Reducing the Number of Annotations in a Verification-oriented Imperative Language

Guido de Caso, Diego Garbervetsky, and Daniel Gorín

Departamento de Computación, FCEyN, Universidad de Buenos Aires
{gdecaso,diegog,dgorin}@dc.uba.ar

Abstract. Automated software verification is a very active field of research which has made enormous progress both in theoretical and practical aspects. Recently, an important amount of research effort has been put into applying these techniques on top of mainstream programming languages. These languages typically provide powerful features such as reflection, aliasing and polymorphism which are handy for practitioners but, in contrast, make verification a real challenge. In this work we present Pest, a simple experimental, while-style, multiprocedural, imperative programming language which was conceived with verifiability as one of its main goals. This language forces developers to concurrently think about both the statements needed to implement an algorithm and the assertions required to prove its correctness. In order to aid programmers, we propose several techniques to reduce the number and complexity of annotations required to successfully verify their programs. In particular, we show that *high-level iteration constructs* may alleviate the need for providing complex loop annotations.

Key words: Annotations, language design, verifiability, high-level iteration constructs.

1 Introduction

Formal automated software verification regained in recent years the attention of the community. There are at least two reasons behind this resurgent success: on the one hand, there were crucial developments in automated theorem proving in the last fifteen years, with SAT-solvers finally reaching industrial strength; on the other, the focus was shifted to *partial specifications* which somehow overcomes many of the objections raised in [20]. Verification of partial specifications must be regarded as an error-detection procedure and, as such, akin to traditional forms of testing.

We will center on the particular form of verification where the source code is annotated with special assertions. These normally take the form of method pre and postconditions, loop and class invariants, etc. Special tools then read these annotated sources, generate verification conditions from them and feed these into automated provers [15]. Spec# [4] and ESC/Java [14] are two of the best-known examples. The former is based on a dialect of C# while the latter takes Java code with JML [17] annotations.

Apart from the abovementioned, there are ongoing research efforts in automated verification for almost every major programming language in use [6, 24, 23]. The rationale is to lower the adoption barrier by giving practitioners tools for verifying the code they are writing today. Now, while this is an undeniably sensible plan, we perceive that, in almost every case, the resulting "programming-language-with-annotations" regarded as a whole ends-up being not entirely satisfying. We consider next some of the reasons for this.

Lack of cohesion. Annotations are usually introduced as a "patch" to the language. Most of the time, this is done in a way such that regular compilers and IDE-tools regard them as mere comments. Moreover, most programming languages provide a way to perform *optional run-time assertion checks*. These are usually used to validate pre and postconditions or invariants and are, thus, the run-time counterparts of the verification annotations. But despite their dual nature, both mechanisms have normally no syntactical relation whatsoever.

Redundancy. In statically typed languages, the type of the input and output variables of a function are clearly part of its contract. But this means one ends-up with two completely unrelated ways of specifying contracts: one enforced by compilers (types) and the other by static checkers (the additional annotations).

Missed optimization opportunities. Optimizing compilers cannot leverage on program annotations in the same way they currently do on type information.

Inadequate semantics. We can most certainly exclude *"to ease automated verifiability"* from the list of goals that have driven the design of most modern-day programming languages. We cannot know for sure if today's mainstream languages would have been as popular without features such as complex inheritance mechanisms, uncontrolled method reentrancy or unrestricted aliasing. Nevertheless, the fact that the designers of SPEC# already had to diverge in slight ways from C#'s semantics [4] is indicating, in our opinion, a new driver for the programming languages to come.

In this paper we report on an ongoing experiment in language design. Our language forces programmers to introduce annotations specifying their intention when writing code. We will show that by coupling tightly the annotations with the language semantics we can rely on a verifying compiler to infer many of the annotations. Moreover, we find that *high level iteration constructs* can effectively be used as means of reducing the annotation burden of invariants.

2 A walk-through of the PEST language

In this section we will briefly present the PEST[1] programming language by way of examples. For a formal description the reader is referred to [8].

[1] After the eastern side of the Danube river in Budapest, pronounced [ˈpɛʃt].

Pest is an multi-procedural, non-recursive, structured, while-style, imperative programming language whose syntax natively incorporates various kinds of annotations. Its main objective is to provide a test-bed for exploring new concepts and ideas.

```
max($a$,$b$,$c$)
:? true
:! $(a \geq b \Rightarrow c = a) \wedge (a < b \Rightarrow c = b)$
:! $a = a$@pre $\wedge$ $b = b$@pre
{
  if $a \geq b$ then
    $c \leftarrow a$
  else
    $c \leftarrow b$
  fi
}
```

Fig. 1. Simple Pest procedure definition.

Figure 1 shows the definition of a simple procedure in Pest. Keywords **:?** and **:!** introduce pre and postconditions respectively. In the postcondition a**@pre** and b**@pre** denote the value of a and b at the beginning of the procedure execution. Therefore, this clause states that the value of a and b does not change; this is necessary since in Pest all values are copied-in and then copied-out.

There are currently only three data types in Pest: booleans, integers and arrays of integers. Variables are monomorphic and their type is inferred from use. For example, in Figure 1, all variables are integers, since the $\geq$ operator takes integers as arguments. Apart from classical boolean operators, boolean expressions may include *bounded* first-order quantification.

Figure 2 lists a Pest procedure containing a while-loop iterating over an array and a procedure call. Loop invariants are introduced using the **:?!** keyword and exactly one loop variant must be provided, using the **:#** keyword. Only variables are allowed as arguments on procedure calls and they are syntactically enforced to be distinct. Observe that since, additionally, values are copied in variable assignments, we impose a strict control over aliasing.

Operationally, annotations in Pest are interpreted as assertions. Roughly speaking, preconditions are evaluated prior to procedure calls and postconditions on return; invariants are checked before evaluating the loop guard. Since most of the semantic rules include the evaluation of annotations, a program execution gets stuck when the interpretation of one of them fails.

To enforce this semantics, annotations can be checked at run-time, but this can be very expensive. Alternatively, a Pest compiler can simply remove an assertion if it can be statically verified. In a way, this is reminiscent of *type*

```
arrayMax($A$, $m$)
:? $|A| > 0$
:! $\forall k \mathbin{/} 0 \leq k < |A| : m \geq A[k]$
:! $\exists k \mathbin{/} 0 \leq k < |A| : m = A[k]$
:! $A = A$@pre
{
  $m \leftarrow A[0]$
  local $i \leftarrow 1$
  while $i < |A|$
      :?! $1 \leq i \leq |A|$
      :?! $\forall k \mathbin{/} 0 \leq k < i : m \geq A[k]$
      :?! $\exists k \mathbin{/} 0 \leq k < i : m = A[k]$
      :?! $A = A$@pre
      :# $|A| - i$
   do
     local $t \leftarrow 0$
     local $e \leftarrow A[i]$
     call max($e$, $m$, $t$)
     $m \leftarrow t$
     $i \leftarrow i + 1$
   od
}
```

Fig. 2. A Pest procedure containing a loop.

erasure. Of course, the programmer will normally want to be aware of which assertions failed to be statically verified.

A Pest compiler can also *infer* pre and postconditions of a procedure. If only the precondition is given a postcondition can be obtained by way of a symbolic computation; on the other hand, starting from a postcondition, a precondition can be obtained using a variation of the notion of *weakest precondition*. In simple cases, like the procedure in Figure 1, one can remove the annotation altogether and rely solely on inference. Details are given in §4.

Eliminating the need for explicitly given loop invariants is, of course, highly desirable. A lot of research has been done in loop invariant inference (for instance, [13, 18]). In §5 we will show that higher-level iteration constructs can be alternatively regarded as encapsulating "directives" on how to build a proper invariant from the loop body.

Using the ideas sketched above, arrayMax can be rewritten in Pest as shown in Figure 3 which requires no annotations for the loop and still enforces its contract. Notice that, in addition to the invariant, in this case we are also able to remove the precondition and part of the postcondition since they can be inferred by the Pest compiler using the rules presented in §4.

```
easyArrayMax($A$,$m$)
:! $\forall k \;/\; 0 \leq k < |A| : m \geq A[k]$
:! $\exists k \;/\; 0 \leq k < |A| : m = A[k]$
{
    $m \leftarrow A[0]$
    for $i$ from $1$ to $|A|$ do
        local $t \leftarrow 0$
        local $e \leftarrow A[i]$
        call max($e$, $m$, $t$)
        $m \leftarrow t$
    od
}
```

Fig. 3. Using a **for** construct to remove annotations.

3 Formal semantics of Pest

In this section we introduce Pest semantics. In §3.1 we briefly comment on Pest (big-step) operational semantics. As we already mentioned in the preceding section, annotations are interpreted as assertions and a computation gets stuck whenever one of these fails. In §3.2, we present a static formulation based on Hoare-style clauses. This characterizes those programs that cannot get stuck according to the operational semantics. In this sense, it is reminiscent of a type system; but, observe that it relies on a semantic entailment relation over formulas ($\models$) that is undecidable in the general case.

In this setting, automatic verification of a Pest program can be performed by checking conformance to its static semantics using a computable approximation ($\vdash$) of the $\models$-relation. Later, in §4 we will derive some calculi from the static semantics to cope with several inference tasks.

3.1 Operational semantics

Pest *operational semantics* is based on the standard notion of while-style programming languages semantics. Annotations are handled by incorporating *conditions* (i.e. assertions) into the semantic rules that disallow the execution of a program that violates them.

The semantic rules are given in terms of state transformations. A state σ is a function that maps program variables to concrete values of the proper type. With $\sigma\{v \mapsto n\}$ we denote a state that coincides with σ except, possibly, in the value for v which, in the former, is n; $\sigma \ominus V$ is the restriction of σ to a domain that does not contain any variable in V. We use $[\![e]\!]_\sigma$ to denote the *value* of an expression e under σ.

We write $\sigma \vartriangleright s \vartriangleright \sigma'$ to express that a Pest sentence s, when run from state σ, finishes correctly and arrives in state σ'. Some rules are depicted in Figure 4; for the complete set of rules refer to [8].

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

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

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

Fig. 4. PEST operational semantics (fragment)

Consider the rules for the **while** statement. Observe first that if $[\![inv]\!]_\sigma \neq$ true then no rule applies and, thus, a computation will stick in that case. When the guard is false, the state is not affected (the language syntax guarantees that guards are free of side-effects). Alternatively, when the guard is true, the variant must be above zero in order to continue. Observe that σ' is the state after an execution of the loop body, including locally-defined variables; if the variant didn't decrease the computation will stuck.

For procedure calls, state ρ binds formal and actual parameters; the precondition of the called procedure must hold for this state. The state ρ', if defined, is the result of executing the procedure body from ρ; the postcondition must hold in ρ'. At the end, actual parameters are updated with the final values assigned to the formal parameters (recall that all parameters are in/out in PEST).

3.2 Hoare-style static semantics

We first need to introduce some preliminary definitions. For σ a state and b a PEST boolean expression[2], we write $\sigma \models b$ if $[\![b]\!]_\sigma =$ true. We will write $b_1 \models b_2$ to indicate that, for all σ, whenever $\sigma \models b_1$ holds, then $\sigma \models b_2$ must hold too (i.e. b_1 is *stronger* than b_2).

We also require a notion of "safeness" for the evaluation of expressions. Given an expression e, we want safe(e) to be a boolean expression such that $\sigma \models$ safe(e) implies that $[\![e]\!]_\sigma$ is defined. For example, we expect safe($a[\,i\,]\ /\ y$) to be the expression "$0 \leq\ i \wedge i < |a|\ \wedge y \neq\ 0$". A formal definition of these conditions is straightforward. Finally, $e_1\lfloor e_2 \mapsto e_3\rfloor$ denotes the expression that results from replacing every occurrence of e_2 by e_3 in e_1. Observe that e_2 and e_3 must be of the same type.

[2] Throughout this section, boolean expressions are augmented with unbounded existential quantification.

Instead of using states like in the operational case, for the static semantics we will use predicates (i.e. boolean expressions) that describe a (possibly infinite) set of states. For boolean expressions p and q and s a Pest sentence, $\{p\}\ s\ \{q\}$ must be read "after executing s from a σ such that $\sigma \models p$, we obtain a σ' such that $\sigma' \models q$".

Figure 5 lists some of the static rules (for the complete list, refer to [8]). Consider the rule for assignments: the first premise guarantees that the program won't get stuck when evaluating e; the second one states that q is a consequence of what was known prior to the assignment ($p\lfloor v \mapsto v' \rfloor$) and its effect ($v = e\lfloor v \mapsto v' \rfloor$). The existentially quantified variable v' stands for the value of v before the assignment (this requires unbounded quantification).

$$\frac{\begin{array}{c} p \models \text{safe}(e) \\ \exists\ v'\ (p\lfloor v \mapsto v' \rfloor\ \wedge\ v = e\lfloor v \mapsto v' \rfloor) \models q \end{array}}{\{p\}\ v \leftarrow e\ \{q\}}\ \text{(S-ASSIGN)}$$

$$\frac{\begin{array}{c} p \models \text{safe}(g) \\ p \wedge g \models p_1 \quad \{p_1\}\ s_1\ \{q_1\} \quad q_1 \models q \\ p \wedge \neg g \models p_2 \quad \{p_2\}\ s_2\ \{q_2\} \quad q_2 \models q \end{array}}{\{p\}\ \textbf{if}\ g\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ \{q\}}\ \text{(S-IF)}$$

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

Fig. 5. Pest static semantics (fragment)

The premises of the rule for **while** can be seen as both a proof of the Fundamental Invariance Theorem for Loops [11] and a proof of termination using the loop variant. The predicate p' represents any state where the invariant and the condition of the while hold; the loop body is augmented with an initial assignment to a *fresh* variable var_0 that is used to prove that the variant decreases.

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:*

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

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

Theorem 1 (Safe programs execute normally) *Let $\pi \in$ SAFE and let p be a procedure in π. For each σ such that $\sigma \models \mathrm{pre}(p)$, there exists a state σ' such that $\sigma \vartriangleright \mathrm{body}(p) \vartriangleright \sigma'$ and $\sigma' \models \mathrm{post}(p)$.*

Proof. It is a longish yet straightforward induction on the length of a derivation of $\{\mathrm{pre}(p)\}\ \mathrm{body}(p)\ \{\mathrm{post}(p)\}$. See [8].

4 Reducing the annotation burden

In the previous section we presented the PEST programming language, including a soundness result showing that programs conforming to PEST static semantics do not go wrong. Still, providing PEST annotations is a heavy and complex task. In this section we discuss two techniques that assist the programmer by inferring and completing annotations.

4.1 Inference of procedure contracts

We will describe a technique for synthesizing procedure pre and postconditions. To do that we will specialize the static semantic rules and turn them into inference rules. This will somehow resemble the process of developing type inference rules from a type system.

Figure 6 shows some of the rules to compute a postcondition $\mathrm{post}(s, p)$ from given p and s. The complete calculus is given in [8].

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

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

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

Fig. 6. PEST postcondition calculus (fragment)

Observe the way in which these rules specialize those in Figure 5. For example, in the specialized rule for assignments, $\mathrm{post}(v \leftarrow e, p)$ is simply the strongest q satisfying the original assignment rule. The rule for **if** eliminates (using an existential closure[3]) the local variables defined in each of the branches from their respective inferred postcondition.

[3] The existential closure of a boolean expression b with respect to a set of variables $\{v_1, \ldots, v_n\}$ is $\exists v_1, \ldots, v_n(b)$

It is worth noticing that the inferred postcondition may contain unbounded existential quantifications. The main drawback of this is that, in that case, it is not valid to literally include it in the code. Observe also that an assertion with unbounded quantification cannot be checked at runtime; nevertheless, the inferred postconditions are correct by construction and the compiler can omit its associated runtime checks.

A similar approach can be used to infer procedure preconditions (denoted $\mathrm{pre}(s, q)$). In this case, we use weakest-preconditions to find a suitable predicate p, except for **while** sentences where we simply use the provided invariant. The rules for computing $\mathrm{pre}(s, q)$ can be found in [8].

Interestingly, we can infer non-trivial pre and postconditions even if the programmer provides no procedure annotations (other than loop annotations). First, we infer a predicate that guarantees the normal execution of a procedure body s:

$$P \overset{\mathrm{def}}{=} \mathrm{pre}(s, \mathbf{true})$$

This gives us a proper precondition for the procedure. Next, we strengthen P by giving a symbolic initial value to each parameter ($p_i = p_i\mathbf{@pre}$) and use it to infer a postcondition Q:

$$Q \overset{\mathrm{def}}{=} \mathrm{post}(s, P \wedge p_1 = p_1\mathbf{@pre} \wedge \ldots \wedge p_k = p_k\mathbf{@pre})$$

The reader should verify that if the pre and postconditions of the procedures in Figures 1 and 2 are omitted, this procedure will infer logically equivalent ones.

4.2 Strengthening annotations

We will now show how we can take advantage of Pest's restrictions on variable aliasing to incorporate inexpensive enhancements of invariants and postconditions. In a nutshell, we can propagate known facts about variables in certain scopes if we know that their contents are not altered. For example, let us say we know that before entering a loop it is always the case that variable j has value e; then if we can determine that the loop body does not update j, we can add $j = e$ to the invariant.

In the absence of aliasing, a simple, sound approximation of the set of unmodified variables is to compute first the set of variables potentially modified and then take its complement. We call $\mathrm{modVars}(s)$ the set of (potentially) modified variables; its definition is straightforward and can be found in [8].

What follows is a definition of our annotation strengthening function which takes a sentence s and an entry point p and produces a sentence s' with possibly strengthened annotations. The translation does not alter assignments, local variable definitions or procedure calls. In the case of sentence sequence, we define it as:

$$\mathrm{tr}^{\mathrm{S}}(s_1\ s_2,\ p) \overset{\mathrm{def}}{=} s_1'\ \mathrm{tr}^{\mathrm{S}}\Big(s_2,\ \mathrm{post}(s_1', p)\Big)$$

where $s_1' = \mathrm{tr}^{\mathrm{S}}(s_1,\ p)$. That is, we first translate s_1 and use its postcondition to translate s_2.

$$\begin{array}{r} \mathbf{1 \leq i \leq |A|} \wedge \forall k \ / \ 0 \leq k < \mathbf{i} : \quad m \geq A[k] \wedge \exists k \ / \ 0 \leq k < \mathbf{i} : \quad m = A[k] \wedge A = A@\text{pre} \\ \forall k \ / \ 0 \leq k < \mathbf{|A|} : m \geq A[k] \wedge \exists k \ / \ 0 \leq k < \mathbf{|A|} : m = A[k] \wedge A = A@\text{pre} \end{array}$$

Fig. 7. Invariant and postcondition of the main loop in the arrayMax procedure. Differences are highlighted.

When dealing with conditional statements we proceed by translating each branch using the conjunction of p with the guard g:

$$\mathrm{tr}^{\mathrm{S}}(\mathbf{if}\ g\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi},\ p) \overset{\text{def}}{=} \begin{array}{l} \mathbf{if}\ g\ \mathbf{then}\ \mathrm{tr}^{\mathrm{S}}(s_1,\ p \wedge g) \\ \quad \mathbf{else}\ \mathrm{tr}^{\mathrm{S}}(s_2,\ p \wedge \neg g)\ \mathbf{fi} \end{array}$$

Finally, in the presence of a loop we strengthen its invariant. We existentially close p over the variables that are potentially modified by the loop body:

$$\mathrm{tr}^{\mathrm{S}}(\mathbf{while}\ g\ \mathbf{:?!}\ inv\ \mathbf{:\#}\ var\ \mathbf{do}\ s\ \mathbf{od},\ p) \overset{\text{def}}{=} \begin{array}{l} \mathbf{while}\ g\ \mathbf{:?!}\ I\ \mathbf{:\#}\ var \\ \quad \mathbf{do}\ \mathrm{tr}^{\mathrm{S}}(s,\ I \wedge g)\ \mathbf{od} \end{array}$$

where $I = inv \wedge Cl_{\exists\,\mathrm{modVars}(s)}(p)$.

Using this technique we can strengthen both invariants and postconditions. For the latter case we simply take the postcondition of the last instruction of the procedure and existentially eliminate the local variables, in order to leave the postcondition only in terms of procedure parameters.

5 High-level iteration constructs as annotations

In this section we focus on what we call "high-level iteration constructs". These capture recurrent *iteration patterns* and are frequently included in programming languages to reduce error-prone boilerplate code. Examples of such constructs are Pascal-style *for*-loop and C#'s *foreach*. We will see next that this kind of constructs can be seen as implicitly carrying their own proof obligations, reducing the need for explicit annotations.

5.1 The *for* construct

We first consider the Pascal-style *for* loop and take as motivating example the arrayMax procedure in §2. In Figure 7 we compare the postcondition of the loop with its invariant. Clearly, they are syntactically very close. In fact, if we are given a postcondition and a *for*-loop, we can simply try to *guess* a candidate invariant for the loop without even considering the loop body. Of course, the correctness of the invariant will have to be statically verified. On the other hand, a correct variant can be trivially obtained.

Formally, given a sentence of the form **for** i **from** l **to** h **do** s **od** (without annotations) and its postcondition in the form of a predicate Q_f, we can macro-expand the sentence into the following *lower-level* code:

```
arrayInc($A$)
{
    local $i \leftarrow 0$
    while $i < |A|$
        :?! $0 \leq i \leq |A|$
        :?! $\forall k \;/\; 0 \leq k < i : A[k] = A$@pre$[k] + 1$
        :?! $\forall k \;/\; i \leq k < |A| : A[k] = A$@pre$[k]$
        :# $|A| - i$
    do
        $A \leftarrow$ update $A$ on $i$ with $A[i] + 1$
        $i \leftarrow i + 1$
    od
}
```

Fig. 8. PEST procedure that increases each element in an array.

```
local $i \leftarrow l$
while $i < h$
    :?! $l \leq i \leq h \wedge Q_f\lfloor h \mapsto i \rfloor$
    :# $h - i$
do
    $s$
    $i \leftarrow i + 1$
od
```

In order to apply this expansion we are required to provide a loop postcondition Q_f. Nevertheless, this can be accomplished using the precondition calculus of §4.1 from the procedure postcondition up to the *for*-sentence we need to expand. If s contains a nested *for*-construct, a postcondition for it can be derived from the inferred invariant and the expansion can be recursively applied.

Observe that in the easyArrayMax procedure of Figure 3, Q_f is simply the postcondition of the procedure, since the *for* construct is located at the very end of the procedure body.

5.2 Declarative iteration constructs

In §4.1 we saw that if the loop-invariants are provided, then postconditions can be omitted; conversely, in the previous section we showed that in the case of *for*-loops invariants can be traded for postconditions. In this section we will show that when more specific iteration constructs are used, both can be dispensed with. We will only consider here the *map*-construct (reminiscent of the *map* function over lists in functional languages) but the idea is easily extensible to other constructs.

Consider the procedure of Figure 8 (notice that the procedure contract was omitted and left for automatic inference). The loop iterates over an array A using an indexing variable i. On each iteration, only the element of A at position i is updated. Furthermore, this is done taking into account only the value of i and $A[i]$. The loop condition is not affected by changes to A. We call this iteration pattern, in which array elements are updated independently of the others, a *map*.

The invariant for this iteration pattern states that the already visited array elements were updated whereas the remaining elements are unchanged. The variant reflects the fact that the array is traversed in a forward direction.

Using our proposed **map** construct, the procedure in Figure 8 can be written like this:

```
arrayInc($A$)
{
  map
    $A[i] \leftarrow A[i] + 1$
  in $A\mathbf{[..}\, i\, \mathbf{..]}$
}
```

Formally, this works as follows. Let A be an array, i a fresh variable and s a sentence such that $\text{modVars}(s) \setminus \text{locals}(s) = \{A\}$ and A is accessed only indexed by i. In that case, **map** s **in** A[**..** i **..**] is well-formed and gets expanded as follows:

```
local $i \leftarrow 0$
while $i < |A|$
    :?! $0 \leq i < |A|$
    :?! $\forall k \,/\, 0 \leq k < i : post_s\lfloor i \mapsto k \rfloor$
    :?! $\forall k \,/\, i \leq k < |A| : A[k] = A\mathbf{@pre}[k]$
    :# $|A| - i$
do
    $s$
    $i \leftarrow i + 1$
od
```

where $post_s = \text{post}(s, 0 \leq i < |A|)$. It can easily be seen that the inferred invariant is always correct. Thus, not only can the compiler erase the assertion checks, it doesn't even need to statically verify them.

Notice that the invariant inferred for the map high-level construct can be further strengthened using the technique presented in §4.2.

6 Experience

In order to test these ideas we developed a tool called BudaPest which is available online[4] as an Eclipse plug-in.

The tool takes Pest programs, statically verifies them and compiles them to Java code. For the verification step the code is translated into an intermediate assume/assert style language similar to BoogiePL [9]. Unlike the Boogie verifier, our tool leverages the structure of the original program to discharge verification conditions one by one to SMT-solvers, such as CVC3 [5], Yices [12] and Z3 [2].

[4] http://lafhis.dc.uba.ar/budapest

We split verification conditions in several pieces in order to allow the provers to work in parallel, so we can leverage each of the provers' power and combine their results. Splitting verification conditions also has the advantage of enabling early detection (during the verification process) of unsatisfiable conditions since once a subformula fails there is no need to continue with the rest of the subformulas.

We tested the usability of this tool in a first-year Computer Science course. The curriculum of the course includes proving correctness and termination of simple imperative programs specified using contracts written in a fragment of first-order logic and proven by way of Hoare axiomatic semantics [16]. With the aid of the tool they were able to automatically verify the correctness of their implementations of algorithms such as bubble and insertion sorting, linear and binary search, etc., which they previously had to do by hand.

The experience was very encouraging for the students since they found that concepts they had learnt during the course (preconditions, postconditions, invariants, variant functions) could be applied in "real" applications and produced code with guarantees of (partial) correctness and termination.

Finally, it is worth mentioning that the integrated nature of the Pest language forced the students to concurrently think about both the statements needed to implement an algorithm and the assertions required to prove its correction.

7 Related Work

There is a plenty of research on the automatic verification of programs. Due to lack of space we will just mention some of those we consider closer to our work.

There are several languages and tools that incorporate Hoare-style specifications to automatically prove partial correctness in imperative languages. Some well known examples are ESC/Java [14], JML [17], ESC/Modula-3 [10], Spec# [4] and SPARK/Ada [3]. These systems enrich the language with user provided annotations which can be checked on runtime or statically analyzed by generating verification conditions that are discharged to a theorem prover.

The approach followed by SPARK/Ada is closer to ours in the sense that they impose limitations to the language in order to make verification possible. We believe they go too far since they limit syntactically to programs where assertions are decidable, making them too restrictive. On the contrary, the problem with the other tools is that they deal with complex languages that provide many features such as polymorphism, concurrency, aliasing or reflection, which are comfortable for program development but complicate verification.

Our work follows the philosophy that a compiler (or interpreter) should try to reject misbehaving programs. This is similar the approach followed by several languages that provide expressive type systems such as Deputy [7], which uses dependent types [19] to type-check low level imperative programs. For functional languages there are many proposals; just to name one, Cayenne [1] extends Haskell with dependent types. Recently, [21] presented a type theory for higher order functional programs which incorporates Hoare style specifications into types, making it possible to enforce correct use of side effects.

8 Final thoughts

In this paper we presented PEST, a simple imperative programming language, which was designed with verifiability in mind. We proposed a series of techniques that aim at mitigating the annotation burden required to verify programs. Firstly, we showed an inference mechanism for procedure pre and postconditions and then a simple method to strengthen loop invariants and postconditions. As a distinguishing contribution we extended PEST with high-level iteration constructs that allow programmers to dramatically reduce annotations and yet maintain the ability to prove correctness of relatively complex pieces of software. Our first experiences showed that an integrated approach such as ours was gracefully adopted by young students with no pre-concepts with respect to programming languages or tools.

We would like to explore, in the near future, the possibility of extending PEST incorporating features that would increase its expressiveness without sacrificing verifiability. We plan to allow the programmer to define and use her own data types and provide means to reason about representation invariants. Adding dynamic memory support is another priority but, in order to keep a verifiable language, we believe we must enforce an alias control mechanism such as [22]. Finally, we are interested in providing the means for programmers to define their own language constructs by feeding the BUDAPEST tool with syntax and macro expansion definitions, and relying on the base tools (i.e. pre and postcondition calculi) that we provided in this paper to ensure correctness.

References

1. L. Augustsson. Cayenne-a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, 1999.
2. T. Ball, S. K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In *LPAR*, 2005.
3. J.G.P. Barnes and Praxis Critical Systems Limited. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.
4. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*. Springer, 2005.
5. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the* 16^{th} *CAV*, 2004.
6. S. Chatterjee, S.K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. 2007.
7. J. Condit, M. Harren, Z. Anderson, D. Gay, and G.C. Necula. Dependent Types for Low-Level Programming. *Lecture Notes in Computer Science*, 4421:520, 2007.
8. G. de Caso, D. Garbervetsky, and D. Gorín. PEST formal specification. Technical report, Universidad de Buenos Aires, `http://lafhis.dc.uba.ar/budapest/` Theory section, 2008.

9. R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
10. D. L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report #159, Palo Alto, USA, 1998.
11. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.
12. B. Dutertre and L. de Moura. The Yices SMT solver. *Available at http://yices.csl.sri.com/, August*, 2006.
13. M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.
14. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02*, 2002.
15. D.I. Good, R.L. London, and WW Bledsoe. An interactive program verification system. In *Proceedings of the international conference on Reliable software table of contents*, pages 482–492. ACM New York, NY, USA, 1975.
16. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
17. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
18. K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.
19. J. McKinna. Why dependent types matter. *Proc. ACM Symp. on Principles of Programming Languages (POPL 2006)*, 2006.
20. R.A. De Millo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.
21. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *ICFP '06*, 2006.
22. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 158–185, London, UK, 1998. Springer-Verlag.
23. N. Norwitz. PyChecker. *SourceForge project http://pychecker. sourceforge. net.*
24. D.N. Xu. Extended static checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59. ACM New York, NY, USA, 2006.