Specification of Logic Programming Languages from Reusable Semantic Building Blocks

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

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

Abstract. We specify a logic programming language using a combination of modular monadic semantics and generic programming concepts. The abstract syntax is modelled as the fixed point of a pattern functor which can be obtained as the sum of several 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 to a base monad, where each monad transformer adds a new notion of computation. In this approach, the prototype interpreters are automatically obtained as a catamorphism over the defined algebras.
When the abstract syntax is composed from mutually recursive categories, it is possible to use many-sorted algebras. As an example, 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. Later, 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. Nevertheless, using monad transformers [16] it is possible to transform a given monad into a new one adding new computational capabilities. The use of monads and monad transformers to specify the semantics of programming languages was called modular monadic semantics in [15].

On the other hand, 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 catamorphisms with modular monadic semantics [5] allowing the independent specification of the abstract syntax, the computational monad and the domain value. In [9,10,11,12] we applied monadic catamorphisms allowing to separate the recursive evaluation and the semantic specification. In [13] we also show that it is possible to apply this approach when the abstract syntax is divided in several mutually recursive categories.

Previous work concentrates on functional and imperative languages. In this paper, we apply that approach to model a logic programming language. The resulting specification identifies the different computational features underlying the logic programming kernel (environment access, global state, backtracking) through different monad transformers and can be combined with other features. As an example, in section 5, we combine the logic programming kernel with an independently specified arithmetic expressions block to obtain a logic programming with arithmetic predicates language.

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

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

will be used as

$$\alpha \,\|\, \beta \;\triangleq\; L\,\alpha \;\mid\; R\,\beta$$

We will also omit the type constructors in some definitions for brevity. The notions we use from category theory are defined in the paper, so it is not a prerequisite.

2 Modular Monadic Semantics

In functional programming, a monad can be defined as a type constructor M and a pair of polymorphic operations $(\gg\!=) : \mathsf{M}\alpha \to (\alpha \to \mathsf{M}\beta) \to \mathsf{M}\beta$ 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.

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 $(\gg\!=)$. Table 1 contains some classes of monads with their operations.

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

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	$failure : \mathsf{M}\,\alpha$ $(\uplus) : \mathsf{M}\,\alpha \to \mathsf{M}\,\alpha \to \mathsf{M}\,\alpha$

Table 1. Some classes of monads

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 [7] it is shown how to derive a backtracking monad transformer from its specification.

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.

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 $map_{\mathsf{F}} : (\alpha \to \beta) \to \mathsf{F}\,\alpha \to \mathsf{F}\,\beta$.

The fixpoint of a functor F can be defined as

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

In the above definition, we explicitly write the type constructor In because we will refer to it later. A recursive datatype can be defined as the fixpoint of a non-recursive functor that captures its shape.

Example 1. The inductive datatype $Term$ defined as

$$Term \triangleq Num\,Int \mid Term + Term$$

can be defined as the fixpoint of the functor T

$$\begin{aligned} \mathsf{T}\,x &\triangleq Num\,Int \mid x + x \\ Term &\triangleq \mu\mathsf{T} \end{aligned}$$

Error handling	Environment Reader
$\begin{array}{lcl} \mathcal{T}_{Err}\ \mathsf{M}\ \alpha & \triangleq & \mathsf{M}\ (\alpha \parallel String) \\ return\ x & = & return\ (L\ x) \\ x \gg\!= f & = & x \gg\!= \lambda y \rightarrow \textbf{case}\ y\ \textbf{of} \\ & & \quad L\ v \rightarrow f\ v \\ & & \quad R\ e \rightarrow e \\ lift\ m & = & m \gg\!= \lambda x \rightarrow return\ (L\ x) \\ err\ msg & = & return\ (R\ msg) \end{array}$	$\begin{array}{lcl} \mathcal{T}_{Env}\ \mathsf{M}\ \alpha & \triangleq & Env \rightarrow \mathsf{M}\ \alpha \\ return\ x & = & \lambda \rho \rightarrow return\ x \\ x \gg\!= f & = & \lambda \rho \rightarrow x\ \rho \gg\!= \lambda a \rightarrow \\ & & \quad f\ a\ \rho \\ lift\ x & = & \lambda \rho \rightarrow x \gg\!= return \\ rdEnv & = & \lambda \rho \rightarrow return\ \rho \\ inEnv\ \rho\ x & = & \lambda \_ \rightarrow x\ \rho \end{array}$
State transformer	**Backtracking**
$\begin{array}{lcl} \mathcal{T}_{State}\ \mathsf{M}\ \alpha & \triangleq & State \rightarrow \mathsf{M}\ (\alpha, State) \\ return\ x & = & \lambda \varsigma \rightarrow return(x, \varsigma) \\ x \gg\!= f & = & \lambda \varsigma \rightarrow x\ \varsigma \gg\!= \lambda (v, \varsigma') \rightarrow \\ & & \quad f\ v\ \varsigma' \\ lift\ m & = & \lambda \varsigma \rightarrow m \gg\!= \lambda x \rightarrow \\ & & \quad return(x, \varsigma) \\ update\ f & = & \lambda \varsigma \rightarrow return(\varsigma, f\varsigma) \end{array}$	$\begin{array}{lcl} \mathcal{T}_{Back}\ \mathsf{M}\ \alpha \triangleq \forall \beta. & & \\ \quad ((\alpha \rightarrow \mathsf{M}\ \beta \rightarrow \mathsf{M}\ \beta) \rightarrow \mathsf{M}\ \beta \rightarrow \mathsf{M}\ \beta) & & \\ return\ x & = & \lambda \kappa \rightarrow \kappa x \\ x \gg\!= f & = & \lambda \kappa \rightarrow x (\lambda v \rightarrow f\ v\ \kappa) \\ lift\ m & = & \lambda\, \kappa\, f \rightarrow m \gg\!= \lambda x \rightarrow \\ & & \quad \kappa\ x\ f \\ failure & = & \lambda \kappa \rightarrow (\lambda\, x \rightarrow x) \\ m \Cup n & = & \lambda\, \kappa\, f \rightarrow m\, \kappa\, (n\, \kappa\, f) \end{array}$

Table 2. Some monad transformers with their definitions

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

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

Example 2. We can obtain a simple evaluator for terms defining a T-algebra whose carrier is the type $\mathsf{M}\ Int$, where M is, in this case, any kind of monad.

$$\begin{array}{lcl} \varphi_{\mathsf{T}} & : & \mathsf{T}\,(\mathsf{M}\ Int) \rightarrow (\mathsf{M}\ Int) \\ \varphi_{\mathsf{T}}\ (Num\ n) & = & return\ n \\ \varphi_{\mathsf{T}}\ (t_1\ +\ t_2) & = & t_1 \gg\!= \lambda v_1 \rightarrow t_2 \gg\!= \lambda v_2 \rightarrow return(v_1 + v_2) \end{array}$$

An interpreter of arithmetic terms is obtained as a catamorphism

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

The functor T with its corresponding T-algebra forms a semantic block which can be incorporated in functional and imperative languages [12,13]. In section 5 we will also reuse that block in a logic programming language to obtain the specification of the arithmetic evaluation predicate *is*.

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 definitions in this section have been used in [13] to specify a simple imperative language with two syntactic categories, expressions and commands. In this paper, these definitions will be applied in section 5 to combine the logic programming kernel with the arithmetic expressions block.

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

$$\mathit{bimap}_{\mathbb{F}} : (\alpha \to \gamma) \;\to\; (\beta \to \delta) \;\to\; (\mathbb{F}\,\alpha\,\beta \to \mathbb{F}\,\gamma\,\delta)$$

The fixpoint 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:

$$\begin{array}{lcl} \mu_1\mathbb{F}\mathbb{G} & \triangleq & \mathit{In}_1\,(\mathbb{F}\,(\mu_1\mathbb{F}\mathbb{G})\,(\mu_2\mathbb{F}\mathbb{G})) \\ \mu_2\mathbb{F}\mathbb{G} & \triangleq & \mathit{In}_2\,(\mathbb{G}\,(\mu_1\mathbb{F}\mathbb{G})\,(\mu_2\mathbb{F}\mathbb{G})) \end{array}$$

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 $(\mathit{In}_1, \mathit{In}_2)$ form the initial object. This allows the definition of bicatamorphisms as:

$$\begin{array}{ll} (\!|\_,\_|\!)_1 & : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to (\mu_1\mathbb{F}\mathbb{G} \to \alpha) \\ (\!|\varphi,\psi|\!)_1\ (\mathit{In}_1\ x) & = \varphi\ (\ \mathit{bimap}_{\mathbb{F}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2\ x) \end{array}$$

$$\begin{array}{ll} (\!|\_,\_|\!)_2 & : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to (\mu_2\mathbb{F}\mathbb{G} \to \beta) \\ (\!|\varphi,\psi|\!)_2\ (\ \mathit{In}_2\ x) & = \psi\ (\ \mathit{bimap}_{\mathbb{G}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2\ x) \end{array}$$

The sum of two bifunctors $\mathbb{F}$ and $\mathbb{G}$ is a new bifunctor $\mathbb{F} \boxplus \mathbb{G}$

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

where the *bimap* operator is

$$\begin{array}{lcl} \mathit{bimap}_{\mathbb{F}\boxplus\mathbb{G}} & : & (\alpha \to \gamma) \to (\beta \to \delta) \to ((\mathbb{F} \boxplus \mathbb{G})\ \alpha\,\beta \to ((\mathbb{F} \boxplus \mathbb{G})\ \gamma\,\delta) \\ \mathit{bimap}_{\mathbb{F}\boxplus\mathbb{G}}\ f\ g\ (L\,x) & = & L\,(\mathit{bimap}_{\mathbb{F}\boxplus\mathbb{G}}\ f\ g\ x) \\ \mathit{bimap}_{\mathbb{F}\boxplus\mathbb{G}}\ f\ g\ (R\,x) & = & R\,(\mathit{bimap}_{\mathbb{F}\boxplus\mathbb{G}}\ f\ g\ x) \end{array}$$

Two-sorted algebras can be extended using the following operators

$$\begin{array}{l} (\boxplus_1) : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \alpha) \to\ (\mathbb{F} \boxplus \mathbb{G})\ \alpha\,\beta \to \alpha \\ (\phi_1\ \boxplus_1\ \phi_2)\,(L\,x)\ =\ \phi_1\ x \\ (\phi_2\ \boxplus_1\ \phi_2)\,(R\,x)\ =\ \phi_2\ x \end{array}$$

$$\begin{array}{l} (\boxplus_2) : (\mathbb{F}\,\alpha\,\beta \to \beta) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to\ (\mathbb{F} \boxplus \mathbb{G})\ \alpha\,\beta \to \beta \\ (\psi_1\ \boxplus_2\ \psi_2)\,(L\,x)\ =\ \psi_1\ x \\ (\psi_2\ \boxplus_2\ \psi_2)\,(R\,x)\ =\ \psi_2\ x \end{array}$$

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 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

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

Given a single sorted algebra, the following operators ϵ^2_1 and ϵ^2_2 obtain the corresponding two-sorted algebras

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

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

4 Specification of a Pure Logic Programming Language

In this section we specify the reusable semantics of a pure logic programming language. The syntactic representation has been inspired by [4].

4.1 Syntactical Structure

Prolog terms are defined as

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

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.

$$\begin{aligned} \mathsf{P}\ e \ &= \mathbf{Def}\ Name\ Name\ e\ e && \text{— Definitions} \\ &\mid\ e \wedge e && \text{— Conjunction} \\ &\mid\ e \vee e && \text{— Disjunction} \end{aligned}$$

$$
\begin{array}{lll}
 & \mid \exists(Name \rightarrow\ e) & \text{— Free variables} \\
 & \mid call\ Name\ Term & \text{— Predicate call} \\
 & \mid Term \stackrel{\circ}{=} Term & \text{— Unification} \\
 & \mid ?Name\ (Name \rightarrow e) & \text{— Goal}
\end{array}
$$

The Prolog language is defined as the fixed point of P

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

Example 3. 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

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

4.2 Unification

In this section we adapt the generic unification algorithm presented in [8]. 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:

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

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

$$
\begin{array}{ll}
lkp_S\ :\ Name \rightarrow Subst \rightarrow Maybe\ Term & \text{— lookup} \\
upd_S\ :\ Name \rightarrow Term \rightarrow Subst \rightarrow Subst & \text{— update}
\end{array}
$$

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

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

The unification algorithm will be:

$$
\begin{array}{ll}
unify_S & :\ Term\ \rightarrow\ Term\ \rightarrow\ Subst\ \rightarrow\ \mathsf{Comp}\ Subst \\
unify_S\ t_1\ t_2\ \sigma & \mid\ isVar\ t_1\ \wedge\ isVar\ t_2\ \wedge\ t_1\ ==\ t_2\ =\ return\ \sigma \\
 & \mid\ isVar\ t_1\ =\ bind\ t_1\ t_2\ \sigma \\
 & \mid\ isVar\ t_2\ =\ bind\ t_2\ t_1\ \sigma \\
 & \mid\ topEq\ t_1\ t_2\ =\ uniTs\ t_1\ t_2\ \sigma \\
 & \mid\ \mathbf{otherwise}\ =\ failure
\end{array}
$$

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

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

4.3 Computational Structure

The computational structure will be described by means of a monad which must support the different operations needed. In this language, we need 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

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

we used the predefined IO monad as the base monad in order to facilitate the communication of solutions to the user. Notice that we used three monad transformers that identify the main computational features. In that way, the computational structure could later be extended without affecting the language specification.

The predicate database will be represented as an abstract datatype *Database* with the following operations:

$$
\begin{aligned}
&\mathit{upd}_{DB} &&: \mathit{Database} \to \mathit{Name} \to \mathit{Predicate} \to \mathit{Database}\\
&\mathit{lkp}_{DB} &&: \mathit{Database} \to \mathit{Name} \to \mathit{Predicate}
\end{aligned}
$$

We use the following domains

$$
\begin{aligned}
&\mathit{Predicate} &&\triangleq (\mathit{Name}, \mathsf{Comp}\,\mathit{Subst}) &&\text{— Predicate Definition}\\
&\mathit{Env} &&\triangleq (\mathit{Database}, \mathit{Subst}) &&\text{— Environment}\\
&\mathit{State} &&\triangleq \mathit{Int} &&\text{— Global state}
\end{aligned}
$$

4.4 Semantic Specification

The semantic specification of the pure logic programming language consists of a P-algebra whose carrier is the computational structure.

$$
\begin{aligned}
&\varphi_{\mathsf{P}} &&:\ \mathsf{P}\ (\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value}\\
&\varphi_{\mathsf{P}}\ (\mathit{Def}\ p\ x\ e_1\ e_2) &&= \mathit{rdEnv} \gg\!= \lambda(\delta, \sigma) \to
\end{aligned}
$$

$$
\begin{array}{ll}
 & \quad inEnv\ (upd_{DB}\ \delta\ p\ (x, e_1))\ e_2 \\[1em]
\varphi_{\mathsf{P}}\ (e_1\ \wedge\ e_2) & = rdEnv {\gg\!=} \lambda(\delta,\sigma) \rightarrow \\
 & \quad e_1 {\gg\!=} \lambda\sigma' \rightarrow \\
 & \quad inEnv\ (\delta,\sigma')\ e_2 \\[1em]
\varphi_{\mathsf{P}}\ (e_1\ \vee\ e_2) & = rdEnv {\gg\!=} \lambda\rho \rightarrow \\
 & \quad inEnv\ \rho\ e_1\ \Cup\ inEnv\ \rho\ e_2 \\[1em]
\varphi_{\mathsf{P}}\ (\exists\, f) & = update\ (+1) {\gg\!=} \lambda n \rightarrow \\
 & \quad f\ (mkFree\ n) \\[1em]
\varphi_{\mathsf{P}}\ (call\ p\ t) & = rdEnv {\gg\!=} \lambda(\delta,\sigma) \rightarrow \\
 & \quad \mathbf{let}\ (x, m)\ =\ lkp_{DB}\ \delta\ p \\
 & \quad \mathbf{in}\ unify_S\ (C\ x)\ t\ \sigma {\gg\!=} \lambda\sigma' \rightarrow \\
 & \qquad inEnv\ (\delta,\sigma')\ m \\
\varphi_{\mathsf{P}}\ (t_1 \mathrel{\mathring{=}} t_2) & = rdEnv {\gg\!=} \lambda(\delta,\sigma) \rightarrow \\
 & \quad unify_S\ t_1\ t_2\ \sigma \\[1em]
\varphi_{\mathsf{P}}\ (?\ x\ f) & = update\ (+1) {\gg\!=} \lambda n \rightarrow \\
 & \quad f\ (mkFree\ n) {\gg\!=} \lambda\sigma \rightarrow \\
 & \quad putAnswer\ x\ (\sigma\ \ v)) {\gg\!=} \lambda\_ \rightarrow \\
 & \quad return\ \sigma
\end{array}
$$

The following auxiliary definitions have been used

- $mkFree : Int \rightarrow Name$, creates a new name
- $putAnswer : Name \rightarrow Term \rightarrow \mathsf{Comp}\ ()$, writes the value of a variable and asks the user for more answers.

The Prolog interpreter is automatically obtained as a catamorphism

$$
\begin{array}{l}
\mathsf{Inter}_{Prolog}\ :\ Prolog \rightarrow \mathsf{Comp}\ Subst \\
\mathsf{Inter}_{Prolog}\ = (\!|\varphi_{\mathsf{P}}|\!)
\end{array}
$$

5 Adding Arithmetic

The Prolog arithmetic evaluation 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 logic programming and arithmetic blocks and combine them to form a new language.

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

$$\mathbb{I}\ g\ e\ \triangleq\ Term\ \mathbf{is}\ e$$

and the semantic specification

$$
\begin{array}{ll}
\varphi_{\mathbb{I}} & : \mathbb{I}\,(\mathsf{Comp}\ \mathit{Subst})\,(\mathsf{Comp}\ \mathit{Int}) \to \mathsf{Comp}\ \mathit{Subst} \\
\varphi_{\mathbb{I}}\ (t\ \mathbf{is}\ e) & = e \gg\!= \lambda v \to \\
& \quad \mathit{rdEnv} \gg\!= \lambda(\delta, \sigma) \to \\
& \quad \mathit{unify}_S\ t\ (\mathit{cnv}\ v)\ \sigma
\end{array}
$$

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

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

and the corresponding interpreter is obtained as a bicatamorphism

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

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. As an example, in this paper 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. 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 [7].

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.

We have developed specifications of imperative [13], functional [12], object-oriented [14] and logic programming languages. 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.

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`.
2. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming - an introduction. In S. D. Swierstra, P. R. Henriques, and J. 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. Utrecht University, 1995.
6. Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, February 1992.
7. Ralf Hinze. Deriving backtracking monad transformers. In Roland Backhouse and J. 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. 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.
10. 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.
11. J. E. Labra, J. M. Cueva, and M. C. Luengo. Modular development of interpreters from semantic building blocks. In *The 12th Nordic Workshop on Programming Theory*, Bergen, Norway, October 2000. University of Bergen.
12. J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. LPS: A language prototyping system using modular monadic semantics. In M. van den Brand and D. Parigot, editors, *First Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova, Italy, April 2001. Electronic Notes in Theoretical Computer Science – Elsevier.
13. 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)*, Lecture Notes in Computer Science, Las Palmas de Gran Canaria – Spain, February 2001. Springer Verlag.
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.