Formal Specification F28FS2, Lecture 10
Limitations of formal techniques

Jamie Gabbay

March 30, 2014

Mathematical limitations (undecidability)

Gödel's Incompleteness Theorem.

It is undecidable whether program P satisfies specification Φ. This means it is impossible in general to test whether a program satisfies a specification!

In practice this may or may not be a problem, depending on the complexity of the program and the specification. However, there are many interesting cases when this is computationally difficult (e.g. 'n is a product of two primes').

That's one reason your mobile phone can't automatically check every app you download to check the predicate 'This app will not steal your identity.'.

Mathematical limitations (inconsistency)

Russell's Paradox is an example of a specification which seems quite reasonable, but is in fact inconsistent.

It is easy to write inconsistent specifications. People do it all the time.

Like bugs in programs, such inconsistency errors can be hard to detect. For many interesting specification languages, the question 'Is this specification consistent?' is either undecidable or computationally extremely expensive.

This is as much a design problem as a mathematical problem and arguably it is a feature of formal specification that such errors may be uncovered.

Nonetheless, it means that making specifications cannot be a push-button operation.

Mathematical limitations (incomplete)

A specification may be incomplete, or vague, meaning that it is not entirely specific.

It may leave major design decisions to the programmer.

'I go home' for example has lots of implementations (road/taxi/train/airplane/Zeppelin/...), all of which get me home.

The canonical incomplete specification is $\top$. Usually read as 'true', in the context of specification it can also be read as 'anything goes'.

Social limitations (ambiguity)

Specification in English may be ambiguous, meaning that it has (at least) two specific interpretations.

'I'm just too cool' might mean, quite specifically:

- I am a seriously hip and trendy lecturer.
- My core temperature is far too low. (Emphasis on the word 'cool'.)
- My core temperature is only just slightly too low. (Emphasis on the word 'just'.)

Note that these meanings are specific, but it may not be clear which specific meaning is intended. It depends on context, which humans are very good at keeping track of.

Z doesn't handle ambiguity well; this is by design. Sometimes, we want ambiguity (e.g. in law we talk about 'what the reasonable person might do').

Social limitations

Specification is expensive. A Z specification, once written, is difficult to modify.

If the customer changes their requirements, you may very easily have to tear up all your specification work and start again.

So Z specification is expensive and inflexible.

Social limitations

Just because a specification is correct does not mean that disaster cannot occur.

The specification may have been in error or may not correspond to actual conditions (e.g. shopping centre built the wrong way around; deep sea submersible has incredibly strong perspex windows when cold, but perspex softens in heat, like, near a deep sea vent).

Users might not understand the specification that the designers were working to (e.g. the warning I once saw on a paint-stripping heat gun "DO NOT USE AS A HAIR DRYER", clearly trying to head off (forgive the pun) possible misunderstanding about this thing that has a handle and jets out hot air).

Think of a few of your own examples.

Benefits of Z specification

Absolute precision, as far as it goes and to the extent that this precision is understood.

No ambiguity, ditto.

Cross-cultural: a set is a set is a set. There is no possibility of cultural bias or misunderstanding.

Susceptible to automation: a fully-formal specification is a logical statement and can be manipulated by a computer. We can use this to verify properties that are either too complex or too repetitive (or both) to be handled by a human. (e.g. chip design.)

Benefits of Z specification

Must balance costs against benefits.

Use full formal specification in safety-critical systems, or embedded systems, or any other scenario where failure just is not an option, no matter what the cost of design and implementation (space shuttle; chip design; automated metro system), or where system is sufficiently complex that formal specification of some kind must be part of the problem-solving and design process itself.

(Israeli fighter plane: they scripted it up in C and tested the hell out of it. I was very disappointed.)

(Great success: Debian package management used formal specification to solve the 'DLL hell' problem in Linux.)

(Intel FDIV bug: cost the company 500 million dollars. Now, all Intel chips undergo rigorous testing and specification.)

Specification vs. implementation

Often presented as contrasts. Actually, two ends of a continuum.

C has types, modules, and header files. These have elements of specification to them (e.g. header files are not intended to be executed; they tell you logical information about function calls).

Java has asserts (and types and . . .).

Scripting languages tend to be low on specification content. So do very machine-oriented languages (assembler).

High level languages tend to be high on specification content (ML/Erlang/Prolog(databases)/. . .).

Ideally, of course, we'd all like a programming environment that will read your mind, negotiate all the contradictions and inconsistencies, find the most efficient implementation, and spit it out for you.

Not going to happen with today's technology.

Different kinds of specification

Not all specification need be fully formal. Two examples:

- Code should be well-commented. A comment in code is a specification of what that code should do (written in English/German/...).
- Variable names and function-calls should be informative. Again, a variable called 'loop-counter' is specifying its intended use.

It doesn't have to be programming! Two examples:

- Legal codes are a very interesting example of a semi-formal specification which must balance precision against flexibility in order to enable the Courts to do justice within the law.
- You renew your mobile phone contract. The choice is dizzying. All those tables of tariffs are a formal specification. Q. Are they designed to inform ...or to confuse potential customers?

Make up your own example from your own experience.

Different kinds of specification

UML (Unified Modelling Language) is one example of an informal but well-structured specification language that uses natural language (English) and diagrams.

It has the pros and cons relative to Z that you could expect: it tends to use natural language so is less scary than Z for more people, and it is more flexible (an advantage also in the design stage, when the spec may be changing).

However, it is less precise than Z, and more ambiguous.

English, UML, and Z, do not compete; they are complementary. Each is a tool for a particular job.

See paper "Formalizing UML Software Models of Safety Critical Systems" (on course webpage), which studies refining a UML spec to a Z spec.

Different metrics for ‘a good program’

Lots of properties make a program ‘good’:

Space-efficient (doesn’t use much memory; embedded chip).

Time-efficient (runs quickly; FPU).

Heat-efficient (requires little power to execute; mobile phone).

Portable (requires little effort to implement on multiple architectures; applet/app).

Obviously correct (requires little effort to debug; not really a buzzword for this one).

Easy to modify (requires little effort to update/modify; ditto).

. . . and so on.

These criteria are both complementary and contradictory. No one ‘right answer’ here: it depends on context.

The place of formal techniques

Formal specification and formal techniques are a tool.

They are an idea; a methodology; a way of thinking.

They are not right. They can slow you down or speed you up. They can keep you out of trouble or draw you into a dead end.

But you can't imagine the modern world without them.

I propose we consider design, specification, and implementation. Specification is a bridge between the two. The type of specification used must be appropriate to the task; sometimes full formal specification is appropriate (FPU design is so complex, algorithms require formal specification; similarly for chip design; also increasingly for security protocols, concurrency, and so on).

Formal techniques are everywhere

Specification is generally becoming more widespread. One reason is obvious: systems are getting larger, more complex, and more abstract.

Most of what you will do to earn a living, most probably, would seem absurdly abstract to someone from the 1950s, or even the 1980s.

Many of you (most? nearly all?) will spend your professional life inhabiting intellectual structures as abstract as anything a university philosopher might have dreamed of, just a generation or two ago.