Extended static checking (ESC) via the PF transform

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI/UM, 2008

Motivation

- Software design is **error-prone**.
- Negative impact of programming errors on software productivity can only be limited by **catching** them early.
- Static checkers (eg. syntax and type checkers) are tools which catch errors at **compile-time**, ie. before running the program.
- Errors such as *null dereferencing*, division by 0 and array bound overflow, are not caught by standard static checking.
- Detecting their presence requires extensive **testing**, and if their presence can not be excluded with certainty, they must be handled at **run-time** via exception mechanisms.

Abstract modeling

- Software **formalists** argue that error checking in the coding phase is too late
- First, a formal **model** should be written, queried, reasoned about, and possibly animated (using eg. a symbolic interpreter).
- Formal modeling relies on *"rich"* datatypes such as eg. finite mappings, finite sequences, and recursive data structures, which **abstract** from much of the complexity found in common imperative programming languages (eg. pointers, loop boundaries)

Dynamic checking

- However, "rich" datatyping is not able to capture *all* properties
- This means that additional **constraints** need to be added to models such as **invariants** (attached to types) and **pre-conditions** (attached to operations)
- Checking such constraints is once again a process which falls outside standard static type-checking, leading to a so-called **dynamic** type checking process.

Extended static checking

- Static checking of formal models involving such constraints is a complex process, relying on generation and discharge of **proof obligations** [5]:

  > The valid objects of *Datec* are those which (...) satisfy inv-*Datec*. This has a profound consequence for the type mechanism of the notation. (...) The inclusion of a sub-typing mechanism which allows truth-valued functions forces the type checking here to rely on proofs.

- While proof obligations can be generated mechanically, their discharge is in general above the **decidability ceiling** in requiring full-fledged formal verification (**theorem proving**).

Extended static checking

- Between the two extremes (cheap, decidable static checking *versus* costly theorem proving), **extended static checking** (ESC) [4] aims to catch more errors at compile-time at the relatively moderate cost of adding annotations to the code which record *design decisions* which were lost throughout the programming process (if ever explicitly recorded).
- ESC tools have been developed for imperative programming languages such as Modula-3 (ESC/Modula-3) and Java (ESC/Java) [4].
- At their heart we find a verification condition generator and the **Simplify** theorem prover [3].

ESC

- Verification conditions are predicates in first-order logic which are computed in weakest pre-condition style.
- Theorem proving is performed by a combination of techniques, including SAT solvers, matching algorithms, and heuristics to guide proof search.
- In what follows we follow the spirit of this approach but intend to apply it much earlier in the design process
- We wish to perform ESC for formal modeling languages such as VDM, Z, and Alloy.
- Since the rich data structures of these modeling languages already preclude by construction the occurrence of errors such as null pointers and array bound overflow, we will aim to catch errors higher on the semantic scale.

PF-ESC

- The main novelty of our approach resides in the chosen method of proof construction: first-order proof obligations are subject to the **PF-transform** [6] before they are reasoned about.
- This transformation eliminates quantifiers and bound variables and reduces complex formulas to algebraic relational expressions which are more agile to calculate with.
- Suitable relational encoding of recursive structures makes it possible to perform **non-inductive** proofs over such structures.

Case study 1 — Lists with no duplicates

Consider the data type of lists with no duplicates:

$$NRList\ X = X^\star$$
$$\textbf{inv}\ \ l \triangleq \mathit{length}\ l = \mathrm{card}\ \mathit{elems}\ l$$

Clearly, the operation which adds an element to the front of a list requires a **pre-condition** for the invariant to be preserved:

$$\mathit{add} : X \to \mathit{NRList}\ X \to \mathit{NRList}\ X$$
$$\mathit{add}\ x\ l \triangleq a : l$$
$$\textbf{pre}\ \ \ldots$$

We can easily guess such pre-condition. But,

- how can we be sure that our guess is the **weakest**?
- more generally, can we **calculate** it instead of inventing and checking?

Case study 2 — mobile phone directory

Requirements fragment:

> *(...) For each list of calls stored in the mobile phone (eg. numbers dialed, SMS messages, lost calls), the store operation should work in a way such that (a) the more recently a call is made the more accessible it is; (b) no number appears twice in a list; (c) only the last 10 entries in each list are stored.*

Functional model:

$$store\ c \quad \triangleq \quad (take\ 10) \cdot (c :) \cdot filter(c \neq) \tag{1}$$

where *take* and *filter* are the obvious functions. The question is:

> *Does store ensure requirements (a)-(c)?*

A (triple) proof obligation is left pending.

Case study 3 — Hashing

Hash tables: given hash function $X \xrightarrow{hash} \mathbb{N}$, define

$$HTable\ X = \mathbb{N} \rightharpoonup \mathcal{P}X$$

$$\textbf{inv}\ \ HT \triangleq \left\langle \begin{array}{c} \forall\ k\ : \\ k \in dom\ HT : \\ HT\ k \neq \{\}\ \wedge \langle \forall\ d\ :\ d \in HT\ k :\ hash\ d = k \rangle \end{array} \right\rangle$$

as well as **representation** function

$$repf : \mathcal{P}X \to HTable\ X$$

$$repf\ S \triangleq \{hash\ x \mapsto \{d \in S : hash\ d = hash\ x\} : x \in S\}$$

Case stdy 3 — Hashing

Questions:

- Are we sure that *repf* builds a proper mapping (**simple** relation) from hash indices to sets of collisions?
- Are we sure that *repf* builds a mapping satisfying the **invariant** on hash tables?

Two **proof obligations** are thus left pending:

$$\langle \forall\ S\ :\ S \subseteq X :\ \mathit{repf}\ S \text{ is simple} \rangle \tag{2}$$

$$\langle \forall\ X, S\ :\ S \subseteq X :\ \text{inv-}(\mathit{HTable}\ X)(\mathit{repf}\ S) \rangle \tag{3}$$

The ESC approach invites us to discharge these obligations at compile-time.

Calculating Invariants and Preconditions

- Wherever a function f does not ensure preservation of invariant inv, there is always a **pre-condition** pre which enforces this at the cost of *partializing* f.
- In the limit, pre is the everywhere false predicate.
- As a rule, the average programmer will become aware of such a pre-condition at runtime, in the **testing** phase.
- One can find it much earlier, at specification time, when trying to discharge the standard proof obligation

$$\langle \forall\ a\ ::\ inv\ a \Rightarrow inv(f\ a) \rangle \tag{4}$$

which then gets extended to

$$\langle \forall\ a\ :\ pre\ a :\ inv\ a \Rightarrow inv(f\ a) \rangle \tag{5}$$

PF-ESC instead of invent & verify

However,

- We seem to be bound to **invent** *pre* and hope we've guessed the **weakest** such pre-condition. Otherwise, future use of this function will be spuriously constrained.
- Can we be sure of having hit the weakest pre-condition?

Our approach (**PF-ESC**) will be as follows:

- We take the PF-transform of $inv(f\ a)$ in (5) — at data level — and attempt to rewrite it to a term involving $inv\ a$ and possibly "something else": the **calculated** pre-condition
- This will be the weakest provided the calculation stays within equivalence steps (as shown in the next slides).

Weakest pre-conditions

- Let us strengthen (5) to equivalence

$$\langle \forall\ a\ ::\ (pre\ a) \wedge (inv\ a) \equiv inv(f\ a) \rangle \tag{6}$$

  which PF-transforms to equality

$$Pre \cdot Inv = \top \cdot Inv \cdot f \tag{7}$$

  for $Pre = \lceil pre \rceil$ and $Inv = \lceil inv \rceil$.

- Below we show that (7) ensures *pre* as the weakest (up to logical equivalence) pre-condition for *inv* to be preserved
- The calculation proceeds by **indirect equality** (31) over coreflexive X:

Weakest pre-conditions

$$
\begin{array}{ll}
 & X \subseteq Pre \cdot Inv \\
\equiv & \quad \{ \ (7) \ \} \\
 & X \subseteq \top \cdot Inv \cdot f \\
\equiv & \quad \{ \ \text{shunting (28) ; converses} \ \} \\
 & f \cdot X \subseteq Inv \cdot \top \\
\equiv & \quad \{ \ \text{range (30)} \ \} \\
 & \rho\,(f \cdot X) \subseteq Inv \\
\equiv & \quad \{ \ \text{weakest pre-condition (25)} \ \} \\
 & X \subseteq f \backslash\!\bullet\, Inv \\
:: & \quad \{ \ \text{indirection (31)} \ \} \\
 & Pre \cdot Inv = f \backslash\!\bullet\, Inv
\end{array}
$$

Case study 1: PF-ESC at work

We want to calculate the WP for

$$add\ x\ l \triangleq a : l$$

to preserve the **no duplicates** invariant on finite lists.

- First step: PF-transform $X^\star$ to $\mathbb{N} \rightharpoonup X$ (**simple** relation telling which elements take which position in list).

  Then the **no duplicates** invariant on L is encoded as $\ker L \subseteq id$ (L is injective)

  Finally, $add\ x\ L$ PF-transforms to

$$\underline{x} \cdot \underline{1}^\circ \cup L \cdot succ^\circ \tag{8}$$

  cf. back to points: $\{1 \mapsto x\} \cup \{i + 1 \mapsto (L\ i) : i \leftarrow \delta\ L\}$.

Case study 1: PF-ESC at work

- Second step: we start from the right hand side $inv(add\ x\ L)$ of (6) and re-write it by successive equivalence steps until we reach:
  - condition $inv\ l$ ...
  - ... "plus something else" — the calculated weakest pre-condition.
- Since the PF-transformed proof has to do with injectivity of union of relations, the following fact

$$R \cup S \text{ is injective } \equiv$$
$$R \text{ is injective} \wedge S \text{ is injective} \wedge R^{\circ} \cdot S \subseteq id \qquad (9)$$

(easy to prove) is likely to be of use.

Case study 1: PF-ESC at work

$$
\begin{array}{ll}
 & \mathit{add}\ x\ L \text{ has no duplicates} \\
\equiv & \quad \{\ \text{cf. (8) etc}\ \} \\
 & \underline{x} \cdot \underline{1}^{\circ} \cup L \cdot \mathit{succ}^{\circ} \text{ is injective} \\
\equiv & \quad \{\ (9)\ \} \\
 & \underline{x} \cdot \underline{1}^{\circ} \text{ is injective } \wedge\ L \cdot \mathit{succ}^{\circ} \text{ is injective } \wedge\ (\underline{x} \cdot \underline{1}^{\circ})^{\circ} \cdot L \cdot \mathit{succ}^{\circ} \subseteq \mathit{id} \\
\equiv & \quad \{\ \text{definition of injective (twice) ; shunting (28)}\ \} \\
 & \underline{1} \cdot \underline{x}^{\circ} \cdot \underline{x} \cdot \underline{1}^{\circ} \subseteq \mathit{id} \wedge \mathit{suc} \cdot L^{\circ} \cdot L \cdot \mathit{succ}^{\circ} \subseteq \mathit{id} \wedge \underline{x}^{\circ} \cdot L \subseteq \underline{1}^{\circ} \cdot \mathit{succ} \\
\equiv & \quad \{\ \text{shunt (28 , 29) as much as possible}\ \} \\
 & \underline{x}^{\circ} \cdot \underline{x} \subseteq \underline{1}^{\circ} \cdot \underline{1} \wedge L^{\circ} \cdot L \subseteq \mathit{succ}^{\circ} \cdot \mathit{succ} \wedge \underline{x}^{\circ} \cdot L \subseteq \underline{1}^{\circ} \cdot \mathit{succ} \\
\equiv & \quad \{\ \text{kernel of constant function is } \top\text{; } \mathit{succ} \text{ is an injection}\ \} \\
 & \textsc{True} \wedge L^{\circ} \cdot L \subseteq \mathit{id} \wedge \underline{x}^{\circ} \cdot L \subseteq \underline{1}^{\circ} \cdot \mathit{succ}
\end{array}
$$

Case study 1: summary

We have thus calculated:

$$\textit{add}\ x\ L \text{ has no duplicates} \quad \equiv \quad \underbrace{L \text{ is injective}}_{\text{no duplicates in } L} \quad \wedge \quad \underbrace{\underline{x}^{\circ} \cdot L \subseteq \underline{1}^{\circ} \cdot \textit{succ}}_{\text{WP}}$$

PW-expansion of the calculated WP:

$$\begin{array}{ll} & \underline{x}^{\circ} \cdot L \subseteq \underline{1}^{\circ} \cdot \textit{succ} \\ \equiv & \quad \{\ \text{go pointwise: (33) twice}\ \} \\ & \langle \forall\ n\ ::\ x\ L\ n \Rightarrow 1 = 1 + n \rangle \\ \equiv & \quad \{\ L \text{ models list } l\ \} \\ & \langle \forall\ n\ :\ n \in \text{inds}\, l :\ x = (l\ n) \Rightarrow 1 = 1 + n \rangle \\ \equiv & \quad \{\ 1 = 1 + n \text{ always false } (n \in \mathbb{N})\ \} \\ & \langle \forall\ n\ :\ n \in \text{inds}\, l :\ (l\ n) \neq x \rangle \end{array}$$

Case study 2: PF-ESC at work

From the mobile phone directory problem we select preservation of the no duplicates invariant by function

$$store\ c \quad \triangleq \quad (take\ 10) \cdot (c :) \cdot filter(c \neq)$$

Remarks:

- It's sufficient to show that $(c :) \cdot filter(c \neq)$ preserves injectivity, since $take\ n\ L \subseteq L$ $(\forall n)$ and smaller than injective is injective
- Defined over PF-transformed lists, *filter* becomes

$$filter(c \neq)L \quad \triangleq \quad (\neg\rho\, \underline{c}) \cdot L \tag{10}$$

where the negated range operator $(\neg\rho)$ satisfies property

$$\Phi \subseteq \neg\rho\, R \quad \equiv \quad \Phi \cdot R \subseteq \bot \tag{11}$$

Case study 2: PF-ESC at work

$$
\begin{array}{ll}
 & c : (\mathit{filter}(c \neq) L) \text{ is injective} \\
\equiv & \quad \{ \text{ case study 1, (10) } \} \\
 & (\neg\rho\, \underline{c}) \cdot L \text{ is injective} \wedge \underline{c}^{\circ} \cdot (\neg\rho\, \underline{c}) \cdot L \subseteq \underline{1}^{\circ} \cdot \mathit{succ} \\
\Leftarrow & \quad \{ \text{ smaller than injective is injective } \} \\
 & L \text{ is injective} \wedge \underline{c}^{\circ} \cdot (\neg\rho\, \underline{c}) \cdot L \subseteq \underline{1}^{\circ} \cdot \mathit{succ} \\
\equiv & \quad \{ \text{ converses } \} \\
 & L \text{ is injective} \wedge L^{\circ} \cdot (\neg\rho\, \underline{c}) \cdot \underline{c} \subseteq \mathit{succ}^{\circ} \cdot \underline{1} \\
\equiv & \quad \{ \ (\neg\rho\, \underline{c}) \cdot \underline{c} = \bot \text{ by left-cancellation of (11) } \} \\
 & L \text{ is injective} \wedge L^{\circ} \cdot \bot \subseteq \mathit{succ}^{\circ} \cdot \underline{1} \\
\equiv & \quad \{ \text{ bottom is below anything } \} \\
 & L \text{ is injective} \wedge \textsc{True}
\end{array}
$$

Case study 2: PF-ESC at work

Moral of this case study:

> *Although the implication in the second step of the reasoning could put weakness of calculated pre-condition at risk, we've calculated the weakest of all conditions anyway (*TRUE*).*

Case study 3: PF-ESC at work

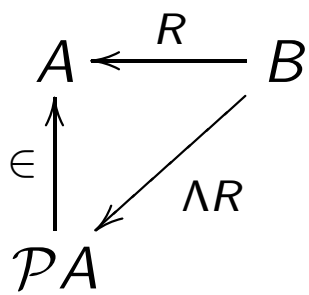
Recall that we want to make sure that

$$repf\ S \triangleq \{hash\ x \mapsto \{d \in S : hash\ d = hash\ x\} : x \in S\}$$

is properly typed, that is, that its result is a valid mapping (**simple** relation). Our reasoning will be based on the PF-transform of *repf*,

$$\lceil repf\ S \rceil = \Lambda(\lceil S \rceil \cdot \ker hash) \cdot \lceil S \rceil \cdot hash^{\circ} \qquad (12)$$

where Λ is the power-transpose operator (34) which captures the view of relations as "set-valued" functions



and $\lceil S \rceil$ is the coreflexive which transforms set S.

(abbreviating *hash* to *h*):

$$
\begin{array}{ll}
 & \lceil repf\ S \rceil \text{ is simple} \\
\equiv & \quad \{\ (12),\ (35)\ \} \\
 & \Lambda(\lceil S \rceil \cdot h^\circ) \cdot h \cdot \lceil S \rceil \cdot h^\circ \cdot h \cdot \lceil S \rceil \cdot h^\circ \cdot \Lambda(\lceil S \rceil \cdot h^\circ)^\circ \subseteq id \\
\equiv & \quad \{\ \text{shunting (29), since } \Lambda(\lceil S \rceil \cdot h^\circ) \text{ is a function}\ \} \\
 & \Lambda(\lceil S \rceil \cdot h^\circ) \cdot h \cdot \lceil S \rceil \cdot h^\circ \cdot h \cdot \lceil S \rceil \subseteq \Lambda(\lceil S \rceil \cdot h^\circ) \cdot h \\
\Leftarrow & \quad \{\ \text{monotonicity of composition}\ \} \\
 & h \cdot \lceil S \rceil \cdot h^\circ \cdot h \cdot \lceil S \rceil \subseteq h \\
\equiv & \quad \{\ \text{shunting; kernels}\ \} \\
 & \ker(h \cdot \lceil S \rceil) \subseteq \ker h \\
\Leftarrow & \quad \{\ \text{ker is monotonic}\ \} \\
 & h \cdot \lceil S \rceil \subseteq h \\
\equiv & \quad \{\ \lceil S \rceil \text{ is coreflexive}\ \}
\end{array}
$$

Generalizing PF-ESC

Back to where we started, let us PF-transform (5):

$$Pre \cdot Inv \subseteq \top \cdot Inv \cdot f \tag{13}$$

If we generalize f above to a (non-functional) post-condition $post$, and Inv to different invariants over source and target types, we obtain

$$Pre \cdot Inv \subseteq \top \cdot Inv' \cdot Post \tag{14}$$

which expands to

$$\langle \forall\ a\ :\ inv\ b :\ pre\ a \Rightarrow \langle \exists\ b\ :\ inv'\ b :\ post(b, a) \rangle \rangle \tag{15}$$

and can be recognized as the **satisfiability** requirement on **pre** / **post** specification pairs [5].

Relationship with Hoare Logic

We finally show that **Hoare triples** such as

$$\{p\}P\{q\} \tag{16}$$

are also instances of ESC proof obligations. First we spell out the meaning of (16):

$$\langle \forall\ s\ :\ p\ s :\ \langle \forall\ s'\ :\ s \xrightarrow{P} s'\ :\ q\ s' \rangle \rangle \tag{17}$$

Then (recording the meaning of program P as relation $[\![P]\!]$ on program states) we PF-transform (17) into

$$\lceil p \rceil \subseteq [\![P]\!] \setminus (\lceil q \rceil \cdot \top) \tag{18}$$

thanks to (27), and then to...

Relationship with Hoare Logic

$$[\![P]\!] \cdot \lceil p \rceil \subseteq \lceil q \rceil \cdot \top \tag{19}$$

thanks to (26). By putting (19) and the PF-transform of (4) aside,

$$f \cdot \lceil inv \rceil \subseteq \lceil inv \rceil \cdot \top \tag{20}$$

we realize both share the same scheme,

$$R \cdot \Phi \subseteq \Psi \cdot \top \tag{21}$$

which is equivalent to

$$R \cdot \Phi \subseteq \Psi \cdot R \tag{22}$$

(see exercise 1) and which one can condense into notation

$$\Phi \xrightarrow{R} \Psi \tag{23}$$

Relationship with Hoare Logic

All in all

- Notation (23) can be regarded as the **type assertion** that, if fed with values (or starting on states) "of type Φ" computation P yields results (changes to states) "of type Ψ" (if it terminates).
- We see that functional invariant preservation and Hoare Logic are one and the same device: a way to **type** computations, be them specified as (allways terminating, deterministic) functions or encoded into (possibly non-terminating, non-deterministic) programs.

Exercises

Exercise 1: Complete the following reasoning which shows that (21) and (22) are equivalent:

$$
\begin{array}{ll}
 & R \cdot \Phi \subseteq \Psi \cdot \top \\
\equiv & \quad \{ \ \ldots\ldots\ldots \ \} \\
 & R \cdot \Phi \subseteq \Psi \cdot \top \cap R \\
\equiv & \quad \{ \ \ldots\ldots\ldots \ \} \\
 & R \cdot \Phi \subseteq \Psi \cdot R
\end{array}
$$

□

Exercise 2: Show that

$$\Phi \xrightarrow{R} \Psi \quad \equiv \quad \Phi \subseteq R \backslash\!\bullet \Psi \tag{24}$$

holds.

□

Background

Weakest (liberal) pre-condition is an upper adjoint [2]

$$\rho\,(R \cdot \Phi) \subseteq \Psi \quad \equiv \quad \Phi \subseteq R \backslash\!\bullet \Psi \tag{25}$$

whose pointwise version $wlp\ R\ \psi$ is:

$$wlp\ R\ \psi \quad \triangleq \quad \langle \bigvee\ \phi\ ::\ \langle \forall\ b, a\ :\ b\ R\ a :\ \phi\ a \Rightarrow \psi\ b \rangle \rangle$$

"Left" division:

$$R \cdot X \subseteq S \quad \equiv \quad X \subseteq R \setminus S \tag{26}$$

where

$$b\ (R \setminus S)\ a \quad \equiv \quad \langle \forall\ c\ :\ c\ R\ b :\ c\ S\ a \rangle \tag{27}$$

Background

Shunting rules:

$$f \cdot R \subseteq S \quad \equiv \quad R \subseteq f^{\circ} \cdot S \tag{28}$$

$$R \cdot f^{\circ} \subseteq S \quad \equiv \quad R \subseteq S \cdot f \tag{29}$$

Range:

$$\rho\, R \subseteq \Phi \quad \equiv \quad R \subseteq \Phi \cdot \top \tag{30}$$

Indirect equality rule:

$$R = S \quad \equiv \quad \langle \forall\ X\ ::\ (X \subseteq R \equiv X \subseteq S) \rangle \tag{31}$$

$$\equiv \quad \langle \forall\ X\ ::\ (R \subseteq X \equiv S \subseteq X) \rangle \tag{32}$$

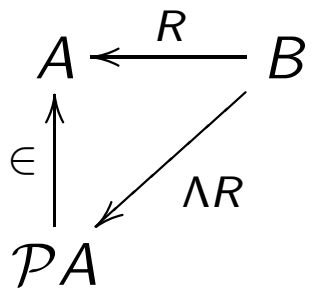
"Guardanapo" rule:

$$b(f^{\circ} \cdot R \cdot g)a \quad \equiv \quad (f\ b)R(g\ a) \tag{33}$$

Background

Power-transpose:

$$f = \Lambda R \quad \equiv \quad R = \in \cdot f \tag{34}$$



Fusion:

$$\Lambda(R \cdot f) \quad = \quad (\Lambda R) \cdot f \tag{35}$$

Property (11) stems from Galois connection [1]

$$\Phi \subseteq \neg\delta\, R \quad \equiv \quad R \subseteq \bot/\Phi \tag{36}$$

[1] This is fact (15.1) of [1], page 288.


C. Aarts, R.C. Backhouse, P. Hoogendijk, E.Voermans, and J. van der Woude.
A relational theory of datatypes, December 1992.
Available from www.cs.nott.ac.uk/~rcb.


R.C. Backhouse.
Fixed point calculus, 2000.
Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Lincoln College, Oxford, UK 10th to 14th April 2000.


David Detlefs, Greg Nelson, and James B. Saxe.
Simplify: a theorem prover for program checking.
JACM, 52(3):365–473, 2005.


Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata.
Extended static checking for java.
In *PLDI*, pages 234–245, 2002.


C.B. Jones.

Systematic Software Development Using VDM.
Series in Computer Science. Prentice-Hall International, 1986.

J.N. Oliveira and C.J. Rodrigues.
Pointfree factorization of operation refinement.
In *FM'06* , volume 4085 of *LNCS*, pages 236–251.
Springer-Verlag, 2006.