Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Epic 1.0 (unconditional) an equational programming language

H.R. Walters and J.F.Th. Kamperman

Computer Science/Department of Software Technology

CS-R9604 1996

Report CS-R9604
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

EPIC 1.0 (unconditional)

An Equational Programming Language

H.R.Walters   J.F.Th.Kamperman

{H.R.Walters,J.Kamperman}@cwi.nl

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

We present EPIC, an equational programming language: its abstract syntax, static and operational semantics, and one of many possible concrete grammars of unconditional EPIC.

CR Subject Classification (1991): D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages, *Algebraic approaches to semantics*; F.4.1 [Mathematical Logic and Formal Languages]: Logic Programming.

AMS Subject Classification (1991): 68N17: Logic Programming, 68Q40: Symbolic computation, 68Q42: Rewriting Systems and 68Q65: Algebraic specification.

Keywords & Phrases: declarative languages, term rewriting, specification languages, formal semantics.

Note: Partial support received from the Foundation for Computer Science Research in the Netherlands (SION) under project 612-17-418, "Generic Tools for Program Analysis and Optimization".

1. INTRODUCTION

Equational programming is the use of (confluent) term rewriting systems as a programming language with don't care non-determinism [MOI95], against a formal background of algebraic specification with term rewriting as a concrete model.

The phrase 'equational programming' was used in the mid-eighties (*cf.* [O'D85, DP86]) to refer to programming based on equations and equational logic. The name has never caught on, probably because the implementations of the time were suitable only to study equational specifications; not to support large scale programming.

Since then, the quality of implementations has increased to such an extent that in many circumstances there is now a real choice between a general purpose language and an implementable specification language: the speed that can be attained using the general purpose language must be weighed against the speed with which an executable specification can be developed.

In order to have an implementable, sufficiently efficient specification language, concessions must be made with respect to expressive power and (operational) semantics: we restrict ourselves to term models and to rewrite systems which must be complete for many results (in order to have don't-care non-determinism)

EPIC is an equational programming language primarily developed as a 'formal system programming language'. That is, it is strongly based on equational specification and term rewriting, but its operational semantics are too specific for a specification language.

EPIC has two main applications:

- It can be used as a 'systems programming language' to write executable specifications in. For example, EPIC's compiler, and several other tools for EPIC, have been implemented in EPIC itself;
- It can be used as a target language, where other specification languages are given an implementation by translating them to EPIC. EPIC is a suitable target for many languages based on pattern matching, tree- (dag-) replacement and term rewriting since it provides precisely the needed primitives, without superfluous detail.

  Historically, EPIC has evolved in the context of ASF+SDF [BHK89]: an algebraic specification and syntax definition formalism which provides algebraic specifications over signatures with user definable syntax. ASF+SDF specifications can be implemented by translating them to EPIC.

For these reasons EPIC's syntax is intentionally abstract: when used as a target language, generating the abstract syntax directly (as a data structure, or in a simple textual format) avoids producing and parsing the concrete text; and when used as a system programming language, a concrete syntax must be available, but can be austere. The EPIC tool set – a collection of software for the support of EPIC, containing, among others, tools constituting the compiler and run-time system, – uses a front-end (written in EPIC) which accepts such an austere syntax and produces EPIC abstract syntax.

Similarly, EPIC's type-system is trivial: it is single-sorted, requiring only the usual restrictions for TRSs (left-hand side of a rule is not a sole variable; all arities coincide; and a variable must be instantiated – in the *lhs* – before it is used), and some concerning modules (free and external functions may not become defined). EPIC's tool set contains a type-checker (incorporated in the compiler) which verifies these requirements.

1.1 EPIC in a nutshell

EPIC features rewrite rules with syntactic specificity ordering [WK95a] (a simplified version of specificity ordering [BBKW89]). It supports external datatypes and separate compilation of modules.

An EPIC module consists of a signature and a set of rules. The signature declares functions, each with an arity (number of arguments). In addition, functions can be declared *external* (i.e., defined in another module, or directly in C), or *free* (i.e., not defined in any module).

The rules are left-linear rewrite rules.

Rules are partially ordered by a syntactic specificity ordering: a more specific rule has higher precedence than a more general rule. When applicable rules are not ordered by syntactic specificity, the choice which rule to apply is free. This makes EPIC a nondeterministic language. In contrast to languages with *don't know nondeterminism* (i.e., the implementation is required to explore all choices) such as Prolog, EPIC is a language with *don't care nondeterminism* (i.e., the programs should be written in such a way that the choice does not matter).

EPIC assumes (rightmost) innermost rewriting; in [KW95] a method is described which makes lazy (outermost) rewriting available by TRS transformation. This method will be added to EPIC in the future.

In [WK95b] a model for I/O in term rewriting systesm is presented, which will be added to EPIC in the future. In [Wal90] so-called hybrid datatypes are introduced as a mechanism to combine, transparently, TRSs with abstract datatypes implemented in any fashion.

1.2 System design filosofy

The development of EPIC and its supporting tools is fueled by our conviction that term rewriting isn't less efficient, intrinsically, than any other implementation mechanism.

Accordingly, all tools relating to EPIC are themselves TRSs written in EPIC; the single exception is the run-time system, which is the abstract rewriting machine μArm discussed in Section 2.

All tools in the EPIC tool set are based on a simple design principle: they consume and produce text. They are usually composed of four parts: a parser, which interprets the input text and builds the term it represents; the essential computation performed by the tool; a (pretty) printer which produces a text given the term resulting from the computation; and a 'top module' which glues the three together.

Clearly intermediate printing and parsing is avoided when tools are combined. Also, a graph exchange language [Kam94] can be used to store or pass on, in a very compact form approaching one byte per node, terms, dags and graphs, where sharing should be preserved.

1.3 A brief overview

Full EPIC features conditional rewrite rules [Klo92] with specificity ordering [KW95]. It supports external datatypes and separate compilation of modules. In this document we only consider unconditional EPIC: rewrite rules are left-linear and unconditional.

An EPIC module consists of a set of types (the signature) and a set of rules. The types declare functions, each with an arity (number of arguments). In addition, functions can be declared *external* (i.e., defined in another module, or directly in C), or *free* (i.e., not defined in any module).

The rules are left-linear pattern-replacement (i.e., rewrite) rules.

Rules are ordered by a syntactic specificity ordering: a more specific rule has higher precedence than a more general rule.

1.4 An Example

As mentioned, the concrete syntax of EPIC is not very relevant. In the sequel we will define one concrete syntax (which is the one we use), but we do not propose that syntax to be 'the' concrete syntax of EPIC; it has none. To provide a first taste of EPIC, however, concrete syntax must be used. This example is intended to illustrate the expressive power of EPIC, and of tool-building with EPIC.

For clarity, we refer to the current version of this concrete language as $\text{EPIC}^{\text{C}}_{1.0}$. $\text{EPIC}^{\text{C}}_{1.0}$ is naively simple in features traditionally considered useful in programming languages or specification languages. Most notably, $\text{EPIC}^{\text{C}}_{1.0}$ is single-sorted, although its syntax allows the expression of argument and result sorts; these are intended for program documentation only, and are not enforced.

Note that EPIC itself is purposefully single-sorted: it is always assumed that typechecking occurs at source-level (if EPIC is a target), or by a separate tool (if EPIC is used for system programming). Operationally, sorts play no role.

The example below defines a simple calculator for binary numbers.

```
module bin-calc
types
   calc: Text   -> Text;
   parse: Text  -> Nat {external};
   print: Num   -> Text {external};
rules
   calc(Txt) = print(parse(Txt));

module io
types
   \n: -> Char;    ’ : -> Char;   ’(: -> Char;   ’): -> Char;
   ’*: -> Char;   ’+: -> Char;   ’0: -> Char;   ’1: -> Char;
   jxt: Nat # Nat        -> Nat {external};
   o:                    -> Nat {external};
   i:                    -> Nat {external};
   plus: Nat # Nat       -> Nat {external};
   times: Nat # Nat      -> Nat {external};
   eos                   -> Text {free};
   str: Char # Text      -> Text {free};
   cat: Text # Text      -> Text {free};
   parse: Text           -> Nat;
   get-val: Tuple        -> Text;
   enc-exp: Tuple        -> Text;
   aft-exp: Num # Text   -> Tuple;
   plus-exp: Num # Tuple -> Tuple;
   mul-exp: Num # Tuple  -> Tuple;
   nb: Text              -> Text;
   parse-num: Text # Nat -> Tuple;
   parse-exp: Text       -> Tuple;
   trail: Text # Nat     -> Tuple;
   tuple: Nat # Text     -> Tuple  {free};
   print: Num            -> Text;
rules
   parse(Txt) = get-val(parse-exp(nb(Txt)));
   get-val(tuple(Val,Rest)) = Val;
   parse-exp(‘(‘+Txt) = enc-exp(parse-exp(nb(Txt)));
   enc-exp(tuple(Val,Rest)) = aft-exp(Val,nb(Rest));
   aft-exp(Val,‘)‘+Rest) = trail(nb(Rest),Val);
   parse-exp(Txt) = parse-num(Txt,o);
   parse-num(‘0‘+Txt,Val) = parse-num(Txt,plus(Val,Val));
   parse-num(‘1‘+Txt,Val) = parse-num(Txt,plus(plus(Val,Val),i));
   parse-num(Txt,Val) = trail(Txt,Val);
   trail(‘+‘+Txt,Val1) = plus-exp(Val1,parse-exp(Txt));
   plus-exp(Val1,tuple(Val2,Rest)) = tuple(plus(Val1,Val2),Rest);
   trail(‘*‘+Txt,Val1) = mul-exp(Val1,parse-exp(Txt));
   mul-exp(Val1,tuple(Val2,Rest)) = tuple(times(Val1,Val2),Rest);
   trail(Txt,Val) = tuple(Val,Txt);
   nb(‘\n‘+Txt) = Txt ;
   nb(‘ ‘+Txt) = Txt ;
   nb(Txt) = Txt;
```

```
      print(jxt(A,B)) = cat(print(A),print(B));
      print(o) = ‘0‘;
      print(i) = ‘1‘;

   module numbers
   types
      o:                -> Nat;
      i:                -> Nat;
      jxt: Nat # Nat    -> Nat;
      plus: Nat # Nat   -> Nat;
      times: Nat # Nat -> Nat;
   rules
      jxt(o,X) = X;
      jxt(X,jxt(Y,Z)) = jxt(plus(X,Y),Z);
      plus(o,X) = X;   plus(i,o) = i ;
      plus(i,i) = jxt(i,o);
      plus(i,jxt(X,Y)) = jxt(X,plus(i,Y));
      plus(jxt(X,Y),Z) = jxt(X,plus(Y,Z));
      times(o,X) = o ;   times(i,X) = X;
      times(jxt(X,Y),Z) = jxt(times(X,Z),times(Y,Z));
```

2. ABSTRACT SYNTAX

The *abstract* syntax of EPIC defines the essential structural information, void of representational aspects. We define the abstract syntax as an abstract datatype: a collection of sorts (corresponding to all distinct notions) and functions (the information that can be retrieved from those notions), and a number of additional properties applicable models should exhibit. This leaves the abstract syntax underspecified: even the signature is only partly given. In Section 6 we present one particular term algebra which is an instance of EPIC's abstract syntax.

There are several reasons for this approach:

- In this manner the syntax is truly abstract: essential aspects are defined, and all irrelevant detail is avoided.
- EPIC is partly an intermediate language. Its major source of input are machine interfaces rather than humans. Whereas humans are text oriented, machine interfaces prefer structured information.
- This approach is more flexible (compared to the traditional approach of defining a graph/tree language as an abstract syntax) w.r.t. future modifications to EPIC.

In this document we indicate specification segments with bars to their left: a single bar signifies syntax (sorts and functions); a double bar signifies semantical information.

`Prog` — An Epic program
`Mod` — An Epic module
`Type` — The type of a function
`Rule` — A rewrite rule
`Term` — A term
$\texttt{Indx}_{\texttt{m}}$,$\texttt{Indx}_{\texttt{f}}$,$\texttt{Indx}_{\texttt{r}}$,$\texttt{Indx}_{\texttt{t}}$, — Indices *(i)*
`Name` — Name
`Number` — Numbers *(ii)*

Notes:

(i): Indices are an abstraction to provide sub-structure selection. The mechanism we define is somewhat abtruse, for the following reason. It models the three most commonly used (different) mechanisms: global, inductively ordered indices (e.g., the natural numbers); context-dependent ordered indices (e.g., field-names); and indices derived from structure (e.g., recursive lists).

To be precise:

- if structures are represented as arrays, then an index is a tuple of such an array and a natural number (i.e., $\langle x, \alpha \rangle$)), the indicated sub-structure is $x[\alpha]$, and the next index is $\langle x, \alpha + 1 \rangle$;
- if lisp-like lists are used for index (and structure), an index would be a `cons`, the indicated sub-structure its `car`, and the next index its `cdr`;
- if field-names and records are used, then an index is a tuple of a record and a field-name ($\langle x, \alpha \rangle$), the sub-structure is $x.\alpha$, and the next index is $\langle x, \texttt{nxt\_fld}(\texttt{tp}(x), \alpha) \rangle$, where `nxt_fld` maps the type of a structure and a field name to the next field name in that type.

(ii): We use `Number` to designate the arity of functions. `Number` need not be the set of natural numbers $\mathbb{N}$ (which is infinite), although, in practice, sufficiently many distinct numbers should exist.

In the remainder of this paper, all formulae are (implicitly) universally quantified (unless otherwise indicated), where the name of variables (possibly with subscript) indicates their range: p for `Prog`, m for `Mod`, f for `Type`(f for function-type), r for `Rule`, t for `Term`, n for `Name`, α for `Number` and i for `Indx` (and, for example, i^t for $\texttt{Indx}_{\texttt{t}}$).

We introduce various auxiliary sorts and overloaded functions in order to reduce the total number of (overloaded) functions and equations, or to reduce trivial conditions. The meaning of a formula is the set of instances that are well-typed using base (i.e., non auxiliary) sorts. We do not consider sub-sorts.

Predicates are logical value (boolean) valued, total functions. Their use in a condition or consequence signifies truth; their negation (e.g., $\neg\texttt{is\_var}(\texttt{lhs}(r))$, or $t \notin r$) signifies falsehood. We assume and use some degree of initiality for predicates: if the value of a predicate isn't defined to be true, then it is taken to be false.

We use the notation $\langle \ldots \rangle$ for tuples (i.e., members of cartesian products). For example, if a and b are of sort A and B, respectively, then $\langle a, b \rangle$ is of sort $A\#B$.

Finally, we take recursively enumerable sets to be a primitive.
Let $\mathtt{Indx} = \mathtt{Indx}_{\mathtt{m}} \cup \mathtt{Indx}_{\mathtt{s}} \cup \mathtt{Indx}_{\mathtt{r}} \cup \mathtt{Indx}_{\mathtt{t}}$ be the sort of all indices.

`mods:`	`Prog`	*predicate*	Predicate expressing if program has (any) modules
$\mathtt{subs}_{\mathtt{m}}$:	`Prog`	-> $\mathtt{Indx}_{\mathtt{m}}$	The first index of a module in the program
`at:`	$\mathtt{Indx}_{\mathtt{m}}$	-> `Mod`	Access *(i)*
`adv:`	$\mathtt{Indx}_{\mathtt{m}}$	-> $\mathtt{Indx}_{\mathtt{m}}$	Advancement
`funs:`	`Mod`	*predicate*	Does module have functions
$\mathtt{subs}_{\mathtt{f}}$:	`Mod`	-> $\mathtt{Indx}_{\mathtt{f}}$	The first index of a function in the module
`at:`	$\mathtt{Indx}_{\mathtt{f}}$	-> `Type`	Access
`adv:`	$\mathtt{Indx}_{\mathtt{f}}$	-> $\mathtt{Indx}_{\mathtt{f}}$	Advancement
`rules:`	`Mod`	*predicate*	Does module have rules
$\mathtt{subs}_{\mathtt{r}}$:	`Mod`	-> $\mathtt{Indx}_{\mathtt{r}}$	The first index of a rule in the module
`at:`	$\mathtt{Indx}_{\mathtt{r}}$	-> `Rule`	Access
`adv:`	$\mathtt{Indx}_{\mathtt{r}}$	-> $\mathtt{Indx}_{\mathtt{r}}$	Advancement
`name:`	`Type`	-> `Id`	the name of a function
`arity:`	`Type`	-> `Number`	The number of arguments a function takes *(ii)*
`external:`	`Type`	*predicate*	Is the function external
`free:`	`Type`	*predicate*	Is the function (globally) free
`lhs:`	`Rule`	-> `Term`	The *lhs* of the rule
`rhs:`	`Rule`	-> `Term`	The *rhs*
`ofs:`	`Term`	-> `Id`	The outermost function symbol
`sub-terms:`	`Term`	*predicate*	Does Term have sub-terms
$\mathtt{subs}_{\mathtt{t}}$:	`Term`	-> $\mathtt{Indx}_{\mathtt{t}}$	The first index of a sub-term of the term
`at:`	$\mathtt{Indx}_{\mathtt{t}}$	-> `Term`	Access
`adv:`	$\mathtt{Indx}_{\mathtt{t}}$	-> $\mathtt{Indx}_{\mathtt{t}}$	Advancement
`is_var:`	`Term`	*predicate*	Is the term a variable
`last:`	`Indx`	*predicate*	is this the last index (or can it be advanced)
0:		-> `Number`	The number zero
_+1:	`Number`	-> `Number`	Successor function

Domains
We do not require all functions to be total, but sub-structure selection should be sufficiently defined as required below. Let $dom(\mathtt{adv})$ denote the union of the domains of all functions `adv`.

$\mathtt{mods}(p) \Longrightarrow p \in dom(\mathtt{subs}_{\mathtt{m}})$
$\mathtt{funs}(m) \Longrightarrow m \in dom(\mathtt{subs}_{\mathtt{f}})$
$\mathtt{rules}(m) \Longrightarrow m \in dom(\mathtt{subs}_{\mathtt{r}})$
$\mathtt{is\_var}(t) \Longrightarrow t \notin dom(\mathtt{ofs}) \wedge t \notin dom(\mathtt{sub\text{-}terms}) \wedge t \notin dom(\mathtt{subs}_{\mathtt{t}})$
$\mathtt{sub\text{-}terms}(t) \Longrightarrow t \in dom(\mathtt{subs}_{\mathtt{t}})$
$\neg\mathtt{last}(i) \Longrightarrow i \in dom(\mathtt{adv})$

3. SEMANTICS

In order to define static and operational semantics, some auxiliary notions are needed, which we will first introduce.

Let $\texttt{Var} = \{t | \texttt{is\_var}(t)\}$ be the set of all variables, and let v, possibly with sub-script, range over $\texttt{Var}$.

Arity

$\texttt{arity:}\quad \texttt{Indx} \quad \texttt{->}\ \texttt{Number}$

$\texttt{last}(i) \Longrightarrow \texttt{arity}(i) = 0$

$\neg\texttt{last}(i) \Longrightarrow \texttt{arity}(i) = \texttt{arity}(\texttt{adv}(i)) + 1$

Containment

Let $\texttt{Mod}^I = \texttt{Mod} \cup \texttt{Indx}_\texttt{m}$, $\texttt{Type}^I = \texttt{Type} \cup \texttt{Indx}_\texttt{f}$, $\texttt{Rule}^I = \texttt{Rule} \cup \texttt{Indx}_\texttt{r}$ and $\texttt{Term}^I = \texttt{Term} \cup \texttt{Indx}_\texttt{t}$ be the union of structures and their indices; let $\texttt{Struct} = \texttt{Prog} \cup \texttt{Mod} \cup \texttt{Type} \cup \texttt{Rule} \cup \texttt{Term}$ be the set of all structures; and let $\texttt{Struct}^I = \texttt{Prog} \cup \texttt{Mod}^I \cup \texttt{Type}^I \cup \texttt{Rule}^I \cup \texttt{Term}^I$.

$\epsilon\texttt{:}\quad \texttt{Struct}^I\ \texttt{\#}\ \texttt{Struct}^I \quad \textit{predicate}$

$x_1 \mathrel{\epsilon} x_2 \wedge x_2 \mathrel{\epsilon} x_3 \Longrightarrow x_1 \mathrel{\epsilon} x_3$

$x \mathrel{\epsilon} x$

$\texttt{mods}(p) \Longrightarrow \texttt{subs}_\texttt{m}(p) \mathrel{\epsilon} p$

$\texttt{funs}(m) \Longrightarrow \texttt{subs}_\texttt{f}(m) \mathrel{\epsilon} m$

$\texttt{rules}(m) \Longrightarrow \texttt{subs}_\texttt{r}(m) \mathrel{\epsilon} m$

$\texttt{lhs}(r) \mathrel{\epsilon} r$

$\texttt{rhs}(r) \mathrel{\epsilon} r$

$\texttt{sub-terms}(t) \Longrightarrow \texttt{subs}_\texttt{t}(t) \mathrel{\epsilon} t$

$\texttt{at}(i) \mathrel{\epsilon} i$

$\neg\texttt{last}(i) \Longrightarrow \texttt{adv}(i) \mathrel{\epsilon} i$

Substitutions

Let $\texttt{Subst} = \mathcal{P}(\texttt{Var\#Term})$ be the set of variable-value pairs which homomorfically generate substitutions, and let σ, possibly with subscript, range over $\texttt{Subst}$.

$\_^{\_}\texttt{:}\quad \texttt{Term}^I\ \texttt{\#}\ \texttt{Subst} \quad \texttt{->}\ \texttt{Term}^I \quad (\text{e.g. } t^\sigma)$

$\langle v, t\rangle \in \sigma \Longrightarrow v^\sigma = t$

$\neg\texttt{is\_var}(t) \Longrightarrow \texttt{ofs}(t^\sigma) = \texttt{ofs}(t)$

$\texttt{sub-terms}(t) \Longrightarrow \texttt{sub-terms}(t^\sigma)$

$\texttt{sub-terms}(t) \Longrightarrow \texttt{subs}_\texttt{t}(t^\sigma) = \texttt{subs}_\texttt{t}(t)^\sigma$

$\texttt{last}(i) \Longrightarrow \texttt{last}(i^\sigma)$

$\texttt{at}(i^\sigma) = \texttt{at}(i)^\sigma$

$\neg\texttt{last}(i) \Longrightarrow \texttt{adv}(i^\sigma) = \texttt{adv}(i)^\sigma$

Contexts

Containment can not be used to express the *position* of sub-terms, as is required in the sequel.

We use the slightly operational notion of *contexts* [Klo92] to express position. With contexts, one can use containment to reason about positions.

Intuitively, a context is a structure with a hole in it. We define contexts by extending the set of terms with the hole ($\Box$). Unlike [Klo92], we take $\Box$ to be a variable; this allows us to use substitution for context instantiation.

$$\begin{array}{l}
\Box\text{:}\quad \texttt{-> Term}\\
\quad \texttt{is\_var}(\Box)
\end{array}$$

Let `Context` be the set of rules and terms, and their indices, which contain exactly one occurrence of $\Box$. We forego the constructive definition of `Context`, which is trivial but tedious. Let γ, γ^t, γ^r and γ^i range over `Context`, $\texttt{Context}\cap\texttt{Term}$, $\texttt{Context}\cap\texttt{Rule}$ and $\texttt{Context}\cap\texttt{Indx}_\texttt{t}$, respectively.

Instantiation of a context coincides with substitution of the hole.

$$\begin{array}{l}
\_[\_]\text{:}\quad \texttt{Context \# Term ->Rule}\\
\qquad\;\; \texttt{Context \# Term ->Term}\\
\quad \texttt{lhs}(\gamma^r[t]) = \texttt{lhs}(\gamma^r)[t]\\
\quad \texttt{rhs}(\gamma^r[t]) = \texttt{rhs}(\gamma^r)[t]\\
\quad \gamma^t[t] = \gamma^{t\{\langle\Box,t\rangle\}}
\end{array}$$

Two contexts are compatible if they can be instantiated to the same

$$\begin{array}{l}
\sim\text{:}\quad \texttt{Context \# Context}\quad \textit{predicate}\\
\quad \gamma_1[t_1] = \gamma_2[t_2] \Longrightarrow \gamma_1 \sim \gamma_2
\end{array}$$

Pre-order: if two contexts are compatible, and $\Box$ occurs above or 'to the left' (picturing `adv` as movement to the right), then that context is smaller in pre-order.

$$\begin{array}{l}
<\text{:}\quad \texttt{Context \# Context}\quad \textit{predicate}\\
\quad \gamma_1 < \gamma_2 \wedge \gamma_2 < \gamma_3 \Longrightarrow \gamma_1 < \gamma_3\\
\quad \gamma_1[\gamma^t] = \gamma_2 \Longrightarrow \gamma_1 < \gamma_2\\
\quad {\gamma^r}_1 \sim {\gamma^r}_2 \wedge \Box \in \texttt{lhs}({\gamma^r}_1) \wedge \Box \in \texttt{rhs}({\gamma^r}_2) \Longrightarrow {\gamma^r}_1 < {\gamma^r}_2\\
\quad {\gamma^i}_1 \sim {\gamma^i}_2 \wedge \neg\texttt{last}({\gamma^i}_2) \wedge \Box \in \texttt{at}({\gamma^i}_1) \wedge \Box \in \texttt{adv}({\gamma^i}_2) \Longrightarrow {\gamma^i}_1 < {\gamma^i}_2
\end{array}$$

Matching

$$\begin{array}{l}
\texttt{matches}\text{:}\quad \texttt{Term}^I \;\#\; \texttt{Term}^I \quad \textit{predicate}\\
\texttt{match}\text{:}\quad\;\;\; \texttt{Term}^I \;\#\; \texttt{Term}^I \quad \texttt{-> Subst}\\
\quad \texttt{matches}(s, v)\\
\quad \neg\texttt{is\_var}(t_1) \wedge \neg\texttt{is\_var}(t_2) \wedge \texttt{ofs}(t_1) = \texttt{ofs}(t_2) \wedge \texttt{matches}(\texttt{subs}_\texttt{t}(t_1), \texttt{subs}_\texttt{t}(t_2))\\
\qquad\quad \Longrightarrow \texttt{matches}(t_1, t_2)\\
\quad \texttt{matches}(\texttt{at}(i_1), \texttt{at}(i_2)) \wedge ((\texttt{last}(i_1) \wedge \texttt{last}(i_2)) \vee \texttt{matches}(\texttt{adv}(i_1), \texttt{adv}(i_2)))\\
\qquad\quad \Longrightarrow \texttt{matches}(i_1, i_2)\\
\quad \texttt{match}(s, v) = \{\langle v, s\rangle\}\\
\quad \neg\texttt{is\_var}(t_1) \wedge \neg\texttt{is\_var}(t_2) \wedge \texttt{ofs}(t_1) = \texttt{ofs}(t_2) \wedge \texttt{matches}(\texttt{subs}_\texttt{t}(t_1), \texttt{subs}_\texttt{t}(t_2))\\
\qquad\quad \Longrightarrow \texttt{match}(t_1, t_2) = \texttt{match}(\texttt{subs}_\texttt{t}(t_1), \texttt{subs}_\texttt{t}(t_2))\\
\quad \texttt{matches}(\texttt{at}(i_1), \texttt{at}(i_2)) \wedge \texttt{last}(i_1) \wedge \texttt{last}(i_2)\\
\qquad\quad \Longrightarrow \texttt{match}(i_1, i_2) = \texttt{match}(\texttt{at}(i_1), \texttt{at}(i_2))\\
\quad \texttt{matches}(\texttt{at}(i_1), \texttt{at}(i_2)) \wedge \neg\texttt{last}(i_1) \wedge \neg\texttt{last}(i_2) \wedge \texttt{matches}(\texttt{adv}(i_1), \texttt{adv}(i_2))\\
\qquad\quad \Longrightarrow \texttt{match}(i_1, i_2) = \texttt{match}(\texttt{at}(i_1), \texttt{at}(i_2)) \cup \texttt{match}(\texttt{adv}(i_1), \texttt{adv}(i_2))
\end{array}$$

Specificity ordering

Intuitively, any non-variable term is more specific than a variable. This is the basis for a partial order on terms: syntactic specificity. The order is extended on rules.

$$\begin{array}{lll}
\prec: & \texttt{Rule} \;\#\; \texttt{Rule} & \textit{predicate} \\
 & \texttt{Term} \;\#\; \texttt{Term} & \textit{predicate} \\
 & \texttt{Indx} \;\#\; \texttt{Indx} & \textit{predicate} \\
\preceq: & \texttt{Term} \;\#\; \texttt{Term} & \textit{predicate} \\
 & \texttt{Indx} \;\#\; \texttt{Indx} & \textit{predicate}
\end{array}$$

$$\begin{array}{l}
\texttt{lhs}(r_1) \prec \texttt{lhs}(r_2) \Longrightarrow r_1 \prec r_2 \\
\neg\texttt{is\_var}(t) \Longrightarrow v \prec t \\
\neg\texttt{is\_var}(t_1)) \wedge \neg\texttt{is\_var}(t_2) \wedge \texttt{ofs}(t_1) = \texttt{ofs}(t_2) \wedge \texttt{subs}_{\texttt{t}}(t_1) \prec \texttt{subs}_{\texttt{t}}(t_2) \Longrightarrow t_1 \prec t_2 \\
\texttt{last}(i_1) \wedge \texttt{last}(i_2) \wedge \texttt{at}(i_1) \prec \texttt{at}(i_2) \Longrightarrow i_1 \prec i_2 \\
\neg\texttt{last}(i_1) \wedge \neg\texttt{last}(i_2) \wedge \texttt{at}(i_1) \prec \texttt{at}(i_2) \wedge \texttt{adv}(i_1) \preceq \texttt{adv}(i_2) \Longrightarrow i_1 \prec i_2 \\
\neg\texttt{last}(i_1) \wedge \neg\texttt{last}(i_2) \wedge \texttt{at}(i_1) \preceq \texttt{at}(i_2) \wedge \texttt{adv}(i_1) \prec \texttt{adv}(i_2) \Longrightarrow i_1 \prec i_2 \\
x_1 \prec x_2 \Longrightarrow x_1 \preceq x_2 \\
x \preceq x \\
v_1 \preceq v_2
\end{array}$$

4. STATIC SEMANTICS

$$\begin{array}{lr}
m_1 \;\epsilon\; p \wedge m_2 \;\epsilon\; p \wedge r_1 \;\epsilon\; m_1 \wedge r_2 \;\epsilon\; m_2 \wedge \texttt{ofs}(\texttt{lhs}(r_1)) = \texttt{ofs}(\texttt{lhs}(r_2)) \Longrightarrow m_1 = m_2 & \textit{(i)} \\
r \;\epsilon\; p \wedge f \;\epsilon\; p \wedge \texttt{ofs}(\texttt{lhs}(r)) = \texttt{name}(f) \Longrightarrow \neg\texttt{free}(f) & \textit{(ii)} \\
r \;\epsilon\; m \wedge s \;\epsilon\; m \wedge \texttt{ofs}(\texttt{lhs}(r)) = \texttt{name}(f) \Longrightarrow \neg\texttt{external}(f) & \textit{(iii)} \\
t \;\epsilon\; p \wedge f \;\epsilon\; p \wedge \texttt{ofs}(t) = \texttt{name}(f) \Longrightarrow \texttt{arity}(f) = \texttt{arity}(\texttt{subs}_{\texttt{t}}(t)) & \textit{(iv)} \\
\neg\texttt{is\_var}(\texttt{lhs}(r)) & \textit{(v)} \\
v \;\epsilon\; \texttt{rhs}(r) \Longrightarrow v \;\epsilon\; \texttt{lhs}(r) & \textit{(vi)} \\
\gamma[v] = \texttt{lhs}(r) \Longrightarrow v \not\epsilon\, \gamma & \textit{(vii)}
\end{array}$$

Notes:

- *(i)*: A function should be defined in one module only (it can be used in more than one module). *This restriction is a consequence of implementational aspects, and should be removed in later versions of* EPIC.;
- *(ii)*: A function that is declared to be *free* should never become defined;
- *(iii)*: A function that is declared to be *external* in a module should not become defined in that module;
- *(iv)*: The number of immediate sub-terms of a term must be in accordance with the arity of the outermost function symbol of that term;
- *(v)*: The left-hand side of a rewrite rule should not be a sole variable;
- *(vi)*: A variable must be defined before it is used.
- *(vii)*: Rules must be left-linear (i.e., unconditional).

5. OPERATIONAL SEMANTICS

An EPIC implementation is a procedure which, given a term and a program, attempts to determine a normal form of that term that can be reached with right-most inner-most reduction and in accordance with syntactic specificity (i.e., given a right-most innermost redex, a most-specific rule must be applied to it).

Right-most inner-most reduction and specificity do not make a rewrite system deterministic: unordered rules, or rules of equal specificity can be applicable to the same redex. Accordingly, we must consider *sets* of reducts and normal forms.

```
potentials:    Term # Prog  -> 𝒫(Context # Term # Rule)
reducts:       Term # Prog  -> 𝒫(Term)
normal_forms:  Term # Prog  -> 𝒫(Term)
```

$$\mathtt{potentials}(t_1,p) = \{\langle\gamma,t_2,r\rangle \mid r \,\epsilon\, p \wedge \gamma[t_2] = t_1 \wedge \mathtt{matches}(t_2,\mathtt{lhs}(r))\}$$

$$\mathtt{reducts}(t_1,p) = \{\gamma[\mathtt{rhs}(r)^{\mathtt{match}(t_2,\mathtt{lhs}(r))}] \mid \exists\langle\gamma,t_2,r\rangle \in \mathtt{potentials}(t_1,p) : \neg\exists\langle\gamma',t',r'\rangle \in \mathtt{potentials}(t_1,p) : \gamma < \gamma' \vee r \prec r'\}$$

$$\mathtt{reducts}(t,p) = \emptyset \Longrightarrow \mathtt{normal\_forms}(t,p) = \{t\}$$

$$\mathtt{reducts}(t_1,p) \neq \emptyset \Longrightarrow \mathtt{normal\_forms}(t_1,p) = \bigcup_{t_2 \in \mathtt{reducts}(t_1,p)} \mathtt{normal\_forms}(t_2,p)$$

An implementation is a procedure which, given a program p and a term t_0, may or may not terminate. If it terminates, it yields a member t_n of $\mathtt{normal\_forms}(t_0,p)$.

6. A MODEL OF THE ABSTRACT SYNTAX

In this section we present a model of the abstract syntax presented earlier.

Consider the following signature:

```
E  — The (single) sort of all EPIC constructs
C  — The sort of characters
```

```
spec:  E              -> E
mod:   E # E          -> E
fun:   E # E # E # E  -> E
rule:  E # E          -> E
ap:    E # E          -> E
var:   E              -> E
cons:  E # E          -> E
nil:                  -> E
str:   C # E          -> E
eos:                  -> E
a:                    -> C
...
z:                    -> C
...
```

We assume a sufficient number of characters can be defined to represent identifiers.

We use characters `f` and `e`, in the appropriate place, to signify free and external functions, respectively (see below).

For each function defined in EPIC's abstract syntax a function should now be added to the signature above, equations should be given, and a 1-1 map between these functions and those in EPIC's abstract syntax should be given. For brevity we will use the same function names as earlier (leaving their signature implicit), and using the identity map.

Without loss of generality we will use sub-structure selection based on recursive structures.

$$
\begin{array}{l}
\texttt{at}(\texttt{cons}(x_1, x_2)) = x_1 \\
\texttt{adv}(\texttt{cons}(x_1, x_2)) = x_2 \\
\texttt{last}(\texttt{cons}(x, \texttt{nil})) \\
\texttt{mods}(\texttt{spec}(\texttt{cons}(x_1, x_2))) \\
\texttt{subs}_{\texttt{m}}(\texttt{spec}(x)) = x \\
\texttt{funs}(\texttt{mod}(\texttt{cons}(x_1, x_2), x_3)) \\
\texttt{subs}_{\texttt{f}}(\texttt{mod}(x_1, x_2)) = x_1 \\
\texttt{rules}(\texttt{mod}(x_1, \texttt{cons}(x_2, x_3))) \\
\texttt{subs}_{\texttt{r}}(\texttt{mod}(x_1, x_2)) = x_2 \\
\texttt{name}(\texttt{fun}(x_1, x_2, x_3, x_4)) = x_1 \\
\texttt{arity}(\texttt{fun}(x_1, x_2, x_3, x_4)) = x_2 \\
\texttt{free}(\texttt{fun}(x_1, x_2, \texttt{f}, x_4)) \\
\texttt{external}(\texttt{fun}(x_1, x_2, x_3, \texttt{e})) \\
\texttt{lhs}(\texttt{rule}(x_1, x_2)) = x_1 \\
\texttt{rhs}(\texttt{rule}(x_1, x_2)) = x_2 \\
\texttt{sub-terms}(\texttt{ap}(x_1, \texttt{cons}(x_2, x_3))) \\
\texttt{subs}_{\texttt{t}}(\texttt{ap}(x_1, x_2)) = x_2 \\
\texttt{ofs}(\texttt{ap}(x_1, x_2)) = x_1 \\
\texttt{is\_var}(\texttt{var}(x))
\end{array}
$$

7. A CONCRETE SYNTAX

In this section we present a concrete syntax of EPIC.

```
Spec      ::= Module Spec | ϵ
Module    ::= "module" LwrId "types" Types "rules" Rules
Types     ::= Type ";" Types | ϵ
Type      ::= FunId "(" Sort Sorts ")" "->" VrSrtId Prop |
              FunId "->" VrSrtId Prop
Prop      ::= "{" free "}" | "{" external "}" | ϵ
Sorts     ::= "," Sort Sorts | ϵ
Sort      ::= VrSrtId | "_"
Rules     ::= Rule ";" Rules | ϵ
Rule      ::= Term "=" Term
Term      ::= Var | FunId | FunId "(" Term Terms ")"
Terms     ::= "," Term Terms | ϵ
Var       ::= VrSrtId
FunId     ::= LwrId |
              "'"[!-~] |  — all printable characters
              "\"[0-2][0-9][0-9]  — all characters; decimal coded
VrSrtId   ::= [A-Z][-_A-Za-z0-9']*
```

```
LwrId    ::= [a-z][-_A-Za-z0-9']*
```

The relation between this concrete syntax and the abstract syntax of the previous section is straightforward. We will look at a few aspects:

- Syntactic rules of the form "`Ss ::= S Ss | .`" are trivially mapped to a **`cons-nil`** list;
- Syntactically, the two `Term` variants `FunId` and "`FunId "(" Terms ")"`" are distinct, but are mapped to the same form with an empty- and non-empty argument list;
- The lexical notions of identifiers are defined in two classes: those starting with a capital, which are used for variables and sorts; and those starting with a lowercase letter, which are used for function symbols.

  In both cases the lexical token should be mapped to a **`str-eos`** representation, each character being mapped to the appropriate function symbol.
- The syntax-less injection of `VrSrtId` into `Var` is represented by the injection **`var`**.

APPENDICES

1. EPIC'S TOOL SET

The EPIC tool set includes the following tools:

- an EPIC parser;
- a (primitive) typechecker;
- a printer for parsed specifications;
- a printer for μArm code;
- a non-linearity annotator. Internally, EPIC requires nonlinearities to be indicated. They are added by this tool;
- a compiler which translates EPIC to μArm. As can be seen, various features not intrinsically in EPIC are added by separate tools. The compiler combines all of the above;
- the μArm interpreter.

In addition several stand-alone tools exist:

- a currifier, which handles function symbol occurrences with too few arguments. EPIC doesn't provide currying, but this tool adds that facility;
- an ML to EPIC translator, which translates a subset of ML to EPIC.
- a μArm to C translator which compiles μArm code into C functions, one for each function in the original TRS. These functions can be linked, statically, to the interpreter.
- a tool which implements associative matching by a TRS transformation.

EPIC is available via www at `http://www.cwi.nl/epic/`

2. A HIGH-PERFORMANCE ENGINE FOR HYBRID TERM REWRITING

μArm is an efficient abstract machine for hybrid term rewriting. Here, efficiency pertains both to run-time efficiency as to efficiency with respect to software-development. In particular, μArm allows for an incremental style of software development and supports the transparant combination of compiled (stable) code with interpreted code still earlier in the software development cycle.

μArm supports external and hybrid datatypes: data types which are entirely opaque, and are manipulated only by external functions, and datatypes which, in addition, can be transparently viewed as formally specified datatypes (as defined in [Wal91]). μArm's dispatcher uses a combination of directly and indirectly threaded code to achieve an efficient, transparent interface between differnet types of functions.

μArm has efficient memory management, where garbage collection takes up less than 5% of the overall execution time. In addition, μArm uses a space-efficient innermost reduction strategy, whilst allowing for lazy rewriting when this is desired (as described in [KW95]).

Finally, μArm is parameterized with a small number of C macro's which can be defined either for portable ANSI C, or for a machine specific variant which performs two to three times better. In this manner ports for SUN SPARC and SGI R5000 using *gcc* have been defined, and a port for Macintosh (680xx) and (Symantec) Think C.

A precursor of μArm is described in [KW93]; a successor in [WK95a].

3. EPIC'S EFFICIENCY

EPIC was designed specifically with efficiency in mind, where a balance was stricken between compilation speed and execution speed. In lieu of the former, an interpreter is used for the intermediate (abstract machine) level; this interpreter has been optimized and fine-tuned to achieve acceptable execution speeds.

In [HF$^+$96] a compute-bound benchmark comparing implementations of functional languages is reported on in which μArm presented itself as the most efficient interpreted system. Since the benchmark relies heavily on floating point computations, with little control-flow overhead, it favors compiling implementations, which fare better in that benchmark.

The (portable; non machine-specific) μArm interpreter performs 350000 simple reductions per second (of the form $f(s(X)) \to f(X)$) on a SUN Sparc station. On the same platform, the Larch Prover (LP 3.1a) performs 488 reductions per second, on the identical example. This is not mentioned as a comment on LP, but rather to provide a basis for comparison with other platforms.

REFERENCES

[BBKW89] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term-rewriting systems with rule priorities. *Theoretical Computer Science*, 67(1):283–301, 1989.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[DP86] N. Dershowitz and D.A. Plaisted. Equational programming. *Machine Intelligence*, 11:21—56, 1986.

[HF$^+$96] Pieter H. Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 1996. Accepted for publication.

[Kam94] J.F.Th. Kamperman. GEL, a graph exchange language. Report CS-R9440, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Kam94.ps.Z.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[KW93] J.F.Th. Kamperman and H.R. Walters. ARM – Abstract Rewriting Machine. In H.A. Wijshoff, editor, *Computing Science in the Netherlands*, pages 193–204, 1993.

[KW95] J.F.Th. Kamperman and H.R. Walters. Lazy rewriting and eager machinery. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, number 914 in Lecture Notes in Computer Science, pages 147–162. Springer-Verlag, 1995.

[MOI95] Aart Middeldorp, Satoshi Okui, and Tesuo Ida. Lazy narrowing: Strong completeness and eager variable elimination. In *Proceedings of the 20th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[O'D85] M.J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[Wal90] H.R. Walters. Hybrid implementations of algebraic specifications. In H. Kirchner and W. Wechler, editors, *Proceedings of the Second International Conference on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 40–54. Springer-Verlag, 1990.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Wal91.ps.Z.

[WK95a] H.R. Walters and J.F.Th. Kamperman. Minimal term rewriting systems. Technical Report CS-R9573, CWI, december 1995. Submitted for publication. Available as http://www.cwi.nl/ gipe/epic/articles/CS-R9573.ps.Z.

[WK95b] H.R. Walters and J.F.Th. Kamperman. A model for I/O in equational languages with don't care non-determinism. Technical Report CS-R9572, CWI, december 1995. Submitted for publication. Available as http://www.cwi.nl/ gipe/epic/articles/CS-R9572.ps.Z.