Reusable Monadic Semantics of Object Oriented Programming Languages

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

[1]Department of Computer Science, University of Oviedo
C/ Calvo Sotelo S/N, 3307, Oviedo, Spain

{labra,cueva,candi,guti}@lsi.uniovi.es

Abstract. *We specify the dynamic semantics of an object oriented programming language in an incremental way. We begin with a simple language of arithmetic and boolean expressions. Then, we add functional abstractions, local declarations, references and assignments obtaining a functional language with imperative features. We finally add objects, classes and subclasses to obtain a prototypical object oriented language with dynamic binding.*
Reusable monadic semantics is a combination of modular monadic semantics and generic programming concepts. The computational structure is defined as a monad obtained from the composition of several monad transformers. The syntax is obtained as the fixpoint of several non-recursive pattern functors. For each functor, an algebra that takes the computational structure as carrier is independently defined. In this way, an interpreter is automatically obtained as a catamorphism over the sum of those algebras.

1. Introduction

E. Moggi [Mog89, Mog91] applied monads to denotational semantics in order to capture the notion of computation and the intuitive idea of separating computations from values. The monadic approach has the problem that, in general, it is not possible to compose two monads to obtain a new monad [JD93]. A proposed solution was the use of monad transformers [LHJ95] which transform a given monad into a new one adding new operations. This approach was called modular monadic semantics [LH96].

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 [BJJM99]. L. Duponcheel [Dup95] proposed the combined use of folds or catamorphisms with modular monadic semantics which enables the independent specification of the abstract syntax, the computational monad and the domain value.

Following that approach, we have developed a Language Prototyping System as an embedded domain-specific language in Haskell [Lab98, LCL99, LCLC01a]. The system also facilitates the combination of monads and catamorphisms and the definition of n-catamorphisms [LCLG01].

A lot of work has been done in the semantic specification of object oriented languages [Car84, Red88, CP94, GM94, Bou01]. The main contribution of this paper is to apply the integration of modular monadic semantics and generic programming concepts to specify the dynamic semantics of an object oriented language using the closure-based model [Red88, KR94].

The paper is organized as follows. In section 2. we give an informal presentation of modular monadic semantics and in section 3. we present the basic generic programming concepts. As an example, we specify a simple language of arithmetic and boolean expressions. In section 4. we extend the language to include functional and imperative features. Finally, in section 5. we extend that language with objects, classes and subclasses.

It is assumed that the reader has some familiarity with a modern functional programming language. Throughout the paper, we use Haskell [JHA+99] notation with some freedom in the use of mathematical symbols and declarations.

2. Modular Monadic Semantics

Definition 1 (Monad) *In functional programming, a monad can be defined as a type constructor* M *with two operations*

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

which satisfy

$$\begin{array}{ll} c \gg\!=_{\mathsf{M}} return_{\mathsf{M}} & \equiv c \\ (return_{\mathsf{M}}\ a) \gg\!=_{\mathsf{M}} k & \equiv k\ a \\ (m \gg\!=_{\mathsf{M}} f \gg\!=_{\mathsf{M}} h & \equiv m \gg\!=_{\mathsf{M}} (\lambda a. f\ a \gg\!=_{\mathsf{M}} h) \end{array}$$

A monad M encapsulates the intuitive notion of computation where $\mathsf{M}\,\alpha$ can be considered as a computation M that returns a value of type α. In Haskell, monads can be defined using constructor classes [Jon95] and it is also possible to use first-class polymorphism which allows one to define monads as first class values [Jon97]. In the rest of the paper, we simply define the type constructor and the corresponding operations and we omit the M subscript when it is clear from the context. We will also use the operator $(\gg)$ defined as

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

Example 1 *The simplest monad is the identity monad*

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

It is possible to define monads that capture different kinds of computations, like partiality, nondeterminism, side-effects, exceptions, continuations, interactions, etc. [Mog89, BHM00].

Example 2 *A monad* M *is an environment reader monad for a given environment* Env *if it has operations with the following types*

$$\begin{array}{ll} rdEnv & : \mathsf{M}\ Env \\ inEnv & : Env \rightarrow \mathsf{M}\ \alpha \rightarrow \mathsf{M}\ \alpha \end{array}$$

which satisfy a number of laws (see [LH96])

Example 3 *A monad* M *is a state transformer monad for a given type* $State$ *if it has operations with the following types*

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

When describing the semantics of a programming language using monads, the main problem is the combination of different classes of monads. In general, it is not possible to compose two monads to obtain a new monad [JD93]. 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 [Mog89] and was later proposed in [LHJ95].

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

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

When defining a monad transformer $\mathcal{T}$, it is necessary to specify the operations $return$, $(\gg\!=)$, $lift$ and the specific operations that the monad transformer adds. The definition of monad transformers is not straightforward because there can be some interactions between the operations of the different monads. These interactions are considered in more detail in [LH96, LHJ95] and in [Hin00] it is shown how to derive from its specification a monad transformer that adds computations with backtracking. In the rest of the paper we suppose that we have defined a monad transformer $\mathcal{T}_{Env}$ that transforms any monad into an environment reader monad and $\mathcal{T}_{State}$ that transforms any monad into a state transformer monad. These definitions can be found in [LCLC01a, LLCC01, LCLG01].

An important feature to facilitate domain value extensibility is the subtyping mechanism defined in [LHJ95] using multi-parameter type classes with overlapping instances. Although we are not going to give the full details, we can assume that if α is a subtype of β, which will be denoted as $\alpha \subseteq \beta$, then we have $\uparrow : \alpha \rightarrow \beta$ and $\downarrow : \beta \rightarrow \alpha$. We also assume that $\alpha \subseteq (\alpha \| \beta)$ and that $\beta \subseteq (\alpha \| \beta)$ where $\alpha \| \beta \triangleq L\ \alpha \mid R\ \beta$. In practice, it would be necessary to take into account that the downcast operator $\downarrow$ is partial.

3. Generic Programming concepts

Definition 3 *A functor* F *can be defined as a type constructor that transforms values of type* α *into values of type* $\mathsf{F}\ \alpha$ *and a function*

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

which preserves identities and composition.

The fixpoint of a functor F can be defined as

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

Notice that we have explicitly written the constructor $In : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F}$ because it will play a special role in the definition of catamorphism.

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

Example 4 *The following inductive datatype for arithmetic expressions*

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

can be defined as the fixpoint of the functor T

$$\begin{array}{lcl} \mathsf{T}\ x & \triangleq & N\ Int\ \mid\ x\ +\ x \\ Term & \triangleq & \mu\mathsf{T} \end{array}$$

where map_{T} *is*[1]

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

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

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

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

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

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

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

Example 5 *We can define a new pattern functor for boolean expressions*

$$\mathsf{B}\,x \;=\; B\;Bool \;\mid\; x == x$$

and the composed recursive datatype of arithmetic and boolean expressions can easily be defined as

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

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

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

where α *is called the carrier.*

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

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

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

$(\!|\varphi|\!)$ is called *fold* or *catamorphism* and satisfies a number of calculational properties [BJJM99, BdM97, MFP91, SF93]. It can be defined as:

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

Example 6 *We can obtain a simple evaluator for arithmetic expressions defining a* T*-algebra whose carrier is the type* $\mathsf{m}\;v$*, where* m *is, in this case, any kind of monad, and* Int *is a subtype of* v*.*

$$\begin{array}{lll}
\varphi_{\mathsf{T}} & : & (Monad\;\mathsf{m},\;Int \subseteq v) \Rightarrow\; \mathsf{T}(\mathsf{m}\;v) \to \mathsf{m}\;v \\
\varphi_{\mathsf{T}}\;(Num\;n) & = & return\;(\uparrow n) \\
\varphi_{\mathsf{T}}\;(c_1\;+\;c_2) & = & c_1 \gg\!\!= \lambda v_1.c_2 \gg\!\!= \lambda v_2.return(\uparrow(\downarrow v_1 + \downarrow v_2))
\end{array}$$

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

$$\begin{array}{ll}
\mathsf{Inter}_{Term} & :\;\; (Monad\;\mathsf{m}, Int \subseteq v) \Rightarrow\;\; Term \;\to\; \mathsf{m}\;v \\
\mathsf{Inter}_{Term} & = (\!|\varphi_{\mathsf{T}}|\!)
\end{array}$$

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

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

Example 7 *The above definition allows one to extend the evaluator of example 6 to arithmetic and boolean expressions without modifying previous definitions. We only specify the semantics of boolean expressions with the following* B*-algebra*

$$
\begin{array}{ll}
\varphi_{\mathsf{B}} & : \; (\mathit{Monad}\ \mathsf{m}, \mathit{Bool} \subseteq v) \Rightarrow \; \mathsf{B}(\mathsf{m}\ v) \to \mathsf{m}\ v \\
\varphi_{\mathsf{B}}\ (B\ b) & = \mathit{return}\ (\uparrow b) \\
\varphi_{\mathsf{B}}\ (c_1\ ==\ c_2) & = c_1 \gg\!\!= \lambda v_1 . c_2 \gg\!\!= \lambda v_2 . \mathit{return}(\uparrow (\downarrow v_1\ == \downarrow v_2))
\end{array}
$$

The new interpreter of expressions is automatically obtained as:

$$
\begin{array}{l}
\mathsf{Inter}_{\mathit{Expr}} \; : \; (\mathit{Monad}\ \mathsf{m},\ \mathit{Int} \subseteq v, \mathit{Bool} \subseteq v) \Rightarrow \mathit{Expr} \to \mathsf{m}\ v \\
\mathsf{Inter}_{\mathit{Expr}} \; = (\!|\varphi_{\mathsf{T}} \oplus \varphi_{\mathsf{B}}|\!)
\end{array}
$$

The theory of catamorphisms can be extended to monadic catamorphisms [Lab98, LCLC01a, LLCC01] and bicatamorphisms [LCLG01] to handle mutually recursive syntactic categories.

4. Functional Language with imperative features

In this section we extend the simple language of expressions to that of a simple functional language with call by value and imperative features similar to ML. The syntax is captured by the following functors:

- F captures functional abstractions, local declarations and variables

$$
\begin{array}{lll}
\mathsf{F}\ e \triangleq & \lambda\ \mathit{Name}\ e & \text{— lambda abstraction} \\
\mid & e\ @\ e & \text{— application} \\
\mid & \mathit{Let}\ \mathit{Name}\ e\ e & \text{— recursive local declarations} \\
\mid & \mathit{Var}\ \mathit{Name} & \text{— variables}
\end{array}
$$

- R captures imperative features

$$
\begin{array}{lll}
\mathsf{R}\ e \triangleq & \mathit{ref}\ e & \text{— new reference} \\
\mid & !\ e & \text{— value of reference} \\
\mid & e\ :=\ e & \text{— assignment} \\
\mid & e\ ;\ e & \text{— sequence}
\end{array}
$$

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:

$$\begin{array}{lll} alloc_H & : \alpha \to Heap\ \alpha \to (Loc, Heap\ \alpha) & \text{— allocate new values} \\ lkp_H & : Loc \to\ Heap\ \alpha \to \alpha & \text{— lookup} \\ upd_H & : Loc \to \alpha \to Heap\ \alpha \to Heap\ \alpha & \text{— update} \end{array}$$

$Table\ \alpha$ is an abstract datatype representing tables addressed by values of type $Name$ with the following operations:

$$\begin{array}{lll} lkp_T & : Name \to Table\ \alpha \to \alpha & \text{— lookup} \\ upd_T & : Name \to \alpha \to Table\ \alpha \to Table\ \alpha & \text{— update} \\ (+\!\!+) & : Table\ \alpha \to Table\ \alpha \to Table\ \alpha & \text{— join two tables} \end{array}$$

In order to specify the semantics of this language, the domain value contains integers, booleans, locations and functions.

$$Value \triangleq Integer \parallel Bool \parallel Loc \parallel Function$$

The computational structure is defined as the composition of $\mathcal{T}_{State}$ and $\mathcal{T}_{Env}$ applied to the identity monad.

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

Functional values are represented as functions over computations

$$Function \triangleq \mathsf{Comp}\ Value \to \mathsf{Comp}\ Value$$

The environment is modelled as a table and the state is modelled as a heap, both storing computations.

$$\begin{array}{ll} Env & \triangleq Table\ (\mathsf{Comp}\ Value) \\ State & \triangleq Heap\ (\mathsf{Comp}\ Value) \end{array}$$

The semantic specification of functional expressions is defined using the following F-algebra

$$\begin{array}{ll} \varphi_{\mathsf{F}} & :\ \mathsf{F}\ (\mathsf{Comp}\ Value) \to \mathsf{Comp}\ Value \\ \varphi_{\mathsf{F}}\ (\lambda\ x\ c) & = rdEnv \gg\!= \lambda\rho. \\ & \quad return\ (\uparrow (\lambda m.m \gg\!= \lambda v.inEnv\ (upd_T\ x\ (return\ v)\ \rho)\ c)) \\ \varphi_{\mathsf{F}}\ (c_1\ @\ c_2) & = c_1 \gg\!= \lambda v. \\ & \quad rdEnv \gg\!= \lambda\rho. \\ & \quad (\uparrow\ v)\ (inEnv\ \rho\ c_2) \\ \varphi_{\mathsf{F}}\ (Let\ x\ c_1\ c_2) & = fetch \gg\!= \lambda\varsigma. \end{array}$$

$$
\begin{array}{lll}
 & & rdEnv \gg\!= \lambda\rho. \\
 & & \quad \mathbf{let} \\
 & & \qquad (loc, \varsigma') \ = alloc_H \ c_1 \ \varsigma \\
 & & \qquad \rho' \qquad\quad = upd_T \ x \ (fetch \gg\!= lkp_H \ loc) \ \rho \\
 & & \quad \mathbf{in} \\
 & & \quad set \ \varsigma' \gg \\
 & & \quad inEnv \ \rho' \ c_1 \gg\!= \lambda v. \\
 & & \quad update \ (upd_H \ loc \ (return \ v)) \gg \\
 & & \quad inEnv \ \rho' \ c_2 \\
\varphi_{\mathsf{F}} \ (Var \ x) & = & rdEnv \gg\!= lkp_T \ x
\end{array}
$$

The imperative features captured by the functor R are specified in the following R-algebra

$$
\begin{array}{lll}
\varphi_{\mathsf{R}} & : & \mathsf{R} \, (\mathsf{Comp} \ Value) \rightarrow \mathsf{Comp} \ Value \\
\varphi_{\mathsf{R}} \ (ref \ c) & = & c \gg\!= \lambda v. \\
 & & fetch \gg\!= \lambda\varsigma. \\
 & & \mathbf{let} \\
 & & \quad (loc, \varsigma') = alloc_H \ (return \ v) \ \varsigma \\
 & & \mathbf{in} \\
 & & set \ \varsigma' \gg \\
 & & return \ (\uparrow \ loc) \\
\varphi_{\mathsf{R}} \ (! \ c) & = & c \gg\!= \lambda v. \\
 & & fetch \gg\!= \lambda\varsigma. \\
 & & lkp_H \ (\downarrow \ v) \ \varsigma \\
\varphi_{\mathsf{R}} \ (c_1 \ := \ c_2) & = & c_1 \gg\!= \lambda v_1. \\
 & & c_2 \gg\!= \lambda v_2. \\
 & & update \ (upd_H \ (\downarrow \ v_1) \ (return \ v_2)) \\
 & & return \ v_2 \\
\varphi_{\mathsf{R}} \ (c_1 \ ; \ c_2) & = & c_1 \gg c_2
\end{array}
$$

The abstract syntax of the ML-like language is defined as

$$
\mathcal{L}_{ML} \ \triangleq \ \mu(\mathsf{T} \oplus \mathsf{B} \oplus \mathsf{F} \oplus \mathsf{R})
$$

and the corresponding interpreter is obtained as a catamorphism

$$
\begin{array}{ll}
\mathsf{Inter}_{\mathcal{L}_{ML}} & : \ \mathcal{L}_{ML} \, (\mathsf{Comp} \ Value) \rightarrow \mathsf{Comp} \ Value \\
\mathsf{Inter}_{\mathcal{L}_{ML}} & = (\!| \varphi_{\mathsf{T}} \oplus \varphi_{\mathsf{B}} \oplus \varphi_{\mathsf{F}} \oplus \varphi_{\mathsf{R}} |\!)
\end{array}
$$

The above specifications have been taken from a functional language with imperative features [LLCC01] and will be used in the following section without any change.

5. Object Oriented Language

In this section, we add objects, classes and subclasses to the previous language.

5.1. Objects

An object can be defined as a set of local variables and a set of methods which have access to the local variables. Outside the object, it is not possible to access the local variables, which can only be accessed through the execution of methods. The functor Obj captures the syntax of object creation and method selection.

$$\begin{array}{lll} \mathsf{Obj}\ c \triangleq & \mathit{Obj}\ [(\mathit{Name}, c)]\ [(\mathit{Label}, c)] & \text{— Object definition} \\ \mid & c \mapsto \mathit{Name} & \text{— Method selection} \end{array}$$

Example 8 *The following expression creates an object p with a local variable x and three methods. The expression evaluates to 2 after setting the local value x of p using the method set and obtaining its value using the method get.*

$$\begin{array}{l} \mathbf{let} \\ \quad p\ = \mathit{Obj}\ [(x, 0)] \\ \qquad\qquad [(get, x) \\ \qquad\qquad , (set, \lambda n.x := n) \\ \qquad\qquad , (eq, \lambda p.\ self \mapsto get\ ==\ p \mapsto get)] \\ \mathbf{in} \\ p \mapsto set\ 2;\ p \mapsto get \end{array}$$

An object will be modelled as a record. From a theoretical point of view, a record is a map from labels to values. We can suppose that we have an abstract datatype $\mathit{Record}\ \alpha$ with the operations

$$\begin{array}{lll} \mathit{emptyRec} & : \mathit{Record}\ v & \text{— empty record} \\ (\mapsto) & : \mathit{Record}\ v \to \mathit{Label} \to v & \text{— value selection} \\ (\uplus) & : \mathit{Record}\ v \to [(\mathit{Label}, v)] \to \mathit{Record}\ v & \text{— record extension} \end{array}$$

An object will be a record of computations

$$\mathit{Object} \triangleq \mathit{Record}\ (\mathsf{Comp}\ \mathit{Value})$$

In order to solve self-reference, the semantic specification will use the fixpoint of a generator function.

$$\mathit{Generator} \triangleq \mathit{Object} \to \mathit{Object}$$

The semantic specification will be defined from the following Obj-algebra

$$\begin{array}{lll} \varphi_{\mathsf{Obj}} & : & \mathsf{Obj}\ (\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value} \\ \varphi_{\mathsf{Obj}}\ (\mathit{Obj}\ ls\ ms) & = & \mathit{rdEnv} \gg\!= \lambda\rho. \\ & & \mathit{updLocals}\ \rho\ ls \gg\!= \lambda\rho'. \\ & & \mathit{alloc}(\mathit{return} \uparrow (\mathit{fix}(\mathit{mkGen}\ \mathit{emptyRec}\ \rho'\ ms))) \gg\!= \lambda loc. \\ & & \mathit{return}\ (\uparrow loc) \end{array}$$

$$
\begin{aligned}
\varphi_{\mathsf{Obj}}\ (c \mapsto m) \quad &= c \gg\!\!= \lambda v. \\
&\quad (\downarrow v) \mapsto\ m
\end{aligned}
$$

$upd\mathit{Locals}$ allocates space for the local variables assigning the corresponding name in the environment. It returns the resulting environment.

$$
\begin{aligned}
&\mathit{updLocals} && : \ \mathit{Env} \to [(\mathit{Name}, \mathsf{Comp}\ \mathit{Value})] \to \mathsf{Comp}\ \mathit{Env} \\
&\mathit{updLocals}\ \rho\ [] && = \mathit{return}\ \rho \\
&\mathit{updLocals}\ \rho\ ((x, m) : \mathit{xs}) && = \mathit{alloc}\ m \gg\!\!= \lambda \mathit{loc}. \\
& && \quad\ \mathit{updLocals}\ (\mathit{upd}_T\ \rho\ x\ (\mathit{return}\ (\uparrow \mathit{loc}))\ \rho)\ \mathit{xs}
\end{aligned}
$$

$\mathit{fix}\ f$ calculates the fixpoint of f.

$$
\begin{aligned}
&\mathit{fix} && : \ (\alpha \to \alpha) \to \alpha \\
&\mathit{fix}\ f && = f\ (\mathit{fix}\ f)
\end{aligned}
$$

mkGen creates a generator modifying the methods adding the "self" variable to their environment.

$$
\begin{aligned}
&\mathit{mkGen} && : \ \mathit{Object} \to\ \mathit{Env}\ \to\ [(\mathit{Label}, \mathsf{Result})] \to \mathit{Generator} \\
&\mathit{mkGen}\ \tau\ \rho\ \mathit{ms} && = \lambda \mathit{self}.\tau\ \uplus\ [(l, \mathit{modify}\ m) | (l, m) \leftarrow \mathit{ms}] \\
&\quad \mathbf{where} && \\
&\qquad \mathit{modify}\ m && = \mathit{inEnv}\ (\mathit{upd}_T\ \text{“}\mathsf{self}\text{”}\ (\mathit{return}\ (\uparrow \mathit{self}))\ \rho)\ m
\end{aligned}
$$

5.2. Classes and inheritance

From a semantic point of view, a class is just an object generator. The syntax will be captured by the Cls functor.

$$
\mathsf{Cls}\ c\ \triangleq \mathit{Class}\ [(\mathit{Name}, c)]\ [(\mathit{Label}, c)]
$$

We also need an operation to create instances from a class.

$$
\mathsf{New}\ c\ \triangleq \mathit{new}\ c
$$

Example 9 *In this example, we create a class Cell. The expression is equivalent to the expression defined in example 8.*

$$
\begin{aligned}
&\mathbf{let} \\
&\quad \mathit{Cell}\ = \mathit{Class}\ [(x, 0)] \\
&\qquad\qquad\qquad [(\mathit{get}, x) \\
&\qquad\qquad\qquad , (\mathit{set}, \lambda n.x := n) \\
&\qquad\qquad\qquad , (\mathit{eq}, \lambda p.\mathit{self} \mapsto \mathit{get}\ ==\ p \mapsto \mathit{get})] \\
&\quad p \qquad = \mathit{new}\quad \mathit{Cell} \\
&\mathbf{in} \\
&p \mapsto \mathit{set}\ 2;\ p \mapsto \mathit{get}
\end{aligned}
$$

Subclasses allow the programmer to extend a given class with new local variables and methods. The syntax is:

$$\mathsf{SubCls}\ c \triangleq \mathit{SubClass}\ c\ [(\mathit{Name}, c)]\ [(\mathit{Label}, c)]$$

Example 10 *In the following example, we create a subclass of* Cell *which adds a* $\mathit{counter}$ *variable that is incremented each time a* set *method is invoked.*

$$
\begin{array}{l}
\mathbf{let} \\
\quad \begin{array}{lll}
\mathit{Cell} & = \ldots & \text{— same definitions as in example 9} \\
\mathit{CCell} & = \mathit{SubClass} & \mathit{Cell}\ [(\mathit{counter},\ 0)] \\
 & & [(\mathit{set}, \lambda n.\mathit{super} \mapsto \mathit{set}; \mathit{counter}\ := !\mathit{counter} + 1) \\
 & & , (\mathit{getC}, \mathit{counter})] \\
q & = \mathit{new} & \mathit{CCell}
\end{array} \\
\mathbf{in} \\
q \mapsto \mathit{set}\ 2;\ p \mapsto \mathit{getC}
\end{array}
$$

A class will be represented as a computation that returns objects.

$$\mathit{Class} \triangleq \mathsf{Comp}\ \mathit{Object}$$

In the case of subclasses, the representation changes if we are modelling static or dynamic binding [Red88]. With static binding, a subclass could be represented as a class. With dynamic binding, a subclass is represented as a computation that returns a generator and an environment.

$$\mathit{SubClass} \triangleq \mathsf{Comp}\ (\mathit{Env},\ \mathit{Generator})$$

The domain value will contain objects, classes and subclasses

$$\mathit{Value} \triangleq \mathit{Int} \parallel \mathit{Bool} \parallel \mathit{Loc} \parallel \mathit{Function} \parallel \mathit{Object} \parallel \mathit{Class} \parallel \mathit{SubClass}$$

and the semantic specification is defined as

$$
\begin{array}{l}
\varphi_{\mathsf{Class}}\ (\mathbf{class}\ \mathit{ls}\ \mathit{ms})\ = \mathit{return}\ \uparrow (\mathit{mkCls}\ \mathit{ls}\ \mathit{ms}) \\
\quad \mathbf{where} \\
\qquad \mathit{mkCls}\ \mathit{ls}\ \mathit{ms}\ = \mathit{rdEnv} \gg\!= \lambda\rho. \\
\qquad\qquad\qquad\qquad \mathit{updLocals}\ \rho\ \mathit{ls} \gg\!= \lambda\rho'. \\
\qquad\qquad\qquad\qquad \mathit{return}\ (\rho',\ \mathit{mkGen}\ \mathit{emptyRec}\ \rho'\ \mathit{ms})
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{SubCls}}\ (\mathbf{subclass}\ c\ \mathit{ls}\ \mathit{ms})\ = c \gg\!= \lambda \mathit{sup}. \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\uparrow (\mathit{mkCls}\ \mathit{sup}\ \mathit{ls}\ \mathit{ms})) \\
\quad \mathbf{where} \\
\qquad \mathit{mkCls}\ \mathit{sup}\ \mathit{ls}\ \mathit{ms}\ = \downarrow \mathit{sup} \gg\!= \lambda(\rho, \mathit{gen}). \\
\qquad\qquad\qquad\qquad\quad \mathit{updLocals}\ \rho\ \mathit{ls} \gg\!= \lambda\rho'. \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\rho',\ \mathit{mkGenSub}\ \rho'\ \mathit{ms}\ \mathit{gen})
\end{array}
$$

$$\begin{aligned}\varphi_{\mathsf{New}}\ (new\ c)\ &= c \gg\!= \lambda v.\\ &\quad close(\downarrow v_{scls}) \gg\!= \lambda\tau.\\ &\quad alloc(return \uparrow \tau) \gg\!= \lambda loc.\\ &\quad return(\uparrow loc)\end{aligned}$$

$mkGenSub$ makes a generator for a subclass. It takes as arguments the environment, the list of methods and the last generator.

$$\begin{aligned}&mkGenSub &&: Env \rightarrow [(Label, \mathsf{Comp}\ Value)] \rightarrow Generator \rightarrow Generator\\ &mkGenSub\ \rho\ ms\ g &&= \lambda self. g\ self\ \uplus\ [(l, modify\ m) | (l, m) \leftarrow ms]\\ &\quad \textbf{where}\\ &\qquad modify\ m &&= rdEnv \gg\!= \lambda\rho'.\\ &&&\quad inEnv\ (mkEnv\ \rho')\ m\\ &\qquad mkEnv\ \rho' &&= (upd_T\ \text{“self”}\ (return\ \uparrow self)\ .\\ &&&\quad upd_T\ \text{“super”}\ (return\ \uparrow (g\ self)))\ (\rho +\!\!+ \rho'))\end{aligned}$$

$close$ creates a class from a subclass, evaluating the fixpoint of the generator specified by the given subclass.

$$\begin{aligned}&close &&: SubClass \rightarrow Class\\ &close\ m_{sub} &&= m_{sub} \gg\!= \lambda(\rho, g).\ return\ (fix\ g)\end{aligned}$$

The abstract syntax of the object-oriented language is defined as the fixpoint of the involved functors

$$\mathcal{L}_{OO} = \mu(\mathsf{T} \oplus \mathsf{B} \oplus \mathsf{F} \oplus \mathsf{R} \oplus \mathsf{Obj} \oplus \mathsf{Cls} \oplus \mathsf{New} \oplus \mathsf{SubCls})$$

Finally, the interpreter is automatically obtained as a catamorphism:

$$\begin{aligned}&\mathsf{Inter}_{\mathcal{L}_{OO}} : \mathcal{L}_{OO} \rightarrow \mathsf{Comp}\ Value\\ &\mathsf{Inter}_{\mathcal{L}_{OO}} = (\!|\varphi_{\mathsf{T}} \oplus \varphi_{\mathsf{B}} \oplus \varphi_{\mathsf{F}} \oplus \varphi_{\mathsf{R}} \oplus \varphi_{\mathsf{Obj}} \oplus \varphi_{\mathsf{Cls}} \oplus \varphi_{\mathsf{New}} \oplus \varphi_{\mathsf{SubCls}}|\!)\end{aligned}$$

6. Conclusions and future work

In this paper we have applied a combination of modular monadic semantics and generic programming concepts to specify an object-oriented programming language with dynamic binding. This approach provides semantic modularity by means of monad transformers and facilitates reusability. As an example, the specification of arithmetic expressions has been taken from a library of semantic blocks and has been reused in the specification of other kinds of programming languages.

We have implemented a Language Prototyping System [LPS01, Lab01] that facilitates the reuse of semantic blocks and provides an interactive framework for language testing. The system has been implemented as a domain-specific language embedded in Haskell [Hud98, Kam00]. Although this approach has some advantages, we are currently assessing the development of an independent meta-language following [Mog97, BHM00].

Future research lines are the axiomatization of different kinds of monads and the derivation of monad transformers and their combination in a more systematic way following [Hin00]. In that line, there has been some definitions of monads that support non-determinism, parallelism and concurrency [Cla99, Wan97]. It will be interesting to check if the introduction of those features can be done in an orthogonal way without modifying previously specified components.

It would also be fruitful to study the combination of algebras, coalgebras, monads and comonads to provide the semantics of interactive and object-oriented features in this framework [Bar00, JP00]. At the same time, the automatic derivation of compilers from the monadic interpreters has already been started in [HK00].

Our approach tackles the problem of specifying a programming language from semantic building blocks. This problem has also been tackled in the Action Semantics framework [DM01, Mos92], where there are also some specifications of object oriented languages [Wat97]. It would be very interesting to make deeper comparisons of these approaches as has already been started in [WB00, Mos98, Lab01, Wan97].

Apart of object oriented languages, we have also specified imperative [LCLG01], functional [LLCC01] and logic programming languages [LCLC01c, LCLC01b]. All the specifications have been made in a modular way reusing the common components of the different languages.

References

[Bar00] 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.

[BdM97] R. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[BHM00] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *International Summer School On Applied Semantics APPSEM'2000*, Caminha, Portugal, 2000.

[BJJM99] 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.

[Bou01] G. Boudol. The recursive record semantics of objects revisited. In *European Symposium on Programming (ESOP)*, number 2028 in Lecture Notes in Computer Science, pages 269–283. Springer-Verlag, 2001.

[Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 51–67. Springer-Verlag, 1984.

[Cla99] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, May 1999.

[CP94] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994.

[DM01] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[Dup95] Luc Duponcheel. Writing modular interpreters using catamorphisms, subtypes and monad transformers. Technical Report (Draft), Utrecht University, 1995.

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Foundations of Computing. The MIT Press, 1994.

[Hin00] 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.

[HK00] 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.

[Hud98] P. Hudak. Domain-specific languages. In Peter H. Salus, editor, *Handbook of Programming Languages*, volume III, Little Languages and Tools. Macmillan Technical Publishing, 1998.

[JD93] Mark P. Jones and L. Duponcheel. Composing monads. YALEU/DCS/RR 1004, Yale University, New Haven, CT, USA, 1993.

[JHA+99] 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.

[Jon95] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[Jon97] 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.

[JP00] 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.

[Kam00] Samuel Kamin. Research on domain specific embedded languages and program generators. In Rance Cleaveland, Michael Mislove, and Philip Mulry, editors, *Electronic Notes in Theoretical Computer Science*, volume 14. Elsevier Science Publishers, 2000.

[KR94] Samuel Kamin and Uday Reddy. Two Semantic Models of Object-Oriented Languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, chapter 13, pages 463–495. MIT Press, 1994.

[Lab98] 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.

[Lab01] Jose E. Labra. *Modular Development of Language Processors from Reusable Semantic Specifications*. PhD thesis, Dept. of Computer Science, University of Oviedo, 2001. In spanish.

[LCL99] J. E. Labra, J. M. Cueva, and C. Luengo. Language prototyping using modular monadic semantics. In *3rd Latin-American Conference on Functional Programming*, Recife - Brazil, March 1999. Available at `http://lsi.uniovi.es/~labra/LPS/Clapf99.ps`.

[LCLC01a] 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.

[LCLC01b] 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.

[LCLC01c] J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. Specification of logic programming languages from reusable semantic building blocks. In *International Workshop on Functional and (Constraint) Logic Programming*, number 2017, Kiel, Germany, September 2001. Christian-Albrechts-Universität Kiel.

[LCLG01] J. E. Labra, J. M. Cueva, M. C. Luengo, 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.

[LH96] 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.

[LHJ95] 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.

[LLCC01] 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.

[LPS01] Language Prototyping System. `http://lsi.uniovi.es/~labra/LPS/LPS.html`, 2001.

[MFP91] 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.

[Mog89] 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.

[Mog91] Eugenio Moggi. Notions of computacion and monads. *Information and Computation*, (93):55–92, 1991.

[Mog97] 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.

[Mos92] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[Mos98] 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.

[Red88] U. S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symposium on LISP and Functional Programming*, pages 289–297, Snowbird, Utah, July 1988.

[SF93] 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.

[Wan97] Keith Wansbrough. A modular monadic action semantics. Master's thesis, Department of Computer Science, University of Auckland, February 1997.

[Wat97] David A. Watt. JOOS action semantics. Technical report, Department of Computing Science, University of Glasgow, 1997.

[WB00] D. A. Watt and D. F. Brown. Formalising the dynamic semantics of java. In *3rd International Workshop on Action Semantics*, number NS-00-6, Recife, Brazil, 2000. BRICS, Dept. of Computer Science, Univ. of Aarhus.