Electronic Notes in Theoretical Computer Science 64 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume64.html 14 pages

Specification of Logic Programming Languages from Reusable Semantic Building Blocks

J. E. Labra Gayo M. C. Luengo Díez J. M. Cueva Lovelle
A. Cernuda del Río [1]

Department of Computer Science, University of Oviedo, C/ Calvo Sotelo S/N, CP 33007, Oviedo, Spain

Abstract

We present a Language Prototyping System that facilitates the modular development of interpreters from independent semantic building blocks. The abstract syntax is modelled as the fixpoint of a pattern functor which can be obtained as the sum of functors. For each functor we define an algebra whose carrier is the computational structure. This structure is obtained as the composition of several monad transformers applied to a base monad, where each monad transformer adds a new notion of computation. When the abstract syntax is composed from mutually recursive categories, we use many-sorted algebras. With this approach, the prototype interpreters are automatically obtained as a catamorphism over the defined algebras.

As an example, in this paper, we independently specify an arithmetic evaluator and a simple logic programming language and combine both specifications to obtain a logic programming language with arithmetic capabilities.

1 Introduction

Monads were applied by E. Moggi [18] to improve the modularity of traditional denotational semantics, capturing the intuitive idea of separating values from computations. P. Wadler [20] popularized the application of monads to the development of modular interpreters and to encapsulate the Input/Output features of the purely functional programming language Haskell. In general, it is not possible to compose two monads to obtain a new monad. However, using monad transformers [16] it is possible to transform a given monad into a new monad adding new computational capabilities. The use of monads and

[1] Email: {labra,candi,cueva,guti}@lsi.uniovi.es

monad transformers to specify the semantics of programming languages was called modular monadic semantics in [15].

In a different context, the definition of recursive datatypes as least fixpoints of pattern functors and the calculating properties that can be obtained by means of folds or catamorphisms led to a complete discipline which could be named as generic programming [2]. Following that approach, L. Duponcheel proposed the combined use of folds or catamorphisms with modular monadic semantics [5] allowing the independent specification of the abstract syntax, the computational monad and the domain value. In [10,13] we also applied monadic catamorphisms, which facilitate the separation between recursive evaluation and semantic specification. In [12] we show that it is possible to apply this approach to model abstract syntax with several categories. That approach was followed to model a logic programming language with arithmetic predicates in [11,14].

There have been several attempts to specify the dynamic semantics of Prolog [19,3]. In [7] it is described an axiomatic semantics with equational logic which will form the basis for the derivation of a backtracking monad transformer [8]. That approach is used in [4] to embed logical variables in Haskell and has been the main inspiration for our encoding of Prolog expressions.

In the paper, it is assumed that the reader has some familiarity with a modern functional programming language. We use Haskell syntax with some freedom in the use of mathematical operators.

2 Modular Monadic Semantics

In functional programming, a monad can be defined as a type constructor M and a pair of polymorphic operations $(\ggg\!=) : \mathsf{M}\,\alpha \to (\alpha \to \mathsf{M}\,\beta) \to \mathsf{M}\,\beta$ and $return : \alpha \to \mathsf{M}\,\alpha$ which satisfy a number of laws. The intuitive idea is that a monad M encapsulates a notion of computation and $\mathsf{M}\,\alpha$ can be considered as a computation M returning a value of type α. In Haskell, the following type class can be used.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ \mathsf{m} \\
\quad \textbf{where} \\
\qquad return : \alpha \ \to\ \mathsf{m}\,\alpha \\
\qquad (\ggg\!=) : \mathsf{m}\,\alpha \to (\alpha \to \mathsf{m}\,\beta) \to \mathsf{m}\,\beta \\
\qquad (\gg) : \mathsf{m}\,\alpha \to \mathsf{m}\,\alpha \to \mathsf{m}\,\alpha \\
\qquad m_1 \gg m_2 \ =\ m_1 \ggg\!= \lambda\_ \to\ m_2
\end{array}
$$

It is possible to define special monads for different notions of computations like exceptions, environment access, state transformers, backtracking, continuations, Input/Output, non-determinism, etc. Each class of monad has some specific operations apart from the predefined *return* and $(\ggg\!=)$. Table 1 contains some classes of monads with their operations.

Name	Operations
Error handling	$err : String \to \mathsf{M}\ \alpha$
Environment Access	$rdEnv : \mathsf{M}\ Env$ $inEnv : Env \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha$
State transformer	$update : (State \to State) \to \mathsf{M}\ State$ $fetch : \mathsf{M}\ State$ $set : State \to \mathsf{M}\ State$
Backtracking	$failure : \mathsf{M}\ \alpha$ $orElse : \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha$

Table 1
Some classes of monads

When describing the semantics of a programming language using monads, the main problem is the combination of different classes of monads. It is not possible to compose two monads to obtain a new monad in general. 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 [18] and was later proposed in [16]. The definition of a monad transformer 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 [15,16] and in [8] it is shown how to derive a backtracking monad transformer from its specification. In Haskell, a monad transformer can be defined using the following multi-parameter type class.

$$
\begin{array}{l}
\textbf{class}\ (Monad\ \mathsf{m}) \Rightarrow MonadT\ \mathcal{T}\ \mathsf{m} \\
\quad \textbf{where} \\
\quad\quad lift : \mathsf{m}\ \alpha \to \mathcal{T}\ \mathsf{m}\ \alpha
\end{array}
$$

Our system contains a library of predefined monad transformers corresponding to each class of monad and the user can also define new monad transformers. When defining a monad transformer $\mathcal{T}$ over a monad M, it is necessary to specify the $return_{\mathcal{T}\ \mathsf{M}}$ and $(\gg\!=_{\mathcal{T}\ \mathsf{M}})$ operations, the $lift : \mathsf{M}\ \alpha \to \mathcal{T}\ \mathsf{M}\ \alpha$ operation transforming any operation in M into an operation in the new monad $\mathcal{T}\mathsf{M}$, and the operations provided for the new monad.

Table 2 presents the definitions of some monad transformers that will be used in the rest of the paper.

Error handling

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{ErrT}\ \mathsf{m}\ \alpha & = & E\ \{\ \mathit{unE}\ :\ \mathsf{m}\ (\mathit{Either}\ \alpha\ \mathit{String})\ \} \\
\mathit{return}\ x & = & E\ (\mathit{return}\ (\mathit{Left}\ x)) \\
m \gg\!= f & = & E\ ((\mathit{unE}\ m) \gg\!= \lambda y.\ \textbf{case}\ y\ \textbf{of} \\
 & & \qquad\qquad\qquad \mathit{Left}\ v \rightarrow \mathit{unE}\ (f\ v) \\
 & & \qquad\qquad\qquad \mathit{Right}\ e \rightarrow \mathit{return}\ (\mathit{Right}\ e) \\
\mathit{lift}\ m & = & E\ (m \gg\!= \lambda x.\mathit{return}\ (\mathit{Left}\ x)) \\
\mathit{err}\ \mathit{msg} & = & E\ (\mathit{return}\ (\mathit{Right}\ \mathit{msg}))
\end{array}
$$

Environment Reader

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{EnvT}\ \ \mathit{Env}\ \mathsf{m}\ \alpha & = & V\ \{\ \mathit{unV}\ :\ \mathit{Env}\ \rightarrow\ \mathsf{m}\ \alpha\ \} \\
\mathit{return}\ x & = & V\ (\lambda\rho.\mathit{return}\ x) \\
m \gg\!= f & = & V\ (\lambda\rho \rightarrow\ \mathit{unV}\ m\ \rho \gg\!= \lambda v.\ \mathit{unV}\ (f\ \ v)\ \rho) \\
\mathit{lift}\ m & = & V\ (\lambda\rho.m \gg\!= \mathit{return}) \\
\mathit{rdEnv} & = & V\ \mathit{return} \\
\mathit{inEnv}\ \rho\ x & = & V(\lambda\_.x\ \ \rho)
\end{array}
$$

State transformer

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{StateT}\ \ \mathit{State}\ \mathsf{m}\ \alpha & = & S\ \{\ \mathit{unS}\ :\ \mathit{State}\ \rightarrow\ \mathsf{m}\ (\alpha, \mathit{State})\ \} \\
\mathit{return}\ x & = & S\ (\lambda\varsigma.\mathit{return}\ (x, \varsigma)) \\
m \gg\!= f & = & S\ (\lambda\varsigma \rightarrow \mathit{unS}\ m\ \varsigma \gg\!= \lambda(v, \varsigma').\ \mathit{unS}\ (f\ \ v)\ \varsigma') \\
\mathit{lift}\ m & = & S\ (\lambda\varsigma.m \gg\!= \lambda x.\ \mathit{return}(x, \varsigma)\ ) \\
\mathit{update}\ f & = & S\ (\lambda\varsigma.\mathit{return}\ (\varsigma,\ f\ \varsigma))
\end{array}
$$

Backtracking

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{BackT}\ \mathsf{m}\ \alpha & = & B\ \{\ \mathit{unB} : \forall\beta.((\alpha \rightarrow \mathsf{m}\ \beta \rightarrow \mathsf{m}\ \beta) \rightarrow \mathsf{m}\ \beta \rightarrow \mathsf{m}\ \beta)\} \\
\mathit{return}\ x & = & B\ (\lambda\kappa.\kappa\ x) \\
m \gg\!= f & = & B\ (\lambda\kappa.(\mathit{unB}\ m)\ (\lambda v.\mathit{unB}\ (f\ \ v)\ \ \kappa)) \\
\mathit{lift}\ m & = & B\ (\lambda\ \kappa\ f\ \ \rightarrow m \gg\!= \lambda x \rightarrow\ \ \kappa\ x\ f) \\
\mathit{failure} & = & B\ (\lambda\kappa \rightarrow (\lambda x \rightarrow x)) \\
\mathit{orElse}\ m\ n & = & B\ (\lambda\kappa\ f \rightarrow\ (\mathit{unB}\ m)\ \kappa\ ((\mathit{unB}\ n)\ \kappa\ f))
\end{array}
$$

Table 2
Some monad transformers with their definitions

3 Generic Programming Concepts

3.1 Functors, Algebras and Catamorphisms

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 $\mathit{map}_{\mathsf{F}} : (\alpha \rightarrow \beta) \rightarrow\ \mathsf{F}\ \alpha\ \rightarrow\ \mathsf{F}\ \beta$.

$$\begin{array}{l} \textbf{class}\ \mathit{Functor}\ \mathsf{f} \\ \quad \textbf{where} \\ \qquad \mathit{map} :\ (\alpha \to \beta) \to \mathsf{f}\,\alpha \to \mathsf{f}\,\beta \end{array}$$

The fixpoint of a functor F can be defined as

$$\textbf{newtype}\ \mathit{Fix}\ \mathsf{F}\ =\ \mathit{In}\ \{\ \mathit{out}\ :\ \mathsf{F}\,(\mathit{Fix}\ \mathsf{F})\ \}$$

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

Example 3.1 The inductive datatype *Term* defined as

$$\mathit{Term}\ \triangleq\ \mathit{Num}\ \mathit{Int}\ \mid\ \mathit{Term}\ +\ \mathit{Term}$$

can be defined as the fixpoint of the functor T

$$\begin{array}{lcl} \textbf{type}\ \mathsf{T}\ x & = & \mathit{Num}\ \mathit{Int}\ \mid\ x\ +\ x \\ \textbf{type}\ \mathit{Term} & = & \mathit{Fix}\ \mathsf{T} \end{array}$$

Given a functor F, an F-algebra is a function $\varphi_{\mathsf{F}} : \mathsf{F}\,\alpha\ \to\ \alpha$ where α is called the carrier. A *fold* or *catamorphism* can be defined as

$$\begin{array}{lcl} \mathit{cata} & : & (\mathit{Functor}\ \mathsf{f}) \Rightarrow (\mathsf{f}\alpha \to \alpha) \to \mathit{Fix}\ \mathsf{f} \to \alpha) \\ \mathit{cata}\ \varphi & = & \varphi\ .\ \mathit{map}\ (\mathit{cata}\ \varphi)\ .\ \mathit{out} \end{array}$$

Example 3.2 We can obtain a simple evaluator for terms defining a T-algebra whose carrier is the type $\mathsf{m}\ \mathit{Int}$, where m is, in this case, any kind of monad.

$$\begin{array}{lcl} \varphi_{\mathsf{T}} & : & (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{T}\,(\mathsf{m}\ \mathit{Int})\ \to\ (\mathsf{m}\ \mathit{Int}) \\ \varphi_{\mathsf{T}}\,(\mathit{Num}\ n) & = & \mathit{return}\ n \\ \varphi_{\mathsf{T}}\,(t_1\ +\ t_2) & = & t_1 \gg\!= \lambda v_1.t_2 \gg\!= \lambda v_2.\mathit{return}(v_1 + v_2) \end{array}$$

An interpreter of arithmetic terms is obtained as a catamorphism

$$\begin{array}{lcl} \mathsf{Inter}_{\mathit{Term}} & : & (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathit{Term}\ \to\ \mathsf{m}\ \mathit{Int} \\ \mathsf{Inter}_{\mathit{Term}} & = & \mathit{cata}\ \varphi_{\mathsf{T}} \end{array}$$

3.2 Two-sorted algebras and catamorphisms

The abstract syntax of a programming language is usually composed from several mutually recursive categories. It is possible to extend the previous definitions to handle many-sorted algebras. In this section, we present the theory for $n = 2$, but it can be defined for any number of sorts [6]. The following definitions will be used in section 5.

A bifunctor f is a type constructor that assigns a type $\mathsf{f}\,\alpha\,\beta$ to a pair of types α and β and an operation *bimap*.

$$
\begin{array}{l}
\textbf{class}\ \mathit{BiFunctor}\ \mathsf{f} \\
\quad \textbf{where} \\
\qquad \mathit{bimap}\ :\ (\alpha \to \gamma) \to (\beta \to \delta) \to \mathsf{f}\,\alpha\,\beta \to \mathsf{f}\,\gamma\,\delta
\end{array}
$$

The fixpoint of two bifunctors f and g is a pair of values ($\mathit{Fix}_1\ \mathsf{f}\ \mathsf{g}$,$\mathit{Fix}_2\ \mathsf{f}\ \mathsf{g}$) that can be defined as:

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Fix}_1\ \mathsf{f}\ \mathsf{g}\ =\ \mathit{In}_1\ \{\ \mathit{out}_1\ :\ \mathsf{f}\ (\mathit{Fix}_1\ \mathsf{f}\ \mathsf{g})\ (\mathit{Fix}_2\ \mathsf{f}\ \mathsf{g})\ \} \\
\textbf{newtype}\ \mathit{Fix}_2\ \mathsf{f}\ \mathsf{g}\ =\ \mathit{In}_2\ \{\ \mathit{out}_2\ :\ \mathsf{g}\ (\mathit{Fix}_1\ \mathsf{f}\ \mathsf{g})\ (\mathit{Fix}_2\ \mathsf{f}\ \mathsf{g})\ \}
\end{array}
$$

Given two bifunctors f and g, a two-sorted f,g-algebra is a pair of functions ($\varphi : \mathsf{f}\,\alpha\,\beta \to \alpha$, $\psi : \mathsf{g}\,\alpha\,\beta \to \beta$) where α, β are called the carriers of the two-sorted algebra.

It is possible to define f,g-homomorphisms and a new category where $(\mathit{In}_1, \mathit{In}_2)$ form the initial object. This allows the definition of bicatamorphisms as:

$$
\begin{array}{lll}
\mathit{cata}^2_1 & : & (\mathit{Bifunctor}\ \mathsf{f}, \mathit{Bifunctor}\ \mathsf{g}) \Rightarrow \\
 & & (\mathsf{f}\,\alpha\,\beta \to \alpha) \to (\mathsf{g}\,\alpha\,\beta \to \beta) \to \mathit{Fix}_1\ \mathsf{f}\ \mathsf{g} \to \alpha \\
\mathit{cata}^2_1\ \varphi\ \psi & = & \varphi\ .\ \mathit{bimap}\ (\mathit{cata}^2_1\ \varphi\ \psi)\ (\mathit{cata}^2_2\ \varphi\ \psi)\ .\ \mathit{out}_1
\end{array}
$$

$$
\begin{array}{lll}
\mathit{cata}^2_2 & : & (\mathit{Bifunctor}\ \mathsf{f}, \mathit{Bifunctor}\ \mathsf{g}) \Rightarrow \\
 & & (\mathsf{f}\,\alpha\,\beta \to \alpha) \to (\mathsf{g}\,\alpha\,\beta \to \beta) \to \mathit{Fix}_2\ \mathsf{f}\ \mathsf{g} \to \beta \\
\mathit{cata}^2_2\ \varphi\ \psi & = & \psi\ .\ \mathit{bimap}\ (\mathit{cata}^2_1\ \varphi\ \psi)\ (\mathit{cata}^2_2\ \varphi\ \psi)\ .\ \mathit{out}_2
\end{array}
$$

The sum of two bifunctors f and g is a new bifunctor $S2\,\mathsf{f}\ \mathsf{g}$

$$
\textbf{newtype}\ S_2\ \mathsf{f}\ \mathsf{g}\,\alpha\,\beta\ =\ S_2\ (\mathit{Either}\ (\mathsf{f}\ \alpha\ \beta)\ (\mathsf{g}\ \alpha\ \beta))
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Bifunctor}\ \mathsf{f},\ \mathit{Bifunctor}\ \mathsf{g}) \Rightarrow\ \mathit{Bifunctor}\ (S_2\ \mathsf{f}\ \mathsf{g}) \\
\quad \textbf{where} \\
\qquad \mathit{bimap}\ f\ g\ (\mathit{Left}\ x)\ \ =\ \mathit{Left}\ (\mathit{bimap}\ f\ g\ x) \\
\qquad \mathit{bimap}\ f\ g\ (\mathit{Right}\ x)\ =\ \mathit{Right}\ (\mathit{bimap}\ f\ g\ x)
\end{array}
$$

Two-sorted algebras can be extended using the following operators

$$
\begin{array}{l}
(\boxplus_1) : (\mathsf{f}\,\alpha\,\beta \to \alpha) \to (\mathsf{g}\,\alpha\,\beta \to \alpha) \to\ S_2\ \mathsf{f}\ \mathsf{g}\,\alpha\,\beta \to \alpha \\
(\phi_1\ \boxplus_1\ \phi_2)\ (S_2\ (\mathit{Left}\ x))\ =\ \phi_1\ x \\
(\phi_2\ \boxplus_1\ \phi_2)\ (S_2\ (\mathit{Right}\ x))\ =\ \phi_2\ x
\end{array}
$$

$$
(\boxplus_2) : (\mathsf{f}\,\alpha\,\beta \to \beta) \to (\mathsf{g}\,\alpha\,\beta \to \beta) \to\ (S_2\ \mathsf{f}\ \mathsf{g})\,\alpha\,\beta \to \beta
$$

$$(\psi_1 \boxplus_2 \psi_2)\,(S_2\,(\mathit{Left}\ x)) \;=\; \psi_1\ x$$
$$(\psi_2 \boxplus_2 \psi_2)\,(S_2\,(\mathit{Right}\ x)) \;=\; \psi_2\ x$$

3.3 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 $P_1^2\,\mathsf{f}$ and $P_2^2\,\mathsf{f}$ as:

$$\textbf{newtype}\ P_1^2\ \mathsf{f}\ \alpha\ \beta \;=\; P_1^2\,(\mathsf{f}\ \alpha)$$
$$\textbf{newtype}\ P_2^2\ \mathsf{f}\ \alpha\ \beta \;=\; P_2^2\,(\mathsf{f}\ \beta)$$

where the *bimap* operations are defined as

$$\textbf{instance}\ \mathit{Functor}\ \mathsf{f} \;\Rightarrow\; \mathit{Bifunctor}\ (P_1^2\ \mathsf{f})$$
$$\quad\textbf{where}$$
$$\qquad \mathit{bimap}\ f\ g\ (P_1^2\ x) \;=\; P_1^2\ (\mathit{map}\ f\ x)$$

$$\textbf{instance}\ \mathit{Functor}\ \mathsf{f} \;\Rightarrow\; \mathit{Bifunctor}\ (P_2^2\ \mathsf{f})$$
$$\quad\textbf{where}$$
$$\qquad \mathit{bimap}\ f\ g\ (P_2^2\ x) \;= P_2^2\ (\mathit{map}\ g\ x)$$

Given a single sorted algebra, the following operators ϵ_1^2 and ϵ_2^2 obtain the corresponding two-sorted algebras

$$\epsilon_1^2\ :\ (\mathsf{f}\ \alpha \to \alpha) \to\ P_1^2\ \mathsf{f}\ \alpha\ \beta \to \alpha$$
$$\epsilon_1^2\ \varphi\ (P_1^2\ x) \;=\; \varphi\ x$$

$$\epsilon_2^2\ :\ (\mathsf{f}\ \beta \to \beta) \to\ P_2^2\ \mathsf{f}\ \alpha\ \beta \to \beta$$
$$\epsilon_2^2\ \varphi\ (P_2^2\ x) \;=\; \varphi\ x$$

4 Specification of Pure Prolog

4.1 Syntactical Structure

Prolog terms are defined as

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

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.

$$
\begin{array}{lllll}
\textbf{data}\,\mathsf{P}\ e & = & \textbf{Def}\ \mathit{Name}\ \mathit{Name}\ e\ e & \text{— Definitions} \\
 & | & e\ \wedge\ e & \text{— Conjunction} \\
 & | & e\ \vee\ e & \text{— Disjunction} \\
 & | & \exists(\mathit{Name} \rightarrow\ e) & \text{— Free variables} \\
 & | & \mathit{call}\ \mathit{Name}\ \mathit{Term} & \text{— Predicate call} \\
 & | & \mathit{Term} \doteq \mathit{Term} & \text{— Unification} \\
 & | & ?\mathit{Name}\ (\mathit{Name} \rightarrow e) & \text{— Goal}
\end{array}
$$

The Prolog language is defined as the fixed point of P

$$\textbf{type}\ \mathit{Prolog}\ =\ \mathit{Fix}\ \mathsf{P}$$

Example 4.1 The Prolog program

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

with the goal $?\,p(x)$ can be codified as

$$\textbf{Def}\ p\ v\ (v \doteq a \vee \exists(\lambda x.v \doteq f(x)\ \wedge\ \mathit{call}\ p\ x))\ (?x(\lambda x.\mathit{call}\ p\ x))$$

4.2 Unification

In this section we present an algorithm adapted from [9] where a polytipic unification algorithm is developed. In that paper, 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:

$$
\begin{array}{lll}
\mathit{isVar} & : \mathit{Term}\ \rightarrow\ \mathit{Bool} & \text{— Checks if a term is a variable} \\
\mathit{topEq} & : \mathit{Term}\ \rightarrow\ \mathit{Term}\ \rightarrow\ \mathit{Bool} & \text{— Checks top equality of two terms} \\
\mathit{args} & : \mathit{Term}\ \rightarrow\ [\mathit{Term}] & \text{— list of arguments of a term}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{lkp}_S & : \mathit{Name} \rightarrow \mathit{Subst} \rightarrow \mathit{Maybe}\ \mathit{Term} & \text{— lookup} \\
\mathit{upd}_S & : \mathit{Name} \rightarrow \mathit{Term} \rightarrow \mathit{Subst} \rightarrow \mathit{Subst} & \text{— update}
\end{array}
$$

where *Maybe* is the predefined datatype:

$$\textbf{data}\ \mathit{Maybe}\ \alpha\ =\ \mathit{Just}\ \alpha\ \mid\ \mathit{Nothing}$$

The unification algorithm will be:

$$\begin{array}{lll}
\mathit{unify}_S & : & \mathit{Term} \to \mathit{Term} \to \mathit{Subst} \to \mathsf{Comp}\ \mathit{Subst} \\
\mathit{unify}_S\ t_1\ t_2\ \sigma & \mid & \mathit{isVar}\ t_1\ \wedge\ \mathit{isVar}\ t_2\ \wedge\ t_1\ ==\ t_2\ =\ \mathit{return}\ \sigma \\
 & \mid & \mathit{isVar}\ t_1\ =\ \mathit{bind}\ t_1\ t_2\ \sigma \\
 & \mid & \mathit{isVar}\ t_2\ =\ \mathit{bind}\ t_2\ t_1\ \sigma \\
 & \mid & \mathit{topEq}\ t_1\ t_2\ =\ \mathit{uniTs}\ t_1\ t_2\ \sigma \\
 & \mid & \textbf{otherwise}\ =\ \mathit{failure}
\end{array}$$

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

$$\begin{array}{lll}
\mathit{bind} & : & \mathit{Name} \to \mathit{Term} \to \mathit{Subst} \to \mathsf{Comp}\ \mathit{Subst} \\
\mathit{bind}\ v\ t\ \sigma & = & \textbf{case}\ \mathit{lkp}_S\ v\ \sigma\ \textbf{of} \\
 & & \quad \mathit{Nothing} \to \mathit{return}\ (\mathit{upd}_S\ v\ t\ \sigma) \\
 & & \quad \mathit{Just}\ t' \quad \to \mathit{unify}_S\ t\ t'\ \sigma \gg\!= \lambda\sigma'. \\
 & & \qquad\qquad \mathit{return}(\mathit{upd}_S\ v\ t\ \sigma')
\end{array}$$

4.3 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 handle errors, backtracking, environment access and to modify a global state. The global state in this simple case is only needed as a supply of fresh variable names. The resulting monad will be

$$\textbf{type}\ \mathsf{Comp}\ =\ \mathit{BackT}\ (\mathit{EnvT}\ \ \mathit{Env}\ (\mathit{StateT}\ \ \mathit{State}\ (\mathit{ErrT}\ \mathit{IO})))$$

we used the predefined *IO* monad as the base monad in order to facilitate the communication of solutions to the user. In this simple case, we use the following domains

$$\begin{array}{llll}
\textbf{type}\ \mathit{Subst} & = & \mathit{Name} \to \mathit{Term} & \text{— Substitutions} \\
\textbf{type}\ \mathit{Database} & = & \mathit{Name} \to (\mathit{Name}, \mathsf{Comp}\ \mathit{Subst}) & \text{— Clause Definitions} \\
\textbf{type}\ \mathit{Env} & = & (\mathit{Database}, \mathit{Subst}) & \text{— Environment} \\
\textbf{type}\ \mathit{State} & = & \mathit{Int} & \text{— Global state}
\end{array}$$

4.4 Semantic Specification

The semantic specification of the Prolog language consist of a P-algebra whose carrier is the computational structure.

$$
\begin{array}{ll}
\varphi_{\mathsf{P}} & : \ \mathsf{P}\ (\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value} \\
\varphi_{\mathsf{P}}\ (\mathit{Def}\ p\ x\ e_1\ e_2) & = \mathit{rdEnv} \gg\!= \lambda(\rho,\sigma). \\
 & \quad \mathit{inEnv}\ (\mathit{updEnv}\ \rho\ p\ (x, e_1))\ e_2 \\
 & \\
\varphi_{\mathsf{P}}\ (e_1\ \wedge\ e_2) & = \mathit{rdEnv} \gg\!= \lambda(\rho,\sigma). \\
 & \quad e_1 \gg\!= \lambda\sigma'. \\
 & \quad \mathit{inEnv}\ (\rho,\sigma')\ e_2 \\
 & \\
\varphi_{\mathsf{P}}\ (e_1\ \vee\ e_2) & = \mathit{rdEnv} \gg\!= \lambda(\rho,\sigma). \\
 & \quad \mathit{orElse}\ (\mathit{inEnv}\ (\rho,\sigma)\ e_1)\ (\mathit{inEnv}\ (\rho,\sigma)\ e_2) \\
 & \\
\varphi_{\mathsf{P}}\ (\exists f) & = \mathit{update}\ (+1) \gg\!= \lambda.f\ (\mathit{mkFree}\ n) \\
 & \\
\varphi_{\mathsf{P}}\ (\mathit{call}\ p\ t) & = \mathit{rdEnv} \gg\!= \lambda(\rho,\sigma). \\
 & \quad \mathbf{let}\ (x, m)\ =\ \rho\ (p, t) \\
 & \quad \mathbf{in}\ \mathit{unify}_S\ (C\ x)\ t\ \sigma \gg\!= \lambda\sigma' \to\ \mathit{inEnv}\ (\rho,\sigma')\ m \\
 & \\
\varphi_{\mathsf{P}}\ (t_1 \stackrel{\circ}{=} t_2) & = \mathit{rdEnv} \gg\!= \lambda(\rho,\sigma). \\
 & \quad \mathit{unify}_S\ t_1\ t_2\ \sigma \\
 & \\
\varphi_{\mathsf{P}}\ (?\ x\ f) & = \mathit{update}\ (+1) \gg\!= \lambda n. \\
 & \quad f\ (\mathit{mkFree}\ n) \gg\!= \lambda\sigma. \\
 & \quad \mathit{putAnswer}\ x\ (\sigma\ \ v)) \gg\!= \lambda\_. \\
 & \quad \mathit{return}\ \sigma
\end{array}
$$

The following auxiliary definitions have been used

- $\mathit{mkFree} : \ \mathit{Int} \to \mathit{Name}$, creates a new name
- $\mathit{putAnswer} : \mathit{Name} \to\ \mathit{Term} \to \mathsf{Comp}\ ()$, writes the value of a variable and asks the user for more answers.
- $\mathit{updEnv} : \mathit{Env}\ \to\ \mathit{Name} \to (\mathit{Name}, \mathsf{Comp}\ \mathit{Subst}) \to \mathit{Env}$, return an environment with a new value for a given predicate.

The Prolog interpreter is automatically obtained as a catamorphism

$$
\begin{array}{l}
\mathsf{Inter}_{\mathit{Prolog}}\ :\ \mathit{Prolog} \to \mathsf{Comp}\ \mathit{Subst} \\
\mathsf{Inter}_{\mathit{Prolog}}\ = \mathit{cata}\ \varphi_{\mathsf{P}}
\end{array}
$$

5 Adding Arithmetic

The Prolog predicate (*is*) opens a new semantic world in the language as it implies the arithmetic evaluation of one of its arguments. Other specifications of Prolog [19,3] often avoid this predicate as it 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 evaluation and combine them to form a new language.

As we are going to use two different categories, we define the bifunctor

$$\mathbf{data}\ \mathsf{Is}\ g\ e\ =\ Is\ Term\ e$$

and the semantic specification

$$\begin{array}{ll}
\varphi_{\mathsf{Is}} & :\ \mathsf{Is}\ (\mathsf{Comp}\ Subst)\ (\mathsf{Comp}\ Int) \to \mathsf{Comp}\ Subst \\
\varphi_{\mathsf{Is}}\ (Is\ t\ e) & = e \gg\!= \lambda v. \\
 & \quad rdEnv \gg\!= \lambda(\rho, \sigma). \\
 & \quad unify_S\ t\ (cnv\ v)\ \sigma
\end{array}$$

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

$$\mathbf{type}\ Prolog^{+}\ =\ Fix_1\ (S_2\ (P_1^2\ \mathsf{P})\ \mathsf{Is})\ (P_2^2\ \mathsf{T})$$

and the corresponding interpreter is obtained as a bicatamorphism

$$\begin{array}{ll}
\mathsf{Inter}_{Prolog^{+}} & :\ Prolog^{+} \to \mathsf{Comp}\ Subst \\
\mathsf{Inter}_{Prolog^{+}} & = cata_1^2\ (\epsilon_1^2\ \varphi_{\mathsf{P}}\ \boxplus_1\ \varphi_{\mathbb{I}})\ (\epsilon_2^2\ \varphi_{\mathsf{T}})
\end{array}$$

6 Conclusions and future work

The integration of modular monadic semantics and generic programming concepts provides a very modular way to specify programming languages from reusable semantic building blocks.

The traditional way to modularize the development of a language processor consist in the identification of the main processes involved: lexical analysis, syntactic analysis, static analysis, evaluation, etc. It is very difficult to identify the code corresponding, for example, to arithmetic expressions and to reuse that code in the development of a processor for a different language. In our approach we independently specify the kernel of a logic programming language and a simple arithmetic expressions block and integrate both to obtain a logic programming with arithmetic capabilities language. The arithmetic expressions block can be reused in a different language without change. Indeed, we have developed specifications of imperative [12], functional [10,13], object-oriented [14] and logic programming languages [11]. The specifications have been made in a modular way by reusing common blocks. With this approach the language designer only needs to concentrate on a particular feature, which can be included and tested in automatically obtained language prototypes.

Moreover, the computational structure of the logic programming language is obtained from the composition of several monad transformers which incrementally add new notions of computation. It would be straightforward to add control facilities like negation or cut by modifying these monad transformers [8].

We have implemented a Language Prototyping System in Haskell. The implementation offers an interactive framework for language testing and is based on a domain-specific meta-language embedded in Haskell. 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 done in [17].

With regard to the current implementation, we have also made a simple version of the system using first-class polymorphism and extensible records. This allows the definition of monads as first class values and monad transformers as functions between monads without the need of type classes. However, this feature is still not fully implemented in current Haskell systems.

This paper is a first attempt to model logic programming languages in this approach. Future work can be done in the specification of other features and in the integration between different modules leading to cross-paradigm programming language designs. More information on the system can be obtained at [1].

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

[4] Koen Claessen and Peter Ljungl´of. Typed logical variables in haskell. In *Haskell Workshop*. ACM SIGPLAN, September 2000.

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

[6] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, February 1992.

[7] Ralf Hinze. Prological features in a functional setting — axioms and implementations. In Masahiko Sato and Yoshihito Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming (FLOPS'98), Kyoto University, Japan*, pages 98–122, Singapore, New Jersey, London, Hong Kong, April 1998. World Scientific.

[8] 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.

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

[10] 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.

[11] J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. Reusable monadic semantics of logic programs with arithmetic predicates. In Luís Moniz Pereira and Paulo Quaresma, editors, *APPIA-GULP-PRODE 2001, Joint Conference on Declarative Programming*, Évora, Portugal, September 2001. Universidade de Evora.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990. ACM Press.