# FACILITATING THE DEFINITION OF PROGRAMMING LANGUAGES BY USING PARAMETRIC CONTEXT-FREE GRAMMARS

NIKOLAOS S. PAPASPYROU AND VASSILIOS C. VESCOUKIS
*National Technical University of Athens*
*Department of Electrical and Computer Engineering*
*Software Engineering Laboratory*
*Polytechnioupoli, 15780 Zografou, Athens, Greece*
*Tel:+30-1-7722486, Fax: +30-1-7722519*
*E-mail: nickie@softlab.ntua.gr, bxb@cs.ntua.gr*

In this paper we propose the use of a new kind of grammars, which we call parametric context-free grammars (PCFG) for the formal definition of programming languages. It combines the simplicity of ordinary context-free grammars with expressive power and context sensitivity that is encountered in significantly more complex formal models. As an example, we consider a small programming language and present a simple and elegant formal definition of its syntax and semantics. The language is based on the principles of Reynolds' *Idealized Algol* and combines a number of orthogonal features, introduced one at a time. Compared to related literature, our approach differs in providing: (i) a rigid mathematical model for the definition of syntax, emphasizing orthogonality; and (ii) a methodology for systematically augmenting the semantic definition of programming languages, by adding orthogonal features in a modular and elegant way.

## 1 Parametric context-free grammars and PBNF

A natural extension of context-free grammars is obtained by introducing parameters denoting arbitrary strings in nonterminal symbols. A *parametric context-free grammar* (PCFG) is defined as a tuple:

$$G = \langle T, NT, a, I, R, s \rangle$$

where $T$ is a set of terminal symbols, $NT$ is a set of nonterminal symbols, $a : NT \to \boldsymbol{N}$ is a function returning the number of parameters that each nonterminal symbol expects, $I$ is a set of identifiers used as names for formal parameters, $R$ is a set of production rules and $s \in NT$ is the initial nonterminal symbol, which must satisfy $a(s) = 0$. The sets $T$, $NT$ and $I$ must be distinct and all sets must be finite. Each rule $r \in R$ has the form:

$$nt(i_1, \ldots, i_p) ::= \alpha$$

where $nt \in NT$ is a nonterminal symbol, $p = a(nt)$ is the number of parameters that $nt$ expects, identifiers $i_k \in I$ are distinct and $\alpha \in S(\{i_1, \ldots, i_p\})$ where:[a]

$$S(P) = \bigcup_{n=0}^{\infty} S^n(P), \quad \text{with} \quad S^0(P) = \emptyset \quad \text{and}$$

$$S^{n+1}(P) = (T \cup P \cup \{nt(\alpha_1, \ldots, \alpha_p) \mid nt \in NT \wedge p = a(nt) \wedge \\ \wedge \forall i : 1 \leq i \leq p.\ \alpha_i \in S^n(P)\})^*$$

We should note here that $S(P)$ is the set of strings consisting of: (i) terminal symbols; (ii) formal parameters, i.e. identifiers drawn from the finite set $P \subseteq I$; and (iii) nonterminal symbols followed by the correct number of actual parameters, which are also elements of $S(P)$.

The one-step production relation $\Rightarrow$ for the parametric context-free grammar $G$ is defined on elements of $S(\emptyset)$, that is, strings containing no free formal parameters. This restriction resolves a number of ambiguities that would result from the possible use of the same identifier as a formal parameter in more than one production rules. The production relation is defined as:

$$\forall p \in \boldsymbol{N}.\ \forall nt \in NT : a(nt) = p.\ \forall \chi, \psi, \beta_1, \ldots, \beta_p, \gamma \in S(\emptyset). \\ \chi\, nt(\beta_1, \ldots, \beta_p)\, \psi \Rightarrow \chi\, \gamma\, \psi \quad \text{iff} \quad \exists\, i_1, \ldots, i_p \in I.\ \exists\, \alpha \in S(\{i_1, \ldots, i_p\}). \\ \alpha[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p] = \gamma \ \wedge \ (nt(i_1, \ldots, i_p) ::= \alpha) \in R$$

where $\alpha[i_1 \mapsto \beta_1, \ldots, i_p \mapsto \beta_p]$ is the result of the textual substitution of all formal parameters $i_1, \ldots, i_p \in I$ by the actual values $\beta_1, \ldots, \beta_p \in S(\emptyset)$ in string $\alpha$. Textual substitution is formally defined in Fig. 1.

The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$ and the language $L(G)$ generated by $G$ is defined by:

$$L(G) = \{\alpha \in T^* \mid s() \Rightarrow^* \alpha\} \subseteq T^*$$

It is easy to show that PCFGs are more expressible than CFGs. First, notice that every CFG can be trivially transformed to an equivalent PCFG, in which all nonterminal symbols expect no parameters. In addition, consider the simple PCFG with $T = \{x\}$, $NT = \{s, p\}$, $I = \{i\}$, $a(s) = 0$, $a(p) = 1$, and three production rules: $s() ::= p(x)$, $p(i) ::= i$ and $p(i) ::= p(i\,i)$. It is not hard

[a]Throughout this paper, we use the notation $\alpha\,\beta$ for the concatenation of strings $\alpha$ and $\beta$, $\epsilon$ for the empty string, $L_1\,L_2$ for the concatenation of languages $L_1$ and $L_2$ defined as $L_1\,L_2 = \{\alpha\,\beta \mid \alpha \in L_1 \wedge \beta \in L_2\}$, $L^n$ for the concatenation of $L$ with itself $n$ times, defined as $L^{n+1} = L\,L_n$ with $L^0 = \{\epsilon\}$, and $L^*$ for the Kleene star of $L$ defined as $L^* = \bigcup_{n=0}^{\infty} L^n$. Moreover, we use small Greek letters $(\alpha, \beta, \chi, \psi)$ to denote strings.

$$
\begin{aligned}
&\forall p \in \boldsymbol{N}.\ \forall i_1,\ldots,i_p \in I.\ \forall \beta_1,\ldots,\beta_p \in S(\emptyset).\ \forall t \in T.\\
&\quad t[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p] = t\\
&\forall p,k \in \boldsymbol{N} : 1 \leq k \leq p.\ \forall i_1,\ldots,i_p \in I.\ \forall \beta_1,\ldots,\beta_p \in S(\emptyset).\\
&\quad i_k[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p] = \beta_k\\
&\forall p,q \in \boldsymbol{N}.\ \forall i_1,\ldots,i_p \in I.\ \forall \beta_1,\ldots,\beta_p \in S(\emptyset).\ \forall nt \in NT : a(nt) = q.\\
&\quad \forall \alpha_1,\ldots,\alpha_q \in S(\{i_1,\ldots,i_p\}).\\
&\qquad nt(\alpha_1,\ldots,\alpha_q)[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p] =\\
&\qquad\quad nt(\alpha_1[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p],\ldots,\alpha_q[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p])\\
&\forall p \in \boldsymbol{N}.\ \forall i_1,\ldots,i_p \in I.\ \forall \beta_1,\ldots,\beta_p \in S(\emptyset).\ \forall \alpha_1,\alpha_2 \in S(\{i_1,\ldots,i_p\}).\\
&\quad (\alpha_1\,\alpha_2)[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p] =\\
&\qquad \alpha_1[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p]\,\alpha_2[i_1 \mapsto \beta_1,\ldots,i_p \mapsto \beta_p]
\end{aligned}
$$

Figure 1. Definition of textual substitution.

to prove that the language generated by this PCFG is $L = \{x^{2^n} \mid n \in \boldsymbol{N}\}$, i.e. the language of strings over $T$ with length equal to a power of 2. However, it is known that this language is not context-free and therefore CFG $\subset$ PCFG.

Following the line of BNF and its variations for CFGs, it is convenient to define a formalism for the representation of PCFGs, which we call Parametric Backus-Naur Form (PBNF). This formalism is based on BNF, with the additional feature that parameters of nonterminal symbols are written inside curly braces ("{" and "}"), in order to distinguish them from grouping parentheses. The braces are omitted altogether if a nonterminal symbol expects no parameters. All other symbols and notational conventions from BNF retain their ordinary meaning in PBNF, i.e. nonterminal symbols are enclosed in angular brackets ("<" and ">") in order to be distinguished from terminal symbols, and a number of rules $nt ::= \alpha_1, \ldots, nt ::= \alpha_m$, with the same nonterminal symbol on the left-hand side can be combined in a rule of the form $nt ::= \alpha_1 \mid \ldots \mid \alpha_m$, with the symbol "|" separating the alternative productions.

## 2 Applications in the definition of programming languages

Parametric context-free grammars are useful in the formal definition of languages with families of similar features. In fact, a production rule for a parametric nonterminal symbol can often replace a family of similar production rules in an ordinary context-free grammar. The unification of similar features facilitates the definition of the language's type system and dynamic semantics. In the following sections, we build an experimental language by starting from a simple core and gradually introducing new features that are orthogonal to each other. The resulting language follows the principles of *Idealized Algol*. [1]

The methodology that is used for the definition of the language divides the definition of semantics in three distinct consecutive phases: static, typing and dynamic semantics. [2,3] Denotational semantics is used for the dynamic part. For each new feature introduced in the language, the following steps are required:

- The abstract syntax of the new orthogonal feature is introduced as a parametric production rule.
- The production rules for the basic syntax domains are extended by adding instances of the new feature, whenever appropriate.
- The static semantics of the language is appropriately revised, if necessary, by extending existing domains and introducing new ones.
- One or more general typing rules for the new feature are introduced.
- The dynamic semantic domains are revised, following the changes in the static semantics.
- For each new typing rule, one general semantic equation is specified.

It should be noted that the revisions in the language's semantics (static or dynamic), that are required for the introduction of a new feature, do not affect the semantics of the language so far. This is implied by the orthogonality of the features and is realized by the use of appropriate *monads* in the definition of the semantics. [4,5] In a naïve definition, a monad is a triple $\langle \mathsf{M}, \mathit{unit}_{\mathsf{M}}, *_{\mathsf{M}} \rangle$, where $\mathsf{M}$ is a domain constructor, $\mathit{unit}_{\mathsf{M}} : A \to \mathsf{M}(A)$ and $\cdot *_{\mathsf{M}} \cdot : \mathsf{M}(A) \times (A \to \mathsf{M}(B)) \to \mathsf{M}(B)$ are polymorphic functions for arbitrary domains $A$ and $B$, satisfying three monad laws. Monads are used in the semantics of programming languages to represent computations. If $D$ is a domain of values, then $\mathsf{M}(D)$ is the domain of computations returning results in $D$ and $\mathit{unit}_{\mathsf{M}}\ v$ is usually the simplest computation returning the value $v$ in $D$. If $m$ is a computation returning a value $v$ in $D$, then the computation denoted by $m *_{\mathsf{M}} f$ is the combined computation of $m$ followed by computation $f\ v$.

## 3 Core language

We begin by defining a simple core language with two basic types: integer and boolean. Expressions in this language are either constants of the two types or applications of unary and binary operators on other expressions. Statements in the language are trivial. However, since the notion of *environment* is crucial in our perception of any programming language, the core language supports values denoted by identifiers, without however supplying any mechanism for binding identifiers to values. The abstract syntax is given by the following two production rules, where <id> and <num> represent the syntactic classes of identifiers and integer constants respectively.

$$\begin{aligned}\langle\text{expr}\rangle &::= \langle\text{id}\rangle \mid \langle\text{num}\rangle \mid \textbf{true} \mid \textbf{false} \mid \textbf{!}\,\langle\text{expr}\rangle \\ &\quad\mid \langle\text{expr}\rangle + \langle\text{expr}\rangle \mid \ldots \\ \langle\text{stmt}\rangle &::= \textbf{skip}\end{aligned}$$

The static semantics of the core language requires the definition of the following static domains, representing *data* and *phrase* types as well as static *environments*, which map identifiers to phrase types.[b]

$$\begin{array}{rcl} \tau : \boldsymbol{TypeDat} &=& \mathit{int} \mid \mathit{bool} \\ \theta : \boldsymbol{TypePhr} &=& \mathit{exp}(\tau) \mid \mathit{stmt} \\ e : \boldsymbol{TypeEnv} &=& \boldsymbol{Ide} \to \boldsymbol{TypePhr} \oplus \boldsymbol{U} \end{array}$$

A number of static semantic functions are also required for extracting static type information from syntactic elements. Function $\{\!|\langle\text{num}\rangle|\!\} : \boldsymbol{N}$, for example, returns the value of an integer constant and function $\{\!|\langle\text{id}\rangle|\!\} : \boldsymbol{Ide}$ maps the syntactic class of identifiers to the corresponding semantic domain.

The following typing rules define the typing semantics of the core language. The static environment is only used to obtain the denoted values of identifiers and is not updated by any typing rule.

$$\frac{I = \{\!|\langle\text{id}\rangle|\!\} \quad e\,I = \mathit{inl}\ \theta}{e \vdash \langle\text{id}\rangle : \theta} \qquad \frac{}{e \vdash \langle\text{num}\rangle : \mathit{exp}(\mathit{int})}$$

$$\frac{}{e \vdash \textbf{true} : \mathit{exp}(\mathit{bool})} \qquad \frac{e \vdash \langle\text{expr}\rangle : \mathit{exp}(\mathit{bool})}{e \vdash \textbf{!}\,\langle\text{expr}\rangle : \mathit{exp}(\mathit{bool})}$$

$$\frac{\begin{array}{c} e \vdash \langle\text{expr}\rangle_1 : \mathit{exp}(\mathit{int}) \\ e \vdash \langle\text{expr}\rangle_2 : \mathit{exp}(\mathit{int}) \end{array}}{e \vdash \langle\text{expr}\rangle_1 + \langle\text{expr}\rangle_2 : \mathit{exp}(\mathit{int})} \qquad \frac{}{e \vdash \textbf{skip} : \mathit{stmt}}$$

[b]In the rest of this paper, ***Ide*** is the domain of identifiers, ***U*** is a domain with a single ordinary element $u$, ***N*** is the flat domain of integer numbers and ***T*** is the flat domain of truth values.

Following the denotational approach, [6] the semantics of a well-formed phrase $P$ of type $\theta$ in environment $e$ can be represented by a semantic function of the form:

$$[\![ e \vdash P : \theta ]\!] \quad : \quad [\![ e ]\!] \to \mathsf{G}([\![ \theta ]\!])$$

where $[\![ e ]\!]$ is the domain of dynamic environments corresponding to the static environment $e$, $[\![ \theta ]\!]$ is the domain of dynamic semantic values corresponding to phrase type $\theta$ and $\mathsf{G}$ is an appropriate monad, representing computations in the core language. The dynamic semantic domains for data and phrase types are defined next, together with the domain of dynamic environments, which is a dependent function domain.

$$\begin{array}{lcl} [\![ \mathit{exp}(\tau) ]\!] & = & [\![ \tau ]\!] \\ [\![ \mathit{stmt} ]\!] & = & \boldsymbol{U} \\ [\![ \mathit{int} ]\!] & = & \boldsymbol{N} \\ [\![ \mathit{bool} ]\!] & = & \boldsymbol{T} \end{array} \qquad \begin{array}{lcl} \rho \ : \ [\![ e ]\!] & = & I : \boldsymbol{Ide} \twoheadrightarrow [\![ e\{I\} ]\!] \\ [\![ e\{I\} ]\!] & = & \begin{cases} \mathsf{G}([\![ \theta ]\!]) & , \text{ if } e\ I = \mathit{inl}\ \theta \\ \boldsymbol{U} & , \text{ otherwise} \end{cases} \end{array}$$

Four representative dynamic semantic equations, corresponding to some of the language's typing rules, are given below.

$$\begin{array}{l} [\![ e \vdash I : \theta ]\!] \ = \ \lambda \rho : [\![ e ]\!].\ \rho\ I \\ [\![ e \vdash \text{<num>} : \mathit{exp}(\mathit{int}) ]\!] \ = \ \lambda \rho : [\![ e ]\!].\ \mathit{unit}\ \{\!|\text{<num>}|\!\} \\ [\![ e \vdash \text{<expr>}_1 + \text{<expr>}_2 : \mathit{exp}(\mathit{int}) ]\!] \ = \ \lambda \rho : [\![ e ]\!]. \\ \quad [\![ e \vdash \text{<expr>}_1 : \mathit{exp}(\mathit{int}) ]\!]\ \rho \ * \ (\lambda n_1 : \boldsymbol{N}. \\ \quad [\![ e \vdash \text{<expr>}_2 : \mathit{exp}(\mathit{int}) ]\!]\ \rho \ * \ (\lambda n_2 : \boldsymbol{N}.\ \mathit{unit}\ (n_1 + n_2))) \\ [\![ e \vdash \textbf{skip} : \mathit{stmt} ]\!] \ = \ \lambda \rho : [\![ e ]\!].\ \mathit{unit}\ u \end{array}$$

Furthermore, the auxiliary function *bind*, defined below, is used in the following sections in order to bind identifiers to denoted values.

$$\begin{array}{lcl} \mathit{bind} & : & I : \boldsymbol{Ide} \twoheadrightarrow \theta : \boldsymbol{TypePhr} \twoheadrightarrow \mathsf{G}([\![ \theta ]\!]) \to [\![ e ]\!] \to \mathsf{G}([\![ e\{I \mapsto \theta\} ]\!]) \\ \mathit{bind} & = & \lambda I : \boldsymbol{Ide}.\ \lambda \theta : \boldsymbol{TypePhr}.\ \lambda x : \mathsf{G}([\![ \theta ]\!]).\ \lambda \rho : [\![ e ]\!]. \\ & & \quad \mathit{unit}\ (\lambda I' : \boldsymbol{Ide}.\ (I = I') \to x\ ,\ \rho\ I') \end{array}$$

## 4 Conditional

The use of the conditional construct applies very similarly to the basic syntactic domains, statements and expressions. In other words, the conditional is a feature orthogonal to the language's core and should be treated uniformly if an elegant language definition is sought. In order to add the conditional construct, the abstract syntax of the language is extended as follows. The last parametric production rule defines the syntax of the new construct and exhibits context-sensitive characteristics.

$$\begin{array}{lll} \langle\text{expr}\rangle & ::= \ldots \mid \langle\text{cond}\rangle\{\langle\text{expr}\rangle\} \\ \langle\text{stmt}\rangle & ::= \ldots \mid \langle\text{cond}\rangle\{\langle\text{stmt}\rangle\} \end{array}$$

$$\langle\text{cond}\rangle\{\alpha\} ::= \mathbf{if}\,\langle\text{expr}\rangle\,\mathbf{then}\,\alpha\,\mathbf{else}\,\alpha$$

One more general typing rule is required for the conditional construct. The rule corresponds to the last production rule and is given below.

$$\frac{e \vdash \langle\text{expr}\rangle : \mathit{exp}(\mathit{bool}) \quad e \vdash \alpha_1 : \theta \quad e \vdash \alpha_2 : \theta}{e \vdash \mathbf{if}\,\langle\text{expr}\rangle\,\mathbf{then}\,\alpha_1\,\mathbf{else}\,\alpha_2 : \theta}$$

The dynamic semantic equation that corresponds to the new typing rule is given below.

$$\begin{array}{l} [\![ e \vdash \mathbf{if}\,\langle\text{expr}\rangle\,\mathbf{then}\,\alpha_1\,\mathbf{else}\,\alpha_2 : \theta ]\!] \;=\; \lambda\rho : [\![ e ]\!] . \\ \quad [\![ e \vdash \langle\text{expr}\rangle : \mathit{exp}(\mathit{bool}) ]\!]\, \rho \,*\, (\lambda b : \boldsymbol{T}. \\ \quad b \rightarrow [\![ e \vdash \alpha_1 : \theta ]\!]\, \rho ,\; [\![ e \vdash \alpha_2 : \theta ]\!]\, \rho) \end{array}$$

## 5 Sequential execution

The next thing to introduce is sequential execution, which can also be treated as an orthogonal feature. The new version of the language allows a statement to be executed before another statement or an expression, and the result is a statement or an expression respectively. The following modifications must be made in the definition of the language.

Abstract syntax:

$$\begin{array}{ll} \langle\text{expr}\rangle & ::= \ldots \mid \langle\text{seq}\rangle\{\langle\text{expr}\rangle\} \\ \langle\text{stmt}\rangle & ::= \ldots \mid \langle\text{seq}\rangle\{\langle\text{stmt}\rangle\} \end{array}$$

$$\langle\text{seq}\rangle\{\alpha\} ::= \langle\text{stmt}\rangle\,\mathbf{;}\,\alpha$$

Typing rule:

$$\frac{e \vdash \langle\text{stmt}\rangle : \mathit{stmt} \quad e \vdash \alpha : \theta}{e \vdash \langle\text{stmt}\rangle\,\mathbf{;}\,\alpha : \theta}$$

Dynamic semantic equation:

$$\begin{array}{l} [\![ e \vdash \langle\text{stmt}\rangle\,\mathbf{;}\,\alpha : \theta ]\!] \;=\; \lambda\rho : [\![ e ]\!] . \\ \quad [\![ e \vdash \langle\text{stmt}\rangle : \mathit{stmt} ]\!]\, \rho \,*\, (\lambda u : \boldsymbol{U}.\; [\![ e \vdash \alpha : \theta ]\!]\, \rho) \end{array}$$

## 6 Variables

The introduction of local variables is the next revision of the language that we attempt. We allow variables to be local to both expressions and statements, and thus they can also be considered as orthogonal to the rest of the language, so far. A parametric production rule is again used to emphasize orthogonality. In order to use variables, two additional language constructs must be introduced: dereference and assignment. Also, a new syntactic domain must be defined, to represent the names of data types.

$$\begin{array}{lll} \text{<expr>} & ::= & \ldots \mid \text{<var>}\{\text{<expr>}\} \mid \textbf{deref}\,\text{<expr>} \\ \text{<stmt>} & ::= & \ldots \mid \text{<var>}\{\text{<stmt>}\} \mid \text{<expr>} \mathrel{\textbf{:=}} \text{<expr>} \\ \text{<var>}\{\alpha\} & ::= & \textbf{new}\,\text{<id>}\,\textbf{:}\,\text{<data>}\,\textbf{in}\,\alpha \\ \text{<data>} & ::= & \textbf{int} \mid \textbf{bool} \end{array}$$

With the introduction of variables, a new phrase type $\mathit{var}(\tau)$ is required for variables of data type $\tau$. The static semantic domain ***TypePhr*** is extended appropriately.

$$\textbf{\textit{TypePhr}} = \ldots \mid \mathit{var}(\tau)$$

In the typing rules that define the typing of the newly introduced feature, the static environment is used in variable definitions, mapping a new variable identifier of data type $\tau$ to the phrase type $\mathit{var}(\tau)$. The static semantic function $\{\!|\text{<data>}|\!\} : \textbf{\textit{TypeDat}}$ must be defined, to map the new syntactic domain to the static semantic domain of data types. The other typing rules are straightfoward.

$$\frac{I = \{\!|\text{<id>}|\!\} \quad \tau = \{\!|\text{<data>}|\!\} \quad e\{I \mapsto \mathit{var}(\tau)\} \vdash \alpha : \theta}{e \vdash \textbf{new}\,\text{<id>}\,\textbf{:}\,\text{<data>}\,\textbf{in}\,\alpha : \theta}$$

$$\frac{e \vdash \text{<expr>} : \mathit{var}(\tau)}{e \vdash \textbf{deref}\,\text{<expr>} : \mathit{exp}(\tau)}$$

$$\frac{e \vdash \text{<expr>}_1 : \mathit{var}(\tau) \quad e \vdash \text{<expr>}_2 : \mathit{exp}(\tau)}{e \vdash \text{<expr>}_1 \mathrel{\textbf{:=}} \text{<expr>}_2 : \mathit{stmt}}$$

A significant revision must be made in the dynamic semantics of the language, at this point. In particular, the computation monad G must support a store, typically a map from objects in the store to dynamic data values of the appropriate type. The dynamic semantic domain ***Obj*** is introduced for representing objects in the store, and the following auxiliary functions are required for allocating, storing and retrieving objects.

$$\begin{array}{lcl} \mathit{newObj} & : & \textbf{\textit{TypeDat}} \to \mathsf{G}(\textbf{\textit{Obj}}) \\ \mathit{getValue} & : & \tau : \textbf{\textit{TypeDat}} \twoheadrightarrow \textbf{\textit{Obj}} \to \mathsf{G}([\![\tau]\!]) \\ \mathit{putValue} & : & \tau : \textbf{\textit{TypeDat}} \twoheadrightarrow \textbf{\textit{Obj}} \to [\![\tau]\!] \to \mathsf{G}(\textbf{\textit{U}}) \end{array}$$

The implementations of an appropriate monad G, domain ***Obj*** and auxiliary functions are not included in this paper, since they are not of primary importance. The dynamic semantic domain for the new phrase type is given below: in the dynamic semantics, variables are denoted by objects in the store.

$$[\![\mathit{var}(\tau)]\!] \;=\; \textbf{\textit{Obj}}$$

Finally, the additional dynamic semantic equations are given below. The three auxiliary functions, as well as *bind*, are used in these equations.

$$\begin{array}{l} [\![e \vdash \textbf{new}\,\text{<id>}\,\textbf{:}\,\text{<data>}\,\textbf{in}\,\alpha : \theta]\!] \;=\; \lambda\,\rho : [\![e]\!]. \\ \quad \mathit{newObj}\ \tau \ *\ (\lambda\,a : \textbf{\textit{Obj}}. \\ \quad \mathit{bind}\ I\ \mathit{var}(\tau)\ (\mathit{unit}\ a)\ \rho\ *\ (\lambda\,\rho' : [\![e\{I \mapsto \mathit{var}(\tau)\}]\!]. \\ \quad [\![e\{I \mapsto \mathit{var}(\tau)\} \vdash \alpha : \theta]\!]\ \rho')) \\ [\![e \vdash \textbf{deref}\,\text{<expr>} : \mathit{exp}(\tau)]\!] \;=\; \lambda\,\rho : [\![e]\!]. \\ \quad [\![e \vdash \text{<expr>} : \mathit{var}(\tau)]\!]\ \rho\ *\ (\lambda\,a : \textbf{\textit{Obj}}.\ \mathit{getValue}\ \tau\ a) \\ [\![e \vdash \text{<expr>}_1 \,\textbf{:=}\, \text{<expr>}_2 : \mathit{stmt}]\!] \;=\; \lambda\,\rho : [\![e]\!]. \\ \quad [\![e \vdash \text{<expr>}_1 : \mathit{var}(\tau)]\!]\ \rho\ *\ (\lambda\,a : \textbf{\textit{Obj}}. \\ \quad [\![e \vdash \text{<expr>}_2 : \mathit{exp}(\tau)]\!]\ \rho\ *\ (\lambda\,v : [\![\tau]\!].\ \mathit{putValue}\ \tau\ a\ v)) \end{array}$$

## 7 Lambda abstraction

In principal, lambda abstraction allows the definition of functions and procedures taking a single parameter of a given type. In our approach, evaluation of parameters will be *lazy*, that is, parameters will not be evaluated when a function or procedure is called but whenever their value is needed. Eager evaluation is more restrictive in what can be passed as a parameter. The intended orthogonality implies that all well-formed phrases of the language may participate in lambda abstractions.[c]

The abstract syntax of the language is again extended to incorporate lambda abstractions and applications. The syntactic class of phrases that can

[c]In our language, valid phrase types correspond grossly to expressions (including variable references) and statements, so lambda abstractions can be regarded as unnamed functions or procedures. However, if the types of the language themselves were considered a phrase type, a third kind of lambda abstraction would be natural. In this way, it would be possible to introduce parametric types and parametric polymorphism in the language.

be passed as arguments is represented by the nonterminal symbol <arg>, that is, parameters may be expressions or statements. In addition, the nonterminal symbol <type> must be introduced for representing the syntactic class of parameter types which, in our approach, covers the whole category of phrase types.

```
<expr>       ::= ... | <abs>{<expr>}
<stmt>       ::= ... | <abs>{<stmt>}
<abs>{α}     ::= lambda <id> : <type> . α | α <arg>
<arg>        ::= <expr> | <stmt>
<type>       ::= <data> exp | stmt | <data> var | <type> -> <type>
```

In the static semantics of the language, a new phrase type must be added, namely the type $\mathit{func}(\theta_1, \theta_2)$ of higher-order functions, taking a parameter of type $\theta_1$ and returning a result of type $\theta_2$.

$$\boldsymbol{TypePhr} \;=\; \ldots \mid \mathit{func}(\theta_1, \theta_2)$$

Also, a new static semantic function $\{\!|\text{<type>}|\!\} : \boldsymbol{TypePhr}$ must be defined, for mapping the syntactic class of parameter types to the static domain of phrase types.

The new typing rules are straightforward. The static environment is again used for storing the type of the parameter.

$$\frac{I = \{\!|\text{<id>}|\!\} \quad \theta_1 = \{\!|\text{<type>}|\!\} \quad e\{I \mapsto \theta_1\} \vdash \alpha : \theta_2}{e \vdash \textbf{lambda}\,\text{<id>}\,\textbf{:}\,\text{<type>}\,\textbf{.}\,\alpha : \mathit{func}(\theta_1, \theta_2)}$$

$$\frac{e \vdash \alpha : \mathit{func}(\theta_1, \theta_2) \quad e \vdash \text{<arg>} : \theta_1}{e \vdash \alpha\,\text{<arg>} : \theta_2}$$

In the revision of the dynamic semantic domains, we are faced with the problem of choosing a dynamic domain for the phrase type of functions. Since we have adopted lazy evaluation, a reasonable option is the domain of functions from computations of the parameter to computations of the result. In this way, the parameters are computed every time they are needed (call by name).

$$[\![\mathit{func}(\theta_1, \theta_2)]\!] \;=\; \mathsf{G}([\![\theta_1]\!]) \to \mathsf{G}([\![\theta_2]\!])$$

Having this settled, it is relatively easy to write the dynamic semantic equations for the new constructs.

$$\begin{array}{l}
[\![ e \vdash \textbf{lambda}\, \text{<id>} \,\texttt{:}\, \text{<type>} \,\texttt{.}\, \alpha : \mathit{func}(\theta_1, \theta_2) ]\!] \;=\; \lambda\, \rho : [\![ e ]\!]. \\
\quad \mathit{unit}\ (\lambda\, x : \mathsf{G}([\![ \theta_1 ]\!]). \\
\qquad \mathit{bind}\ I\ \theta_1\ x\ \rho\ *\ (\lambda\, \rho' : [\![ e\{I \mapsto \theta_1\} ]\!]. \\
\qquad [\![ e\{I \mapsto \theta_1\} \vdash \alpha : \theta_2 ]\!]\ \rho')) \\
[\![ e \vdash \alpha\, \text{<arg>} : \theta_2 ]\!] \;=\; \lambda\, \rho : [\![ e ]\!]. \\
\quad [\![ e \vdash \alpha : \mathit{func}(\theta_1, \theta_2) ]\!]\ \rho\ *\ (\lambda\, f : \mathsf{G}([\![ \theta_1 ]\!]) \to \mathsf{G}([\![ \theta_2 ]\!]). \\
\quad f([\![ e \vdash \text{<arg>} : \theta_1 ]\!]\ \rho))
\end{array}$$

## 8 Recursion

Recursion is the last feature that we will consider in this paper. In the approach that we use, the new feature is not orthogonal to lambda abstraction, since it depends on it. However, it is orthogonal to all other features of the language that have been considered so far. Two illustrative examples will be very helpful in understanding the way in which we introduce recursion in the language. First, consider the factorial function (assuming a range of integer and boolean operators).

```
rec (lambda f : int exp -> int exp. lambda n : int exp.
        if n=0 then 1 else n * f (n-1))
```

The parameter of the recursion's body provides a way to refer to the result of the recursion. Consider also the following encoding of a simple *while* statement, calculating the sum of numbers between 1 and 100. The example is less intuitive and, this time, the parameter and the result of the recursion are statements, instead of functions.

```
new s : int in
new i : int in
s := 0; i := 1;
rec (lambda c : stmt.
   if deref i <= 100 then
      s := deref s + deref i; i := deref i + 1; c
   else skip)
```

In order to introduce recursion, we extend the abstract syntax of the language as follows.[d]

[d]Returning to the footnote of the previous section, the combination of recursion with the parametric types would produce recursively defined types, which are present in most popular programming languages.

$$\begin{array}{lll} \text{<expr>} & ::= \ldots \mid \text{<req>}\{\text{<expr>}\} \\ \text{<stmt>} & ::= \ldots \mid \text{<req>}\{\text{<stmt>}\} \end{array}$$

$$\text{<req>}\{\alpha\} ::= \mathbf{rec}\,\alpha$$

The typing rule for the new construct is given below:

$$\frac{e \vdash \alpha : \mathit{func}(\theta, \theta)}{e \vdash \mathbf{rec}\,\alpha : \theta}$$

Finally, the dynamic semantic equation uses the least fixed point operator on domain $\mathsf{G}([\![\theta]\!])$.[e]

$$[\![e \vdash \mathbf{rec}\,\alpha : \theta]\!] \quad = \quad \lambda\,\rho : [\![e]\!].\ \mathit{fix}\ ([\![e \vdash \alpha : \mathit{func}(\theta, \theta)]\!]\ \rho)$$

## 9 Conclusion

As demonstrated in this paper, programming languages composed of a number of orthogonal features can be given simple and elegant formal definitions. Apart from the language features that have been considered in the previous sections, it is also possible to treat others in a similar way, including as records, elements of object-oriented programming and polymorphic types. The proposed methodology uses the new formalism of parametric context-free grammars to enhance the modularity and elegance of programming language definitions.

On the one hand, the use of PCFGs in the definition of syntax emphasizes the presence of orthogonal features, which interact with the basic syntactic domains of the language in a uniform way, and reduces the complexity of the definition. On the other hand, the presence of parametric production rules facilitates the definition of semantics by reducing the number of typing rules and of semantic equations that is required.

## References

1. J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, The Netherlands, 1981. North-Holland.
2. N. S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Software Engineering Laboratory, February 1998.

[e]The least fixed point operator $\mathit{fix} : (a \to a) \to a$ has the property $\mathit{fix}\ f = f\ (\mathit{fix}\ f)$.

3. N. S. Papaspyrou. A methodology for the definition of programming languages. In N. Mastorakis, editor, *Software and Hardware Engineering for the 21st Century, Proceedings of the 3rd Multiconference on Circuits, Systems, Communications and Computers (CSCC'99)*, pages 67–78. World Scientific Publishing, July 1999.
4. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.
5. P. Wadler. The essence of functional programming. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92)*, January 1992.
6. R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, Englewood Cliffs, NJ, 1991.