PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 35, NO. 3, PP. 165-192 (1991)

A BMF FOR SEMANTICS[1]

E. MEIJER

Informatics Department
University of Nijmegen

Received July 31,1991

Abstract

We show how the Bird-Meertens formalism (BMF) can be based on continuous algebras such that finite and infinite datatypes may peacefully coexist. Until recently the theory could only deal with either finite datatypes (= initial algebra) or infinite datatypes (= final co-algebra). In the context of continuous algebras the initial algebra coincides with the final co-algebra. Elements of this algebra can be finite, infinite or partial. We intend to use EBMF for semantics directed compiler generation by combining initial algebra semantics with the calculational power of BMF.

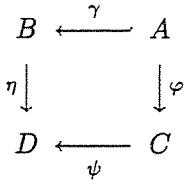
1 Introduction

In contrast to the consensus about the formal definition of syntax by means of some form of context free grammars, e.g. Backus-Nauer Form (BNF), there is no agreement about a concise, readable and easily manipulatable formalism for defining the semantics of programming languages. We think that the *Initial Algebra Semantics* of the ADJ group (GOGUEN, et al, REYNOLDS 1977) combined with the calculational power of the *Bird-Meertens formalism* or *Squiggol* is very adequate for this purpose. In this paper we show how BMF (BACKHOUSE, 1988; BIRD, 1976; MEERTENS, 1986; MALCOLM, 1989b) can be built on the same basis as IAS: continuous algebras. Since programs may or may not be recursive, our formalism should be able to deal with both finite and infinite datatypes at the same time. Continuous algebras allow this.

Both BMF and IAS are firmly based on algebraic grounds. The extensive use of initiality properties is perhaps their most characteristic feature. Datatypes are initial algebras and programs are homomorphisms (called *catamorphisms* by Meertens) from these algebras to other, similar algebras. In the context of semantics 'programs' are datatypes and semantic functions are catamorphisms. If A is initial then there exists a unique cata-

[1]Report on the Phœnix project: Esprit Basic Research Action 3147

morphism from A to any other algebra of the same signature, thence the following diagram commutes:



In other words $\eta \cdot \gamma = \psi \cdot \varphi$. Initiality forms the basis for transformation steps, the *promotion* law, by putting $\eta = id$ or $\psi = id$ and for correctness proofs, the *unique extension property*, by putting $\eta = id$ and $\psi = id$. The beauty about this is that explicit induction proofs can be circumvented by simply checking some homomorphic properties, i.e. by calculation using equational reasoning.

We use the above ideas to develop a transformational approach to semantics directed compiler generation. First a semantics is defined (as a catamorphism) which is as abstract as possible. Next an efficiency improving transformation is defined as an injective homomorphism on the target algebra. By applying the promotion law we can calculate a new, more efficient, semantics.

This paper is couched in fairly informal terms, a more formal account is (MEIJER) and (FOKINGA), 1991.

1.1 Overview

First we review the basic notions of continuous functions, cpos and least fixed points (SCHMIDT, 1986). Then in section 4 we extend some of the Squiggol theory (FOKINGA, 1990); (MANES) and (ARBIB, 1986) to continuous algebras (GOGUEN et al, 1977). Various techniques for reasoning and calculating with fixed points are discussed in sections 3 and 5 . The most important ones are based on a promotion law for fixed points, which allows us to extend transformations on nonrecursive constructs to recursive constructs. Finally we show several examples among which an alternative theory of infinitary objects and the definition of the meaning of flowcharts.

2 Continuous Functions and their Least Fixed Points

In this section a short overview of continuous functions and their fixed points is given, thereby introducing a number of notational conventions that come from Meerten's and Bird's Algorithmics. Readers already familiar with this notation may skip to section 4.

2.1 Partially Ordered Sets

A partially ordered set or *poset* is a pair $(D, \sqsubseteq)$ consisting of a set D together with a partial order $\sqsubseteq$ on D that is
reflexive $a \sqsubseteq a$
transitive $a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$
antisymmetric $a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$

A set $X \subseteq D$ is a *chain* if all elements in X are comparable. Thus for all $a, b \in X$

$$a \sqsubseteq b \vee \sqsubseteq a.$$

The terms 'totally ordered set' and 'linearly ordered set' are used as synonyms for chain.

The least element, if any, in a poset is usually denoted by $\bot$, i.e. for all $a \in D$

$$\bot \sqsubseteq a.$$

Given $a, b \in D$, their *join* $a \sqcup b$ is the least element in D that is greater than both a and b. In general it is not true that every two elements in D have joins

$$c = a \sqcup b \equiv (\forall d :: c \sqsubseteq d \equiv a \sqsubseteq d \wedge b \sqsubseteq d).$$

The dual counterpart of join is called *meet* and is denoted by $\sqcap$. An easy to verify law concerning join is $\bot \sqcup a = a \sqcup \bot = a$. The *least upperbound* of a subset $X \subseteq D$ is denoted by $\sqcup/$, conventionally written as $\bigsqcup$. Not every $X \subseteq D$ need have a lub

$$a = \sqcup/X \equiv (\forall c :: a \sqsubseteq c \equiv (\forall b \in X :: b \sqsubseteq c)).$$

2.2 CPOs

A poset D is a complete partial order or *ω-cpo* if it contains a bottom element and each chain D has a lub, so $\sqcup/X$ does exist for any chain $X \subseteq D$. For a chain X we can interpret $\sqcup/X$ as the 'reduction' of $\sqcup$ over X, since $\sqcup/$ satisfies:

$$\sqcup/\{\} = \bot,$$
$$\sqcup/(\{x\} \cup X) = x \sqcup (\sqcup/X),$$

(this is *not* a definition of $\sqcup/$!) From now on we assume that all sources and targets of functions are cpos unless stated otherwise.

A function $f \in D' \leftarrow D$ is *monotonic* if it respects the ordering on D,

$$f\ a \sqsubseteq f\ b \Leftarrow a \sqsubseteq b.$$

A function $f \in D' \leftarrow D$ is *continuous* if it respects lubs of chains, let X be a chain:

$$(f \cdot \sqcup/)\ X = (\sqcup/ \cdot f*)\ X,$$

where $f*X = \{f\ x \mid x \in X\}$, so $f*$ satisfies the following equations:

$$f*\{\} = \{\},$$
$$f*(\{x\} \cup X) = \{f\ x\} \cup (f*X).$$

If f is continuous then it follows that f is monotonic by taking the chain $X = a \sqsubseteq b$.

2.3 Building Domains

It is a reassuring fact that any (recursive) domain built from cpos using sums, products and arrows is again a cpo and that any function built using lambda notation is continuous.

The (lazy) product $D\|D'$ of two cpos D and D' and its corresponding operation on functions are defined as:

$$D\|D' = \{(d, d') \mid d \in D,\ d' \in D'\},$$
$$(f\|g)\ (x, x') = (f\ x, g\ x'),$$

with ordering $(x_1, x_2) \sqsubseteq (y_1, y_2) \equiv x_1 \sqsubseteq y_1 \wedge x_2 \sqsubseteq y_2$. The name *lazy* product stems from the fact that the least element of $D\|D'$ is $(\bot, \bot)$.

The (lifted) sum $D \mid D'$ of D and D' and the corresponding operation on functions are defined as:

$$D \mid D' = \{\bot\} \cup \{(0, d) \mid d \in D\} \cup \{(1, d') \mid d' \in D'\},$$
$$(f \mid g)\ \bot = \bot,$$
$$(f \mid g)\ (0, x) = (0, f\ x),$$
$$(f \mid g)\ (1, x') = (1, g\ x'),$$

with ordering $x \sqsubseteq y \equiv (x = \bot) \vee (x = (i, x') \wedge y = (i, y') \wedge x' \sqsubseteq y')$.

The function space $D' \leftarrow D$ of continuous functions from D to D' is ordered by $f \sqsubseteq g \equiv (\forall d \in D :: f\ d \sqsubseteq g\ d)$. The corresponding action on

functions is the 'enveloping' function:

$$(f \leftarrow g)\ h = f \cdot h \cdot g.$$

Note that we write arrows from right to left; this is because function composition also goes right to left. If we curry a function we write the arrow from left to right as usual.

Lifting a domain D adds an additional bottom element: $D_\perp = \{\perp\} \cup D$, with ordering $d \sqsubseteq d' \equiv d = \perp \vee d \sqsubseteq_D d'$. Its associated operation on functions is the *strictifying* function:

$$f_\perp\ \perp = \perp,$$
$$f_\perp\ d = f\ d, d \in D,$$

The one point set is denoted by $\mathbf{1}$ and can be used to lift constants of type A into nullary functions of type $A \leftarrow \mathbf{1}$. The only member of $\mathbf{1}$ called *void* is denoted by ().

2.3.1 Functors

The above domain constructors or *bi-functors* illustrate the principle that for each binary domain constructor $\ddagger$ there is a corresponding combinator structuring functions $f \in B \leftarrow A, g \in D \leftarrow C$ into $f \ddagger g \in B \ddagger D \leftarrow A \ddagger C$ that respects identity and composition:

$$id \ddagger id = id,$$
$$f \ddagger g \cdot h \ddagger i = (f \cdot h) \ddagger (g \cdot i).$$

A *mono-functor* $\dagger$ is a unary type constructor; its operation on functions satisfies:

$$id\dagger = id,$$
$$f\dagger \cdot g\dagger = (f \cdot g)\dagger.$$

A *polynomial* functor is any functor which can be constructed from constant functors $D\dagger_C = C, f\dagger = id$ or the identity functor $D\dagger = D, f\dagger = f$ through (recursive) combinations of product and sum operations. We assume all our functors to be polynomial.

Apart from the combinators $|$ (*sum*), $\leftarrow$ (*arrow*) and $\|$ (*product*) we will also need the related combinators Δ (*doubling*), $\Uparrow$ (*sharing*), $\downarrow$ (*selection*), $^\circ, ^\frown$ (*lifting*), $\langle\!\langle , \rangle\!\rangle$ (*projections*), and $^\sim$ (*reversal*).

$$
\begin{aligned}
(f \overset{\dagger}{\leftarrow} g)\ h &= f \cdot h\dagger \cdot g, & & \\
& & x \mathbin{(\!(} y &= x, \\
\Delta\ x &= (x, x), & x \mathbin{)\!)} y &= y, \\
(f \Uparrow g) &= f \| g \cdot \Delta, & & \\
& & c^{\circ}\ x &= c, \\
(f \downarrow g)\ \bot &= \bot, & (f \mathbin{\widehat{\oplus}} g)\ x &= (f\ x) \oplus (g\ x), \\
(f \downarrow g)\ (0, x) &= f\ x, & x \mathbin{\tilde{\oplus}} y &= y \oplus x, \\
(f \downarrow g)\ (1, x') &= g\ x'. & &
\end{aligned}
$$

For the above combinators a large number of laws hold of which we state only a few:

$$
\begin{aligned}
a \| b \cdot c \| d &= (a \cdot c) \| (b \cdot d), \\
a \| b \cdot c \Uparrow d &= (a \cdot c) \Uparrow (b \cdot d), \\
a \Uparrow b \cdot c &= (a \cdot c) \Uparrow (b \cdot c),
\end{aligned}
$$

$$
\begin{aligned}
(\!(\cdot a \Uparrow b &= a, \\
(\!(\cdot a \| b &= a \cdot (\!(.
\end{aligned}
$$

In subsequent derivations the following two simple λ-promotion laws will save a lot of work, since we don't have to invent variable-free versions of functions in order to calculate

$$
\begin{aligned}
f \cdot (\lambda x.E[x]) &= \lambda x.f\ (E[x]), \\
(\lambda x.E[x]) \cdot g &= \lambda x.E[g\ x].
\end{aligned}
$$

We prove them both in one go

$$
\begin{aligned}
& (f \cdot \lambda x.E[x] \cdot g)\ y \\
&= \\
& (f \cdot \lambda x.E[x])\ (g\ y) \\
&= \\
& f\ (E[g\ y]) \\
&= \\
& (\lambda x.f\ E[g\ x])\ y.
\end{aligned}
$$

2.4 Least Fixed Points

An element $d \in D$ is a fixed point of $f \in D \leftarrow D$ if $f\ d = d$, it is a *least* fixed point if for any other fixed point d' it holds that $d \sqsubseteq d'$.

We now have collected enough definitions to prove that continuous functions do have least fixed points. Let D be a cpo and $f \in D \leftarrow D$ a continuous function. Then f has a least fixed point μf. The proof is by construction of the required element. Let $iterate \in (D \leftarrow D) \rightarrow (\{D\} \leftarrow D)$ be defined as

$$iterate\ f\ x = \{f^i\ x \mid i \in \mathbb{N}\ \}.$$

Then *iterate* satifies $iterate\ f = \{\_\}\ \hat{\cup}\ (f* \cdot iterate\ f)$, where $\{\_\}\ x = \{x\}$.

The least fixed point of f is obtained by (this is *Kleenes first recursion theorem*):

$$\mu \in (D \leftarrow D) \rightarrow D,$$
$$\mu f = (\sqcup / \cdot iterate\ f)\ \bot.$$

First we show that $f\ (\mu f) = \mu f$ by calculating:

$$\begin{aligned}
& f\ (\mu f) \\
& = \\
& (f \cdot \sqcup / \cdot iterate\ f)\ \bot \\
& = \{iterate\ f\ \bot \text{ is a chain}\} \\
& (\sqcup / \cdot f* \cdot iterate\ f)\ \bot \\
& = \{\bot \sqcup x = x\} \\
& \bot \sqcup (\sqcup / \cdot f* \cdot iterate\ f)\ \bot \\
& = \{\text{property of } \sqcup\} \\
& \sqcup\ /\ (\{\bot\} \cup (f* \cdot iterate\ f)\ \bot) \\
& = \{\text{lifting}\} \\
& \sqcup\ /\ ((\{_\}\ \hat{\cup}\ (f* \cdot iterate\ f))\ \bot) \\
& = \{\text{property of } iterate\} \\
& (\sqcup / \cdot iterate\ f)\ \bot \\
& = \\
& \mu f.
\end{aligned}$$

Next we show that μf is also least. Assume that e is also a fixed point of f then $f\ e \sqsubseteq e$ and

$$
\begin{aligned}
&\mu f \sqsubseteq e \\
&\equiv \{f\ e \sqsubseteq e\} \\
&(\sqcup/ \cdot \mathit{iterate}\ f)\ \bot \sqsubseteq (\sqcup/ \cdot \mathit{iterate}\ f)\ e \\
&\Leftarrow \{f \text{ monotonic}\} \\
&\bot \sqsubseteq e \\
&\equiv \\
&\mathit{true}.
\end{aligned}
$$

An important fact is that the least fixed point operator μ itself is continuous.

3 Calculation and Induction Rules for Fixed Points

Although we will try to avoid using (any form of) induction as much as possible, there are occasions where it is inevitable to use the *Fixed Point Induction Principle* (STOY, 1977). The three *Fixed Point Promotion Laws* allow easy calculation in the presence of fixed points.

3.1 Fixed Point Induction

Let $F \in D \leftarrow D$ a continuous function from cpo D into itself and P an *inclusive* predicate on D, then:

$$
\begin{aligned}
P\ (\mu F) &\Leftarrow P\ \bot \\
&\wedge\ P \cdot F \underset{\forall}{\Leftarrow} P.
\end{aligned}
$$

3.1.1 Inclusive Predicates

A predicate is called *inclusive* or *admissible* if it respects lubs of chains, for any chain X:

$$(\wedge/ \cdot P*)\ X \Rightarrow (P \cdot \sqcup/)\ X.$$

Clearly not every predicate is inclusive, but if f and g are continuous the predicate $P\ (f,g) \equiv f = g$ is inclusive and if f is continuous and g monotonic then $P\ (f,g) \equiv f \sqsubseteq g$ is inclusive (BIRD, 1976).

3.2 Fixed Point Promotion Theorems

In our derivations we often end up at the expression $g\ (\mu f)$, and the question is: can we proceed by putting $g\ (\mu f) = \mu h$ for some h. This question is answered by the following *fixed point promotion theorems.*

3.2.1 FPPT1

- A, B cpos.
- $f \in B \leftarrow B$, $h \in A \leftarrow A$ continuous.
- $g \in A \leftarrow B$ strict and continuous.
- $g \cdot f = h \cdot g$.

Then μh exists and satisfies

$$g\ (\mu f) = \mu h.$$

In diagrammatic form the theorem reads

$$\begin{array}{ccc} B & \xleftarrow{f} & B \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle g} \\ A & \xleftarrow[h]{} & A \end{array} \qquad \begin{array}{c} \mu f \\ \downarrow{\scriptstyle g} \\ \mu h \end{array}$$

A large number of applications of the fixed point promotion theorem were investigated by (MEYER, 1985). He actually proved a slightly stronger theorem which only requires h to be *monotonic.* Although mentioned as an example property of fixed points in (STOY, 1977), the theorem was not put into use by that author. The following fixed point induction proof is taken from (STOY 1977). Let $P\ (x, y) \equiv g\ x = y$. Since g is assumed to be strict $P\ (\bot, \bot)$ holds. So we may assume $P\ (x, y)$ and calculate:

$$\begin{array}{l} P\ (f\ x, h\ y) \\ \equiv \\ g\ (f\ x) = h\ y \\ \equiv \{\text{assumption}\} \\ h\ (g\ x) = h\ y \\ \equiv \{\text{induction hypothesis}\} \\ h\ y = h\ y \\ \equiv \\ true. \end{array}$$

Hence $P\ (\mu f, \mu h)$, i.e., $g\ (\mu f) = \mu h$.

A fixed point operator Φ is called *uniform* if it satisfies FPPT1. The least fixed point operator μ is the unique uniform fixed point operator (GUNTER, et al 1989).

3.2.2 FPPT2

When constructing circular datatypes, bound variables are used a lot to 'tie knots'. The following direct corollary of FPPT1 is formulated in terms of bound variables

$$g\ \mu(\lambda x.E[x]) = \mu(\lambda x.E'[x]) \Leftarrow g\ E[x] = E'[g\ x].$$

The proof uses the lambda-promotion law twice:

$$\begin{aligned} & g \cdot (\lambda x.E[x]) \\ & = \\ & \lambda x.g\ E[x] \\ & = \\ & \lambda x.E'[g\ x] \\ & = \\ & (\lambda x.E[x]) \cdot g. \end{aligned}$$

3.2.3 FPPT3

The third fixed point promotion theorem can be used if all else fails. In order to apply it f and g must be known but not h. If in addition we know that $i \in C \leftarrow C'$ is a continuous inclusion such that $g \cdot i = id$, in other words C' is a retract of C (this happens quite often), we can uniquely calculate h provided f is *canonical*. If C' is a retract of C we write $C' \leq_{i,g} C$.

Let $C' \leq_{i,j} C$ and $D' \leq_{k,l} D$, then $f \in D \leftarrow C$ is canonical iff

$$j\ c = j\ c' \Rightarrow (l \cdot f)\ c = (l \cdot f)\ c'.$$

We can now formulate a second promotion theorem FPPT3 (DE BRUIN, and DE VINK, 1989) as follows. Let,

- C, C' cpos such that $C' \leq_{i,g} C$, so $g \cdot i = id$,
- $f \in C \leftarrow C$ continuous and canonical, so $g\ c = g\ c' \Rightarrow (g \cdot f)\ c = (g \cdot f)\ c'$,
- g strict and continuous, then

$$(g \cdot \mu)\ f = (\mu \cdot g \leftarrow i)\ f.$$

In order to apply FPPT1 we have to show that f being canonical implies that $g \cdot f = h \cdot g$ where $h = (g \leftarrow i)\ f = g \cdot f \cdot i$.

$$\begin{aligned}
&g\ c = g\ c \\
&\{i \cdot g = id\} \equiv \\
&(g \cdot i \cdot g)\ c = g\ c \\
&\{f \text{ canonical}\} \Rightarrow \\
&(g \cdot f \cdot i \cdot g)\ c = (g \cdot f)\ c \\
&\{h = g \cdot f \cdot i\} \equiv \\
&(h \cdot g)\ c = (g \cdot f)\ c.
\end{aligned}$$

4 Reflexive Domains and Continuous Algebras

It is possible to work with infinitary objects when a datatype definition is interpreted as a reflexive domain equation or initial continuous algebra. A continuous †-algebra (A, φ) is a pair consisting of a domain A called the *carrier set* and a strict continuous *signature* $\varphi \in A \leftarrow A\dagger$ (GOGUEN, et al, REYNOLDS, 1977). Homomorphisms between such algebras are strict and continuous structure preserving mappings. Let (A, φ) and (B, ψ) be two continuous †-algebras, a function $h \in B \leftarrow A$ is called a †-homomorphism, if

- h is strict (i.e. $h\ \bot = \bot$) and continuous.
- $h \cdot \varphi = \psi \cdot h\dagger$.

In diagrammatic form the homomorphism property states that the following diagram commutes:

$$\begin{array}{ccc}
A & \xleftarrow{\varphi} & A\dagger \\
{\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h\dagger} \\
B & \xleftarrow[\psi]{} & B\dagger
\end{array}$$

4.1 Catamorphisms

A *catamorphism* (from the Greek preposition $\kappa\alpha\tau\alpha$ meaning 'downwards' as in catastrophe) is a homomorphism with as source an initial algebra.

The class of continuous †-algebras has initial algebra $\mu\dagger$ consisting of a type L and a function $\mathtt{in} \in L \leftarrow L\dagger$ such that

$$f \cdot \mathtt{in} = \varphi \cdot f\dagger$$

has the unique strict solution $f = (\!|\varphi|\!)_\dagger$ for any strict $\varphi \in A \leftarrow A\dagger$. When the subscript † is clear from the context it will be omitted. In other words $f = (\!|\varphi|\!)$ is the unique homomorphism that makes the following diagram commute.

$$\begin{array}{ccc} L & \xleftarrow{\mathtt{in}} & L\dagger \\ {\scriptstyle (\!|\varphi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\varphi|\!)\dagger} \\ A & \xleftarrow[\varphi]{} & A\dagger \end{array}$$

The initial algebra $(L, \mathtt{in})$ which is the 'least' solution of the domain equation $L = L\dagger$ (hence our notation $\mu\dagger$) can be constructed by means of Scott's *inverse limit construction*. In short, the inverse limit construction constructs a chain of cpos whose lub is the required solution. The algebra $(L, \mathtt{in})$ is isomorphic with the algebra $(L\dagger, \mathtt{in}\dagger)$, with $\mathtt{out} \in L\dagger \leftarrow L$ and $\mathtt{in} \in L \leftarrow L\dagger$ strict functions such that $\mathtt{out} \cdot \mathtt{in} = id$, $\mathtt{in} \cdot \mathtt{out} = id$ and $\mu(\mathtt{in} \overset{\dagger}{\leftarrow} \mathtt{out}) = id$. For more details on the construction of L we refer to (SCHMIDT, 1986) since this requires a fair amount of purely technical detail. We shall only prove that L is initial indeed, i.e.:

$$h = (\!|\varphi|\!) := \mu(\varphi \overset{\dagger}{\leftarrow} \mathtt{out}) \equiv (h \text{ strict}) \wedge (h \cdot \mathtt{in} = \varphi \cdot h\dagger). \tag{1}$$

The $\Rightarrow$ part of the proof follows from the proposition that for strictness preserving functors † the function $\mu(\varphi \overset{\dagger}{\leftarrow} \psi)$ is strict, if φ and ψ are strict.

For proving $\Leftarrow$ we use fixed point induction using the inclusive predicate $P(F, G) = h \cdot F = G$ with $F = \mathtt{in} \overset{\dagger}{\leftarrow} \mathtt{out}$ and $G = \varphi \overset{\dagger}{\leftarrow} \mathtt{out}$. The base case $P\ (\bot, \bot)$ follows from strictness of h. So assume $P\ (f, g) \equiv h \cdot f = g$ and calculate:

$$
\begin{array}{l}
P\ (F\ f, G\ g) \\
\equiv \\
h \cdot \mathtt{in} \cdot f\dagger \cdot \mathtt{out} = \varphi \cdot g\dagger \cdot \mathtt{out} \\
\equiv \{h \cdot \mathtt{in} = \varphi \cdot h\dagger, \text{ functor calculus}\} \\
\varphi \cdot (h \cdot f)\dagger \cdot \mathtt{out} = \varphi \cdot g\dagger \cdot \mathtt{out} \\
\equiv \{\text{induction hypothesis}\} \\
\varphi \cdot g\dagger \cdot \mathtt{out} = \varphi \cdot g\dagger \cdot \mathtt{out} \\
\equiv \\
true.
\end{array}
$$

Hence $P\ ((\!|\mathtt{in}|\!), (\!|\varphi|\!))$, and since $(\!|\mathtt{in}|\!) = id$ we are done.

In the framework of continuous algebras an initial algebra is far richer than the conventional word algebra or Herbrand universe. Initial continuous algebras contain proper, partially defined and infinite elements. The imposition of strictness and continuity upon homomorphisms forces these elements to behave. *Especially the strictness requirement is not to be taken too lightly*, least fixed point semantics is a useful and powerful tool but whenever possible we should do without it.

Often we will use the traditional 'algebraic data type' notation for initial algebras, i.e.,

$$
\begin{array}{rl}
I\!N & ::= 0 \\
& \mid \sigma\ I\!N
\end{array}
$$

instead of $(I\!N\ , 0 \downarrow \sigma) = \mu\dagger$ where $N\dagger = \mathbb{1} \mid N$. Also instead of $(\!|\varphi_0 \downarrow \ldots \downarrow \varphi_{n-1}|\!)$ we write $(\!|\varphi_0, \ldots, \varphi_{n-1}|\!)$. Using such instantiations for specific datatypes makes formulas much more readable.

When interpreting the definition for $I\!N$ as the initial word algebra we have:

$$I\!N = \{0, \sigma\ 0, (\sigma \cdot \sigma)\ 0, \ldots\}.$$

When interpreting it as an initial continuous algebra we get:

$$I\!N = \{\bot, 0, \sigma\ \bot, \sigma\ 0, (\sigma \cdot \sigma)\ \bot, (\sigma \cdot \sigma)\ 0, \ldots, \infty\}.$$

Intuitively we can see $\bot$ as a yet unfinished number.

4.2 Anamorphisms

By reversing the arrows in the catamorphism diagram we get *anamorphisms* (from the Greek preposition $\alpha\nu\alpha$ meaning 'upwards' as in anabolism):

$$\begin{array}{ccc} L & \xrightarrow{\mathtt{out}} & L\dagger \\ [\![\varphi]\!]\uparrow & & \uparrow[\![\varphi]\!]\dagger \\ A & \xleftarrow[\varphi]{} & A\dagger \end{array}$$

with $[\![\varphi]\!]$ a strict and continuous function which makes the diagram commute. In (PATTERSON, 1988) anamorphisms are called *generators*, since they generate an L object of the required structure from a given 'seed' value taken from A.

By dualizing the proof of (1) we can prove that $(L, \mathtt{out})$ is a final co-algebra:

$$h = [\![\varphi]\!] := \mu(\mathtt{in} \overset{\dagger}{\leftarrow} \varphi) \equiv (\mathtt{out} \cdot h = h\dagger \cdot \varphi)$$

4.3 Paramorphisms

Not all functions on a data type are cata- or anamorphisms. There is for example no *simple* φ such that:

$$\begin{aligned} &fac \in I\!N \ \rightarrow I\!N\ , \\ &fac = (\!|\varphi|\!). \end{aligned}$$

The problem with the factorial function is that it 'eats its argument and keeps it too' (WADLER, 1987).

$$\begin{aligned} fac\ 0 &= \sigma\ 0, \\ fac\ (\sigma\ n) &= \sigma\ n \times fac\ n \\ &= ((\times) \cdot (\sigma \Uparrow fac))\ n. \end{aligned}$$

Paramorphisms were invented by (MEERTENS, 1990) to cover this type of recursive pattern, for strict φ:

$$[\![\varphi]\!]_\dagger = \mu(\lambda f.\varphi \cdot (id \Uparrow f)\dagger \cdot \mathtt{out}).$$

The factorial function above can be defined as $[\![\sigma\ 0 \mid (\times) \cdot (\sigma \Uparrow fac)]\!]$.

A nice result is that any paramorphism can be written as the composition of a cata- and an anamorphism. Let $(L, \mathtt{in}) = \mu\dagger$ be given, then define

$$\begin{aligned} X\ddagger &= (L\|X)\dagger, \\ h\ddagger &= (id\|h)\dagger, \\ (M, \mathtt{IN}) &= \mu\ddagger. \end{aligned}$$

For the naturals we get:

$$\begin{aligned} X\ddagger &= (I\!N\ \|X)\dagger \\ &= \mathbf{1} \mid I\!N\ \|X \\ (I\!N* [\,] \downarrow \succ\!\!+) &= \mu\dagger, \end{aligned}$$

hence $\mu\ddagger$ is the type of 'lists of natural numbers'.

Now define $preds \in M \leftarrow L$ as follows:

$$preds = [\![\Delta\dagger \cdot \mathtt{out}_\dagger]\!]_\ddagger.$$

For the naturals we get:

$$\begin{aligned} &preds \in I\!N* \leftarrow I\!N\ , \\ &preds = [\![id \mid \Delta \cdot \mathtt{out}]\!]. \end{aligned}$$

That is given a natural number $N = \sigma^n\ 0$, the expression $preds\ N$ yields the list $\sigma^{n-1}\ 0 \succ\!\!+ \ldots \succ\!\!+ 0 \succ\!\!+ [\,]$.

Using $preds$ we start calculating:

$$\begin{aligned} &(\!|\varphi|\!)_\ddagger \cdot preds \\ &= \\ &(\!|\varphi|\!)_\ddagger \cdot [\![\Delta\dagger \cdot \mathtt{out}_\dagger]\!]_\ddagger \\ &= \{\text{theorem proved in section 6.1}\} \\ &\mu(\lambda f.\varphi \cdot f\ddagger \cdot \Delta\dagger \cdot \mathtt{out}) \\ &= \{\text{definition } \ddagger \text{ and } \Delta\} \\ &\mu(\lambda f.\varphi \cdot (id\|f)\dagger \cdot (id \Uparrow id)\dagger \cdot \mathtt{out}) \\ &= \{\text{functor calculus}\} \\ &\mu(\lambda f.\varphi \cdot (id \Uparrow f)\dagger \cdot \mathtt{out}) \\ &= \\ &[\![\varphi]\!]_\dagger. \end{aligned}$$

Thus $[\![\varphi]\!]_\dagger = (\!|\varphi|\!)_\ddagger \cdot preds$. Since $(\!|\mathtt{IN}|\!)_\ddagger = id$ we immediately get $preds = [\![\mathtt{IN}]\!]_\dagger$.

5 Calculation and Induction Rules for Data Structures

5.1 Unique Extension Property

Since cata- and anamorphisms are unique, we have the *unique extension property:*

$$f = g = (\!|\varphi|\!) \equiv (f \text{ strict} \wedge f \cdot \mathtt{in} = \varphi \cdot f\dagger) \wedge (g \text{ strict} \wedge g \cdot \mathtt{in} = \varphi \cdot g\dagger).$$

$$f = g = [\![\varphi]\!] \equiv \mathtt{out} \cdot f = f\dagger \cdot \varphi) \wedge \mathtt{out} \cdot g = g\dagger \cdot \varphi).$$

The UEP can be used to prove equality of two functions f and g without using induction, but by only checking the promotability property.

5.2 Promotion Law

The *promotion law*:

$$f \cdot (\!|\psi|\!) = (\!|\varphi|\!) \Leftarrow (f \cdot \psi = \varphi \cdot f\dagger) \wedge (f \text{ strict})$$

$$[\![\varphi]\!] = [\![\psi]\!] \cdot f \Leftarrow (\psi \cdot f = f\dagger \cdot \varphi)$$

can be used to transform one function into another (computationally more efficient) one.

The promotion law for catamorphisms follows from the FPPT (FOKKINGA, 1990):

$$
\begin{aligned}
& f \cdot (\!|\psi|\!) = (\!|\varphi|\!) \\
& \equiv \\
& f \cdot \mu(\psi \overset{\dagger}{\leftarrow} \mathtt{out}) = \mu(\varphi \overset{\dagger}{\leftarrow} \mathtt{out}) \\
& \Leftarrow \{\text{FPPT1}\} \\
& ((f\cdot) \cdot (\psi \overset{\dagger}{\leftarrow} \mathtt{out}))\ g = ((\varphi \overset{\dagger}{\leftarrow} \mathtt{out}) \cdot (f\cdot))\ g \\
& \equiv \\
& f \cdot \psi \cdot g\dagger \cdot \mathtt{out} = \varphi \cdot (f \cdot g)\dagger \cdot \mathtt{out} \\
& \equiv \\
& f \cdot \psi \cdot g\dagger \cdot \mathtt{out} = \varphi \cdot f\dagger \cdot g\dagger \cdot \mathtt{out} \\
& \equiv \{f \cdot \psi = \varphi \cdot f\dagger\} \\
& \varphi \cdot f\dagger \cdot g\dagger \cdot \mathtt{out} = \varphi \cdot f\dagger \cdot g\dagger \cdot \mathtt{out} \\
& \equiv \\
& \mathit{true}.
\end{aligned}
$$

The promotion law for anamorphisms can be proved by fixed point induction using the inclusive predicate $P\ (H, G) \equiv H = G \cdot f$ with $H = \mathtt{in} \overset{\dagger}{\leftarrow} \varphi$ and $G = \mathtt{in} \overset{\dagger}{\leftarrow} \psi$.

5.3 Partial Structural Induction

Fixed point induction is useful when we want to prove a property for a specific recursively defined object. If we want to prove a property for each $a \in A$ we use *partial structural induction.* Let (A, φ) be a continuous $\dagger$-algebra and P an *inclusive* predicate on A, then

$$
\begin{aligned}
P\ a \Leftarrow\ & P\ \bot \\
& \wedge\ P \cdot \varphi \underset{\forall}{\Leftarrow} \bigwedge_{\dagger} \cdot P\dagger,
\end{aligned}
$$

where $\bigwedge_{\dagger} \in \mathit{Bool} \leftarrow \mathit{Bool}\dagger$, generalizes normal conjunction.

6 Examples

As an illustration of the elegance of cata- and anamorphisms based on initial continuous algebras, we will present an alternative theory of infinite data structures by building infinite objects using the fixed point operator explicitly and compare them to infinite objects built by anamorphisms.

Subsection 6.3 uses the FPPT to give semantics to a language of flowcharts, an archetypical example of the use of our extended BMF for deriving programming language implementations. We start by giving some interesting theorems about ana- and cata-morphisms taken from the excellent thesis (PATERSON, 1988).

6.1 Theorems about Ana- and Catamorphisms

The composition of a catamorphism $(\!|\varphi|\!)$ and an anamorphism $[\!(\psi)\!]$ is equal to:

$$(\!|\varphi|\!) \cdot [\!(\psi)\!] = \mu(\varphi \overleftarrow{\dagger} \psi).$$

The proof is by fixed point induction using the inclusive predicate $P\ (F, G, H) \equiv F \cdot G = H$ with $F = \varphi \overleftarrow{\dagger} \mathtt{out}$, $G = \mathtt{in} \overleftarrow{\dagger} \psi$ and $H = \varphi \overleftarrow{\dagger} \psi$. The base case $P\ (\bot, \bot, \bot)$ is obviously true. So assuming $f \cdot g = h$ we calculate:

$$\begin{aligned}
&\varphi \cdot f\dagger \cdot \mathtt{out} \cdot \mathtt{in} \cdot g\dagger \cdot \psi = \varphi \cdot h\dagger \cdot \psi \\
&\equiv \\
&\varphi \cdot (f \cdot g)\dagger \cdot \psi = \varphi \cdot h\dagger \cdot \psi \\
&\equiv \\
&\varphi \cdot h\dagger \cdot \psi = \varphi \cdot h\dagger \cdot \psi \\
&\equiv \\
&\mathit{true}.
\end{aligned}$$

The following interesting laws can be used in conjunction with the 'theorems for free theorem'. Informally the RWB theorem (WADLER, 1989) states that any polymorphic function $f \in \alpha\ddagger \leftarrow \alpha\dagger$ is a natural transformation $\ddagger \leftarrow \dagger$, i.e. for strict g we get the following theorem for free:

$$g\ddagger \cdot f = f \cdot g\dagger.$$

The first law shows the equivalence of some cata- and anamorphisms:

$$[\!(\varphi \cdot \mathtt{out}_\dagger)\!]_\ddagger = (\!|\mathtt{in}_\ddagger \cdot \varphi|\!)_\dagger \Leftarrow \varphi \cdot f\dagger = f\ddagger \cdot \varphi.$$

The proof is straightforward:

$$
\begin{aligned}
&[\![\varphi \cdot \mathtt{out}_\dagger]\!]_\ddagger \\
&= \\
&\mu(\lambda f.\mathtt{in}_\ddagger \cdot f\ddagger \cdot \varphi \cdot \mathtt{out}_\dagger) \\
&= \\
&\mu(\lambda f.\mathtt{in}_\ddagger \cdot \varphi \cdot f\dagger \cdot \mathtt{out}_\dagger) \\
&= \\
&(\!|\mathtt{in} \cdot \varphi|\!)_\dagger .
\end{aligned}
$$

As an example of this law take the binary trees based on the functor

$$
\begin{aligned}
X\dagger &= \mathbf{1} \mid \alpha \mid X \| X, \\
(tree\ \alpha, [\,] \downarrow [\_] \downarrow +\!\!+) &= \mu\dagger .
\end{aligned}
$$

Then reversing a binary tree can be defined as either:

$$
\begin{aligned}
reverse &= (\!|\mathtt{in} \cdot id \mid id \mid \sim|\!), \\
reverse &= [\![id \mid id \mid \sim \cdot \mathtt{out}]\!],
\end{aligned}
$$

Another useful law concerns the composition of two morphisms of different signature:

$$
\begin{aligned}
(\!|\varphi|\!)_\dagger \cdot (\!|\mathtt{in} \cdot \psi|\!)_\ddagger &= (\!|\varphi \cdot \psi|\!)_\ddagger \Leftarrow f\dagger \cdot \psi = \psi \cdot f\ddagger, \\
[\![\psi \cdot \mathtt{out}]\!]_\ddagger \cdot [\![\varphi]\!]_\dagger &= [\![\psi \cdot \varphi]\!]_\ddagger \Leftarrow \psi \cdot f\dagger = f\ddagger \cdot \psi.
\end{aligned}
$$

The proof is the promotion theorem:

$$
\begin{aligned}
&(\!|\varphi|\!)_\dagger \cdot \mathtt{in} \cdot \psi \\
&= \\
&\varphi \cdot (\!|\varphi|\!)\dagger \cdot \psi \\
&= \\
&\varphi \cdot \psi \cdot (\!|\varphi|\!)\ddagger,
\end{aligned}
$$

hence by the promotion law for catamorphisms we have $(\!|\varphi|\!)_\dagger \cdot (\!|\mathtt{in} \cdot \psi|\!)_\ddagger = (\!|\varphi \cdot \psi|\!)_\ddagger$.

A nice application of this law is the fact that $reverse \cdot reverse = id$, this directly follows from the fact that $id \mid id \mid \sim \in \dagger \leftarrow \dagger$ and $\sim \cdot \sim = id$.

6.2 An Alternative Theory of Infinite Data Structures

In (MALCOLM, 1989a) infinite data structures were presented using a variant of anamorphisms. In his approach the objects generated by anamorphisms constitute a completely different type than the types reduced by catamorphisms. Technically speaking Malcolm works in the category *Set* where initial algebras are least fixed point of functors while greatest fixed points yield final co-algebras which contain only infinite elements (MANES and ARBIB, 1986). As we have seen however it is quite nice to be able to mix cata- and anamorphisms and also many catamorphisms can be described as anamorphisms. In this section we present an alternative theory of infinite data structures. We will repeat some of Malcolms calculations by building infinite objects using both anamorphisms as well as using the fixed point operator explicitly. It is not clear to us yet when it is better to use the one or the other. As a rule of thumb we have that systematic generation of infinite objects is most naturally described by anamorphisms while building a specific infinite object is done by tying knots with the least fixed point operator.

6.2.1 List of Function Results

The data type of cons-lists are defined as follows:

$$\begin{aligned} X\dagger &= \mathbb{1} \mid \alpha \| X, \\ h\dagger &= id \mid id \| h, \\ (\alpha *, \mathtt{in}) &= \mu\dagger \ where \ \mathtt{in} = [\,] \downarrow \succ\!\!\!+. \end{aligned}$$

As discussed earlier, this type contains finite, partial and infinite lists. Infinite lists can be constructed using the least fixed point operator μ. An example of an infinite list is $zeros = \mu(\lambda zeros.0 \succ\!\!\!+ zeros)$.

List catamorphisms $(\!|\varphi|\!)$ and anamorphisms $[\![\psi]\!]$ are defined as follows

$$\begin{aligned} (\!|\varphi|\!) &\in \beta \leftarrow \alpha *, \\ (\!|\varphi|\!) \cdot \mathtt{in} &= \varphi \cdot (\!|\varphi|\!)\dagger, \\ [\![\psi]\!] &\in \alpha * \leftarrow \beta, \\ \mathtt{out} \cdot [\![\psi]\!] &= [\![\psi]\!]\dagger \cdot \psi. \end{aligned}$$

Let $\varphi = e \downarrow \oplus$, then according to the conventions introduced in section 4.1 we write $(\!|e, \oplus|\!)$. From strictness and by instantiating in and $(\!|e, \oplus|\!)\dagger$ we get the following equations

$$(\!|e, \oplus|\!)\ \bot = \bot,$$
$$(\!|e, \oplus|\!)\ [\,] = e,$$
$$(\!|e, \oplus|\!) \cdot \rightarrowtail = \oplus \cdot id \| (\!|e, \oplus|\!).$$

$(\!|e, \oplus|\!)$ recursively replaces $[\,]$ by e and $\rightarrowtail$ by $\oplus$.

For readers not familiar with the Squiggol notation the above definition probably may look a little daunting. In a more traditional functional programming setting one would write:

$$\alpha* ::= [\,] \mid \alpha \rightarrowtail \alpha*$$

and list catamorphisms are written using pattern matching as:

$$h\ [\,] = e,$$
$$h\ (a \rightarrowtail as) = a \oplus (h\ as).$$

The promotion theorem is instantiated for lists as follows: Let $f \in \gamma \leftarrow \beta$ be a strict function such that $f \cdot \oplus = \otimes \cdot id \| f$, in other words $f\ (a \oplus as) = a \otimes (f\ as)$, then $f \cdot (\!|e, \oplus|\!) = (\!|f\ e, \otimes|\!)$.

Perhaps the most often used homomorphism on lists is *map*; define $f\ddagger = id \mid f \| id$, then for $f \in \beta \leftarrow \alpha$,

$$f* \in \beta* \leftarrow \alpha*,$$
$$f* = (\!|\mathtt{in} \cdot f\ddagger|\!)$$
$$= (\!|[\,], \rightarrowtail \cdot f \| id|\!)$$
$$f* = [\![f\ddagger \cdot \mathtt{out}]\!].$$

In more traditional notation: $f*[\,] = [\,], f*(a \rightarrowtail as) = (f\ a) \rightarrowtail (f*as)$.

Using list promotion it is easy to verify that $f* \cdot g* = (f \cdot g)*$ and $id* = id$; $*$ is a functor.

Another law concerning $*$ that we will need is the following, let f be a strict function, then

$$f \cdot (\!|\varphi|\!) = (\!|\psi|\!) \cdot g* \Leftarrow f \cdot \varphi = \psi \cdot f\dagger \cdot g\ddagger,$$
$$[\![\varphi]\!] \cdot f = g* \cdot [\![\psi]\!] \Leftarrow \varphi \cdot f = g\ddagger \cdot f\dagger \cdot \psi.$$

The law can be proved (for catamorphisms) by fixed point induction using the inclusive predicate $P\ (A, B, C) \equiv f \cdot A = B \cdot C$ where $A = \varphi \xleftarrow{\dagger} \mathtt{out}$, $B = \psi \xleftarrow{\dagger} \mathtt{out}$ and $C = \mathtt{in} \cdot g \xleftarrow{\dagger} \mathtt{out}$.

We have already seen the definition of natural numbers:

$$\begin{aligned} I\!N &::= 0 \\ &\mid \sigma\ I\!N\ . \end{aligned}$$

An $I\!N$ -homomorphism $(\!|e, \varphi|\!) \in \alpha \leftarrow I\!N$ is defined as:

$$\begin{aligned} (\!|e, \varphi|\!)\ 0 &= e, \\ (\!|e, \varphi|\!) \cdot \sigma &= \varphi \cdot (\!|e, \varphi|\!). \end{aligned}$$

A more useful example of an infinite list than *zeros* as given above, is the list of all natural numbers:

$$nats = \mu(\lambda ns.0 \rightarrowtail \sigma * ns).$$

The recursive pattern occurring in *nats* appears quite often (BIRD, and WADLER, 1988) so it deserves to be defined as a separate higher order function:

$$\begin{aligned} iterate &\in (\alpha * \leftarrow \alpha) \leftarrow (\alpha \leftarrow \alpha), \\ iterate\ f\ x &= \mu(\lambda xs.x \rightarrowtail f * xs). \end{aligned}$$

The first n elements of $iterate\ f\ x$ can be computed in $\mathcal{O}(n)$ steps assuming that $f\ x$ can be computed in $\mathcal{O}(1)$ steps. It is also possible to specify *iterate* by means of an anamorphism:

$$\begin{aligned} iterate'\ f &= [\![(1,) \cdot id \Uparrow f]\!] \\ &= \mu(\lambda h. \rightarrowtail \cdot id \Uparrow (h \cdot f)). \end{aligned}$$

(Remember that $(1,)$ is the right injection function). This anamorphism also builds the infinite list $x \rightarrowtail f\ x \rightarrowtail (f \cdot f)\ x \rightarrowtail \ldots$ without building a cyclic list but by using an accumulating argument. A big difference between *iterate* and $iterate'$ is that $iterate'$ is by definition strict in its second argument.

Now let $g \cdot f = h \cdot g$ then

$$g * \cdot iterate\ f = iterate\ h \cdot g.$$

Assuming that $(g \cdot f^n)\ x$ takes $\mathcal{O}(n)$ steps, this law turns an $\mathcal{O}(n^2)$ algorithm into an $\mathcal{O}(n)$ algorithm. Using FPPT2 we can calculate:

$$
\begin{array}{l}
(g* \cdot \mathit{iterate}\ f)\ x = (\mathit{iterate}\ h \cdot g)\ x \\
\equiv \\
g*\ \mu(\lambda xs.x \rightarrowtail f*xs) = \mu(\lambda xs.g\ x \rightarrowtail h*xs) \\
\Leftarrow \{\text{FPPT2}\} \\
g*\ (x \rightarrowtail f*xs) = g\ x \rightarrowtail (h \cdot g)*xs \\
\equiv \\
g\ x \rightarrowtail (g \cdot f)*xs = g\ x \rightarrowtail (h \cdot g)*xs \\
\equiv \\
g\ x \rightarrowtail (h \cdot g)*xs = g\ x \rightarrowtail (h \cdot g)*xs \\
\equiv \\
\mathit{true}.
\end{array}
$$

For strict g it is also possible to apply the law we derived for maps since we have:

$$
\begin{array}{l}
(1,) \cdot \mathit{id} \Uparrow f \cdot g \\
= \\
(1,) \cdot g \Uparrow (f \cdot g) \\
= \\
(1,) \cdot g \Uparrow (g \cdot h) \\
= \\
(1,) \cdot g \| g \cdot \mathit{id} \Uparrow h \\
= \\
g\ddagger \cdot g\dagger \cdot (1,) \cdot \mathit{id} \Uparrow h.
\end{array}
$$

The following corollary may be used to derive an efficient computation of the list of function results. Let $f = (\!|e, \varphi|\!)$ be a $I\!N$ -homomorphism, then $f*\mathit{nats} = \mathit{iterate}\ e\ \varphi$. Informally it states that instead of first building an intermediate list of naturals, and then replacing each occurrence of 0 by e and σ by φ in each natural in that list, we might as well directly build a list of β's built of e and φ.

6.2.2 Initial Segments

Besides the recursive pattern $\ldots \rightarrowtail f^i\ x \rightarrowtail \ldots$ generated by $\mathit{iterate}\ f\ x$, the pattern $a \rightarrowtail a \oplus a_0 \rightarrowtail (a \oplus a_0) \oplus a_1 \rightarrowtail \ldots$ also appears quite often in

practice. The function *scan* takes a possibly infinite list $a_0 \succ a_1 \succ \ldots$ into the latter sequence given a seed a and an operator $\oplus$

$$scan\ (\oplus)\ a\ as = \mu(\lambda xs.a \succ (xs \underset{\oplus}{Y} as)).$$

Informally $\underset{\oplus}{Y}$ (*zip*, think of a 'zipper') is defined as:

$$(a_0 \succ a_1 \succ \ldots) \underset{\oplus}{Y} (b_0 \succ b_1 \succ \ldots) = a_0 \oplus b_0 \succ a_1 \oplus b_1 \succ \ldots .$$

Just like *iterate* the first n elements of $scan\ (\oplus)\ a\ as$ can be computed in $\mathcal{O}(n)$ steps assuming that $x \oplus y$ can be computed in $\mathcal{O}(1)$ steps. An application of *scan* is the function *inits* which returns the list of the reversed initial segments of a cons-list

$$inits \in \alpha** \leftarrow \alpha*,$$
$$inits = scan\ (\tilde{\succ})\ [\,].$$

To get the initial segments in the right order one could use

$$scan\ (\tilde{\oplus})\ [\,]\ where\ a \oplus l = (\!|a \succ [\,], \succ|\!)\ l.$$

The operator $x \oplus l$ appends the element x at the end of the list l.

In what follows we need the following *zip promotion law* JEURING, 1989): let $f \cdot \oplus = \otimes \cdot g \| h$ and g, h strict, then

$$f* \cdot \underset{\oplus}{Y} = \underset{\otimes}{Y} \cdot g* \| h*.$$

Using zip and fixed point promotion it is not difficult to prove:

$$(\!|e, \oplus|\!)* \cdot inits = scan\ (\oplus)\ e.$$

Again an $\mathcal{O}(n^2)$ algorithm is turned into a linear one. If f is strict we get the following theorem for free:

$$inits \cdot f* = f** \cdot inits.$$

Now if $f \cdot \oplus = \otimes \cdot g \| f$ and f, g strict, we can show that

$$f* \cdot scan\ (\oplus)\ e = scan\ (\otimes)\ (f\ e) \cdot g*.$$

6.3 Semantics Directed Compiler Generation

The last introductory application of the FPPT will be the semantics of flowcharts REYNOLDS, 1977). The abstract syntax of flowcharts is defined as follows:

$$\begin{aligned} E, F \in prog ::= & \texttt{skip} \\ & \mid var := expr \\ & \mid prog \ ; \ prog \\ & \mid expr \texttt{ --> } prog \texttt{ [] } prog \end{aligned}$$

The meaning of a flowchart program is given by the catamorphism $\mathcal{M}[\![\_]\!] \in success \leftarrow prog$ where

$$\begin{aligned} \eta \in state &== var \rightarrow I\!N \\ statetransf &== state \leftarrow state \\ \mathcal{M}[\![\_]\!] &= (\!|\ \texttt{SKIP}, :=, ;, \_ \rightarrow \_, \_\ |\!) \\ \texttt{SKIP}\ \eta &= \eta \\ (x := e)\ \eta &= \eta[x := e\ \eta] \\ ; &= \tilde{\circ} \end{aligned}$$

assuming that $\mathcal{E}[\![\_]\!] \in expr \rightarrow (I\!N \leftarrow state)$ is given separately. We model sequential composition as strict composition for the following reason. Let E be a $prog$ with $\mathcal{M}[\![E]\!]\ \eta = \bot$, then we want the $\mathcal{E}[\![E;F]\!]$ to denote $\bot$ as well, regardless of the meaning of F.

Although $prog$ does not contain a syntactic recursion operator, we can build circular (recursive) programs and get recursion on the semantic level as well

$$\begin{aligned} & fac = \texttt{n := read ; r := 1 ; } fac', \\ & fac' = \mu(\lambda fac'.\texttt{n = 0 --> skip [] (r := r*n ; n := n-1 ; } fac')). \end{aligned}$$

The FPPT shows that the meaning of the above program is the following recursive function:

$$\begin{aligned} & fac = n := \texttt{READ} \ ; r := 1 \ ; fac', \\ & fac' = \mu(\lambda fac'.n = 0 \rightarrow \texttt{SKIP}, (r := r * n \ ; n := n - 1 \ ; fac')). \end{aligned}$$

For an arbitrary circular program $\mu(\lambda f \in prog.E[f])$ where E is an expression built from the constructors of $prog$ we have $\mathcal{M}[\![E[f]]\!] = E'[\mathcal{M}[\![f]]\!]$ and thus by FPPT2:

$$\mathcal{M}[\![\mu(\lambda f \in prog.E[f])]\!] = \mu(\lambda f \in statetransf.E'[f]).$$

The μ in the rhs shows that we have 'semantic recursion'.

6.3.1 Continuation Semantics

It is remarkably simple to transform the above direct semantics into a *continuation* semantics; just define $Cont\ e\ \sigma = e\ ;\ \sigma$, and calculate:

$$\begin{aligned} & Cont\ (e\ ;\ f)\ \sigma \\ & = \\ & e\ ;\ f\ ;\ \sigma \\ & = \\ & (Cont\ e \cdot Cont\ f)\ \sigma. \end{aligned}$$

Hence $Cont \cdot (;) = (\cdot) \cdot Cont||Cont$. Similarly it follows that

$$\begin{aligned} Cont\ \bot &= \bot, \\ Cont \cdot (\_ \to \_, \_) &= (\_ \to \_, \_) \cdot id||Cont||Cont. \end{aligned}$$

For the elementary actions we show the derivation of the new semantic function $:\grave{=}$ from the old one $:\acute{=}$:

$$\begin{aligned} & Cont\ (x{:}\acute{=}e)\ \sigma\ \eta \\ & = \\ & ((x{:}\acute{=}e)\ ;\ \sigma)\ \eta \\ & = \\ & \sigma\ \eta[x := e\ \eta] \\ & = \\ & (x{:}\grave{=}e)\ \sigma\ \eta. \end{aligned}$$

Similarly we can derive $\grave{\texttt{SKIP}}\ \sigma = \sigma$. Hence $Cont$ is a homomorphisms mapping direct semantics into continuation semantics.The promotion law shows that

$$\grave{\mathcal{M}}[\![\ ]\!] = Cont \cdot \acute{\mathcal{M}}[\![\ ]\!] = (\!|\grave{\texttt{SKIP}}, :\grave{=}, \cdot, \_ \to \_, \_|\!) \in statetransf \leftarrow statetransf.$$

We can go from the continuation semantics to the direct semantics by means of $Dir\ e = e\ id$. It is straightforward to prove that $(Dir \cdot Cont) = id$;

so the continuation semantics is a correct implementation of the direct semantics.

Acknowledgements

We would like to thank John-Jules Meyer, Hans Meijer and the members of the STOP Algorithmics club, especially Maarten Fokkinga, for helpful discussions on the topics treated here.

References

BACKHOUSE, R. (1988): An Exploration of BMF. Technical Report CS 8810, RUG.

BIRD, R. (1976): Programs and Machines: An Introduction to the Theory of Computation. Wiley, 1976.

BIRD, R. (1988): Constructive Functional Programming. In *Marktoberdorf International Summer School on Constructive Methods in Computer Science.*

BIRD, R. – WADLER P. (1988): Introduction to Functional Programming. Prentice-Hall, 1988.

DE BRUIN, A – DE VINK, E.P. (1989): Retractions in Comparing Prolog Semantics. In *Computer Science in the Netherlands 1989,*, pp. 71–90. SION.

FOKKINGA, M. M. (1990) personal communication.

FOKKINGA, M. M. (1990): Homo- and Catamorphisms, Reductions and Maps, an Overview. *STOP Algorithmics Internal Note.* February 1990.

FOKKINGA, M. – MEIJER, E. (1991) Program calculation properties of continnous algebras. *Techmical report 1991* 4. CWI Amsterdam. University of Nijmegen, CWI.

GOGUEN, J.A. – THATCHER, J.W. – WAGNER, E.G. – WRIGHT, J.B. (1977): Initial Algebra Semantics and Continuous Algebras. *JACM*, vol 24(1) pp. 68–95.

GUNTER, C. – MOSSES, P. – SCOTT, D. Semantic Domains and Denotational Semantics. In *Marktoberdorf International Summer School on Logic, Algebra and Computation*, 1989. to appear in: Handbook of Theoretical Computer Science, North Holland.

JEURING, J. (1989): Deriving Algorithms on Binary Trees. In *Computer Science in the Netherlands 1989*, pp. 229–249. SION.

MALCOLM. G. (1989a): An Algebraic Approach to Infinite Data Structures. Technical Report CS 8909, RUG.

MALCOLM, G. (1989b): Homomorphisms and Promotability. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction: LNCS 375*, pp. 335–347.

MANES, E. G. – ARBIB, M. A. (1986): Algebraic Approaches to Program Semantics. Springer Verlag.

MEERTENS, L. (1986): Algorithmics — towards Programming as a Mathematical Activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pp. 289–334. North-Holland.

MEERTENS, L. (1989): Constructing a Calculus of Programs. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction: LNCS 375*, pp 66–90.

MEERTENS, L. (1990): Paramorphisms. *STOP Algorithmics Internal Note,* February 1990.

MEYER, J-J. CH. (1985): Programming Calculi Based on Fixed Point Transformations: Semantics and Applications. PhD Thesis, Vrije Universiteit, Amsterdam.

PATERSON, R. (1988): Reasoning about Functional Programs. PhD Thesis, University of Queensland, Brisbane.

REYNOLDS, J. C. (1977): Semantics of the Domain of Flowcharts. *ACM Toplas*, vol. 24(3) pp. 484–503.

SCHMIDT, D. A. (1986): Denotational Semantics. Allyn and Bacon.

STOY, J. E. (1977): Denotational Semantics, The Scott-Strachey Approach to Programming Language Theory. The MIT press.

WADLER, P. (1987): Views: A Way for Pattern Matching to Cohabit with Data Abstraction. Technical Report 34, Programming Methodology Group, University of Göteborg and Chalmers University of Technology, March 1987.

WADLER, P. (1989): Theorems for free ! In *Proc. 1989 ACM Conference on Lisp and Functional Programming*, pp. 347–359.

Address:

Erik MEIJER
Informatics Department
University of Nijmegen
Toernooiveld 1
NL-6525 ED Nijmegen
The Netherlands