PEST Formal Specification

Guido de Caso Diego Garbervetsky Daniel Gorn
Departamento de Computacin, FCEyN, Universidad de Buenos Aires
{gdecaso, diegog, dgorin}@dc.uba.ar

May 7, 2012

1 Introduction

This is the PEST programming language formal specification version 1.1. PEST is an multiprocedural, imperative, while-style programming language that includes annotations as part of its native syntax.

2 Syntax

We use α to express types (e.g., integers, arrays) and Γ as a typing function that assigns one unique type to each variable in scope. The set $\text{Exp}^{\alpha}_{\Gamma}$ contains the expressions with type α using variables according to Γ.

As basic types, PEST requires the presence of integers and booleans as they are used in annotations and guards. Note that boolean expressions may quantification, provided it is always bounded (i.e., decidable in run-time)[1]. All expressions may also refer to v**@pre**, the value of the variable v at the beginning of the current procedure.

We use $\text{Sentence}_{\Gamma,\pi,\Gamma}$ as the set of every possible sentence that starts with Γ as a typing function and ends with Γ', potentially calling procedures defined in program π. A sentence may not change the typing of any existing variable, but it may extend the typing functions with new variables that appear in the program.

The PEST sentences are defined according to the rules in Figure 1.

$$\frac{\Gamma(v) = \alpha \quad e \in \text{Exp}^{\alpha}_{\Gamma}}{v \leftarrow e \in \text{Sentence}_{\Gamma,\pi,\Gamma}}\,(\text{ASSIGN}) \qquad \frac{v \notin \text{dom}(\Gamma) \quad e \in \text{Exp}^{\alpha}_{\Gamma}}{\textbf{local } v \leftarrow e \in \text{Sentence}_{\Gamma,\pi,\Gamma\{v\mapsto\alpha\}}}\,(\text{DEF})$$

$$\frac{}{\textbf{skip} \in \text{Sentence}_{\Gamma,\pi,\Gamma}}\,(\text{SKIP}) \qquad \frac{s_1 \in \text{Sentence}_{\Gamma,\pi,\Gamma'} \quad s_2 \in \text{Sentence}_{\Gamma',\pi,\Gamma''}}{s_1\ s_2 \in \text{Sentence}_{\Gamma,\pi,\Gamma''}}\,(\text{SEQ})$$

$$\frac{g \in \text{Exp}^{Bool}_{\Gamma} \quad g \text{ is quantifier free} \quad s_1 \in \text{Sentence}_{\Gamma,\pi,\Gamma'} \quad s_2 \in \text{Sentence}_{\Gamma,\pi,\Gamma''}}{\textbf{if } g \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \in \text{Sentence}_{\Gamma,\pi,\Gamma}}\,(\text{IF})$$

$$\frac{g \in \text{Exp}^{Bool}_{\Gamma} \quad g \text{ is quantifier free} \quad inv \in \text{Exp}^{Bool}_{\Gamma} \quad var \in \text{Exp}^{Int}_{\Gamma} \quad s \in \text{Sentence}_{\Gamma,\pi,\Gamma'}}{\textbf{while } g \textbf{ :?! } inv \textbf{ :\# } var \textbf{ do } s \textbf{ od} \in \text{Sentence}_{\Gamma,\pi,\Gamma}}\,(\text{WHILE})$$

$$\frac{\begin{array}{c} proc \in \pi \\ i \neq j \Rightarrow cp_i \neq cp_j \quad \text{pars}(proc) = p_1, \ldots, p_k \\ \Gamma(cp_i) = \Gamma_{proc}(p_i) \end{array}}{\textbf{call } proc(cp_1, \ldots, cp_k) \in \text{Sentence}_{\Gamma,\pi,\Gamma}}\,(\text{CALL})$$

Figure 1: PEST syntax

PEST programs are defined either as empty or by extending an existing program with a new procedure that potentially calls procedures in that program. Formally, the program set Prog is the smallest set that satisfies:

$$\frac{}{\emptyset \in \text{Prog}}\,(\text{PROG-EMPTY})$$

[1] Note that this does not affect the fact that statically resolving the truth value of a boolean expression is undecidable.

$$\frac{\begin{array}{c}\pi \in \text{Prog} \quad proc \notin \pi \\ \text{dom}(\Gamma_{proc}) = \{p_1, \ldots, p_k\} \quad i \neq j \Rightarrow p_i \neq p_j \\ pre, post \in \text{BoolExp}_{\Gamma_{proc}} \\ s \in \text{Sentence}_{\Gamma_{proc},\ \pi,\ \Gamma'}\end{array}}{\pi,\ proc(p_1, \ldots, p_k)\ \textbf{:?}\ pre\ \textbf{:!}\ post\ \{\ s\ \} \in \text{Prog}}\ (\text{PROG-EXTEND})$$

We define the local variables of a given sentence s as the set of variables that are incorporated by s, formally:

$$\text{locals}(s) = \text{dom}(\Gamma') \setminus \text{dom}(\Gamma)$$

Note that both procedures and cycles are augmented with annotations such as preconditions, invariants, variants and postconditions. The following sections make use of these in order to establish a safe notion of sentence semantics.

3 Operational semantics

A valuation σ is a function that maps each variable to a concrete value in its type domain. Valuations can be updated to reflect that the new value for an already defined variable v is n (noted $\sigma\{v \mapsto n\}$), extended to incorporate a new variable v with an initial value n (noted $\sigma\{\text{new } v \mapsto n\}$) or cropped to forget the value of a set of variables V (noted $\sigma \ominus V$).

Valuations can be extended to assign values to expressions if the according rules for each expression type are provided. We will denote $[\![e]\!]_\sigma = n$ to say that the valuation σ can assign value n to the expression e. Note that this is not always possible, for instance e may refer to a variable for which σ has no associated value.

Finally, the operational semantics of a Pest sentence s that starts from the valuation σ and finishes correctly rendering a valuation σ' is noted $\sigma \ \triangleright\ s \ \triangleright\ \sigma'$. The Pest language operational semantics is defined according to the rules in Figure 2.

$$\frac{[\![e]\!]_\sigma = n}{\sigma \ \triangleright\ v \leftarrow e \ \triangleright\ \sigma\{v \mapsto n\}}\ (\text{O-ASSIGN}) \qquad \frac{[\![g]\!]_\sigma = \text{true} \quad \sigma \ \triangleright\ s_1 \ \triangleright\ \sigma'}{\sigma \ \triangleright\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \ \triangleright\ \sigma' \ominus \text{locals}(s_1)}\ (\text{O-IF-T})$$

$$\frac{[\![e]\!]_\sigma = n}{\sigma \ \triangleright\ \textbf{local}\ v \leftarrow e \ \triangleright\ \sigma\{\text{new } \alpha \mapsto n\}}\ (\text{O-DEF}) \qquad \frac{[\![g]\!]_\sigma = \text{false} \quad \sigma \ \triangleright\ s_2 \ \triangleright\ \sigma'}{\sigma \ \triangleright\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \ \triangleright\ \sigma' \ominus \text{locals}(s_2)}\ (\text{O-IF-F})$$

$$\frac{}{\sigma \ \triangleright\ \textbf{skip} \ \triangleright\ \sigma}\ (\text{O-SKIP}) \qquad \frac{[\![g]\!]_\sigma = \text{false} \quad [\![inv]\!]_\sigma = \text{true}}{\sigma \ \triangleright\ \textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od} \ \triangleright\ \sigma}\ (\text{O-WHILE-F})$$

$$\frac{\sigma \ \triangleright\ s_1 \ \triangleright\ \sigma' \quad \sigma' \ \triangleright\ s_2 \ \triangleright\ \sigma''}{\sigma \ \triangleright\ s_1\ s_2 \ \triangleright\ \sigma''}\ (\text{O-SEQ}) \qquad \frac{\begin{array}{c}[\![g]\!]_\sigma = \text{true} \quad [\![inv]\!]_\sigma = \text{true} \quad [\![var]\!]_\sigma > 0 \\ \sigma \ \triangleright\ s \ \triangleright\ \sigma' \\ [\![var]\!]_{\sigma'} < [\![var]\!]_\sigma \\ \sigma' \ominus \text{locals}(s) \ \triangleright\ \textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od} \ \triangleright\ \sigma''\end{array}}{\sigma \ \triangleright\ \textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od} \ \triangleright\ \sigma''}\ (\text{O-WHILE-T})$$

$$\frac{[\![\text{pre}(proc)]\!]_\rho = \text{true} \quad \rho \ \triangleright\ \text{body}(proc) \ \triangleright\ \rho' \quad [\![\text{post}(proc)]\!]_{\rho'} = \text{true}}{\sigma \ \triangleright\ \textbf{call}\ proc(cp_1, \ldots, cp_k) \ \triangleright\ \sigma\{cp_1 \mapsto \rho'(p_1), \ldots\}}\ (\text{O-CALL}) \qquad \begin{array}{rcl}\rho(p_i) & \stackrel{\text{def}}{=} & \sigma(cp_i) \\ \rho(p_i\textbf{@pre}) & \stackrel{\text{def}}{=} & \sigma(cp_i)\end{array}$$

Figure 2: Pest operational semantics

Note that sentence semantics may lock if something goes wrong. For instance, if a loop is cycling and at a given moment the variant function does not decrease, then the semantics is not defined. The same happens if a called procedure precondition does not hold in the current valuation, a loop invariant is broken, etc.

4 Static semantics

In order to define static semantics we will first introduce the concept of safe expression condition. Given an expression e, safe(e) is a boolean expression guaranteeing that every valuation σ that makes it true can provide a value for e. For instance, safe($4/y$) $=$ $y \neq 0$.

$$\frac{p \models \text{safe}(e) \quad \exists\, v'\ (p\lfloor v \mapsto v' \rfloor \ \wedge\ v = e\lfloor v \mapsto v' \rfloor) \models q}{\{p\}\ v \leftarrow e\ \{q\}}\ \text{(S-ASSIGN)} \qquad \frac{p \models \text{safe}(e) \quad p \ \wedge\ v = e \models q}{\{p\}\ \textbf{local}\ v \leftarrow e\ \{q\}}\ \text{(S-DEF)}$$

$$\frac{p \models q}{\{p\}\ \textbf{skip}\ \{q\}}\ \text{(S-SKIP)} \qquad \frac{\{p\}\ s_1\ \{r\} \quad \{r\}\ s_2\ \{q\}}{\{p\}\ s_1\ s_2\ \{q\}}\ \text{(S-SEQ)}$$

$$\frac{\begin{array}{c}\textbf{true} \models \text{safe}(inv) \quad inv \models \text{safe}(g) \quad inv \models \text{safe}(var) \\ p \models inv \quad inv \wedge g \models p' \quad p' \models var > 0 \\ \{p'\}\ var_0 \leftarrow var\ \ s\ \{q'\} \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\{p\}\ \textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od}\ \{q\}}\ \text{(S-WHILE)} \qquad \frac{\begin{array}{c}p \models \text{safe}(g) \\ p \wedge g \models p_1 \quad \{p_1\}\ s_1\ \{q_1\} \quad q_1 \models q \\ p \wedge \neg g \models p_2 \quad \{p_2\}\ s_2\ \{q_2\} \quad q_2 \models q\end{array}}{\{p\}\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ \{q\}}\ \text{(S-IF)}$$

$$\frac{\begin{array}{c}1 \le i \le k \quad p\lfloor cp_i \mapsto p_i \rfloor \models \text{pre}(proc) \\ \exists o_1, \ldots, o_k\ (p\lfloor cp_i \mapsto o_i \rfloor \ \wedge\ \text{post}(proc)\lfloor p_i \mapsto cp_i,\ p_i\textbf{@pre} \mapsto o_i \rfloor) \models q\end{array}}{\{p\}\ \textbf{call}\ proc(cp_1, \ldots, cp_k)\ \{q\}}\ \text{(S-CALL)}$$

Figure 3: PEST static semantics

In general, given a boolean expression e in negated normal form (NNF), we can compute safe as follows:

$$\begin{array}{rcl}\text{safe}(\neg e) & = & \text{safe}(e) \\ \text{safe}(e_1 \wedge e_2) & = & \text{safe}(e_1) \wedge (e_1 \Rightarrow \text{safe}(e_2)) \\ \text{safe}(e_1 \vee e_2) & = & \text{safe}(e_1) \wedge (-e_1 \Rightarrow \text{safe}(e_2))\end{array}$$

In the presence of quantifiers, we can extend this:

$$\begin{array}{rcl}\text{safe}(\exists x.P(x)) & = & \forall x.\, \text{safe}(P(x)) \\ \text{safe}(\forall x.P(x)) & = & \forall x.\, \text{safe}(P(x))\end{array}$$

We will use $b_1 \models b_2$ to indicate that the boolean expression b_1 is stronger than the boolean expression b_2. That is, whenever a valuation makes b_1 true, then it makes b_2 true as well. Notice that in presence of unbounded quantification this is an undecidable problem.

Sustitution will be noted $e_1\lfloor e_2 \mapsto e_3 \rfloor$, meaning the expression that results from changing in e_1 each occurrence of e_2, putting e_3 instead. Notice that e_2 and e_3 must be of the same type.

We now define the static semantics for a PEST sentence. Instead of operational semantics that act in terms of valuation updates, static semantics acts by modifying boolean expressions that model many valuations. If p is a boolean expression that describes the possible valuations before the sentence s, and q is a boolean expression that describes the possible valuations after s, then $\{p\}\ s\ \{q\}$ will be the static semantics of s.

The PEST language static semantics is defined according to the rules in Figure 3.

5 Safe programs

There is a clear correlation between PEST's operational and static semantics. Using the latter, we can give a notion of *safe* program. In what follows, if π is a program and p a procedure, then π, p is the program obtained by appending p to π.

Definition 1 (Safe programs)**.** The set SAFE of programs is inductively defined as follows:

$$\frac{}{\emptyset \in \text{SAFE}}\ \text{(SAFE-EMPTY)}$$

$$\frac{\pi \in \text{SAFE} \quad \{\text{pre}(p)\}\ \text{body}(p)\ \{\text{post}(p)\}}{\pi, p \in \text{SAFE}}\ \text{(SAFE-EXTEND)}$$

Safe programs are the ones that respect their annotations on any run. That is, for each procedure whenever the precondition holds then the execution flows normally, satisfying every called procedure precondition and loop invariants, decreasing variants on each cycle and satisfying the postcondition. Formally:

Theorem 1 (Safe programs execute normally)**.**
Let π be a safe program and p a procedure in π. Then:

$$\begin{array}{c}\textit{for each valuation } \sigma \textit{ if } [\![\mathrm{pre}(p)]\!]_\sigma = \mathrm{true} \\ \textit{then it exists a valuation } \sigma' \textit{ such that } \sigma \triangleright \mathrm{body}(p) \triangleright \sigma' \textit{ and } [\![\mathrm{post}(p)]\!]_{\sigma'} = \mathrm{true}\end{array}$$

Proof: By induction in the structure of the derivations $\{\mathrm{pre}(p)\}\ \mathrm{body}(p)\ \{\mathrm{post}(p)\}$. See [1].

6 Postcondition Calculus

The static semantics rules of Figure 3 conform an axiomatic system to determine, given p, s and q wether or not $\{p\}\ s\ \{q\}$ holds. In some ocassions (such as trying to infer annotations for a procedure definition) it may probe useful to infer some q such that, for a given p and s $\{p\}\ s\ \{q\}$ holds. It is desirable that the obtained q was the strongest to satisfy the static semantics. For decidability's sake we will get a q that, hopefully, is strong enough for our purpouses. We note this calculated postcondition $\mathrm{post}(s,p) = q$ and we formally obtain it using the rules in Figure 4.

$$\frac{p \models \mathrm{safe}(e)}{\mathrm{post}(v \leftarrow e, p) = \exists\ v'\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = e\lfloor v \mapsto v' \rfloor)}\ (\text{Q-ASSIGN})$$

$$\frac{p \models \mathrm{safe}(e)}{\mathrm{post}(\textbf{local}\ v \leftarrow e, p) = p\ \wedge\ v = e}\ (\text{Q-DEF}) \qquad \frac{}{\mathrm{post}(\textbf{skip}, p) = p}\ (\text{Q-SKIP}) \qquad \frac{\mathrm{post}(s_1, p) = r \quad \mathrm{post}(s_2, r) = q}{\mathrm{post}(s_1\ s_2, p) = q}\ (\text{Q-SEQ})$$

$$\frac{p \models \mathrm{safe}(g)}{\mathrm{post}(\textbf{if}\ g\ \textbf{then}\ s\ \textbf{else}\ t\ \textbf{fi}, p) = Cl_{\exists\,\mathrm{locals}(s)}(\mathrm{post}(s, p \wedge g)) \vee Cl_{\exists\,\mathrm{locals}(t)}(\mathrm{post}(t, p \wedge \neg g))}\ (\text{Q-IF})$$

$$\frac{\begin{array}{c}\textbf{true} \models \mathrm{safe}(inv) \quad inv \models \mathrm{safe}(g) \quad inv \models \mathrm{safe}(var) \\ p \models inv \quad inv \wedge g \models var > 0 \\ \mathrm{post}(var_0 \leftarrow var\ \ s, inv \wedge g) = q' \\ q' \models inv \quad q' \models var < var_0\end{array}}{\mathrm{post}(\textbf{while}\ g\ \textbf{:?!}\ inv\ \textbf{:\#}\ var\ \textbf{do}\ s\ \textbf{od}, p) = inv\ \wedge\ \neg g}\ (\text{Q-WHILE})$$

$$\frac{1 \leq i \leq k \quad p\lfloor cp_i \mapsto p_i \rfloor \models \mathrm{pre}(pr)}{\mathrm{post}(\textbf{call}\ pr(cp_1, \ldots, cp_k), p) = \exists_{o_1,\ldots,o_k}\ (p\lfloor cp_i \mapsto o_i \rfloor\ \wedge\ \mathrm{post}(pr)\lfloor p_i \mapsto cp_i,\ p_i\textbf{@pre} \mapsto o_i \rfloor)}\ (\text{Q-CALL})$$

Figure 4: Pest postcondition calculus

The notation $Cl_{\exists V}(b)$ refers to the existencial closure of the boolean expression b with respect to every variable in V.

7 Precondition Calculus

Analogously to the previous section, we say that $\mathrm{pre}(s,q) = p$ when p is the calculated precondition of s with respect to the boolean expression q that characterises the state after executing s. The precondition we get will not necessarily be the weakest one. The formal rules are given in Figure 5.

Revision history

1.1 May 7th, 2012. Added definition for safe.

1.0 October 9th, 2008. Initial release.

References

[1] Guido de Caso. High-level iteration constructs as annotations for automated software verification. Master's thesis, Universidad de Buenos Aires, Tutored by Diego Garbervetsky and Daniel Gorín, `http://lafhis.dc.uba.ar/~gdecaso/tesisLicGuidodeCaso.pdf`, 2007.

$$\frac{}{\mathrm{pre}(v \leftarrow e, q) = \mathrm{safe}(e) \ \wedge \ q\lfloor v \mapsto e \rfloor}\ (\text{P-ASSIGN}) \qquad \frac{}{\mathrm{pre}(\mathbf{skip}, q) = q}\ (\text{P-SKIP})$$

$$\frac{}{\mathrm{pre}(\mathbf{local}\ v \leftarrow e, q) = \mathrm{safe}(e) \ \wedge \ q\lfloor v \mapsto e \rfloor}\ (\text{P-DEF}) \qquad \frac{\mathrm{pre}(s_1, r) = p \quad \mathrm{pre}(s_2, q) = r}{\mathrm{pre}(s_1 \ s_2, q) = p}\ (\text{P-SEQ})$$

$$\frac{}{\mathrm{pre}(\mathbf{if}\ g\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}, q) = \mathrm{safe}(g) \ \wedge \ (g \wedge \mathrm{pre}(s_1, q) \ \vee \ \neg g \wedge \mathrm{pre}(s_2, q))}\ (\text{P-IF})$$

$$\frac{\begin{array}{c}\mathbf{true} \models \mathrm{safe}(inv) \quad inv \models \mathrm{safe}(g) \quad inv \models \mathrm{safe}(var) \\ inv \wedge g \models p' \quad p' \models var > 0 \\ \mathrm{post}(var_0 \leftarrow var \ \ s, inv \wedge g) = q' \\ q' \models inv \quad q' \models var < var_0 \quad inv \wedge \neg g \models q\end{array}}{\mathrm{pre}(\mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od}, q) = inv}\ (\text{P-WHILE})$$

$$\frac{1 \le i \le k}{\begin{array}{c}\mathrm{pre}(\mathbf{call}\ proc(cp_1, \ldots, cp_k), q) = \mathrm{pre}(proc)\lfloor p_i \mapsto cp_i \rfloor \ \wedge \\ (\exists a_1, \ldots, a_k (\mathrm{post}(proc)\lfloor p_i \mapsto a_i, \ p_i\mathbf{@pre} \mapsto cp_i \rfloor \ \wedge \ q\lfloor cp_i \mapsto a_i \rfloor))\end{array}}\ (\text{P-CALL})$$

Figure 5: PEST precondition calculus