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

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI/UM, 2007 (Updated 2008-10; 2012)

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:

$$\begin{aligned} Alt &= \mathbb{R} \\ Lat &= \mathbb{R} \\ Lon &= \mathbb{R} \end{aligned}$$

However,

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

In maths we would have defined:

$$\begin{aligned} Alt &= \{a \in \mathbb{R} : a \geq 0\} \\ Lat &= \{x \in \mathbb{R} : -90 \leq x \leq 90\} \\ Lon &= \{y \in \mathbb{R} : -180 \leq y \leq 180\} \end{aligned}$$

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:

$$\begin{aligned} Alt &= \mathbb{R} \\ Lat &= \mathbb{R} \\ Lon &= \mathbb{R} \end{aligned}$$

However,

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

In maths we would have defined:

$$\begin{aligned} Alt &= \{a \in \mathbb{R} : a \geq 0\} \\ Lat &= \{x \in \mathbb{R} : -90 \leq x \leq 90\} \\ Lon &= \{y \in \mathbb{R} : -180 \leq y \leq 180\} \end{aligned}$$

Data type invariants "a la" VDM

Standard notation (VDM [1] family)

$$Alt = \mathbb{R}$$
$$\textbf{inv } a \triangleq a \geq 0$$

implicitly defines predicate

$$inv\text{-}Alt : \mathbb{R} \to \mathbb{B}$$
$$inv\text{-}Alt(a) \triangleq a \geq 0$$

known as the **invariant** property of Alt.

[1] VDM=Vienna Development Method, one of the earliest formal methods developed in IBM Vienna in the 1970s.

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

$$ListOfCalls = Call^\star$$
$$\textbf{inv}\ l \triangleq length\ l \leq 10$$

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 $1 \leq i \leq length\ l$.

□

Invariants are *inevitable*

Case study — why the Western dating system?

> *The sidereal year — absolute time the Earth takes to complete one revolution around the Sun — is not a multiple of the solar day — time the Earth takes to spin 360 degrees over itself.*

If you divide the first by the second of these times (to know how many days are there in one solar year) you get an infinite number, something like

$$365.24219879...$$

You are bound to *round* this number. 365 days being too coarse, 365.25 is a convenient approximation since

$$365\ \mathit{days} + 0.25 \times 24\ \mathit{hours} \quad = \quad 365\ \mathit{days} + 6\ \mathit{hours}$$

— still losing around $-0.00780121...$ days per year.

Problem — how to record time

This reasoning gave birth to the calender adopted in the Roman Empire, in which the 365 days were packaged as follows:

Martius	31
Aprilis	30
Maius	31
Iunius	30
Quintilis	31
Sextilis	31
September	30
October	31
November	30
December	31
Ianuarius	31
Februarius	28
	365

What about the 6 hours? And the $-0.00780121...$ error (=+-11 mins per year)?

Problem — how to record time

The problem was handled as the following text shows:

> '**Do Ano e Sua Divisão** — *(...) Júlio César instituiu o ano, de que hoje usamos, de 365 dias e 6 horas, a qual quantidade não é exacta, pois vemos claramente adiantar-se o tempo; (...) a Santa Madre Igreja usa do ano que instituiu Júlio César, tomando em cada ano as 6 horas, que formam um dia inteiro em cada quatro anos, chamando-se* **bissexto** *a esse ano, a que se acrescenta um dia (...).*

(Extracted from *Lunário de Prognóstico Perpétuo, para Todos os Reinos e Províncias*, por Jerónimo Cortez, Valenciano (18c), re-edited by Lello & Irmão, 1910.)

Problem — how to record time

How does one specify that a given date is **valid** according to Julius Cæsar's calender?

First we define the types (already with invariants):

$$Year = \mathbb{N}$$

$$Month = \mathbb{N}$$
$$\textbf{inv}\ m \triangleq\ m \leq 12$$

$$Day = \mathbb{N}$$
$$\textbf{inv}\ d \triangleq\ d \leq 31$$

$$Date = Year \times Month \times Day$$

Problem — how to record time

Next we constrain type *Date* by assigning days to months:

$$
\begin{array}{lll}
Date = Year \times Month \times Day & & \\
\textbf{inv}(y, m, d) \triangleq & \textit{if } m \in \{1,3,5,7,8,10,12\} & \textit{then } d \leq 31 \\
& \textit{else if } m \in \{4,6,9,11\} & \textit{then } d \leq 30 \\
& \textit{else if } m = 2 \wedge \textit{leapYear}(y) & \textit{then } d \leq 29 \\
& \textit{else if } m = 2 \wedge \neg \textit{leapYear}(y) & \textit{then } d \leq 28 \\
& \textit{else } \textsc{False}; &
\end{array}
$$

where

$$
\begin{array}{lll}
\textit{leapYear} & : & Year \to \mathbb{B} \\
\textit{leapYear } y & \triangleq & \textit{rem}(y, 4) = 0
\end{array}
$$

Problem — how to record time

Eventually, the inaccuracy of the round down made itself evident:

> "**Da Reforma do Calendário**
> — *Tendo-se observado, que desde a celebração do concílio de Niceia, em 325, até ao ano de 1582, se haviam antecipado os equinócios 10 dias do assento fixo em que os colocara Dionísio Romano; (...) mandou o papa Gregório XIII proceder à reforma do Calendário, em virtude da qual se determinou: (...)*

Pope Gregory XIII

Problem — how to record time

Birth of the Gregorian calender (1582):

> *(...) 1 que no mês de Outubro de 1582 se suprimissem 10 dias, contando 4 no dia de S. Francisco, e 15 no seguinte; 2 que em cada 400 anos se suprimissem 3 dias, principiando de 1700, 1800, 1900, 2100, 2200, 2300, 2500, etc (que por isso não são bissextos), para diminuir o excesso do ano sinodal ao civil, e os equinócios ficarem imóveis a 21 de Março e 23 de Setembro"*

Thus the new version of clause

$$\begin{array}{l} \mathit{if}\ \ m \in \{1,3,5,7,8,10,12\}\ \mathit{then} \\ d \leq 31 \wedge ((y = 1582 \wedge m = 10) \Rightarrow (d < 5 \vee 14 < d)) \\ \mathit{else}\ \ldots \end{array}$$

as well as a new version of

$$\begin{array}{lcl} \mathit{leapYear} & : & \mathbb{N} \to \mathbb{B} \\ \mathit{leapYear}\ y & \triangleq & 0 = \mathit{rem}(y, \mathit{if}\ \ y \geq 1700 \wedge \mathit{rem}(y,100) = 0 \\ & & \mathit{then}\ 400\ \mathit{else}\ 4) \end{array}$$

Problem — how to record time

Clearly, writing a function such as eg.

$$tomorrow : Date \to Date$$

is not as easy as before — one has to ensure that, given a *valid* date (y, m, d), $tomorrow(y, m, d)$ is also a valid date.

Exercise 2: Give a definition of $tomorrow : Date \to Date$ which ensures date validity. Suggestion: resort to the following auxiliary functions:

$sucy(y,m,d)=(y+1,1,1)$
$sucm(y,m,d)=(y,m+1,1)$
$sucd(y,m,d)=(y,m,d+1)$

□

Invariants are *inevitable*

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

$DbSAUM = \ldots$
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 registe
(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 $p : A \to \mathbb{B}$, data type declaration

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

  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 3: (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 4: Write clause (b) of inv-*ListOfCalls* (recall exercise 1) using $\forall$ notation.

□

Invariant preservation

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

$$store : Call \to ListOfCalls \to ListOfCalls$$
$$store\ c\ l \triangleq take\ 10\ (c : [\ a \mid a \leftarrow l, a \neq c\ ])$$

The fact that *ListOfCalls* has invariant properties (b)+(c),

$$ListOfCalls = Call^\star$$
$$\textbf{inv}\ l \triangleq length\ l \leq 10 \wedge$$
$$\langle \forall\ i, j\ :\ 1 \leq i, j \leq length\ l :\ (l\ i) = (l\ j) \Rightarrow i = j \rangle$$

leads to **proof obligation**

$$\langle \forall\ c, l\ :\ l \in ListOfCalls :\ (store\ c\ l) \in ListOfCalls \rangle \qquad (1)$$

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,

$$\langle \forall\ a\ :\ a \in A :\ (f\ a) \in B \rangle \tag{2}$$

equivalent to

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

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:

$$\begin{array}{rcll} \text{test unveils errors} & \Rightarrow & \text{program has errors} & (p \Rightarrow q) \\ \text{test unveils no errors} & \not\Rightarrow & \text{program has no errors} & (\neg p \not\Rightarrow \neg q) \end{array}$$

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.*

Follow up

- Verification conditions can be handled by model checkers such as eg. the **Alloy** Analyser (Jackson, 2012).
- They can also be discharged by **calculation**, using the **Algebra of Programming** (AoP) (Bird and de Moor, 1997).
- The two approaches together lead to the "Alloy Meets the AoP" (Oliveira and Ferreira, 2012) approach which will be followed in this course.

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 [2]:

- **Trading:**

$$\langle \forall\ i\ :\ R \wedge S :\ T \rangle \quad = \quad \langle \forall\ i\ :\ R :\ S \Rightarrow T \rangle \tag{4}$$

$$\langle \exists\ i\ :\ R \wedge S :\ T \rangle \quad = \quad \langle \exists\ i\ :\ R :\ S \wedge T \rangle \tag{5}$$

Exercise 5: Check rule

$$\langle \exists\ i\ :\ R :\ T \rangle \quad = \quad \langle \exists\ i\ :\ T :\ R \rangle \tag{6}$$

□

---

[2] Warning: 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:

$$\langle \forall\ j\ :\ R :\ \langle \forall\ k\ :\ S :\ T \rangle \rangle \quad = \quad \langle \forall\ k\ :\ \langle \exists\ j\ :\ R :\ S \rangle :\ T \rangle \tag{7}$$

$$\langle \exists\ j\ :\ R :\ \langle \exists\ k\ :\ S :\ T \rangle \rangle \quad = \quad \langle \exists\ k\ :\ \langle \exists\ j\ :\ R :\ S \rangle :\ T \rangle \tag{8}$$

One-point:

$$\langle \forall\ k\ :\ k = e :\ T \rangle \quad = \quad T[k := e] \tag{9}$$

$$\langle \exists\ k\ :\ k = e :\ T \rangle \quad = \quad T[k := e] \tag{10}$$

Nesting:

$$\langle \forall\ a, b\ :\ R \wedge S :\ T \rangle \quad = \quad \langle \forall\ a\ :\ R :\ \langle \forall\ b\ :\ S :\ T \rangle \rangle \tag{11}$$

$$\langle \exists\ a, b\ :\ R \wedge S :\ T \rangle \quad = \quad \langle \exists\ a\ :\ R :\ \langle \exists\ b\ :\ S :\ T \rangle \rangle \tag{12}$$

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:

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

Exercises

Exercise 6: Infer tautologies

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

from (13).

□

Exercise 7: Check **carefully** which rules of the quantifier calculus need to be applied to prove that predicate

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

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

R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.

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.

D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge Mass., 2012. Revised edition, ISBN 0-262-01715-2.

J.N. Oliveira and M.A. Ferreira. Alloy meets the algebra of programming: a case study, 2012. To appear in IEEE Transactions on Software Engineering.