Reducing the number of annotations in a verification-oriented imperative language

Guido de Caso Diego Garbervetsky Daniel Gorín

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

APV '09

Static typing: a successful form of program verification

```
main() {
    tmp = "hello";
    return foo(tmp);
}

foo(x) {
    return x^2;
}
```

Figure: Runtime error

```
int main() {
    string tmp = "hello";
    return foo(tmp);
}

int foo(int x) {
    return x^2;
}
```

Figure: Compile-time error

The type system is a core part of a programming language

A type system added to an untyped language at a later stage...

...may lead to a rather awkward construction!

The *type–contract* analogy

Static typing	**Program verification**
Type	Contract
Typing rules	Semantics
Type checking	Contract verification
Type inference	Contract inference

Our long-term goal

A programming language

- designed from first principles to be automatically verified,
- inspired on the *"type–contract analogy"*,
- combining *well-known* and *new* ideas in a coherent way.

Why?

- We can pick properties we want the language to *enforce*
- We can deviate from traditional practices when needed

Our tiny first step in that direction...

Pest 1.0

- Basic while-style language
- Ints, bools and arrays only
- Pre/post inference*

Pest 1.1

- More iteration constructs
- (no invariants required)

This is Pest

```
sumN(n, s)
:? n ≥ 0
:! s = n * (n+1) / 2
:! n = n@pre
{
    s ← 0
    local i ← 1
    while i ≤ n
        :?! 1 ≤ i ∧ i ≤ n+1 ∧ s = (i−1) * i / 2
        :# n − i + 1
    do
        s ← s + i
        i ← i + 1
    od
}
```

Pest static semantics

Example (The *while* sentence)

$$\frac{\begin{array}{c} \mathit{true} \models \mathsf{safe}(\mathit{inv}) \quad \mathit{inv} \models \mathsf{safe}(g) \quad \mathit{inv} \models \mathsf{safe}(\mathit{var}) \\ p \models \mathit{inv} \quad \mathit{inv} \wedge g \models p' \quad p' \models \mathit{var} > 0 \\ \{p'\}\ \mathit{var}_0 \leftarrow \mathit{var}\ \ s\ \{q'\} \\ q' \models \mathit{inv} \quad q' \models \mathit{var} < \mathit{var}_0 \quad \mathit{inv} \wedge \neg g \models q \end{array}}{\{p\}\ \mathit{while}\ g\ :?!\ \mathit{inv}\ :\#\ \mathit{var}\ \mathit{do}\ s\ \mathit{od}\ \{q\}}\ \text{(S-WHILE)}$$

Pre and postcondition inference

```
max(a,b,c)
:? true
:! a ≥ b ⇒ c = a
:! a < b ⇒ c = b
:! a = a@pre ∧ b = b@pre
{
    if a ≥ b then
        c ← a
    else
        c ← b
    fi
}
```

Figure: Annotated Pest procedure

```
max(a,b,c)
{
    if a ≥ b then
        c ← a
    else
        c ← b
    fi
}
```

Figure: Annotations out!

OK, but what can I do about loops?

```
arrayInc(a[])
{
    local k ← 0
    while k < |a|
        :?! 0 ≤ k ∧ k ≤ |a|
        :?! ∀ i from 0 to k−1: a[i] = a@pre[i] + 1
        :?! ∀ i from k to |a|−1: a[i] = a@pre[i]
        :# |a| − k
    do
        a[k] ← a[k] + 1
        k ← k + 1
    od
}
```

Figure: Pest procedure with a loop

Invariants = Headaches

A lesson learnt from functional programming

Common recursion pattern

$$f\ [\,] = [\,]$$
$$f\ (x{:}xs) = x{+}1 : f\ xs$$

$$g\ [\,] = [\,]$$
$$g\ (x{:}xs) = x{*}x : g\ xs$$

Recusion pattern abstracted away

$$\mathrm{map}\ fun\ [\,] = [\,]$$
$$\mathrm{map}\ fun\ (x{:}xs) = fun\ x : \mathrm{map}\ fun\ xs$$

$$f\ xs = \mathrm{map}\ (+1)\ xs$$
$$g\ xs = \mathrm{map}\ (\lambda x \rightarrow x{*}x)\ xs$$

The *map* iteration pattern

```
arrayInc(a[])
{
    local k ← 0
    while k < |a|
        :?! 0 ≤ k ∧ k ≤ |a|
        :?! ∀ i from 0 to k−1: a[i] = a@pre[i] + 1
        :?! ∀ i from k to |a|−1: a[i] = a@pre[i]
        :# |a| − k
    do
        a[k] ← a[k] + 1
        k ← k + 1
    od
}
```

Figure: While-based array iteration

The *map* iteration pattern

```
mapIteration(a[])
{
    local k ← 0
    while k < |a|
        :?! 0 ≤ k ∧ k ≤ |a|
        :?! ∀ i from 0 to k−1: a[i] = Trans(a@pre[i], i)
        :?! ∀ i from k to |a|−1: a[i] = a@pre[i]
        :# |a| − k
    do
        a[k] ← Trans(a[k], k)
        k ← k + 1
    od
}
```

Figure: The *map* iteration pattern

The *map* iteration pattern

```
easyArrayInc(a[])
{
    map e in a[..k..] do
        e ← e + 1
    od
}
```

Figure: Array iteration with the *map* looping construct

The *for* iteration pattern

```
sumN(n, s)
:?  n ≥ 0
:!  s = n * (n+1) / 2
:!  n=n@pre
{
    s ← 0
    local  i ← 1
    while  i ≤ n
        :?!  s = (i−1) * i / 2
        :#  n − i + 1
    do
        s ← s + i
        i ← i + 1
    od
}
```

Figure: While-based sum of first n integers

The *for* iteration pattern

```
sumN(n, s)
:? n ≥ 0
:! s = n * (n+1) / 2
:! n=n@pre
{
    s ← 0
    for i from 1 to n
        :?! 1 ≤ i ∧ i ≤ n+1 ∧ s = (i−1) * i / 2
    do
        s ← s + i
    od
}
```

Figure: Sum of first n integers using a classic *for*

The *for* iteration pattern

```
easySumN(n, s)
:!  s = n * (n+1) / 2
:!  n = n@pre
{
    s ← 0
    for i from 1 to n do
        s ← s + i
    od
}
```

Figure: Sum of first n integers using the *for* looping construct

Inferring the invariant of a *for*-loop

$$:!\ \ s = n * (n{+}1)\ /\ 2$$

Figure: *sumN* postcondition

$$:?!\ \ 1 \leq\ i \wedge i \leq\ n{+}1 \wedge s = (i{-}1) * i\ /\ 2$$

Figure: *sumN* loop invariant

Demo

Tool demo

- Tool is available at `http://lafhis.dc.uba.ar/budapest`

Future work

- Extending the language with ADTs.
- User defined high-level constructs.
- Increasing the expressiveness of predicates.