CAM's denotational domain, are basic values in the modified CAM. We add a constructor to represent pairs: Pair(., .) explicitly.
A new constructor is added to represent singleton sets: `single(.)`. An expression of the form:

```
expr1 ∪ expr2
```

can be interpreted as `expr1` or `expr2`. 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).Exp.
```

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: `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 `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...
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 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 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 =/= 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 point 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.

<table>
<thead>
<tr>
<th>OLD STATE</th>
<th>NEW STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>V</td>
</tr>
<tr>
<td>br(C1,C2);C (true,V)</td>
<td>S hp</td>
</tr>
<tr>
<td>br(C1,C2);C (false,V)</td>
<td>S hp</td>
</tr>
<tr>
<td>br(C1,C2);C (LV(X),V)</td>
<td>S hp</td>
</tr>
<tr>
<td>not:X</td>
<td>true</td>
</tr>
<tr>
<td>not:X</td>
<td>false</td>
</tr>
<tr>
<td>not:X</td>
<td>(LV(X),V)</td>
</tr>
</tbody>
</table>

C[A]_p = quote(A_p)
C[λx . E]_p = cur(C[E]_(p,x))
C[(Expr1 . Expr2)]_p =
  push; free(C[Expr1]_p); swap;
  free(C[Expr2]_p); cons
C[Expr . Expr2]_p = push; C[Expr]_p; swap;
  free(C[Expr2]_p); cons; app

639
Where \( a \) is an atom

\[
\text{IsA}(a); \text{LV}(x) \quad \text{IsA}(a); \text{LV}(x) \\
\text{if comp}(X, Y) = \text{true} \quad \text{false} \quad \text{true} \quad \text{true} \quad \text{false} \quad \text{true} \\
\text{if comp}(X, Y) = \text{false} \quad \text{false} \quad \text{true} \quad \text{true} \quad \text{false} \quad \text{true} \\
\text{if comp}(X, Y) = \text{unknown} \quad \text{false} \quad \text{true} \quad \text{true} \quad \text{false} \quad \text{true} \\
\]

\[
C = \text{true} \quad S = \text{hp} _1 \quad \text{cp} = \text{CS} \\
C = \text{false} \quad S = \text{hp} _1 \quad \text{cp} = \text{CS} \\
C = \text{true} \quad S = \text{hp}[a/X] \quad \text{cp} = \text{CS} \\
\]

Figure 5: Transition Tables for AeqA with Logical Variables

Where \( \text{hp} _1 = \text{new}(t _1, y, x) \odot \text{new}(t _2, z, x) \odot \text{hp}[\text{Pair}(y, z)/x] \)

Where \( x \) is of type \( t _1 \ast t _2 \)

\[
\text{old state} \quad \text{new state} \\
\text{retre}(K) ; \text{eon}(K) \quad \text{retre}(K) ; \text{eon}(K) \\
\text{where } K \text{ is not of the form } \text{eon}(, \) \\
\text{where } e \text{ is of type } t = C T 1 \mid \ldots \mid C n T n \\
\]

Where \( \text{CS} ' = \) \( \text{cp}(C, z _2, S, h p _2) \) ::

... 

\( \text{cp}(C, z _n, S, h p _n) \)

and \( x _i = \text{if } K = C, \text{then } \text{LogVar}(u _i) \) else .bot 

and \( \text{hp} _i = \text{new}(t _i, u _i, x) \odot \text{hp}[\text{ConsApp}(C _i, u _i)/x] \)

Figure 6: Narrowing for User-defined types
\[ C[\text{ConstructorApp}(K, \text{Expr})]_s = \]
\[ \text{free}(C[\text{Expr}]_s); \]
\[ \text{mkCnstr}(K) \]
\[ C[\text{Left}(\text{Expr})]_s = C[\text{Expr}]_s; \text{unf}; \text{fst} \]
\[ C[\text{RetrCnstr}(K, \text{Expr})]_s = C[\text{Expr}]_s; \text{unf}; \text{retrC} \]
\[ C[\text{type}(x).\text{Expr}]_s = \text{curv}(\text{type}, C[\text{Expr}]_{(s,x)}; \text{unf}) \]

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 OCAMAL.

References


