

Data type invariants — starting where (static) type checking stops

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI/UM, 2007 (Updated 2008; 2009; 2010)

Types for software quality

Data type evolution:

- **Assembly** (1950s) — one single primitive data type: machine binary
- **Fortran** (1960s) — primitive types for numeric processing (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL data types)
- **Pascal** (1970s) — user defined (monomorphic) data types (eg. records, files)
- **ML, Haskell** etc (\geq 1980s) — user defined (polymorphic) data types (eg. *List a* for all *a*)

Type checking for software quality

Why data types?

- **Fortran** anecdote: non-terminating loop `DO I = 1.10` once went unnoticed due to poor type-checking
- Diagnosis: compiler unable to prevent using a real number where a discrete value (eg. integer, enumerated type) was expected
- Solution: improve grammar + static type checker

(static means *done at compile time*)

Data type invariants

In a system for monitoring the flight paths of aircrafts in a controlled airspace, we need to define altitude, latitude and longitude:

$$Alt = \mathbb{R}$$

$$Lat = \mathbb{R}$$

$$Lon = \mathbb{R}$$

However,

- altitude cannot be negative
- latitude ranges between -90 and 90
- longitude ranges between -180 and 180

In maths we would have defined:

$$Alt = \{a \in \mathbb{R} : a \geq 0\}$$

$$Lat = \{x \in \mathbb{R} : -90 \leq x \leq 90\}$$

$$Lon = \{y \in \mathbb{R} : -180 \leq y \leq 180\}$$

Data type invariants

In a system for monitoring the flight paths of aircrafts in a controlled airspace, we need to define altitude, latitude and longitude:

$$Alt = \mathbb{R}$$

$$Lat = \mathbb{R}$$

$$Lon = \mathbb{R}$$

However,

- altitude cannot be negative
- latitude ranges between -90 and 90
- longitude ranges between -180 and 180

In maths we would have defined:

$$Alt = \{a \in \mathbb{R} : a \geq 0\}$$

$$Lat = \{x \in \mathbb{R} : -90 \leq x \leq 90\}$$

$$Lon = \{y \in \mathbb{R} : -180 \leq y \leq 180\}$$

Data type invariants “a la” VDM

Standard notation (VDM family)

$$Alt = \mathbb{R}$$

$$\mathbf{inv} \ a \triangleq a \geq 0$$

implicitly defines predicate

$$inv\text{-}Alt : \mathbb{R} \rightarrow \mathbb{B}$$

$$inv\text{-}Alt(a) \triangleq a \geq 0$$

known as the *invariant* property of Alt .

Data Type invariants

Recall the following requirements from mobile phone manufacturer

*(...) 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) each list stores up to 10 entries.*

Clause (c) leads to

$$\text{ListOfCalls} = \text{Call}^*$$
$$\text{inv } l \triangleq \text{length } l \leq 10$$

Exercise 1: Think of a natural language definition of clause (b) to *inv-ListOfCalls* involving denotation $l\ i$ of the i -th element of l , for $1 \leq i \leq \text{length } l$.

□

Invariants are *inevitable*

Modeling the Western dating system:

$$\textit{Year} = \mathbb{N}$$

$$\textit{Month} = \mathbb{N}$$

$$\textit{inv } m \triangleq m \leq 12$$

$$\textit{Day} = \mathbb{N}$$

$$\textit{inv } d \triangleq d \leq 31$$

$$\textit{Date} = \textit{Year} \times \textit{Month} \times \textit{Day}$$

However, $12 \times 31 = 372$, while one year has 365.2425... days.
Thus the *Julian calendar* (45 BC, which introduced *leap years*) and the much more complex *Gregorian calendar* (1582), which fine tuned it to

Invariants are *inevitable*

$Date = Year \times Month \times Day$

$\mathbf{inv}(y, m, d) \triangleq$ if $m \in \{1, 3, 5, 7, 8, 10, 12\}$ then
 $d \leq 31 \wedge$
 $((y = 1582 \wedge m = 10) \Rightarrow (d < 5 \vee 14 < d))$
 else if $m \in \{4, 6, 9, 11\}$ then $d \leq 30$
 else if $m = 2 \wedge \mathit{leapYear}(y)$ then $d \leq 29$
 else if $m = 2 \wedge \neg \mathit{leapYear}(y)$ then $d \leq 28$
 else FALSE;

where

$\mathit{leapYear} \quad : \quad \mathbb{N} \rightarrow \mathbb{B}$

$\mathit{leapYear} \ y \triangleq 0 = \mathit{rem}(y, 4) \wedge (y \geq 1700 \wedge \mathit{rem}(y, 100) = 0$
 then 400 else 4)

Invariants are *inevitable*

Real-life conventions, laws, rules, norms, acts lead to invariants,
eg. **RIAPA** (U.Minho internal students' course follow-up rules):

DbSAUM = ...

inv *db* \triangleq (a) */*student's current degree course must exist */*
 (b) */*student's current plan must belong to degree course */*
 (c) */*student' past registrations obey to constraint (b) */*
 (d) */*students cannot do exams of courses they are not regist*
 (e) */*student is registered in one degree course only in the bac*
 (f) */*courses in all academic years must belong to degree plan*
 (g) */*same as (f) concerning every student */*
 (...) */*..... etc etc */*

Summing up

- Given a datatype A and a predicate $p : A \rightarrow \mathbb{B}$, data type declaration

$$B = A$$
$$\mathbf{inv} \ x \triangleq \ p \ x$$

means the type whose extension is

$$B = \{x \in A : p \ x\}$$

- p is referred to as the invariant property of B
- Therefore, writing $a \in B$ means $a \in A \wedge (p \ a)$.

How does one write invariants?

We resort to first order predicate logic and set theory, which you have studied in your 1st degree. Warming up:

Exercise 2: (adapted from exercise 5.1.4 in C.B. Jones's *Systematic Software Development Using VDM*):

Hotel room numbers are pairs (l, r) where l indicates a floor and r a door number in floor l . Write the invariant on room numbers which captures the following rules valid in a particular hotel with 25 floors, 60 rooms per floor:

- 1. there is no floor number 13; (guess why)*
- 2. level 1 is an open area and has no rooms;*
- 3. the top five floors consist of large suites and these are numbered with even integers.*



Quantifier notation

Most invariants require quantified expressions. Here is how we write them:

- $\langle \forall k : R : T \rangle$ meaning “for all k in range R it is the case that T ”
- $\langle \exists k : R : T \rangle$ meaning “there exists k in range R case such that T ”

Exercise 3: Write clause (b) of *inv-ListOfCalls* (recall exercise 1) using \forall notation.

□

Invariant preservation

Proposed model for operation *store* in the mobile phone problem,

$$\begin{aligned} \textit{store} &: \textit{Call} \rightarrow \textit{ListOfCalls} \rightarrow \textit{ListOfCalls} \\ \textit{store } c \ l &\triangleq \textit{take } 10 \ (c : [a \mid a \leftarrow l, a \neq c]) \end{aligned}$$

The fact that *ListOfCalls* has an invariant property

$$\begin{aligned} \textit{ListOfCalls} &= \textit{Call}^* \\ \mathbf{inv } \ l &\triangleq \textit{length } l \leq 10 \wedge \\ &\langle \forall i, j : 1 \leq i, j \leq \textit{length } l : (l \ i) = (l \ j) \Rightarrow i = j \rangle \end{aligned}$$

leads to **proof obligation**

$$\langle \forall c, l : l \in \textit{ListOfCalls} : (\textit{store } c \ l) \in \textit{ListOfCalls} \rangle \quad (1)$$

Invariant preservation (functions)

In general, given a function $A \xrightarrow{f} B$ where both A and B have invariants, extended **type checking** requires the following

Proof obligation

f should be invariant-preserving, that is,

$$\langle \forall a : a \in A : (f a) \in B \rangle \quad (2)$$

equivalent to

$$\langle \forall a : \text{inv-}A a : \text{inv-}B(f a) \rangle \quad (3)$$

holds.

(Our example above is a special case of this, for $A = B$.)

Dealing with proof obligations

- The essence of formal methods consists in regarding conjectures such as (2) as **proof obligations** (aka **verification conditions**) which, once discharged, add quality and confidence to the design
- In lightweight approaches, one regards (2) as the subject of as many **test cases** as possible, either using smart testing techniques or **model checking** techniques.
- These techniques, however, only prove the existence of **counter-examples** — not their absence:

test unveils errors \Rightarrow program has errors $(p \Rightarrow q)$
test unveils no errors $\not\Rightarrow$ program has no errors $(\neg p \not\Rightarrow \neg q)$

Dealing with proof obligations

- In full-fledged formal techniques, one is obliged to provide a **mathematical proof** that conjectures such as (2) do hold for **any** *a*.
- Such proofs can either be performed as paper-and-pencil exercises or, in case of very complex invariants, be supported by **theorem provers**
- If automatic, discharging such proofs can be regarded as **extended static checking** (ESC)
- As we shall see, *all* the above approaches to adding quality to a formal model are useful and have their place in software engineering using formal methods.

The Verifying Compiler GC

Quoting Hoare (2003):

(...) I revive an old challenge: the construction and application of a verifying compiler that guarantees correctness of a program before running it. (...)

Still Hoare (2003):

A verifying compiler [should use] automated mathematical and logical reasoning methods to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations that are associated with the code of the program, often inferred automatically, and increasingly often supplied by the original programmer.

Background — Eindhoven quantifier calculus

When writing \forall, \exists -quantified expressions is useful to know a number of rules which help in reasoning about them. Below we list some of these rules ¹:

- **Trading:**

$$\langle \forall i : R \wedge S : T \rangle = \langle \forall i : R : S \Rightarrow T \rangle \quad (4)$$

$$\langle \exists i : R \wedge S : T \rangle = \langle \exists i : R : S \wedge T \rangle \quad (5)$$

Exercise 4: Check rule

$$\langle \exists i : R : T \rangle = \langle \exists i : T : R \rangle \quad (6)$$

□

¹Warning: the application of a rule is invalid if (a) it results in the capture of free variables or release of bound variables; (b) a variable ends up occurring more than once in a list of dummies.

Background — Eindhoven quantifier calculus

Splitting:

$$\langle \forall j : R : \langle \forall k : S : T \rangle \rangle = \langle \forall k : \langle \exists j : R : S \rangle : T \rangle \quad (7)$$

$$\langle \exists j : R : \langle \exists k : S : T \rangle \rangle = \langle \exists k : \langle \exists j : R : S \rangle : T \rangle \quad (8)$$

One-point:

$$\langle \forall k : k = e : T \rangle = T[k := e] \quad (9)$$

$$\langle \exists k : k = e : T \rangle = T[k := e] \quad (10)$$

Nesting:

$$\langle \forall a, b : R \wedge S : T \rangle = \langle \forall a : R : \langle \forall b : S : T \rangle \rangle \quad (11)$$

$$\langle \exists a, b : R \wedge S : T \rangle = \langle \exists a : R : \langle \exists b : S : T \rangle \rangle \quad (12)$$

Background — set-theoretical membership

Above we have seen the important rôle of membership (\in) tests in (formal) type checking. How do we characterize \in ?

- given a set S , let $(\in S)$ denote the predicate such that $(\in S)a \stackrel{\text{def}}{=} a \in S$
- the following universal property holds, for all S, p :

$$p = (\in S) \Leftrightarrow S = \{a : p a\} \quad (13)$$

Exercises

Exercise 5: Infer tautologies

$$S = \{a : a \in S\} \quad , \quad p a \Leftrightarrow a \in \{a : p a\}$$

from (13).

□

Exercise 6: Check **carefully** which rules of the quantifier calculus need to be applied to prove that predicate

$$\langle \forall b, a : \langle \exists c : b = f c : r(c, a) \rangle : s(b, a) \rangle \quad (14)$$

is the same as

$$\langle \forall c, a : r(c, a) : s(f c, a) \rangle$$

where f is a function and r , s are binary predicates.

□

References

C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. In Görel Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 2003. ISBN 3-540-00904-3.