Translating Fork Specifications into Logic Programs

Gabriel Baum
LIFIA
Facultad de Informática
Universidad Nacional de la Plata
gbaum@sol.info.unlp.edu.ar

Marcelo Frias*
Departamento de Computación
Facultad de Ciencias Exactas, Físicas y Naturales
Universidad de Buenos Aires
mfrias@sol.info.unlp.edu.ar

Nazareno Aguirre Marcelo Arroyo
Área de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
{naguirre, marroyo}@exa.unrc.edu.ar

Abstract

In this work a compiler from fork specifications into logic programs is presented. The technique implemented by the compiler consists of transforming a set of fork equations (with some restrictions) into normal logic programs in such a way that the semantics of the fork equations is preserved.

After translating a fork specification, it can be executed by consulting the generated logic program.

1 Introduction

Fork algebras are a kind of algebras of binary relations specially developed for program specification and construction [3]. Every relation in this formalism represents a program, relating the input data (problem domain) with

*The participation of Marcelo Frias in this work has been partially supported by LIFIA, Facultad de Informática, Universidad Nacional de La Plata.

the output data (solutions). Fork algebras have several important properties [4][6]. One of their main features as specification language is their relational nature, which allows to write specifications very easily, specially for non-deterministic tasks [5]. The main property of (abstract) fork algebras as environment for calculating programs is the representability of abstract fork algebras into proper ones [7]; due to this property, the programmer can port knowledge from the problem domain to the abstract calculus.

However, fork algebras, as many other formal languages, are not widely used, partly because of the absence of software tools for development within the methodology.

There exist some software tools for supporting relational methods, such as *RELVIEW* [11], *RALF* [10] and *LIBRA* [8]. *RELVIEW* is a relational evaluator, *RALF* is a theorem prover, and *LIBRA*, closer to our compiler, is a programming language based on the algebra of binary relations. However, *LIBRA* is different to our compiler, because it is not focused on an abstract calculus of relations.

Here we present a compiler that allows to translate fork specifications into normal logic programs, that could be executed by a common Prolog intepreter in a sound way.

2 Fork Algebras

Proper Fork algebras are algebras of binary relations extended with a binary operation called *fork*. For formally defining proper fork algebras, it is necessary first to define the class of Full$*$PFA:

DEFINITION 2.1 A Full$*$PFAis a two sorted structure with domains $\mathcal{P}(U \times U)$ and U

$$\langle \mathcal{P}(U \times U), U, \cup, \cap, ^{-}, \emptyset, U \times U, |, Id, ^{\smile}, \underline{\nabla}, * \rangle$$

such that

1. $|$, Id, $^{\smile}$ and $^{-}$ denote respectively composition of binary relations, the identity relation on U, converse of a binary relation and set complementation w.r.t. $U \times U$,
2. $* : U \times U \to U$ is an injective function,
3. $R \underline{\nabla} S = \{ \langle x, *(y, z) \rangle : x\, R\, y \text{ and } x\, S\, z \}$.

DEFINITION 2.2 The class of FullPFA is defined as RdFull$*$PFA, where Rd takes reducts of the similarity type $\langle \cup, \cap, ^{-}, \emptyset, U \times U, |, Id, ^{\smile}, \underline{\nabla} \rangle$, and the class PFA is defined as S P FullPFA, where S takes subalgebras and P closes a class under direct product.

The abstract counterpart of the class PFA is the class of abstract fork algebras, which is defined as follows:

DEFINITION 2.3 An abstract fork algebra is an algebraic structure

$$\langle R, +, \cdot, ^{-}, 0, 1, ;, 1', ^{\smile}, \nabla \rangle$$

where $+, \cdot, ;, \nabla$ are binary operations, $^{-}$ and $^{\smile}$ are unary, and $0, 1, 1'$ are constants, and the following set of axioms is satisfied:

Those axioms stating that $\langle R, +, \cdot, ^{-}, 0, 1 \rangle$ is a Boolean Algebra,

$$x\,;(y\,;z) = (x\,;y)\,;z, \tag{Ax. 1}$$

$$(x+y)\,;z = x\,;z \;+\; y\,;z, \tag{Ax. 2}$$

$$(x+y)^{\smile} = \breve{x}+\breve{y}, \tag{Ax. 3}$$

$$\breve{\breve{x}} = x, \tag{Ax. 4}$$

$$x\,;1' = 1'\,;x = x, \tag{Ax. 5}$$

$$(x\,;y)^{\smile} = \breve{y}\,;\breve{x}, \tag{Ax. 6}$$

$$x\,;y \;\cdot\; z = 0 \quad \text{iff} \quad z\,;\breve{y} \;\cdot\; x = 0 \quad \text{iff} \quad \breve{x}\,;z \;\cdot\; y = 0. \tag{Ax. 7}$$

$$r \nabla s = (r\,;(1' \nabla 1)) \cdot (s\,;(1 \nabla 1'))\,, \tag{Ax. 8}$$

$$(r \nabla s)\,;(t \nabla q)^{\smile} = \left(r\,;\breve{t}\right) \cdot (s\,;\breve{q})\,, \tag{Ax. 9}$$

$$(1' \nabla 1)^{\smile} \nabla (1 \nabla 1')^{\smile} \;+\; 1' = 1'. \tag{Ax. 10}$$

An useful operator for program specification using AFA is the *operator cross* (denoted by $\otimes$), which can be defined from the other operations as

$$R \otimes S = ((1' \nabla 1)^{\smile}\,;R) \;\nabla\; ((1 \nabla 1')^{\smile}\,;S).$$

and whose meaning in the standard models of AFA is depicted in Figure 1.

Also, operations that behave as projections in standard models can be abstractly defined as follows

$$\pi = (1' \nabla 1)^{\smile} \quad \text{and} \quad \rho = (1 \nabla 1')^{\smile}$$

The interpretation of π and ρ in the standard models is described pictorally in Figure 2.

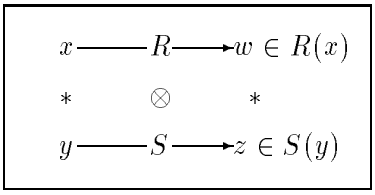

Figure 1: The operator *cross*.

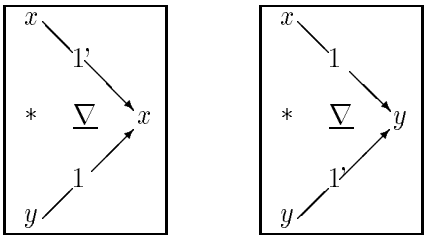

Figure 2: The projections π and ρ.

2.1 Fork Algebras as Specification Language

Within fork algebras, program specifications are made up by sets of abstract fork equations. The intended meaning of a fork equation is a binary relation that relates data (input) to results (output); hence, relational composition represents sequential composition of programs, relational join represents program joining, and so on. Program transformation rules are the theorems of abstract fork algebras. for example, the term $t_1 + t_2$ can be transformed into $t_2 + t_1$ due to Commutativity of $+$ (recall that the structure $\langle R, +, \cdot, ^-, 0, 1 \rangle$ is a Boolean algebra).

Programs are homogeneous relations, so programs can have input or output from multiple data types in this formalism.

Some extra constant relations are included, and its meaning is related to basic operations on datatypes.

During the development process, the *fork* and *cross* operations are very important and useful. The reason is that they are convenient for specifying programs composed by subprograms that share data, and, in the case of *cross*, it allows to perform parallel computations on data constructed by $\star$.

EXAMPLE: Let us consider the operation that sums the elements of a list of natural numbers. Let us suppose further that $1'_{L=0}$ is the *partial identity* on the empty list, and $1'_{L>0}$ be the partial identity on nonempty lists, *zero* be the constant that relates any element with the natural number 0; finally, let *add* be the relation that sums two natural numbers. The operation SUM

can be specified as follows:

$$SUM = 1'_{L=0}; zero + 1'_{L>0}; \begin{array}{ccc} hd & & 1' \\ \nabla & ; & \otimes \\ tl & & SUM \end{array} ; add$$

where hd and tl yield respectively the head and the tail of a (nonempty) list.

Let us explain the meaning of the above specification. The sum of the elements of the empty sequence is zero; if a list is nonempty we calculate the sum of the tail, and we add the head to that result.

As it is shown in the previous example, the combination of partial identities and the operator $+$ can be used to construct case-like compositions of programs.

3 Logic Programs

Opposed to the untyped setting of common logic programming systems, we consider a typed universe; instead of using the Herbrand universe, we choose a restriction of it, in which the terms are constructed as follows:
Suppose that the language supports types $\tau_1, ..., \tau_k$. Consider a first order language $\mathcal{L}$ composed by:

- A numerable set of variable symbols,
- A numerable set of predicate symbols,
- for each n-ary constructor f from a type τ_i, we include f in the alphabet as an n-ary function symbol of type τ_i ,
- a binary function symbol $\star$, which will be called *star*.

Constructors from types must be injective functions, and the ranges of two different constructors of the same type must be disjoint sets.

DEFINITION 3.1 The set of *ur-terms* for language $\mathcal{L}$ is constructed as follows:

- Each variable symbol is an ur-term,
- each 0-ary function symbol f_0 of type τ_i is an ur-term of type τ_i
- If f is a k-ary function symbol, where $k \geq 1$, corresponding to a constructor

  $$f : \tau_{j_1}, ..., \tau_{j_k} \rightarrow \tau_i$$

  and $t_1, ..., t_k$ are ur-terms of type $\tau_{j_1}, ..., \tau_{j_k}$ respectively, then $f(t_1, ..., t_k)$ is an ur-term of type τ_i.

DEFINITION 3.2 The set of *terms* for language $\mathcal{L}$ is constructed in the following way:

- If t is an ur-term then it is a term,
- if t_1, t_2 are terms, so is $\star(t_1, t_2)$.

3.1 Syntax of Programs

Let $\mathcal{L}$ be a language as described above. If p is an n-ary predicate symbol, and $t_1, ..., t_n$ are terms, then $p(t_1, ..., t_n)$ is an *atom*. A *literal* is an atom or a negated atom (it is to say, $\neg\langle atom\rangle$).

The *S-base* of $\mathcal{L}$, denoted by $B_{\mathcal{L}}$, is the set of all ground atoms (i.e., the set of all atoms that do not contain variables).

A *clause* is an expression of the form:

$$p \leftarrow p_1, ..., p_n$$

for each $n \geq 0$, where p is an atom and every p_i, $1 \leq i \leq n$ is a literal.

A *program* is a pair

$$\langle P, m\rangle$$

where P is a set of clauses and m is a predicate symbol.

We will denote the class of all logic programs by $Prog$.

3.2 Semantics of Programs

Let S be a subset of $B_{\mathcal{L}}$ and Cl be a set of clauses. We will say that S is a *model* of Cl if S satisfies every clause in Cl. A clause of the form

$$p \leftarrow p_1, ..., p_k$$

is interpreted as the universal closure of the formula $p_1 \wedge ... \wedge p_k \rightarrow p$ (interpreting the symbol $\neg$ as logical negation).

We cannot choose as semantics for our programs the minimal model semantics, because negation is allowed in the body of clauses. This produces that many distinct minimal models could exist for a particular program; it could be worst: a set of clauses could be inconsistent. So, we restrict the class $Prog$ to a set of programs, called stratified, for which always there exist a minimal model. We consider for these programs the *standard model* semantics [2], which consists in dividing a program into (monotonic) strata, where each stratum uses negatively only predicates from previous strata, and construct the minimal model of each stratum based on the result on the previous one.

DEFINITION 3.3 Let $\langle P, m\rangle \in Prog$. We construct the *dependecy graph* $DG(P)$ for P as follows:

- For every predicate symbol q occuring in P, there is a node in $DG(P)$ labeled by q,
- if there exists a clause in P of the form:

$$q(...) \leftarrow ..., p(...), ...$$

then there is an arc in $DG(P)$ from the node labeled by p to the node labeled by q,

- if there exists a clause in P of the form:

$$q(...) \leftarrow ..., \neg p(...), ...$$

then there is an arc in $DG(P)$ labeled by '$\neg$' from the node labeled by p to the node labeled by q.

We will say that $\langle P, m\rangle$ is *stratified* if $DG(P)$ has no cycles with an arc labeled by $\neg$.

We will denote by $Prog_{Strat}$ the class of all the stratified programs.

DEFINITION 3.4 Consider a set of clauses Cl. $Cl = Cl_1 \cup Cl_2 \cup ... \cup Cl_n$ is called a *stratification* of Cl is for $i \in [1, n]$ Cl_i uses

- positively only predicates defined in $\bigcup_{j=1}^{i} Cl_j$,
- negatively only predicates defined in $\bigcup_{j=1}^{i-1} Cl_j$.

DEFINITION 3.5 Let Cl be a stratified set of clauses. Assume a stratification $Cl = Cl_1 \cup Cl_2 \cup ... \cup Cl_n$, and let $M|S$, where $M \subseteq B_{\mathcal{L}}$ and S is a set of clauses, denote the interpretation M restricted to predicates in clauses of S. Then, we define:
$M_1 =$ minimal model for Cl_1,
$M_2 =$ minimal model for Cl_2 such that $M_2|Cl_1 = M_1$,
...
$M_n =$ minimal model for Cl_n such that $M_n|Cl_1 = M_1, ..., M_n|Cl_{n-1} = M_{n-1}$.

M_n is called the *standard model* of Cl.

It is shown in [2] that the standard model is minimal and supported, and that it does not depend on the stratification.

DEFINITION 3.6 Let $\langle P, m\rangle \in Prog_{Strat}$. We will call *general meaning* of $\langle P, m\rangle$ the standard model of P. The *meaning* of $\langle P, m\rangle$, denoted by $M(\langle P, m\rangle)$ is the set of atoms in the general meaning that have m as predicate symbol.

4 The Language

Basically, a specification is a set of fork equations, where a fork equation has the form

$$\langle\, variable \,\rangle = \langle\, term \,\rangle$$

An equation may be thought of as the definition of a program module, where the variable is the "name" of the module and the term is its implementation. The term may contain variables, that may be seen as "calls" to other program modules.

A variable is simply an identifier, composed by any sequence of characters, not beginning with '\'. A term is a (possibly nonground) abstract fork term, where the names of the fork operations are the following:

1	$\backslash univ$
$1'$	$\backslash id$
0	$\backslash empty$
π	$\backslash pi$
ρ	$\backslash rho$
$arg_1 ; arg_2$	$arg_1 ; arg_2$
$arg_1 + arg_2$	$\backslash join\{arg_1\}\{arg_2\}$
$arg_1 \cdot arg_2$	$\backslash meet\{arg_1\}\{arg_2\}$
$arg_1 \nabla arg_2$	$\backslash fork\{arg_1\}\{arg_2\}$
$arg_1 \otimes arg_2$	$\backslash cross\{arg_1\}\{arg_2\}$
$arg^{\breve{}}$	$\backslash conver\{arg\}$
$\overline{arg}$	$\backslash compl\{arg\}$

EXAMPLE: Consider the following fork specification

$$TWO\_PARALLEL\_X = \begin{matrix} 1' & X \\ \nabla & ; & \otimes \\ 1' & X \end{matrix}$$

that intuitively performs two parallel computations of X to the same argument; in our language it is written as follows:

```
TWO_PARALLEL_X = \fork{\id}{\id} ; \cross{X}{X}
```

4.1 Types

It is obvious that without further constant relations it is not possible to write interesting specifications; in fact, we could not use datatypes if only the basic fork operations are available.

Thus, we include some extra operations whose behavior is related to datatype manipulation.

Natural numbers

The extra relational operations that our language supports for manipulating natural numbers are:

- $\backslash zero$: This operation relates any element (an element from any datatype) to the natural number zero.
- $\backslash succ$: relates a natural number to its successor.
- $\backslash pred$: relates a nonzero natural number to its predecessor.

Lists of natural numbers

The relations that act on lists of natural numbers are:

- $\backslash nil$: relates any element to the empty list.
- $\backslash cons$: Given a pair, constructed by '$\star$', whose first component is a natural number n and the second one is a list l, relation $\backslash cons$ relates this pair to the list constructed by putting n in front of l.
- $\backslash hd$: relates a nonempty list to its head.
- $\backslash tl$: relates a nonempty list to its tail.

Booleans

The relations that manipulate boolean values are:

- $\backslash true$: relates any element to the boolean value $true$.
- $\backslash false$: relates any element to the boolean value $false$.

Binary Trees of natural numbers

The following relations allow to use binary trees:

- $\backslash niltree$: relates any element to the empty tree.
- $\backslash maketree$: Given a 3-uple (actually is a pair) whose first component is a natural number n, and whose second and third components are binary trees b_1 and b_2 respectively, $\backslash maketree$ relates this triuple to the tree composed by b_1 as left child, b_2 as right child, and n as root.
- $\backslash lch$: relates a nonempty tree to its left child.
- $\backslash rch$: relates a nonempty tree to its right child.
- $\backslash root$: relates a nonempty tree to its root.

We include also relations that correspond to "filters" on the range of constructor relations, such as $\backslash idnil$, which is the partial identity on the empty list. The grammar of our specification language is shown in Figure 3.

EXAMPLE:

1. Let us consider the following specification:

$$LENGTH = 1'_{L=0}; zero + 1'_{L>0}; tl; LENGTH; succ$$

where $1'_{L=0}$ is the partial identity on the empty list, $1'_{L>0}$ is the partial identity on nonempty lists, *zero* relates any element to zero, *tl* calculates the tail of a list, and *succ* adds 1 to a natural number. Clearly, $LENGTH$ recursively computes the length of a list. In our language it is written as:

```
LENGTH = \join{\idnil;\zero}
              {\idcons;\tl;LENGTH;\succ}
```

2. Let us consider now the following specification:

$$add = \begin{matrix} 1'_{N=0} \\ \otimes \\ 1'_{Nat} \end{matrix} ; \rho + \begin{matrix} 1'_{N>0} \\ \otimes \\ 1'_{Nat} \end{matrix} ; \begin{matrix} pred \\ \otimes \\ 1' \end{matrix} ; add; succ$$

where *pred* calculates the predecessor of a natural number, and the identities $1'_{N=0}$, $1'_{N>0}$, $1'_{Nat}$ are respectively filters on the natural zero, nonzero natural numbers and natural numbers. The relation *add* computes the sum of two natural numbers. This specification can be written in our language in the following way:

```
add = \join{\cross{\idzero}{\join{\idzero}{\idsucc}};\rho}
           {\cross{\idzero}{\idsucc};
            \cross{\pred}{\id};
            add;
            \succ
           }
```

4.2 Restrictions

There exist some restrictions on the specifications. As it is indicated in [1], a set of equations must be stratified with respect to complementation; this means that if a relation R depends on the complementation of S then S cannot depend on R.

It is also necessary that all the equations from a set have different variables in their left-hand side (no multiple definitions of relations).

```
<program>  : <eqlist>

<eqlist>   : <equation>
           | <eqlist> . <equation>

<equation> : <VAR> = <term_list>

<term_list>: <term>
           | <term_list> ; <term>

<term>     : <VAR> | \fork <arg> <arg> | \join <arg> <arg>
           | \meet <arg> <arg> | \cross <arg> <arg> | \cons
           | \conver <arg> | \compl <arg> | \succ | \pred
           | \hd | \tl | \dom <arg> | \ran <arg> | \id
           | \idnil | \idcons | \idzero | \idsucc | \univ
           | \empty | \pi | \rho | \zero | \nil | \niltree
           | \maketree | \root | \leftchild | \rightchild
           | \idniltree | \idnvtree | \true | \false
           | \idfalse | \idtrue

<arg>      : { <term_list> }
```

Figure 3: Grammar of the specification language.

4.3 Semantics of Fork Specifications

Although the semantics of fork specifications will not be studied in this paper, it is important to note that, as it is explained and proved in [1], the way in which specifications are defined and translated is completely natural, and yields a straightforward definition for semantics of stratified sets of equations.

5 The Fork Compiler

The fork compiler that we describe in this section is a tool that allows to execute fork specifications. It works translating a fork specification into a normal logic program in such a manner that the semantics of the original specification is preserved.

5.1 Executing a Specification

Once a specification is written, it can be translated into a logic program by using the fork compiler. Then, the programmer can execute the specification by interpreting the generated code in a logic programming interpreter.

For each relation definition of the form

$$X = T$$

a predicate p_X is generated in the output logic program, so the programmer may consult predicate p_X to execute the relational program X.

EXAMPLE: Let us consider the $LENGTH$ specification given above. The logic program generated by the compiler include the predicate p_LENGTH, which can be used to execute $LENGTH$. If the programmer wants to compute the length of the list $[1, 2, 3, 4]$, he would make the following consult to the generated program:

```
p_LENGTH([1,2,3,4], X).
```

5.2 Using a Common Prolog Interpreter to Execute Specifications

The declarative meaning of logic programs is given by the standard model semantics [2]; although Prolog does not support this semantics, the refutation procedure [9] (procedural semantics of programs in Prolog) is sound with respect to this meaning. Thus, a Prolog interpreter can be used to evaluate a program generated by the fork compiler. However, two problems could arise if a Prolog interpreter is used to evaluate fork logic programs:

- the refutation procedure could fail to find successful results, specially when the original fork specification use the complementation operation,
- because of the untyped nature of Prolog semantics, a program could yield meaningless terms as results if it is evaluated on a non-well-formed term (it is not controlled by the interpreter). For example, if the relation *pred* (that relates a nonzero natural number to its predecessor) is evaluated on the term $s(\star(0,0))$, which is not well-formed, it will yield $\star(0,0)$ as result.

5.3 Optimizations to the Generated Code

Some simple optimizations can be made to the generated program, that does not affect the performance of the program, but improve its reading. They are unfolding of predicates, in order to avoid the use of unnecessary predicate definitions, and the elimination of repeated predicate definitions, produced by the sharing of some subexpressions in fork specifications.

6 Conclusions

The tool that we have presented allows to translate a fork specification into a logic program preserving the meaning of the original specification. This tool can be extended in many ways. At first, an interpreter that checks the correct construction of predicate arguments when consulting a program should be made; also, the stratification of specifications should be checked staticly.

Another useful extension is the construction of a visual tool for editing fork specifications, and language extension for supporting further datatypes.

Acknowledgements

Part of this work was written while Marcelo Frias was visiting the Universidad Nacional de Río Cuarto. The hospitality of Jorge Aguirre, Ricardo Medel, Verónica Gentile, Laura Sabatini, Javier Smaldone, Nicolás Merkis and Ricky is gratefully acknowledged.

References

[1] Aguirre, N., *A logical interpretation of abstract fork specifications*, in Proceedings of Workshop Argentino de Informática Teórica WAIT'99, 28^o Jornadas Argentinas de Informática e Investigación Operativa 28 JAIIO, 1999.

[2] Apt, k.R., Blair, H.A., Walker, A., *Towards a Theory of Declarative Knowledge*, in J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Pub., Washington D. C., 1988.

[3] Baum, G.A., Frias, M.F., Haeberer, A.M. and Martínez López, P.E., *From Specifications to Programs: A Fork–algebraic Approach to Bridge the Gap*, in Proceedings of MFCS'96, LNCS 1113, Springer-Verlag, pp. 180–191, 1996.

[4] Brink, C., Kahl, W. and Schmidt, G. (Eds.), *Relational Methods in Computer Science*, Springer, Wien New York, 1997.

[5] Frias, M. F., Baum, G. A. and Haeberer, A. M., *Representability and Program Construction within Fork Algebras*, to appear in Journal of the IGPL.

[6] Frias, M. F., Baum, G. A., Haeberer, A. M. and Veloso, P. A. S., *A Representation Theorem for Fork Algebras*, (Technical Report) MCC. 29/93, PUC-RJ, August 1993.

[7] Frias, M. F., Baum, G. A., Haeberer, A. M. and Veloso, P. A. S., *Fork Algebras are Representable*, in Bulletin of the Section of Logic, University of Łódź, Vol. 24, No. 2, pp.64–75, 1995.

[8] Libra Programming Language, Department of Computer Science, University of Adelaide, Adelaide, South Australia, URL: www.cs.adelaide.edu.au/users/dwyer/TR95-10_TOC.html.

[9] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1987.

[10] Ralf System, Home Page of RelMiCS, URL: inf2-www.informatik.unibw-muenchen.de/Research/Tools/Ralf/ralfmanual.html.

[11] Relview System, Department of Computer Science and Applied Mathematics, University of Kiel, Germany. URL: www.informatik.uni-kiel.de/~progsys/relview.html.