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

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI/UM, 2007 (Last update: Sep 2014)

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*)
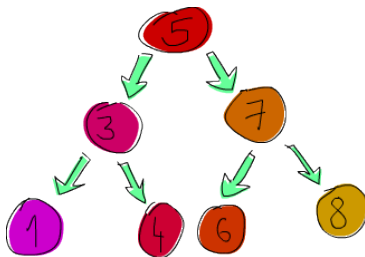
Type checking for software quality

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 aircraft flight paths 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 aircraft flight paths 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 [1] family)

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

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.

---

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

Data Type invariants

Consider the following requirements from a (toy) 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

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

Exercise 1: Let $s\ n$ denote the n-th element of a list s. Complete the following alternative formulation of clause (b) of inv-*ListOfCalls*:

Should $(l\ i)$ and $(l\ j)$ be the same, then .... for all ....

□

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

$$365 \; days + 0.25 \times 24 \; hours \;\; = \;\; 365 \; days + 6 \; hours$$

— 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 type

$$Date = Year \times Month \times Day \tag{1}$$

where

$$
\left.
\begin{array}{l}
Year = \mathbb{N} \\[2mm]
Month = \mathbb{N} \\
\textbf{inv } m \triangleq\ m \leq 12 \\[2mm]
Day = \mathbb{N} \\
\textbf{inv } d \triangleq\ d \leq 31
\end{array}
\right\} \tag{2}
$$

are auxiliary types (already with invariants).

Problem — how to record time

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

$$Date = Year \times Month \times Day$$
$$\mathbf{inv}(y, m, d) \triangleq \quad if \ m \in \{1, 3, 5, 7, 8, 10, 12\} \ then \ d \leq 31$$
$$else \ if \ m \in \{4, 6, 9, 11\} \ then \ d \leq 30$$
$$else \ if \ m = 2 \land leapYear(y) \ then \ d \leq 29$$
$$else \ if \ m = 2 \land \neg leapYear(y) \ then \ d \leq 28$$
$$else \ \text{FALSE};$$

where

$$leapYear \quad : \quad Year \rightarrow I\!B$$
$$leapYear \ y \quad \triangleq \quad rem(y, 4) = 0$$

Problem — how to record time

Eventually, the inaccuracy of the
rounding made itself evident:

Pope Gregory XIII

" **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: (...)*

Problem — how to record time

Birth of the Gregorian calender (1582):

> "(...) 1.º que no mês de Outubro de 1582 se suprimissem 10
> dias, contando 4 no dia de S. Francisco, e 15 no seguinte; 2.º
> 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

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

as well as a new version of

$$leapYear \quad : \quad \mathbb{N} \to \mathbb{B}$$
$$leapYear \ y \quad \triangleq \quad 0 = rem(y, if \ y \geq 1700 \wedge rem(y, 100) = 0$$
$$then \ 400 \ else \ 4)$$

Problem — how to record time

Clearly, writing a function such as eg.

$$tomorrow : Date \rightarrow Date$$

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: For *Date* defined solely by (1,2) above, give definitions for the auxiliary functions y, m and d of

$$tomorrow : Date \rightarrow Date$$
$$tomorrow\ x \triangleq (y\ x, m\ x, d\ x)$$

which respectively give tomorrow's year, month and day. Then *consider the effort* required by repeating the exercise while ensuring **full** date **validity** within the Gregorian calender.

☐

Invariants are *inevitable*

Real-life conventions, **laws**, rules, norms, acts — all lead to
invariants.

Example: typical students' **course follow-up** rules such as e.g.:

> $DbSAUM = \ldots$
> **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)/*unregistered students cannot do exams */
> (e)/*student is registered in one degree course only */
> (f)/*courses in all academic years belong to degree plan */
> (g)/*same as (f) concerning every student */
> $(\ldots)$/*........... etc .... etc .... */

(Can you guess how many **invariants** are hidden in the recent
RT-41/2014?)

In general

Summing up:

- Given a **data type** A and a **predicate** $p : A \to \mathbb{B}$, the new **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, by writing $a \in B$ one 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:

Exercise 3: (adapted from exercise 5.1.4 in C.B. Jones's *Systematic Software Development Using VDM*):

> *Hotel room numbers are pairs (f, d) where f indicates a floor and d a door number in floor f. 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.*

NB: assume predicate *even* on natural numbers.

□

Quantifier notation

Most invariants require **quantified** expressions. Here is how we shall write these:

- $\langle \forall\ k\ :\ R\ :\ T \rangle$ meaning *"for all k in range R it is the case that T holds"*

- $\langle \exists\ k\ :\ R\ :\ T \rangle$ meaning *"there exists at least one k in range R case such that T holds"*

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

□

How does one *ensure* invariants?

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,*

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

equivalent to

$$\langle \forall\ a\ :\ inv\text{-}A\ a :\ inv\text{-}B(f\ a) \rangle \tag{4}$$

should hold.

In the example of the next slide, $A = B$.

How does one *ensure* invariants?

Proposed model for operation *store* in the mobile phone problem has type

$$store : Call \rightarrow ListOfCalls \rightarrow ListOfCalls$$

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

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

leads to **proof obligation**

$$\langle \forall\ c, l\ :\ l \in ListOfCalls :\ (store\ c\ l) \in ListOfCalls \rangle \qquad (5)$$

How does one *ensure* invariants?

Therefore, for any candidate definition for *store*, eg.

$$store\ c\ l \triangleq\ take\ 10\ (c : [\ a\ |\ a \leftarrow l, a \neq c\ ]) \tag{6}$$

one has to **ensure** that (5) holds for **any** inputs c and l.

NB: *take n s* denotes the list of the first n elements of list s, if they exist (using Haskell terminology; in Python, for instance, one will write $s[0 : n]$ for *take n s*). Notation $a : x$ means adding a to the front of list x.

Dealing with proof obligations

The essence of any **formal method** consists in regarding conjectures such as (3) as **proof obligations** (aka **verification conditions**) which, once discharged, add quality and confidence to the design.

In *lightweight* approaches, one regards (3) 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:

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

Dealing with proof obligations

- In full-fledged formal techniques, one is obliged to provide a **mathematical proof** that conjectures such as (3) 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 <u>**extended**</u> 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, 2013) 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:**

$$\langle \forall\, i\, :\, R \wedge S\, :\, T \rangle \;\; = \;\; \langle \forall\, i\, :\, R\, :\, S \Rightarrow T \rangle \tag{7}$$

$$\langle \exists\, i\, :\, R \wedge S\, :\, T \rangle \;\; = \;\; \langle \exists\, i\, :\, R\, :\, S \wedge T \rangle \tag{8}$$

Exercise 5: Check rule

$$\langle \exists\, i\, :\, R\, :\, T \rangle \;\; = \;\; \langle \exists\, i\, :\, T\, :\, R \rangle \tag{9}$$

□

---

[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

Trading:

$$\langle \forall\ k\ :\ R \wedge S\ :\ T \rangle\ =\ \langle \forall\ k\ :\ R\ :\ S \Rightarrow T \rangle \tag{10}$$

$$\langle \exists\ k\ :\ R \wedge S\ :\ T \rangle\ =\ \langle \exists\ k\ :\ R\ :\ S \wedge T \rangle \tag{11}$$

de Morgan:

$$\neg \langle \forall\ k\ :\ R\ :\ T \rangle\ =\ \langle \exists\ k\ :\ R\ :\ \neg T \rangle \tag{12}$$

$$\neg \langle \exists\ k\ :\ R\ :\ T \rangle\ =\ \langle \forall\ k\ :\ R\ :\ \neg T \rangle \tag{13}$$

One-point:

$$\langle \forall\ k\ :\ k = e\ :\ T \rangle\ =\ T[k := e] \tag{14}$$

$$\langle \exists\ k\ :\ k = e\ :\ T \rangle\ =\ T[k := e] \tag{15}$$

Background — Eindhoven quantifier calculus

Nesting:

$$\langle \forall\, a, b\, :\, R \wedge S\, :\, T \rangle \;=\; \langle \forall\, a\, :\, R\, :\, \langle \forall\, b\, :\, S\, :\, T \rangle \rangle \tag{16}$$

$$\langle \exists\, a, b\, :\, R \wedge S\, :\, T \rangle \;=\; \langle \exists\, a\, :\, R\, :\, \langle \exists\, b\, :\, S\, :\, T \rangle \rangle \tag{17}$$

Rearranging-$\forall$:

$$\langle \forall\, k\, :\, R \vee S\, :\, T \rangle \;=\; \langle \forall\, k\, :\, R\, :\, T \rangle \wedge \langle \forall\, k\, :\, S\, :\, T \rangle \tag{18}$$

$$\langle \forall\, k\, :\, R\, :\, T \wedge S \rangle \;=\; \langle \forall\, k\, :\, R\, :\, T \rangle \wedge \langle \forall\, k\, :\, R\, :\, S \rangle \tag{19}$$

Rearranging-$\exists$:

$$\langle \exists\, k\, :\, R\, :\, T \vee S \rangle \;=\; \langle \exists\, k\, :\, R\, :\, T \rangle \vee \langle \exists\, k\, :\, R\, :\, S \rangle \tag{20}$$

$$\langle \exists\, k\, :\, R \vee S\, :\, T \rangle \;=\; \langle \exists\, k\, :\, R\, :\, T \rangle \vee \langle \exists\, k\, :\, S\, :\, T \rangle \tag{21}$$

Splitting:

$$\langle \forall\, j\, :\, R\, :\, \langle \forall\, k\, :\, S\, :\, T \rangle \rangle \;=\; \langle \forall\, k\, :\, \langle \exists\, j\, :\, R\, :\, S \rangle\, :\, T \rangle \tag{22}$$

$$\langle \exists\, j\, :\, R\, :\, \langle \exists\, k\, :\, S\, :\, T \rangle \rangle \;=\; \langle \exists\, k\, :\, \langle \exists\, j\, :\, R\, :\, S \rangle\, :\, T \rangle \tag{23}$$

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 $(\in S)a \stackrel{\mathrm{def}}{=} a \in S$

- the following universal property holds, for all S, p:

$$p = (\in S) \iff S = \{a : p\ a\} \qquad (24)$$

Exercises

Exercise 6: Infer tautologies

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

from (24).
□

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

$$\langle \forall\ b, a\ :\ \langle \exists\ c\ :\ b = f\ c\ :\ r(c, a) \rangle\ :\ s(b, a) \rangle \tag{25}$$

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. *IEEE Transactions on Software Engineering*, 39(3):305–326, 2013. doi: 10.1109/TSE.2012.15. .