Electronic Notes in Theoretical Computer Science 44 No. 2 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 22 pages

# LPS: A Language Prototyping System Using Modular Monadic Semantics

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**

This paper describes LPS, a Language Prototyping System that facilitates the modular development of interpreters from semantic building blocks. The system is based on the integration of ideas from Modular Monadic Semantics and Generic Programming.

To define a new programming language, the abstract syntax is described as the fixpoint of non-recursive pattern functors. For each functor an algebra is defined whose carrier is the computational monad obtained from the application of several monad transformers to a base monad. The interpreter is automatically generated by a catamorphism or, in some special cases, a monadic catamorphism.

The system has been implemented as a domain-specific language embedded in Haskell and we have also implemented an interactive framework for language testing.

## 1 Introduction

The lack of modularity and reusability of traditional denotational semantics has already been recognized [45]. Monads were applied by E. Moggi [43] to capture the intuitive idea of separating values from computations. After his work, P. Wadler [52,53] applied monads to the development of modular interpreters and to encapsulate the Input/Output features of the purely functional programming language Haskell [27]. That work produced a growing interest in the development of modular interpreters using monads [10,47,53]. However, the monad approach has a problem. In general, it is not possible to compose two monads to obtain a new monad [26]. A proposed solution was the use of monad transformers which transform a given monad into a new one adding

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

new operations [38]. The use of monads and monad transformers to specify the semantics of programming languages was called modular monadic semantics by S. Liang et al. [37,36]. The close relationship between modular monadic semantics and action semantics was described in [54] where they present a system that combines both approaches.

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 [39,40,3].

Following that approach, L. Duponcheel proposed the combined use of folds or catamorphisms with modular monadic semantics [9] allowing the independent specification of the abstract syntax, the computational monad and the domain value.

Monadic catamorphisms were studied in [11,19] and applied to practical functional programming in [41]. Inspired by that work, we applied monadic folds [30,31,32] to modular monadic semantics, allowing the separation between recursive evaluation and semantic specification in some special cases.

The paper is structured as follows: in section 2, we give a brief overview of the underlying theory, in section 3, we describe the architecture of the Language Prototyping System, section 4 describes the structure of the semantic specifications, and section 5 the interactive framework. As an example, section 6 describes the specification of a functional programming language with some imperative features and different evaluation semantics. Finally, we discuss some conclusions and directions for future work.

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 Theoretical Background

### *2.1 Monads and Monad Transformers*

Although the notion of monad comes from category theory, in functional programming a monad can be defined as a triplet $\langle \mathsf{M}, return_{\mathsf{M}}, \gg\!=_{\mathsf{M}} \rangle$ with a type

| Description | Name | Operations |
|---|---|---|
| Error handling | ErrM | $err : String \to \mathsf{ErrM}\ \alpha$ |
| Environment Access | EM | $rdEnv : \mathsf{EM}\ Env$<br>$inEnv : Env \to \mathsf{EM}\ \alpha \to \mathsf{EM}\ \alpha$ |
| State transformer | SM | $update : (State \to State) \to \mathsf{SM}\ State$<br>$fetch : \mathsf{SM}\ State$<br>$set : State \to \mathsf{SM}\ State$ |
| Continuations | CM | $callcc : ((\mathsf{CM}\ \alpha \to \mathsf{CM}\ \beta) \to \mathsf{CM}\ \alpha) \to \mathsf{CM}\ \alpha$ |

Table 1
Some classes of monads

constructor M and a pair of polymorphic operations:

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

which must satisfy three laws (see, for example, [53]). The basic idea is that a monad M encapsulates a notion of computation and a value of type $\mathsf{M}\ \alpha$ can be considered as a computation M that returns a value of type $\alpha$.

The simplest monad of all is the identity monad Id which can be defined as

$$\begin{aligned} \mathsf{Id}\ x &\triangleq x \\ return\ x &= x \\ m \gg\!= f &= f\ m \end{aligned}$$

In the rest of the paper we use a special syntax, called the *do*-notation. The conversion rules are:

$$\begin{aligned} \mathbf{do}\ \{\ m;\ e\ \} &\equiv m \gg\!= \lambda\ \_ \to\ \mathbf{do}\ \{\ e\ \} \\ \mathbf{do}\ \{\ x \leftarrow\ m;\ e\ \} &\equiv m \gg\!= \lambda\ x\ \to\ \mathbf{do}\ \{\ e\ \} \\ \mathbf{do}\ \{\ \mathbf{let}\ exp;\ e\ \} &\equiv \mathbf{let}\ exp\ \mathbf{in}\ \mathbf{do}\ \{\ e\ \} \\ \mathbf{do}\ \{\ e\ \} &\equiv e \end{aligned}$$

It is possible to define special classes of monads for different notions of computations like state transformers, environment access, continuations, exceptions, Input/Output, non-determinism, resumptions, backtracking, etc. Each class of monad has some specific operations apart from the predefined $return_{\mathsf{M}}$ and $(\gg\!=_{\mathsf{M}})$. 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 [26]. Nevertheless, a monad transformer $\mathcal{T}$ can transform a given monad $\mathsf{M}$ into a new monad $\mathcal{T}\ \mathsf{M}$ that has new operations and maintains the operations of $\mathsf{M}$. The idea of monad transformer is based on the notion of monad morphism that appeared in Moggi's work [43] and was later proposed in [38]. 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 [36,37,38] and in [17] 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 $\mathsf{M}$, it is necessary to specify the $return_{\mathcal{T}\,\mathsf{M}}$ and $(\gg=_{\mathcal{T}\,\mathsf{M}})$ operations, the $lift : \mathsf{M}\,\alpha \rightarrow \mathcal{T}\,\mathsf{M}\,\alpha$ operation transforming any operation in $\mathsf{M}$ 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.

### *2.2 Functors, Algebras and Catamorphisms*

As in the case of monads, functors are also derived from category theory but can easily be defined in a functional programming setting. A functor $\mathsf{F}$ can be defined as a type constructor that transforms values of type $\alpha$ into values of type $\mathsf{F}\,\alpha$ and a function $map_{\mathsf{F}} : (\alpha \rightarrow \beta) \rightarrow \mathsf{F}\,\alpha \rightarrow \mathsf{F}\,\beta$.

The fixpoint of a functor $\mathsf{F}$ 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. For example, the inductive datatype *Term* defined as

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

can be defined as the fixpoint of the functor $\mathsf{T}$

$$\mathsf{T}\,x \triangleq Num\,Int \mid x + x \mid x - x$$

where the $map_{\mathsf{T}}$ is defined as[2]:

$$map_{\mathsf{T}} \quad : \quad (\alpha \rightarrow \beta) \rightarrow (\mathsf{T}\,\alpha \rightarrow \mathsf{T}\beta)$$

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

| | |
|---|---|
| Error handling | $\mathcal{T}_{Err}\ \mathsf{M}\ \alpha \triangleq \mathsf{M}\ (\alpha \parallel String)$<br>$return\ x = return\ (L\ x)$<br>$x \gg= f = x \gg= \lambda y \rightarrow$ **case** $y$ **of**<br>$\quad L\ v \rightarrow f\ v$<br>$\quad R\ e \rightarrow e$<br>$lift\ m = m \gg= \lambda x \rightarrow return\ (L\ x)$<br>$err\ msg = return\ (R\ msg)$ |
| Environment | $\mathcal{T}_{Env}\ \mathsf{M}\ \alpha \triangleq Env \rightarrow \mathsf{M}\ \alpha$<br>$return\ x = \lambda\rho \rightarrow returnx$<br>$x \gg= f = \lambda\rho \rightarrow (x\rho) \gg= (\lambda a \rightarrow f\ a\ \rho)$<br>$lift\ x = \lambda\rho \rightarrow x \gg= return$<br>$rdEnv = \lambda\rho \rightarrow return\rho$<br>$inEnv\ \rho\ x = \lambda\_ \rightarrow x\ \rho$ |
| State transformer | $\mathcal{T}_{State}\ \mathsf{M}\ \alpha \triangleq State \rightarrow \mathsf{M}\ (\alpha, State)$<br>$return\ x = \lambda\varsigma \rightarrow return(x, \varsigma)$<br>$x \gg= f = \lambda\varsigma \rightarrow (x\varsigma) \gg= (\lambda(v, \varsigma') \rightarrow f\ v\ \varsigma')$<br>$lift\ x = \lambda\varsigma \rightarrow x \gg= (\lambda x \rightarrow return(x, \varsigma))$<br>$update\ f = \lambda\varsigma \rightarrow return(\varsigma, f\varsigma)$<br>$fetch = update\ (\lambda\varsigma \rightarrow \varsigma)$<br>$set\ \varsigma = update\ (\lambda\_ \rightarrow \varsigma)$ |
| Continuations | $\mathcal{T}_{Cont}\ \mathsf{M}\ \omega\ \alpha \triangleq (\alpha \rightarrow \mathsf{M}\omega) \rightarrow \mathsf{M}\omega$<br>$return\ x = \lambda\kappa \rightarrow \kappa x$<br>$x \gg= f = \lambda\kappa \rightarrow x(\lambda v \rightarrow f\ v\ \kappa)$<br>$lift\ x = \lambda\kappa \rightarrow x \gg= \kappa$<br>$callcc\ f = \lambda\kappa \rightarrow (f\ (\lambda m \rightarrow\ (\lambda\_ \rightarrow m\ \kappa))\ \kappa)$ |

Table 2
Some monad transformers with their definitions

$$
\begin{aligned}
map_{\mathsf{T}}\ f\ (Num\ n) &= n \\
map_{\mathsf{T}}\ f\ (x_1\ +\ x_2) &= f\ x_1\ +\ f\ x_2 \\
map_{\mathsf{T}}\ f\ (x_1\ -\ x_2) &= f\ x_1\ -\ f\ x_2
\end{aligned}
$$

Once we have the shape functor $\mathsf{T}$, we can obtain the recursive datatype as the fixpoint of $\mathsf{T}$

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

As an example, the term $3 + 4$ can be represented as

$$In\ ((In\ (Num\ 3))\ +\ (In\ (Num\ 4))) : Term$$

The sum of two functors $\mathsf{F}$ and $\mathsf{G}$, denoted by $\mathsf{F} \oplus \mathsf{G}$ can be defined as

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

where $map_{\mathsf{F} \oplus \mathsf{G}}$ is defined as

$$\begin{aligned} map_{\mathsf{F} \oplus \mathsf{G}}\, f\ (L\, x) &= L\, (map_{\mathsf{F}}\, f\ x) \\ map_{\mathsf{F} \oplus \mathsf{G}}\, f\ (R\, x) &= R\, (map_{\mathsf{G}}\, f\ x) \end{aligned}$$

Using the sum of two functors, it is possible to extend recursive datatypes. For example, we can define a new pattern functor for factors as

$$\mathsf{F}\, x \triangleq x \times x \mid x \div x$$

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

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

Given a functor $\mathsf{F}$, an $\mathsf{F}$-algebra is a function $\varphi_{\mathsf{F}} : \mathsf{F}\, \alpha \to \alpha$ where $\alpha$ is called the carrier. An homomorphism between two $\mathsf{F}$-algebras $\varphi : \mathsf{F}\, \alpha \to \alpha$ and $\psi : \mathsf{F}\, \beta \to \beta$ is a function $h : \alpha \to \beta$ which satisfies

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

It is possible to consider a new category with $\mathsf{F}$-algebras as objects and homomorphisms between $\mathsf{F}$-algebras as morphisms. In this category, $In : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F}$ is an initial object, i.e. for any $\mathsf{F}$-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 [3,6,40,46]. It can be defined as:

$$\begin{aligned} (\!|\_|\!) &: (\mathsf{F}\alpha \to \alpha) \to (\mu\mathsf{F} \to \alpha) \\ (\!|\varphi|\!) &= \varphi \,.\, map_{\mathsf{F}}\, (\!|\varphi|\!) \,.\, out \end{aligned}$$

where

$$\begin{aligned} out &: \mu\mathsf{F} \to \mathsf{F}\, (\mu\mathsf{F}) \\ out\ (In\ x) &= x \end{aligned}$$

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

$$\varphi_{\mathsf{T}} \quad : \quad \mathsf{T}\, (\mathsf{M}\, Int) \to (\mathsf{M}\, Int)$$

$$
\begin{aligned}
\varphi_{\mathsf{T}}\,(Num\ n) &= return\ n \\
\varphi_{\mathsf{T}}\,(t_1\ +\ t_2) &= \mathbf{do} \\
&\quad v_1 \leftarrow t_1 \\
&\quad v_2 \leftarrow t_2 \\
&\quad return\ (v_1\ +\ v_2) \\
\varphi_{\mathsf{T}}\,(t_1\ -\ t_2) &= \mathbf{do} \\
&\quad v_1 \leftarrow t_1 \\
&\quad v_2 \leftarrow t_2 \\
&\quad return\ (v_1\ -\ v_2)
\end{aligned}
$$

Applying a catamorphism over $\varphi_{\mathsf{T}}$ we obtain the evaluation function for terms:

$$
\begin{aligned}
&eval_{Term} && :\ Term \rightarrow \mathsf{M}\,Int \\
&eval_{Term} = eval_{\mu\mathsf{T}} && = (\!|\varphi_{\mathsf{T}}|\!)
\end{aligned}
$$

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

$$
\begin{aligned}
&\oplus :\ (\mathsf{F}\,\alpha \rightarrow \alpha) \rightarrow (\mathsf{G}\,\alpha \rightarrow \alpha) \rightarrow (\mathsf{F} \oplus \mathsf{G})\alpha \rightarrow \alpha \\
&(\varphi \oplus \psi)(L\,x) = \varphi\,x \\
&(\varphi \oplus \psi)(R\,x) = \psi\,x
\end{aligned}
$$

The above definition allows to extend the evaluator for terms and factors without modifying the existing definitions. If we want to add factors, we only need to define the corresponding $\mathsf{F}$-algebra over $\mathsf{M}\,Int$ as:

$$
\begin{aligned}
\varphi_{\mathsf{F}}\,(t_1\ \times\ t_2) &= \mathbf{do} \\
&\quad v_1 \leftarrow t_1 \\
&\quad v_2 \leftarrow t_2 \\
&\quad return\ (v_1\ \times\ v_2) \\
\varphi_{\mathsf{F}}\,(t_1\ \div\ t_2) &= \mathbf{do} \\
&\quad v_1 \leftarrow t_1 \\
&\quad v_2 \leftarrow t_2 \\
&\quad \mathbf{if}\ v_2 == 0\ \mathbf{then} \\
&\quad\quad err\ \text{``Divide by zero''} \\
&\quad \mathbf{else} \\
&\quad\quad return\ (v_1\ \div\ v_2)
\end{aligned}
$$

Notice that, in this case, the monad $\mathsf{M}$ must support the *err* operation, i.e. it must support partial computations. Now, a new evaluator for expressions is automatically obtained by means of a catamorphism over the $(\mathsf{T} \oplus \mathsf{F})$-algebra.

$$
\begin{aligned}
&eval_{Expr} && :\ \mu(\mathsf{T} \oplus \mathsf{F}) \rightarrow \mathsf{M}\,Int \\
&eval_{Expr} = eval_{\mu(\mathsf{T}\oplus\mathsf{F})} && = (\!|\varphi_{\mathsf{T}} \oplus \varphi_{\mathsf{F}}|\!)
\end{aligned}
$$

The theory of catamorphisms can be extended to monadic catamorphisms as described in [11,19,30,32]. Given a monad $\mathsf{M}$, we define a monadic function $f : \alpha \to \mathsf{M}\,\beta$. For some combinations of monads and functors $\mathsf{F}$, we define the monadic extension of a functor $\mathsf{F}^{\mathsf{m}}$ declaring the function

$$map^{\mathsf{m}}_{\mathsf{F}} \;:\; (\alpha \to \mathsf{M}\,\beta) \to (\mathsf{F}\,\alpha \to \mathsf{M}\,(\mathsf{F}\,\beta))$$

In the same way, we can define monadic $\mathsf{F}$-algebras as $\varpi_{\mathsf{F}} : \mathsf{F}\,\alpha \to \mathsf{M}\,\alpha$ and monadic catamorphisms as

$$\begin{aligned}
(\!\!\left[\!\left[_\right]\!\right]\!\!) \;&:\; (\mathsf{F}\,\alpha \to \mathsf{M}\,\alpha) \to \mu\mathsf{F} \;\to\; \mathsf{M}\,\alpha \\
(\!\!\left[\!\left[\varpi_{\mathsf{F}}\right]\!\right]\!\!) &= \varpi_{\mathsf{F}} \;@\; map^{\mathsf{m}}_{\mathsf{F}}\; (\!\!\left[\!\left[\varpi_{\mathsf{F}}\right]\!\right]\!\!) \;@\; return \;.\; out
\end{aligned}$$

where @ represents the composition of monadic functions and can be defined as

$$\begin{aligned}
(@) \;&:\; (\beta \to \mathsf{M}\,\gamma) \to (\alpha \to \mathsf{M}\,\beta) \to (\alpha \to \mathsf{M}\,\gamma) \\
f \;@\; g \;&= \lambda x \to g\; x \gg\!\!= f
\end{aligned}$$

Using monadic catamorphisms, it is possible to separate the recursive evaluation from the semantic specification. In the simple evaluator example, we can define the monadic extension of the functor $\mathsf{T}$ as:

$$\begin{aligned}
map^{\mathsf{m}}_{\mathsf{T}} \;&:\; (\alpha \;\to\; \mathsf{M}\,\beta) \;\to\; (\mathsf{T}\,\alpha \to\; (\mathsf{M}\,(\mathsf{T}\,\beta))) \\
map^{\mathsf{m}}_{\mathsf{T}}\, f\; (Num\; n) \;&= return\; (Num\; n) \\
map^{\mathsf{m}}_{\mathsf{T}}\, f\; (x_1 \;+\; x_2) \;&= \;\mathbf{do} \\
&\qquad v_1 \;\leftarrow\; f\; x_1 \\
&\qquad v_2 \;\leftarrow\; f\; x_2 \\
&\qquad return\; (v_1 \;+\; v_2) \\
map^{\mathsf{m}}_{\mathsf{T}}\, f\; (x_1 \;-\; x_2) \;&= \;\mathbf{do} \\
&\qquad v_1 \;\leftarrow\; f\; x_1 \\
&\qquad v_2 \;\leftarrow\; f\; x_2 \\
&\qquad return\; (v_1 \;-\; v_2)
\end{aligned}$$

Notice that the above definition could have been obtained automatically. However, it specifies an explicit order of evaluation and a mandatory recursive evaluation of subcomponents, which could be inappropiate for other expressions.

Now, the semantic specification consists of a simple monadic $\mathsf{T}$-algebra

$$\begin{aligned}
\varpi_{\mathsf{T}} \;&:\; \mathsf{T}\,\alpha \;\to\; \mathsf{M}\,\alpha \\
\varpi_{\mathsf{T}}\; (Num\; n) \;&=\; return\; n \\
\varpi_{\mathsf{T}}\; (v_1 \;+\; v_2) \;&=\; return\; (v_1 \;+\; v_2) \\
\varpi_{\mathsf{T}}\; (v_1 \;-\; v_2) \;&=\; return\; (v_1 \;-\; v_2)
\end{aligned}$$

and the evaluation of terms is automatically obtained as a monadic catamorphism

$$
\begin{aligned}
eval_{Term} &\quad : \quad Term \;\rightarrow\; \mathsf{M}\, Int \\
eval_{Term} &= eval_{\mu\mathsf{T}} = (\!\![\varpi_{\mathsf{T}}]\!\!)
\end{aligned}
$$

It is possible to define the sum of two monadic algebras

$$
\begin{aligned}
&\oplus_{\mathsf{m}} \;:\; (\mathsf{F}\,\alpha \rightarrow \mathsf{M}\,\alpha) \rightarrow (\mathsf{G}\,\alpha \rightarrow \mathsf{M}\,\alpha) \rightarrow ((\mathsf{F} \oplus \mathsf{G})\,\alpha \rightarrow \mathsf{M}\,\alpha) \\
&(\varpi_{\mathsf{F}} \oplus_{\mathsf{m}} \varpi_{\mathsf{G}})(L\,x) = \varpi_{\mathsf{F}}\,x \\
&(\varpi_{\mathsf{F}} \oplus_{\mathsf{m}} \varpi_{\mathsf{G}})(R\,x) = \varpi_{\mathsf{G}}\,x
\end{aligned}
$$

Finally, it is possible to combine catamorphisms and monadic catamorphisms with the following definition

$$
\begin{aligned}
&(\!\![\_\,,\_]\!\!) \quad : \; (\mathsf{F}\,\alpha \rightarrow \mathsf{M}\,\alpha) \rightarrow (\mathsf{G}(\mathsf{M}\,\alpha) \rightarrow \mathsf{M}\,\alpha) \rightarrow \mu(\mathsf{F} \oplus \mathsf{G}) \rightarrow \mathsf{M}\,\alpha \\
&(\!\![\varpi, \varphi]\!\!)\; x = \mathbf{case}\; out\; x\; \mathbf{of} \\
&\qquad\qquad L\,v \;\rightarrow (\varpi \;@\; map^{\mathsf{m}}_{\mathsf{F}}\; (\!\![\varpi,\varphi]\!\!) \;@\; return)\; v \\
&\qquad\qquad R\,v \;\rightarrow (\varphi \,.\, map\; (\!\![\varpi,\varphi]\!\!))\; v
\end{aligned}
$$

## 3 Architecture of the *Language Prototyping System*

The Language Prototyping System (LPS) is defined as a domain specific language embedded in Haskell. The structure of LPS is divided in several parts:

- There are different *programming language descriptions* and the user can define new languages. If the user wants to add a new language, it is necessary to define the parser, the pretty-printer and the semantic specification.
- The *interactive framework* allows runtime selection and interpretation of the different programming languages that were defined.
- There are some modules for *common tools.* These tools give support to theoretical concepts like functors, algebras, catamorphisms, etc. and to common structures like heaps, stacks, symbol tables, etc.
- Finally, the *semantic blocks* will allow the definition of the computational monad. The system includes a library of some specific kinds of monads (with their corresponding monad transformers) but the user can also define new blocks.

## 4 Semantic specifications

The main goal is to obtain extensible and reusable semantic descriptions which will form the basis for different programming languages. In general, the semantic specification of a programming language can be obtained as a function $\mu\,\mathsf{F} \;\rightarrow\; \mathsf{M}\;\mathsf{V}$ where:

- $\mathsf{M}$ is the computational monad which can be defined as $(\mathcal{T}_1 \,.\, \mathcal{T}_2 \,\ldots\, \mathcal{T}_n)\,\mathsf{M}'$ where $\mathcal{T}_i$ is a monad transformer that adds some notion of computation and $\mathsf{M}'$ is the base monad. In this way, it is possible to add or remove computational features to a programming language without changing the rest of the specification.

- $\mathsf{V}$ is the value type. It can be defined using extensible union types which facilitate the incremental extension of value types. To achieve this, we use multi-parameter type classes with overlapping instances currently implemented in the main Haskell systems. A more detailed presentation of this approach can be found in [38]. In the rest of the paper, we assume that the components of a value $\alpha \,\|\, \beta$ are subtypes of it, and that, if $\alpha$ is a subtype of $\gamma$, then we have the operations $\uparrow\colon \alpha \to \gamma$ and $\downarrow\colon \gamma \to \alpha$.
- $\mu\,\mathsf{F}$ is the fixpoint of a functor $\mathsf{F}$ that describes the shape of the abstract syntax tree. $\mathsf{F}$ can usually be decomposed as $\mathsf{F}_1 \oplus \mathsf{F}_2 \oplus \ldots \oplus \mathsf{F}_n$ where $\mathsf{F}_i$ are different pattern functors that capture syntactic entities as arithmetic expressions, comparisons, declarations, etc. For each $\mathsf{F}_i$ we define an $\mathsf{F}_i$-algebra or a monadic $\mathsf{F}_i$-algebra.

  Therefore, the interpreter function $\mu\mathsf{F} \to \mathsf{M}\,\mathsf{V}$ can be obtained as a catamorphism or a monadic catamorphism.

## 5 Interactive Framework

We have implemented an application which allows runtime selection of interpreted programming languages and provides a common framework for language testing. In order to use programming languages of different types in the same data structure, we used the approach described in [34] combining existential types with type classes.

In order to integrate a new language to be interpreted under our framework, it is necessary to supply the parser, the pretty printer and the semantic specification. We use the *Parsec* combinator library [35] which is based on the parser combinators described in [22], but the system does not depend on any particular parser library. Regarding pretty printing, we use the library developed in [21]. As in the case of parsing, the system does not depend on this particular library.

The interactive framework can be configured with a list of languages

$$L_s = [l_1, l_2, \ldots, l_n]$$

At any moment the system contains an active programming language $l_i \in L_s$ and it allows the following operations:

- Loading a program $p_i$ written in the current language $l_i$.
- Execute that program $p_i$.
- Select a different language.
- Interrupt and debug the language that it is executing.
- Show information about the loaded program and the current language.

We have implemented descriptions of simple imperative, functional, object-oriented and logic programming languages.

## 6 Specification of MLambda

As an example, in this section we apply LPS to the specification of *MLambda*, a simple functional language with some imperative features.

### *6.1 Syntactical Structure*

In order to simplify the presentation, the syntactical structure of MLambda consists of a single category of expressions. It will be divided in different syntactical components which will allow an independent semantic specification. The syntactical components will be:

- Arithmetic expressions.

$$Arith\, x \triangleq Num\, Int \mid x + x \mid x - x \mid x \times x \mid x \div x$$

- *Boolean expressions*

$$Boolean\, x \triangleq B\, Bool \mid x \wedge x \mid x \vee x$$

- *Comparisons*

$$Cmp\, x \triangleq x < x \mid x > x \mid x \leq x \mid x \geq x \mid x = x \mid x \neq x$$

- *Variables*

$$Var\, x \triangleq V\, Name$$

- *References and assignments*

$$\mathsf{Ref}\, x \triangleq ref\ x \mid !\, x \mid x := x \mid x\, ;\, x$$

  This block offers reference variables and assignments. *ref* $e$ allocates a new location in the heap with the value of $e$ and returns the new location, $!\, x$ obtains the value from the position referenced by the value of $e$, $e_1 := e_2$ assigns the value of $e_2$ to the position referenced by the value of $e_1$, and finally, $e_1\, ;\, e_2$ evaluates $e_2$ after $e_1$.

- *Functional Abstractions*

$$\mathsf{Func}\, x \triangleq \lambda_N\ Name\ x \mid \lambda_V\ Name\ x \mid \lambda_L\ Name\ x \mid x\, @\, x$$

  $\lambda_X\ n\ e$ indicates lambda abstraction (for example, $\lambda n \to n + 3$). We use three different types of evaluation, by name ($\lambda_N$), by value ($\lambda_V$) and lazy ($\lambda_L$). $e_1\, @\, e_2$ indicates the application of $e_1$ to $e_2$.

- *Local Declarations*

$$\mathsf{Dec}\, x \triangleq Let_N\ Name\ x\ x \mid Let_V\ Name\ x\ x \mid Let_L\ Name\ x\ x$$

$Let_X\ n\ e_1\ e_2$ indicates the evaluation of $e_2$ assigning the value of $e_1$ to $x$. We will allow recursive evaluation in three ways, by name ($Let_N$), by value ($Let_V$) and lazy ($Let_L$).

- *First class continuations*

$$\mathsf{Callcc}\, x \triangleq Callcc$$

The language can be defined as the fixpoint of the sum of the defined functors

$$\mathcal{L} = \mu(\mathsf{Arith} \oplus \mathsf{Boolean} \oplus \mathsf{Cmp} \oplus \mathsf{Var} \oplus \mathsf{Ref} \oplus \mathsf{Func} \oplus \mathsf{Dec})$$

### *6.2 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: environment access, state update, partial computations, and continuations.

The resulting monad can be obtained applying the corresponding monad transformers to a base monad. In this example, we use the identity monad Id as the base monad but in a more practical language we could have been used other monads, like the predefined IO monad to obtain direct communication with the external world.

This computational structure is defined as

$$\mathsf{Comp} \triangleq (\mathcal{T}_{Err} . \mathcal{T}_{State} . \mathcal{T}_{Env} . \mathcal{T}_{Cont})\ \mathsf{Id}$$

### *6.3 Domain value*

The domain value will consist of two primitive types, integers and booleans, and the combined type of functions. Functions will be represented as values of type $\mathsf{Comp}\ Value \rightarrow \mathsf{Comp}\ Value$. The Domain Value can be described as:

$$Value \triangleq Int \| Bool \| Loc \| \mathsf{Comp}\ Value \rightarrow \mathsf{Comp}\ Value$$

### *6.4 Semantic Specification*

#### *6.4.1 Auxiliary Functions*

In order to facilitate the semantic specifications, we declare some auxiliary functions.

- *evalWith* will be used for arithmetic and boolean evaluation. In the following definitions, $\alpha, \beta$ are considered subtypes of $\gamma$.

$$\begin{aligned} &\mathit{evalWith} &&: (\alpha \to \alpha \to \beta) \to \gamma \to \gamma \to \mathsf{M}\,\gamma \\ &\mathit{evalWith}\ \odot\ v_x\ v_y &&= \mathit{return}\ \uparrow (\downarrow v_x \odot \downarrow v_y) \end{aligned}$$

- Although we are not going to present the whole implementation, we assume that we have some utility modules implementing common data structures. *Heap* $\alpha$ is an abstract datatype addressed by locations of type *Loc* with the following operations:

$$\begin{aligned} &\mathit{alloc}_H &&: \alpha \to \mathit{Heap}\ \alpha \to (\mathit{Loc}, \mathit{Heap}\ \alpha) && \text{— allocate new values} \\ &\mathit{lkp}_H &&: \mathit{Loc} \to \mathit{Heap}\ \alpha \to \alpha && \text{— lookup} \\ &\mathit{upd}_H &&: \mathit{Loc} \to \alpha \to \mathit{Heap}\ \alpha \to \mathit{Heap}\ \alpha && \text{— update} \end{aligned}$$

  We will also use a *Table* $\alpha$ data structure with the following operations:

$$\begin{aligned} &\mathit{lkp}_T &&: \mathit{Name} \to \mathit{Table}\ \alpha \to \alpha && \text{— lookup} \\ &\mathit{upd}_T &&: \mathit{Name} \to \alpha \to \mathit{Table}\ \alpha \to \mathit{Table}\ \alpha && \text{— update} \end{aligned}$$

  We will store computations in both structures, i.e. the environment will be a value of type *Table* (Comp *Value*) and the state will be a value of type *Heap* (Comp *Value*)

### *6.4.2 Algebras and Monadic Algebras*

For each of the syntactical components we must specify an algebra or a monadic algebra. If the component always requires the recursive evaluation of subcomponents, we only need a monadic algebra $\varpi$, otherwise, we need an algebra $\varphi$.

- Arithmetic Expressions

$$\begin{aligned} &\varpi_{\mathsf{Arith}}\,[\![\mathit{Num}\ n]\!] &&= \mathit{return}\ (\uparrow n) \\ &\varpi_{\mathsf{Arith}}\,[\![x + y]\!] &&= \mathit{evalWith}\ (+)\ x\ y \\ &\varpi_{\mathsf{Arith}}\,[\![x - y]\!] &&= \mathit{evalWith}\ (-)\ x\ y \\ &\varpi_{\mathsf{Arith}}\,[\![x \times y]\!] &&= \mathit{evalWith}\ (\times)\ x\ y \\ &\varpi_{\mathsf{Arith}}\,[\![x \div y]\!] &&= \mathit{evalWith}\ (\div)\ x\ y \end{aligned}$$

- Boolean Expressions

$$\begin{aligned} &\varpi_{\mathsf{Boolean}}\,[\![B\ b]\!] &&= \mathit{return}\ (\uparrow b) \\ &\varpi_{\mathsf{Boolean}}\,[\![x \wedge y]\!] &&= \mathit{evalWith}\ (\wedge)\ x\ y \\ &\varpi_{\mathsf{Boolean}}\,[\![x \vee y]\!] &&= \mathit{evalWith}\ (\vee)\ x\ y \end{aligned}$$

- Comparisons

$$\varpi_{\mathsf{Cmp}}\,[\![x > y]\!] = \mathit{evalWith}\ (>)\ x\ y$$

$$
\begin{aligned}
\varpi_{\mathsf{Cmp}}\, [\![x\ <\ y]\!] &= \mathit{evalWith}\ (<)\ x\ y\\
\varpi_{\mathsf{Cmp}}\, [\![x\ \geq\ y]\!] &= \mathit{evalWith}\ (\geq)\ x\ y\\
\varpi_{\mathsf{Cmp}}\, [\![x\ \leq\ y]\!] &= \mathit{evalWith}\ (\leq)\ x\ y\\
\varpi_{\mathsf{Cmp}}\, [\![x\ ==\ y]\!] &= \mathit{evalWith}\ (==)\ x\ y\\
\varpi_{\mathsf{Cmp}}\, [\![x\ \neq\ y]\!] &= \mathit{evalWith}\ (\neq)\ x\ y
\end{aligned}
$$

- Variables. To obtain the value of a variable we only need to access the environment and to search the name in the symbol table.

$$
\begin{aligned}
\varpi_{\mathsf{Var}}\, [\![V\ x]\!] \ = \ &\mathbf{do}\\
&\quad \rho \ \leftarrow\ \mathit{rdEnv}\\
&\quad \mathit{lkp}_T\ x\ \rho
\end{aligned}
$$

- References and assignments. We will need to change the state so we will have to use the operators *fetch*, *set* and *update* from SM.

$$
\begin{aligned}
\varphi_{\mathsf{Ref}}\, [\![\mathit{ref}\ \ e]\!] \ = \ &\mathbf{do}\\
&\quad v\ \leftarrow\ e\\
&\quad h\ \leftarrow\ \mathit{fetch}\\
&\quad \mathbf{let}\ (\mathit{loc}, h')\ =\ \mathit{alloc}_H\ (\mathit{return}\ v)\ h\\
&\quad \mathit{set}\ h'\\
&\quad \mathit{return}\ \mathit{loc}
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{\mathsf{Ref}}\, [\![!\ e]\!] \ = \ &\mathbf{do}\\
&\quad v_{loc}\ \leftarrow\ e\\
&\quad h\ \leftarrow\ \mathit{fetch}\\
&\quad \mathit{lkp}_H\ (\downarrow v_{loc})\ h
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{\mathsf{Ref}}\, [\![e_1\ :=\ e_2]\!] \ = \ &\mathbf{do}\\
&\quad v_{loc}\ \leftarrow\ e_1\\
&\quad v\ \leftarrow\ e_2\\
&\quad \mathit{update}\ (\mathit{upd}_H\ (\downarrow v_{loc})\ (\mathit{return}\ v))\\
&\quad \mathit{return}\ v
\end{aligned}
$$

$$
\varphi_{\mathsf{Ref}}\, [\![e_1\ ;\ e_2]\!] \ = \ \mathbf{do}\ \{\ e_1\ ;\ e_2\ \}
$$

- Functional abstractions. In this specification we show the difference between different evaluation mechanisms. Either we evaluate before returning the function ($\lambda_V$), we return the function that will evaluate the argument if it is needed ($\lambda_N$) or we create a thunk that will only be evaluated the first time it is needed ($\lambda_L$).

$$
\begin{aligned}
\varphi_{\mathsf{Fun}}\, [\![\lambda_V\ x\ e]\!] \ = \ & \mathbf{do} \\
& \rho \leftarrow rdEnv \\
& return\ (\uparrow\ (\lambda m \rightarrow \mathbf{do} \\
& \qquad\qquad v \leftarrow m \\
& \qquad\qquad inEnv\ (upd_T\ x\ (return\ v)\ \rho)\ e \\
& ))
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{\mathsf{Fun}}\, [\![\lambda_N\ x\ e]\!] \ = \ & \mathbf{do} \\
& \rho \leftarrow rdEnv \\
& return\ (\uparrow (\lambda m \rightarrow inEnv\ (upd_T\ x\ m\ \rho)\ e\ ))
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{\mathsf{Fun}}\, [\![\lambda_L\ x\ e]\!] \ = \ & \mathbf{do} \\
& \rho \leftarrow rdEnv \\
& return\ (\uparrow (\lambda m \rightarrow \mathbf{do} \\
& \qquad\qquad h \leftarrow fetch \\
& \qquad\qquad \mathbf{let}\ (loc, h') \ = \ alloc_H\ m\ h \\
& \qquad\qquad set\ (upd_H\ loc\ (mkThunk\ loc\ m)\ h') \\
& \qquad\qquad inEnv\ (upd_T\ x\ (fetch \gg\!= lkp_H\ loc)\ \rho)\ e \\
& ))
\end{aligned}
$$

$$
\begin{aligned}
\varphi_{\mathsf{Fun}}\, [\![e_1 \mathbin{@} e_2]\!] \ = \ & \mathbf{do} \\
& v_f \leftarrow e_1 \\
& \rho \leftarrow rdEnv \\
& (\downarrow v_f)\ (inEnv\ \rho\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
& mkThunk \quad : \ Loc \rightarrow \mathsf{Comp}\ Value \rightarrow \mathsf{Comp}\ Value \\
& mkThunk\ loc\ m \ = \ \mathbf{do} \\
& \qquad v \leftarrow m \\
& \qquad update\ (upd_H\ loc\ (return\ v)) \\
& \qquad return\ v
\end{aligned}
$$

- Local declarations. We create a new location to store the local declaration in the heap and, depending on the evaluation mechanism, we evaluate $e_1$ and store its value in that location before calling $e_2$ ($LET_V$), we store in that location the computation that will evaluate $e_1$ when it is needed ($LET_N$), or we create and store in that location a thunk that will only evaluate $e_1$ the first time it is needed ($LET_L$).

$$
\begin{aligned}
\varphi_{\mathsf{Dec}}\, [\![Let_V\ x\ e_1\ e_2]\!] \ = \ & \mathbf{do} \\
& (loc, \rho) \leftarrow prepareDecl\ e_1\ x \\
& v \leftarrow inEnv\ \rho\ e_1 \\
& update\ (upd_H\ loc\ (return\ v)) \\
& inEnv\ \rho\ e_2
\end{aligned}
$$

$$\begin{aligned}\varphi_{\mathsf{Dec}}\,[\![Let_N\ x\ e_1\ e_2]\!] \;=\; & \mathbf{do}\\ & (loc,\rho) \leftarrow prepareDecl\ e_1\ x\\ & update\ (upd_H\ loc\ (inEnv\ \rho\ e_1))\\ & inEnv\ \rho\ e_2\end{aligned}$$

$$\begin{aligned}\varphi_{\mathsf{Dec}}\,[\![Let_L\ x\ e_1\ e_2]\!] \;=\; & \mathbf{do}\\ & (loc,\rho) \leftarrow prepareDecl\ e_1\ x\\ & update\ (upd_H\ loc\ (mkThunk\ loc\ (inEnv\ \rho\ e_1)))\\ & inEnv\ \rho\ e_2\end{aligned}$$

$$\begin{aligned}& prepareDecl \quad : \ \mathsf{Comp}\ Value \rightarrow\ Name \rightarrow \mathsf{Comp}\ (Loc,\ Env)\\ & prepareDecl\ m\ x \ = \ \mathbf{do}\\ & \qquad h \leftarrow fetch\\ & \qquad \mathbf{let}\ (loc, h') \ = \ alloc_H\ m\ h\\ & \qquad set\ h'\\ & \qquad \rho \leftarrow rdEnv\\ & \qquad return\ (loc,\ upd_T\ x\ (fetch \ggg\!= lkp_H\ loc)\ \rho)\end{aligned}$$

- First class continuations are directly obtained using the *callcc* operator from the *ContM*.

$$\varphi_{\mathsf{Callcc}}\,[\![Callcc]\!] = return\ (\uparrow fcc)$$

$$\begin{aligned}& \mathbf{where}\\ & \quad fcc\ m \ = \ \mathbf{do}\\ & \qquad v_f \leftarrow m\\ & \qquad callcc\ (\lambda\kappa \rightarrow (\downarrow v_f)\ (return\ (\uparrow \kappa)))\end{aligned}$$

Once we have specified the algebras and monadic algebras, we define the corresponding interpreter as a combination of catamorphism and monadic catamorphism:

$$\begin{aligned}& \varpi_{\mathcal{L}} \ : \ (\mathsf{Arith} \oplus \mathsf{Boolean} \oplus \mathsf{Cmp} \oplus \mathsf{Var})\ Value \rightarrow \mathsf{Comp}\ Value\\ & \varpi_{\mathcal{L}} \ = \varpi_{\mathsf{Arith}} \oplus_{\mathsf{m}} \varpi_{\mathsf{Boolean}} \oplus_{\mathsf{m}} \varpi_{\mathsf{Cmp}} \oplus \varpi_{\mathsf{Var}}\end{aligned}$$

$$\begin{aligned}& \varphi_{\mathcal{L}} \ : \ (\mathsf{Ref} \oplus \mathsf{Fun} \oplus \mathsf{Dec})(\mathsf{Comp}\ Value) \rightarrow \mathsf{Comp}\ Value\\ & \varphi_{\mathcal{L}} \ = \varphi_{\mathsf{Ref}} \oplus \varphi_{\mathsf{Fun}} \oplus \varphi_{\mathsf{Dec}}\end{aligned}$$

$$\begin{aligned}& \mathsf{Inter}_{\mathcal{L}} \ : \ \mathcal{L} \rightarrow \mathsf{Comp}\ Value\\ & \mathsf{Inter}_{\mathcal{L}} \ = (\![\varpi_{\mathcal{L}}, \varphi_{\mathcal{L}}]\!)\end{aligned}$$

## 7 Conclusions and future work

The Language Prototyping System is a combination of generic programming concepts and modular monadic semantics, which offers a *very* modular way to specify the semantics of programming languages. It allows the definition of reusable semantic blocks and provides an interactive system for language testing.

The system can be considered as another example of a domain-specific language embedded in Haskell [20,28,51]. This approach has some advantages: the development is easier as we can rely on the fairly good type system of Haskell, it is possible to obtain direct access to Haskell libraries and tools, and we do not need to define a new language with its syntax, semantics, type system, etc. At the same time, the main disadvantages are the mixture of error messages from the domain-specific language and the host language, some Haskell type system limitations and the Haskell dependency, which impedes obtaining executable prototypes implemented in other languages. At this moment we are assessing whether it would be better to define an independent domain specific meta-language for monadic semantic specifications. Some work in this direction can be found in [42,5].

On the theoretical side, [17] shows how to derive a backtracking monad transformer from its specification. That approach should be applied to other types of monad transformers in order to prove the correctness of the system. It would be interesting to study the combination of algebras, coalgebras, monads and comonads to provide the semantics of interactive and object-oriented features [4,24,23,29,49]. Another line of research is the automatic derivation of compilers from the obtained interpreters. This line has already been started in [13,14].

LPS allows the definition of a language from reusable semantic building blocks. In [8], the same problem is solved in the Action Semantics framework. In our approach, however, there are no conflicts leading to inconsistencies because the combined constructions belong to different abstract syntax entities. It would be very interesting to make deeper comparisons of the modularity of semantic specification techniques as has been started in [44,33].

With regard to the implementation, we have also made a simple version of the system using first-class polymorphism [25] and extensible records [12]. 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. The current implementation could benefit from recent advances in generic programming [18] which would allow the automatic generation of some definitions. Although there is a proposal for a Generic Haskell [16], it has not been implemented yet.

In order to obtain a complete language design assistant [15] it would be interesting to develop a Graphical User Interface or to integrate LPS in other tools like the ASF+SDF Meta-Environment [50].

Finally, the initial goal of our research was the development of prototypes for the abstract machines underlying the Integral Object-Oriented Operating System Oviedo3 [2] with the aim to test new features as security, concurrency, reflectiveness and distribution [7,48]. More information on the system can be obtained at [1].

## References


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

[2] D. Álvarez, L. Tajes, F. Álvarez, M. A. Díaz, R. Izquierdo, and J. M. Cueva. An object-oriented abstract machine as the substrate for an object-oriented operating system. In J. Bosch and S. Mitchell, editors, *Object Oriented Technology ECOOP'97*, Jÿvaskÿla, Finland, June 1997. Springer Verlag, LNCS 1357.

[3] 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.

[4] L. S. Barbosa. Components as processes: An exercise in coalgebraic modeling. In S. F. Smith and C. L. Talcott, editors, *FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems*, pages 397–417, Stanford, USA, September 2000. Kluwer Academic Publishers.

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

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

[7] M. A. Díaz, D. Álvarez, A. García-Mendoza, F. Álvarez, L. Tajes, and J. M. Cueva. Merging capabilities with the object model of an object-oriented abstract machine. In S. Demeyer and J. Bosch, editors, *Ecoop'98 Workshop on Distributed Object Security*, volume 1543, pages 273–276. LNCS, 1998.

[8] K. Doh and P. Mosses. Composing programming languages by combining action-semantics modules. In M. van den Brand, M. Mernik, and D. Parigot, editors, *First workshop on Language, Descriptions, Tools and Applications*, Genova, Italy, April 2001.

[9] Luc Duponcheel. Writing modular interpreters using catamorphisms, subtypes and monad transformers. Utrecht University, 1995.

[10] David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[11] Maarten M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, Dept. of Computer Science, Univ. of Twente, June 1994.

[12] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

[13] William Harrison and Samuel Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.

[14] William Harrison and Samuel Kamin. Compilation as metacomputation: Binding time separation in modular compilers. In *5th Mathematics of Program Construction Conference, MPC2000*, Ponte de Lima, Portugal, June 2000.

[15] Jan Heering. Application software, domain-specific languages and language design assistants. In *Proceedings SSGRR 2000 International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, L'Aquila, Italy, 2000.

[16] R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28.

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

[18] Ralf Hinze. A new approach to generic functional programming. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.

[19] Z. Hu and H. Iwasaki. Promotional transformation of monadic programs. In *Workshop on Functional and Logic Programming*, Susono, Japan, 1995. World Scientific.

[20] P. Hudak. Domain-specific languages. In Peter H. Salus, editor, *Handbook of Programming Languages*, volume III, Little Languages and Tools. Macmillan Technical Publishing, 1998.

[21] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *First International School on Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[22] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, December 1996.

[23] B. Jacobs and E. Poll. A monad for basic java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology*, number 1816 in LNCS. Springer, 2000.

[24] Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. In *Coalgebraic Methods in Computer Science*, number 11. Electronic Notes in Computer Science, 1998.

[25] 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.

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

[27] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999.

[28] S. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science, Elsevier Press*, 12, 1998.

[29] Richard B. Kieburtz. Designing and implementing closed domain-specific languages. Invited talk at the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG), 2000.

[30] 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.

[31] 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.

[32] 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.

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

[34] Konstantin Laufer. Combining type classes and existential types. In *Proceedings of the Latin American Informatics Conference*, Mexico, 1994.

[35] Daan Leijen. Parsec. `http://www.cs.uu.nl/~daan/parsec.html`, 2000.

[36] Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Graduate School of Yale University, May 1998.

[37] 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.

[38] 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.

[39] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[40] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.

[41] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 228–266. Springer-Verlag, 1995.

[42] 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.

[43] 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.

[44] P. D. Mosses. Semantics, modularity, and rewriting logic. In C.Kirchner and H. Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

[45] 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.

[46] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM, New York, 1993.

[47] Guy L. Steele, Jr. Building interpreters by composing monads. In *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, USA, 1994. ACM Press.

[48] L. Tajes, F. Álvarez, D. Álvarez, M. A. Díaz, and J.M. Cueva. A computational model for a distributed object-oriented operating system based on a reflective abstract machine. In S. Demeyer and J. Bosch, editors, *Ecoop'98 Workshop on Reflective Object-Oriented Programming and Systems*, volume 1543, pages 382–383. LNCS, 1998.

[49] Daniele Turi and Jan Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8(5):481–540, 1998.

[50] M. van den Brand et al. The asf+sdf meta-environment: a component-based language development environment. In *Compiler Construction*, LNCS. Springer-Verlag, 2001.

[51] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

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

[53] P. Wadler. The essence of functional programming. In *POPL '92, Albuquerque*, 1992.

[54] Keith Wansbrough and John Hamer. A modular monadic action semantics. In *The Conference on Domain-Specific Languages*, pages 157–170, Santa Barbara, California, 1997. The USENIX Association.