Reusable Monadic Semantics of Logic Programs with Arithmetic Predicates

J. E. Labra Gayo J. M. Cueva Lovelle M. C. Luengo Díez
A. Cernuda del Río

Department of Computer Science, University of Oviedo
C/ Calvo Sotelo S/N, 3307, Oviedo, Spain
{labra,cueva,candi,guti}@lsi.uniovi.es

Abstract

We present a combination of modular monadic semantics and generic programming concepts that improves the reusability of semantic specifications.

The computational structure is defined as the composition of several monad transformers, where each monad transformer adds a new notion of computation to a given monad. The abstract syntax is defined as the fixed point of several non-recursive pattern functors. In the case of several syntactic categories, it is possible to define many sorted algebras and n-catamorphisms.

As an application, we combine the kernel of a pure logic programming language with independently specified arithmetic expressions obtaining a logic programming language with arithmetic predicates.

Keywords: Programming Language, Logic programming, Semantic specification, Interpreter, Monad, Reusability

1 Introduction

Traditional denotational semantics was developed with the aim to identify the main notions of programming languages in a formal setting. Although it has been applied to describe a number of simple languages, it has rarely been used in the design of practical ones [26]. Some reasons for this situation could be its lack of modularity and reusability [23]. Reusable monadic semantics is an attempt to solve those problems combining modular monadic semantics with generic programming concepts.

Modular monadic semantics was proposed in [20, 19] where they use monads and monad transformers to separate values from computations and to capture the different notions of computation like environment access, global state, input-output, non-determinism, etc.

In a different context, generic programming [2] has been developed into a complete discipline from the study of the calculational properties of recursive datatypes and patterns.

The combination of modular monadic semantics and generic programming was firstly proposed by L. Duponcheel [6], allowing the independent specification of the abstract syntax, the computational monad and the domain value. Following that approach, we developed a Language Prototyping System [12, 13, 14, 17] where we also apply monadic catamorphisms, which facilitate the separation between recursive evaluation and semantic specification. In [16] we extend our previous work to handle mutually recursive syntactical categories using many sorted algebras and in [15] we apply that work to the semantics of logic programming languages. In this paper we describe the combination of logic programming features with arithmetic predicates which are independently described. The main advantages of this approach are the automatic derivation of an interpreter from the semantic description, as well as the modularity and reusability of the descriptions which allow to obtain a whole programming language from independently specified semantic building blocks.

It is assumed that the reader has some familiarity with a modern functional programming language. Along the paper, we use Haskell notation with some freedom in the use of mathematical symbols and declarations. As an example, the predefined Haskell datatype

$$\textbf{data}\ Either\ a\ b\ =\ Left\ a\ \mid\ Right\ b$$

will be used as

$$\alpha \parallel \beta\ \triangleq\ L\,\alpha\ \mid\ R\,\beta$$

2 Modular Monadic Semantics

The notion of monad was taken by E. Moggi [22] from Category Theory and was later adapted to a functional programming setting by P. Wadler [25].

Definition 1 (Monad) *A monad can be defined as a type constructor* M *with two operations*

$$\begin{aligned} &return\ : \alpha \to \mathsf{M}\alpha \\ &(\gg\!=)\ \ : \mathsf{M}\alpha \to (\alpha \to \mathsf{M}\beta) \to \mathsf{M}\beta \end{aligned}$$

which satisfy

$$\begin{aligned} &c \gg\!= return \quad\ \equiv c \\ &(return\ a) \gg\!= k\ \equiv k\ a \\ &(m \gg\!= f) \gg\!= h\ \equiv m \gg\!= (\lambda a.f\ a \gg\!= h) \end{aligned}$$

A monad M encapsulates the intuitive notion of computation where $\mathsf{M}\,\alpha$ can be considered as a computation M that returns a value of type α. In Haskell, monads can be defined using constructor classes [9] and it is also possible to use first-class polymorphism [10]. In the rest of the paper, we simply define the type constructor

and the corresponding operations. In this paper, we will also use the operator ($\gg$) defined as

$$\begin{array}{ll}(\gg) & : \mathsf{M}\,\alpha \to \mathsf{M}\,\beta \to \mathsf{M}\,\beta \\ c_1 \gg c_2 & = c_1 \gg\!= \lambda x.c_2\end{array}$$

Example 1 *The simplest monad is the identity monad*

$$\begin{array}{ll}\mathsf{Id}\mathsf{M}\,\alpha & \triangleq \alpha \\ return & = \lambda x.x \\ m \gg\!= f & = f\ x\end{array}$$

It is possible to define monads that capture different kinds of computations, like partiality, nondeterminism, side-effects, exceptions, backtracking, continuations, interactions, etc. [22, 3].

Example 2 *The environment reader monad adds the following operations*

$$\begin{array}{l}rdEnv\ : \mathsf{M}\ Env \\ inEnv\ : Env \to \mathsf{M}\,\alpha \to \mathsf{M}\,\alpha\end{array}$$

Example 3 *The state transformer monad adds the operations*

$$\begin{array}{ll}update & : (State \to State) \to \mathsf{M}\ State \\ fetch & : \mathsf{M}\ State \\ set & : State \to \mathsf{M}\ State\end{array}$$

Example 4 *The backtracking monad adds two operations to handle backtracking*

$$\begin{array}{lll}failure & : \mathsf{M}\,\alpha & \text{— } failure \text{ failing computation} \\ (\Cup) & : \mathsf{M}\,\alpha \to \mathsf{M}\,\alpha \to \mathsf{M}\,\alpha & \text{— } m_1 \Cup m_2 \text{ executes } m_1\text{, if it fails, executes } m_2\end{array}$$

All of the above kinds of monads must satisfy a number of observational laws, which are described in more detail in [19, 7, 18].

When describing the semantics of a programming language using monads, the main problem is the combination of different classes of monads. In general, it is not possible to compose two monads to obtain a new monad [11]. Nevertheless, a monad transformer $\mathcal{T}$ can transform a given monad M into a new monad $\mathcal{T}_{\mathsf{M}}$ that has new operations and maintains the operations of M. The idea of monad transformer is based on the notion of monad morphism that appeared in Moggi's work [22] and was later proposed in [20].

Definition 2 (Monad transformer) *A monad transformer is a type constructor* $\mathcal{T}$ *with an associated operation* $lift : \mathsf{M}\,\alpha \to \mathcal{T}_{\mathsf{M}}\,\alpha$ *that transforms a monad* M *into a new monad* $\mathcal{T}_{\mathsf{M}}$ *and satisfies*

$$\begin{aligned} lift\ .\ return_{\mathsf{M}} \quad &= return_{\mathcal{T}_{\mathsf{M}}} \\ lift\ (m \gg\!=_{\mathsf{M}} k) \ &= (lift\ m) \gg\!=_{\mathcal{T}_{\mathsf{M}}} (lift\ .\ k) \end{aligned}$$

When defining a monad transformer $\mathcal{T}$, it is necessary to specify the operations *return*, $(\gg\!=)$, *lift* and the specific operations that the monad transformer adds. The definition of monad transformers is not straightforward because there can be some interactions between the intervening operations of the different monads. These interactions are considered in more detail in [19, 20] and in [7] it is shown how to derive a backtracking monad transformer from its specification. In the rest of the paper we suppose that we have defined three monad transformers: $\mathcal{T}_{Env}$ transforms any monad into an environment reader monad, $\mathcal{T}_{State}$ transforms any monad into a state transformer monad, and $\mathcal{T}_{Back}$ transforms any monad into a backtracking monad. These definitions can be found in [17, 16, 15].

3 Arithmetic Expressions

In this section, we present the semantics of a simple arithmetic expressions language. The presentation is done in an incremental way. Firstly, we specify terms (constants and additions) and secondly, we add factors (multiplications). At the same time, we introduce the basic concepts of functor, algebras and catamorphisms.

3.1 Extensible abstract syntax

Functors allow the extensible definition of the abstract syntax.

Definition 3 *A functor* F *can be defined as a type constructor that transforms values of type* α *into values of type* $\mathsf{F}\,\alpha$ *and a function*
$map_{\mathsf{F}} : (\alpha \to \beta) \to \mathsf{F}\,\alpha \to \mathsf{F}\,\beta$ *which preserves identities and composition.*

The fixed point of a functor F can be defined as

$$\mu\mathsf{F} \triangleq In\ (\mathsf{F}\ (\mu\mathsf{F}))$$

A recursive datatype can be defined as the fixed point of a non-recursive functor that captures its shape.

Example 5 *The following inductive datatype for arithmetic expressions*

$$Term \triangleq N\ Int \mid Term + Term$$

can be defined as the fixed point of the functor T

$$\mathsf{T}\ x \quad \triangleq N\ Int \mid x + x$$

$$Term \triangleq \mu\mathsf{T}$$

where map_{T} *is*[1]

$$\begin{array}{ll} map_{\mathsf{T}} & : (\alpha \to \beta) \to (\mathsf{T}\,\alpha \to \mathsf{T}\beta) \\ map_{\mathsf{T}}\, f\, (N\ n) & = n \\ map_{\mathsf{T}}\, f\, (x_1 + x_2) & = f\ x_1 + f\ x_2 \end{array}$$

Definition 4 (Sum of two functors) *The sum of two functors* F *and* G, $\mathsf{F} \oplus \mathsf{G}$ *is defined as*

$$(\mathsf{F} \oplus \mathsf{G})\, x \triangleq \mathsf{F}\, x \parallel \mathsf{G}\, x$$

where $map_{\mathsf{F} \oplus \mathsf{G}}$ *is*

$$\begin{array}{ll} map_{\mathsf{F} \oplus \mathsf{G}} & : (\alpha \to \beta) \to (\mathsf{F} \oplus \mathsf{G})\, \alpha \to (\mathsf{F} \oplus \mathsf{G})\, \beta \\ map_{\mathsf{F} \oplus \mathsf{G}}\, f\, (L\, x) & = L\, (map_{\mathsf{F}}\, f\ x) \\ map_{\mathsf{F} \oplus \mathsf{G}}\, f\, (R\, x) & = R\, (map_{\mathsf{G}}\, f\ x) \end{array}$$

Using the sum of two functors, it is possible to extend recursive datatypes.

Example 6 *We can define a new pattern functor for variables*

$$\mathsf{F}\, x = x \times x$$

and the composed recursive datatype of terms and factors can easily be defined as

$$Expr \triangleq \mu(\mathsf{T} \oplus \mathsf{F})$$

3.2 Reusable Semantic Specification

Definition 5 (F-Algebra) *Given a functor* F, *an* F*-algebra is a function*

$$\varphi_{\mathsf{F}} : \mathsf{F}\, \alpha \to \alpha$$

where α *is called the carrier.*

Definition 6 (Homomorphism between F-algebras) *A homomorphism between two* F*-algebras* $\varphi : \mathsf{F}\, \alpha \to \alpha$ *and* $\psi : \mathsf{F}\, \beta \to \beta$ *is a function* $h : \alpha \to \beta$ *which satisfies*

$$h\, .\, \varphi = \psi\, .\, map_{\mathsf{F}}\, h$$

[1] In the rest of the paper, we omit the definition of *map* functions as they can automatically be derived from the shape of the functor.

We consider a category with F-algebras as objects and homomorphisms between F-algebras as morphisms. In this category, $In : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F}$ is an initial object, i.e. for any F-algebra $\varphi : \mathsf{F}\,\alpha \to \alpha$ there is a unique homomorphism $(\!|\varphi|\!) : \mu\mathsf{F} \to \alpha$ satisfying the above equation.

$(\!|\varphi|\!)$ is called *fold* or *catamorphism* and satisfies a number of calculational properties [2, 4]. It can be defined as:

$$
\begin{array}{ll}
(\!|\_|\!) & : \ (\mathsf{F}\alpha \to \alpha) \to (\mu\mathsf{F} \to \alpha) \\
(\!|\varphi|\!)\ (In\ x) & = \varphi\ (\ map_{\mathsf{F}}\ (\!|\varphi|\!)\ x)
\end{array}
$$

Example 7 *We can obtain a simple evaluator for arithmetic expressions defining a* T*-algebra whose carrier is the type* $\mathsf{M}\ Int$*, where* M *is, in this case, any kind of monad.*

$$
\begin{array}{lcl}
\varphi_{\mathsf{T}} & : & \mathsf{T}(\mathsf{M}\ Int) \to \mathsf{M}\ Int \\
\varphi_{\mathsf{T}}\ (Num\ n) & = & return\ (\uparrow n) \\
\varphi_{\mathsf{T}}\ (e_1\ +\ e_2) & = & e_1 \gg\!= \lambda v_1 . e_2 \gg\!= \lambda v_2 . return(v_1\ +\ v_2)
\end{array}
$$

Applying a catamorphism over φ_{T} *we obtain an interpreter for terms:*

$$
\begin{array}{lcl}
\mathsf{Inter}_{Term} & : & Term\ \to\ \mathsf{M}\ Int \\
\mathsf{Inter}_{Term} & = & (\!|\varphi_{\mathsf{T}}|\!)
\end{array}
$$

The operator $\oplus$ allows to obtain a $(\mathsf{F} \oplus \mathsf{G})$-algebra from an F-algebra φ and a G-algebra ψ

$$
\begin{array}{l}
\oplus : \ (\mathsf{F}\ \alpha \to \alpha) \to (\mathsf{G}\ \alpha \to \alpha) \to (\mathsf{F} \oplus \mathsf{G})\alpha \to \alpha \\
(\varphi \oplus \psi)(L\ x)\ =\ \varphi\ x \\
(\varphi \oplus \psi)(R\ x)\ =\ \psi\ x
\end{array}
$$

Example 8 *The above definition allows to extend the evaluator of example 7 to terms and factors without modifying previous definitions. We only specify the semantics of variables with the following* F*-algebra*

$$
\begin{array}{ll}
\varphi_{\mathsf{F}} & : \ \mathsf{F}(\mathsf{M}\ Int) \to \mathsf{M}\ Int \\
\varphi_{\mathsf{F}}\ (e_1\ \times\ e_2) & = e_1 \gg\!= \lambda v_1 . e_2 \gg\!= \lambda v_2 . return(v_1\ \times\ v_2)
\end{array}
$$

And the new interpreter of expressions is automatically obtained as:

$$
\begin{array}{lcl}
\mathsf{Inter}_{Expr} & : & Expr \to \mathsf{M}\ Int \\
\mathsf{Inter}_{Expr} & = & (\!|\varphi_{\mathsf{T}} \oplus \varphi_{\mathsf{F}}|\!)
\end{array}
$$

The theory of catamorphisms can be extended to monadic catamorphisms [12, 14, 17] which allow to separate the recursive evaluation from the semantic specification.

4 Pure Logic Programming Language

4.1 Term representation

Prolog terms are defined as

$$\begin{array}{lll} \mathsf{Term} & = C\ Name & \text{— Constants} \\ & \mid\ V\ Name & \text{— Variables} \\ & \mid\ F\ Name\ [Term] & \text{— Compound terms} \end{array}$$

Facts and rules will be represented as local declarations, leaving the goal as an executable expression. We will use the functor P to capture the abstract syntax of the language. Our abstract syntax assumes all predicates to be unary, this simplifies the definition of the semantics without loss of generality.

$$\begin{array}{lll} \mathsf{P}\ g & = \mathbf{Def}\ Name\ Name\ g\ g & \text{— Definitions} \\ & \mid\ g\ \wedge\ g & \text{— Conjunction} \\ & \mid\ g\ \vee\ g & \text{— Disjunction} \\ & \mid\ \exists(Name \rightarrow\ g) & \text{— Free variables} \\ & \mid\ call\ Name\ Term & \text{— Predicate call} \\ & \mid\ Term \stackrel{\circ}{=} Term & \text{— Unification} \\ & \mid\ ?Name\ (Name \rightarrow g) & \text{— Goal} \end{array}$$

Example 9 *The Prolog program*

$$\begin{array}{l} p(a). \\ p(f(x)) \leftarrow\ p(x) \end{array}$$

with the goal $?\ p(x)$ *could be codified as*

$$\mathbf{Def}\ p\ v\ (v \stackrel{\circ}{=} a \vee \exists(\lambda x.v \stackrel{\circ}{=} f(x)\ \wedge\ call\ p\ x))\ (?x(\lambda x.call\ p\ x))$$

4.2 Computational Structure

The computational structure will be described by means of a monad, which must support the different operations needed. In this sample language, we need to capture backtracking, environment access and global state modification. The global state in this simple case is only needed as a supply of fresh variable names. The resulting monad will be

$$\mathsf{Comp}\ =\ (\mathcal{T}_{Back}\ .\ \mathcal{T}_{Env}\ .\ \mathcal{T}_{State}\ )\ IO$$

we used the predefined IO monad as the base monad in order to facilitate the communication of solutions to the user. We use the following domains

$$\begin{array}{lll} Database & \triangleq\ Name \rightarrow (Name, \mathsf{Comp}\ Subst) & \text{— Clause Definitions} \\ Env & \triangleq\ (Database, Subst) & \text{— Environment} \end{array}$$

$State \quad \triangleq \; Int$ — Global state

We suppose that we have the following operations to lookup and update values in the database.

$$lkp_{DB} \; : Database \to Name \to (Name, \mathsf{Comp}\; Subst)$$
$$upd_{DB} \; : Database \to Name \to (Name, \mathsf{Comp}\; Subst) \to Database$$

4.3 Unification

In this section we present an algorithm adapted from [8] where a polytipic unification algorithm is developed. Genericity is obtained through the definition of type classes and the corresponding instance declarations. We omit those declarations for brevity and just assume that we have the following functions:

$isVar \; : Term \to Bool$ — Checks if a term is a variable
$topEq \; : Term \to Term \to Bool$ — Checks top equality of two terms
$args \; : Term \to [Term]$ — list of arguments of a term

A substitution could be represented as an abstract datatype *Subst* with the following operations:

$lkp_S \; : \; Name \to Subst \to Maybe\; Term$ — lookup
$upd_S \; : \; Name \to Term \to Subst \to Subst$ — update

where *Maybe* is the predefined datatype which could be defined as:

$$Maybe\; \alpha \triangleq Just\; \alpha \mid Nothing$$

The unification algorithm will be:

$$\begin{array}{l}
unify_S \quad\quad : Term \to Term \to Subst \to \mathsf{Comp}\; Subst \\
unify_S\; t_1\; t_2\; \sigma \mid isVar\; t_1 \wedge isVar\; t_2 \wedge t_1 == t_2 = return\; \sigma \\
\qquad\qquad\quad \mid isVar\; t_1 = bind\; t_1\; t_2\; \sigma \\
\qquad\qquad\quad \mid isVar\; t_2 = bind\; t_2\; t_1\; \sigma \\
\qquad\qquad\quad \mid topEq\; t_1\; t_2 = uniTs\; t_1\; t_2\; \sigma \\
\qquad\qquad\quad \mid \mathbf{otherwise} = failure
\end{array}$$

$$\begin{array}{l}
uniTs \; : \; Term \to Term \to Subst \to \mathsf{Comp}\; Subst \\
uniTs\; t_1\; t_2\; \sigma = foldr\; f\; (return\; \sigma)\; (zip\; (args\; t_1)\; (args\; t_2)) \\
\quad \mathbf{where} \\
\qquad f\; (a_1, a_2)\; r = r \gg\!= \lambda\sigma'.unify_S\; a_1\; a_2\; \sigma'
\end{array}$$

$$\begin{array}{l}
bind \quad\quad : \; Name \to Term \to Subst \to \mathsf{Comp}\; Subst \\
bind\; v\; t\; \sigma = \mathbf{case}\; lkp_S\; v\; \sigma\; \mathbf{of} \\
\qquad\qquad\quad Nothing \to return\; (upd_S\; v\; t\; \sigma) \\
\qquad\qquad\quad Just\; t' \quad \to unify_S\; t\; t'\; \sigma \gg\!= \lambda\sigma'.return(upd_S\; v\; t\; \sigma')
\end{array}$$

4.4 Semantic specification

The semantic specification consists of the following P-algebra that takes the computational structure $\mathsf{Comp}\ \mathit{Subst}$ as carrier.

$$
\begin{array}{lll}
\varphi_{\mathsf{P}} & : & \mathsf{P}\ (\mathsf{Comp}\ \mathit{Subst}) \to \mathsf{Comp}\ \mathit{Subst} \\
\varphi_{\mathsf{P}}\ (\mathit{Def}\ p\ x\ g_1\ g_2) & = & \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\
 & & \mathit{inEnv}\ (\mathit{upd}_{DB}\ \rho\ p\ (x, g_1))\ g_2 \\
 & & \\
\varphi_{\mathsf{P}}\ (g_1\ \wedge\ g_2) & = & \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\
 & & g_1 \gg\!= \lambda\sigma'. \\
 & & \mathit{inEnv}\ (\rho, \sigma')\ g_2 \\
 & & \\
\varphi_{\mathsf{P}}\ (g_1\ \vee\ g_2) & = & \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\
 & & \mathit{inEnv}\ (\rho, \sigma)\ g_1\ \Cup\ \mathit{inEnv}\ (\rho, \sigma)\ g_2 \\
 & & \\
\varphi_{\mathsf{P}}\ (\exists\, f) & = & \mathit{update}\ (+1) \gg\!= \lambda n \to f\ (\mathit{mkFree}\ n) \\
 & & \\
\varphi_{\mathsf{P}}\ (\mathit{call}\ p\ t) & = & \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\
 & & \mathbf{let}\ (x, g)\ =\ \mathit{lkp}_{DB}\ \rho\ p \\
 & & \mathbf{in}\ \mathit{unify}_S\ (C\ x)\ t\ \sigma \gg\!= \lambda\sigma'.\ \mathit{inEnv}\ (\rho, \sigma')\ g \\
 & & \\
\varphi_{\mathsf{P}}\ (t_1 \stackrel{\circ}{=} t_2) & = & \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\
 & & \mathit{unify}_S\ t_1\ t_2\ \sigma \\
 & & \\
\varphi_{\mathsf{P}}\ (?\ x\ f) & = & \mathit{update}\ (+1) \gg\!= \lambda n. \\
 & & f\ (\mathit{mkFree}\ n) \gg\!= \lambda\sigma. \\
 & & \mathit{putAnswer}\ x\ (\sigma\ v)) \gg\!= \lambda y. \\
 & & \mathit{return}\ \sigma
\end{array}
$$

The following auxiliary definitions have been used[2].

- $\mathit{mkFree} :\ \mathit{Int} \to \mathit{Name}$, creates a fresh variable name from a given integer
- $\mathit{putAnswer} : \mathit{Name} \to\ \mathit{Term} \to \mathsf{Comp}\ ()$, writes the value of a variable and asks the user for more answers.

The Prolog language is defined as the fixed point of P

$$\mathit{Prolog}\ \triangleq\ \mu\mathsf{P}$$

and the interpreter is automatically obtained as a catamorphism

$$
\begin{array}{lll}
\mathsf{Inter}_{\mathit{Prolog}} & : & \mathit{Prolog} \to \mathsf{Comp}\ \mathit{Subst} \\
\mathsf{Inter}_{\mathit{Prolog}} & = & (\!|\varphi_{\mathsf{P}}|\!)
\end{array}
$$

[2]The detailed definition of the auxiliary functions could be included in the full paper

5 Prolog + Arithmetic Predicates

Arithmetic predicates open a new semantic world in logic programming languages. Other semantic specifications of Prolog [24, 5] often avoid these predicates as they can interfere with the understanding of the particular aspects of Prolog. In our approach, it is possible to reuse the independent specifications of pure logic programming and arithmetic expressions and combine them to form a new language. In order to combine two syntactic categories (goals and expressions), we will extend previous definitions of algebras to 2-sorted algebras in the following section.

5.1 2-sorted abstract syntax and bicatamorphisms

Definition 7 (Bifunctor) *A bifunctor $\mathbb{F}$ is a type constructor that assigns a type $\mathbb{F}\,\alpha\,\beta$ to a pair of types α and β and an operation*

$$bimap_{\mathbb{F}} : (\alpha \to \gamma) \to (\beta \to \delta) \to (\mathbb{F}\,\alpha\,\beta \to \mathbb{F}\,\gamma\,\delta)$$

The fixed point of two bifunctors $\mathbb{F}$ and $\mathbb{G}$ is a pair of values ($\mu_1\mathbb{F}\mathbb{G}$,$\mu_2\mathbb{F}\mathbb{G}$) that can be defined as:

$$\begin{array}{l} \mu_1\mathbb{F}\mathbb{G} \triangleq In_1\ (\mathbb{F}\ (\mu_1\mathbb{F}\mathbb{G})\ (\mu_2\mathbb{F}\mathbb{G})) \\ \mu_2\mathbb{F}\mathbb{G} \triangleq In_2\ (\mathbb{G}\ (\mu_1\mathbb{F}\mathbb{G})\ (\mu_2\mathbb{F}\mathbb{G})) \end{array}$$

Example 10 *The syntax of a simple imperative language with two mutually recursive syntactical categories, expressions and commands, could be modelled by the following bifunctors.*

$$\begin{array}{l} \mathbb{E}\ e\ c = e + e \mid Num\ Int \mid Var\ Name \\ \mathbb{C}\ e\ c = Name := e \mid c\,;\,c \mid While\ e\ c \end{array}$$

and we can obtain commands as the fixed point of $\mathbb{E}$ and $\mathbb{C}$

$$Comm = \mu_2\,\mathbb{E}\,\mathbb{C}$$

Definition 8 (Two-sorted algebra) *Given two bifunctors $\mathbb{F}$ and $\mathbb{G}$, a two-sorted $\mathbb{F},\mathbb{G}$-algebra is a pair of functions ($\varphi : \mathbb{F}\,\alpha\,\beta \to \alpha$, $\psi : \mathbb{G}\,\alpha\,\beta \to \beta$) where α, β are called the carriers of the two-sorted algebra.*

It is possible to define $\mathbb{F},\mathbb{G}$-homomorphisms and a new category where (In_1, In_2) form the initial object. This allows the definition of bicatamorphisms as:

$$\begin{array}{l} (\!|\_,\_|\!)_1 : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to (\mu_1\mathbb{F}\mathbb{G} \to \alpha) \\ (\!|\varphi,\psi|\!)_1\ (In_1\ x) = \varphi\ (\ bimap_{\mathbb{F}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2\ x) \end{array}$$

$$\begin{array}{l} (\!|\_,\_|\!)_2 : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to (\mu_2\mathbb{F}\mathbb{G} \to \beta) \\ (\!|\varphi,\psi|\!)_2\ (\ In_2\ x) = \psi\ (\ bimap_{\mathbb{G}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2\ x) \end{array}$$

Example 11 *We could define the semantics of the imperative language of example 10 by defining the following two sorted $\mathbb{E},\mathbb{C}$-algebra (see [16] for details).*

$$\begin{aligned}\varphi_{\mathbb{E}} &: \mathbb{E}\,(\mathsf{M}\;Int)\,(\mathsf{M}\;())\to \mathsf{M}\;Int\\ \psi_{\mathbb{C}} &: \mathbb{E}\,(\mathsf{M}\;Int)\,(\mathsf{M}\;())\to \mathsf{M}\;()\end{aligned}$$

And we could automatically obtain the corresponding interpreter as a bicatamorphism.

$$\begin{aligned}\mathsf{Inter}_{Comm} &: Comm\to \mathsf{M}\;()\\ \mathsf{Inter}_{Comm} &= (\!|\varphi_{\mathbb{E}},\psi_{\mathbb{C}}|\!)_2\end{aligned}$$

The sum of two functors will be useful to break the specification of a syntactical category into several parts.

Definition 9 (Sum of two bifunctors) *The sum of two bifunctors $\mathbb{F}$ and $\mathbb{G}$ is a new bifunctor $\mathbb{F}\boxplus\mathbb{G}$*

$$(\mathbb{F}\boxplus\mathbb{G})\;\alpha\;\beta \triangleq \mathbb{F}\;\alpha\;\beta \parallel \mathbb{G}\;\alpha\;\beta$$

where the bimap operator is

$$\begin{aligned}bimap_{\mathbb{F}\boxplus\mathbb{G}} &: (\alpha\to\gamma)\to(\beta\to\delta)\to((\mathbb{F}\boxplus\mathbb{G})\;\alpha\;\beta\to((\mathbb{F}\boxplus\mathbb{G})\;\gamma\;\delta)\\ bimap_{\mathbb{F}\boxplus\mathbb{G}}\;f\;g\;(L\;x) &= L\;(bimap_{\mathbb{F}\boxplus\mathbb{G}}\;f\;g\;x)\\ bimap_{\mathbb{F}\boxplus\mathbb{G}}\;f\;g\;(R\;x) &= R\;(bimap_{\mathbb{F}\boxplus\mathbb{G}}\;f\;g\;x)\end{aligned}$$

Two-sorted algebras can be extended using the following operators

$$\begin{aligned}&(\boxplus_1) : (\mathbb{F}\;\alpha\;\beta\to\alpha)\to(\mathbb{G}\;\alpha\;\beta\to\alpha)\to\;(\mathbb{F}\boxplus\mathbb{G})\;\alpha\;\beta\to\alpha\\ &(\phi_1\;\boxplus_1\;\phi_2)\;(L\;x) = \phi_1\;x\\ &(\phi_2\;\boxplus_1\;\phi_2)\;(R\;x) = \phi_2\;x\end{aligned}$$

$$\begin{aligned}&(\boxplus_2) : (\mathbb{F}\;\alpha\;\beta\to\beta)\to(\mathbb{G}\;\alpha\;\beta\to\beta)\to\;(\mathbb{F}\boxplus\mathbb{G})\;\alpha\;\beta\to\beta\\ &(\psi_1\;\boxplus_2\;\psi_2)\;(L\;x) = \psi_1\;x\\ &(\psi_2\;\boxplus_2\;\psi_2)\;(R\;x) = \psi_2\;x\end{aligned}$$

5.2 From functors to bifunctors

When specifying several programming languages, it is very important to be able to share common blocks and to reuse the corresponding specifications. In order to reuse specifications made using single-sorted algebras in a two-sorted framework, it is necessary to extend functors to bifunctors.

Given a functor F, we define the bifunctors F^2_1 and F^2_2 as:

$$\begin{aligned}\mathsf{F}^2_1\;\alpha\;\beta &\triangleq \mathsf{F}\;\alpha\\ \mathsf{F}^2_2\;\alpha\;\beta &\triangleq \mathsf{F}\;\beta\end{aligned}$$

where the *bimap* operations are defined as

$$\begin{aligned} &bimap_{\mathsf{F}_1^2}\ f\ g\ x\ = f\ x \\ &bimap_{\mathsf{F}_2^2}\ f\ g\ x\ = g\ x \end{aligned}$$

Given a single sorted algebra, the operators $(\_)_1^2$ and $(\_)_2^2$ obtain the corresponding two-sorted algebras

$$\begin{aligned} &(\_)_1^2\ :\ (\mathsf{F}\ \alpha \to \alpha) \to \mathsf{F}_1^2\ \alpha\ \beta \to \alpha \\ &\varphi_1^2\ x\ =\ \varphi\ x \end{aligned}$$

$$\begin{aligned} &(\_)_2^2\ :\ (\mathsf{F}\ \beta \to \beta) \to \mathsf{F}_2^2\ \alpha\ \beta \to \beta \\ &\varphi_2^2\ x\ =\ \varphi\ x \end{aligned}$$

5.3 Adding Arithmetic Predicates to Prolog

We define the bifunctor $\mathbb{A}$ which captures the predicates *is* and $=:=$[3].

$$\mathbb{A}\ g\ e\ \triangleq\ \mathit{Term}\ \mathbf{is}\ e\ \mid\ e\ =:=\ e$$

The semantic specification is defined as the following two sorted algebra.

$$\begin{aligned} &\varphi_{\mathbb{A}} && :\ \mathbb{A}\ (\mathsf{Comp}\ \mathit{Subst})\ (\mathsf{Comp}\ \mathit{Int}) \to \mathsf{Comp}\ \mathit{Subst} \\ &\varphi_{\mathbb{A}}\ (t\ \mathbf{is}\ e) && = e \gg\!= \lambda v. \\ &&& \quad \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\ &&& \quad \mathit{unify}_S\ t\ (\mathit{cnv}\ v)\ \sigma \\ &\varphi_{\mathbb{A}}\ (e_1\ =:=\ e_2) && = e_1 \gg\!= \lambda v_1. \\ &&& \quad e_2 \gg\!= \lambda v_2. \\ &&& \quad \mathit{rdEnv} \gg\!= \lambda(\rho, \sigma). \\ &&& \quad \mathbf{if}\ v_1 == v_2\ \mathbf{then} \\ &&& \quad\quad \mathit{return}\ \sigma \\ &&& \quad \mathbf{else} \\ &&& \quad\quad \mathit{failure} \end{aligned}$$

where $\mathit{cnv} : \mathit{Int} \to \mathit{Term}$ converts an integer into a constant term.
The extended language can be defined as

$$\mathit{Prolog}^+\ \triangleq\ \mu_1\ (\mathsf{P}_1^2\ \boxplus_1\ \mathbb{A})\ (\mathsf{T} \oplus \mathsf{F})_2^2$$

and the corresponding interpreter is obtained as a bicatamorphism

$$\begin{aligned} &\mathsf{Inter}_{\mathit{Prolog}^+}\ :\ \mathit{Prolog}^+ \to \mathsf{Comp}\ \mathit{Subst} \\ &\mathsf{Inter}_{\mathit{Prolog}^+}\ = (\!|\varphi_{\mathsf{P}_1}^2 \boxplus_1 \varphi_{\mathbb{A}}, (\varphi_{\mathsf{T}} \oplus \varphi_{\mathsf{F}})_2^2|\!)_1 \end{aligned}$$

[3]Other arithmetic predicates could easily be added

6 Conclusions and Future Work

The proposed approach allows the development of reusable semantic specifications of programming languages as an integration of modular monadic semantics and generic programming concepts.

Modular monadic semantics allows to identify the computational structure through the notion of monad. Monads are used to distinguish between values and computations. In the case of logic programming languages, the value is the answer substitution, while the computation encapsulates the notions of backtracking, environment access, fresh name supply, etc. Monads can incrementally be defined using monad transformers, where each transformer adds a given notion of computation. In this way, to add new computational features, like the control mechanisms of Prolog it is only needed to change the corresponding transformer [7].

Generic programming concepts allow the definition of extensible abstract syntax of a programming language identifying the shape of the different entities using non-recursive pattern functors. Semantic specifications are independently defined through algebras that take the computational structure as carrier. This allows to automatically obtain the interpreter as a catamorphism.

The proposed approach has been implemented in a *Language Prototyping System* [1]. The system consists of a domain specific meta-language embedded in Haskell and it also contains an interactive framework for language testing. This approach offers easier development and the fairly good type system of Haskell. Nevertheless, there are some disadvantages like the mixture of error messages between the host language and the metalanguage, Haskell dependency and some type system limitations. We are currently planning to develop an independent meta-language. Some work in this direction has been already done by E. Moggi [21].

The Language Prototyping System has been used to specify imperative, functional, object-oriented and logic programming languages [16, 17, 18, 15]. All the specifications share common blocks, like arithmetic expressions. Future work can be done in the specification of other features and in the integration between different blocks leading to cross-paradigm programming language designs.

References

[1] Language Prototyping System. `http://lsi.uniovi.es/~labra/LPS/LPS.html`, 2001.

[2] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming - an introduction. In S. Swierstra, P. Henriques, and Jose N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*. Springer, 1999.

[3] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *International Summer School On Applied Semantics APPSEM'2000*, Caminha, Portugal, 2000.

[4] R. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[5] E. Börger and D. Rosenzweig. A mathematical definition of full prolog. *Science of Computer Programming*, 1994.

[6] Luc Duponcheel. Writing modular interpreters using catamorphisms, subtypes and monad transformers. Technical Report (Draft), Utrecht University, 1995.

[7] Ralf Hinze. Deriving backtracking monad transformers. In Roland Backhouse and Jose N. Oliveira, editors, *Proceedings of the 2000 International Conference on Functional Programming, Montreal, Canada*, September 2000.

[8] P. Jansson and J. Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.

[9] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[10] Mark P. Jones. First-class Polymorphism with Type Inference. In *Proceedings of the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 15-17 1997.

[11] Mark P. Jones and L. Duponcheel. Composing monads. YALEU/DCS/RR 1004, Yale University, New Haven, CT, USA, 1993.

[12] J. E. Labra. An implementation of modular monadic semantics using folds and monadic folds. In *Workshop on Research Themes on Functional Programming, Third International Summer School on Advanced Functional Programming*, Braga - Portugal, 1998.

[13] J. E. Labra, J. M. Cueva, and C. Luengo. Language prototyping using modular monadic semantics. In *3rd Latin-American Conference on Functional Programming*, Recife - Brazil, March 1999. Available at `http://lsi.uniovi.es/~labra/LPS/Clapf99.ps`.

[14] J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. Modular development of interpreters from semantic building blocks. *Nordic Journal of Computing*, 8(3), 2001. To appear.

[15] J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. Specification of logic programming languages from reusable semantic building blocks. In *International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, September 2001. University of Kiel. To appear.

[16] J. E. Labra, J. M. Cueva Lovelle, M. C. Luengo Díez, and B. M. González. A language prototyping tool based on semantic building blocks. In *Eight International Conference on Computer Aided Systems Theory and Technology (EUROCAST'01)*, volume 2178 of *Lecture Notes in Computer Science*, Las Palmas de Gran Canaria – Spain, February 2001. Springer Verlag.

[17] J.E. Labra, M.C. Luengo, J.M. Cueva, and A. Cernuda. LPS: A language prototyping system using modular monadic semantics. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[18] Jose E. Labra. *Modular Development of Language Processors from Reusable Semantic Specifications*. PhD thesis, Dept. of Computer Science, University of Oviedo, 2001. In spanish.

[19] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.

[20] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages, San Francisco, CA*. ACM, January 1995.

[21] E. Moggi. Metalanguages and applications. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute. Cambridge University Press, 1997.

[22] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Dept. of Computer Science, June 1989. Lecture Notes for course CS 359, Stanford University.

[23] Peter D. Mosses. Theory and practice of action semantics. In *21st Int. Symp. on Mathematical Foundations of Computer Science*, volume 1113, pages 37–61, Cracow, Poland, Sept 1996. Lecture Notes in Computer Science, Springer-Verlag.

[24] T. Nicholson and N. Foo. A denotational semantics for prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, 1989.

[25] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 19 – 22, 1992. ACM Press.

[26] David A. Watt. Why don't programming language designers use formal methods? In *Seminario Integrado de Software e Hardware - SEMISH'96*, pages 1–6, Recife, Brazil, 1996. University of Pernambuco.