Formal specification using Z

A. J. van Leeuwen

2006 09 18
Overview

Introduction

Preliminaries

Schemas

Schema Operators

A worked example

Questions

The assignment
Formal specification

- **Formal specification** consists of describing the *required properties* of a system using *formal notation*.
- The required properties are what the customer wants.
- The formal notation allows you to write things down in a *precise* and *unambiguous* manner.
- Formality helps you figure out exactly what the customer wants.
- Formality helps others (including computers) help you.
- Formality allows formal verification.
- Formality requires discipline and hard work.
The Z notation

▶ Z is a standardized notation (ISO/IEC 13568:2002).
▶ Z was introduced in the late 1970’s.
▶ Z has been used both commercially (IBM, Logica/CMG) and academically.

▶ Z provides conventions and notation for presenting mathematical formulas, to model a computer system.
▶ A Z specification document consists of a sequence of paragraphs.
▶ Some of these paragraphs are formal and consist of structured mathematics
▶ Other paragraphs are in narrative text, in natural language
Set theory

- Formal Z is based on Zermelo-Fränkel set theory.
- The consequence is that *everything* in Z is a set.
- The axioms of ZF set theory are:
  - extensionality
  - empty set
  - pairing
  - union
  - infinity
  - separation
  - replacement
  - power set
  - regularity
  - choice
- Z adds the notion of *types* (in essence named sets).
Propositions and Predicates

- A proposition is a *formal statement*.
- Propositions can be combined using conjunction, disjunction, implication, equivalence and negation.
- Propositions may be tautological or contradictory.

- A predicate is an *abstracted* proposition.
- It abstracts over a variable named in the proposition.
- By binding the variable with a *quantifier* the predicate is turned into a proposition again.
- The usual quantifiers are *for all* ($\forall$) and *exists* ($\exists$), but others exist, such as *always* ($\Box$) and *eventually* ($\Diamond$).
Equality and Definition

- Equivalence relates propositions, equality relates values.
- Equality, or rather inequality, enables distinction and counting.
- Equality enables the one-point rule:
  \[(\exists x : a \cdot p \land x = t) \iff (t \in a \land p[t/x])\]
- Equality also supports unique definition: \((\mu x : a \mid p)\)
- Defining objects allows one to reason about them.
- Declaration is the simplest form of definition: it only defines the name of an object.
- Abbreviations introduces names for arbitrary terms, using \(\equiv\).
- Axiomatic definitions further introduce constraints on names.
- Definitions may be generic, that is, parameterized by some set.
Relations and Functions

- Relations express links between (elements of) sets.
- Relations have domains and ranges.
- Relations may be restricted to smaller domains and ranges.
- Relations can be homogeneous and heterogeneous.
- Homogeneous relations can be reflexive, symmetric, asymmetric or antisymmetric, or transitive.
- Relations can be inverted, composed and closed.

- A function is a relation that allows at most one element in the domain to point to any element in the range.
- Functions can be partial or total.
- Functions can be injective, surjective or bijective.
- Functions can be overridden by maps.

Note: it is all sets.
Sequences and Free Types

- Sequences are ordered collections of objects.
- Sequences behave like lists in functional programming.
- Bags are unordered sequences.
- Both sequences and bags have special notation.

- Free types are models of sets.
- The allow for more concise definition of sets.
- Free types are similar to algebraic datatypes.
- A free type introduces both a set and its elements.
- Finitary construction keeps free types consistent.
The Schema

- **Schemas** provide structure to specifications
- A schema consists of two parts
  1. a declaration of variables
  2. a predicate constraining the values of the variables
- Schemas can be displayed in two ways:
  \[
  \text{Schema} \equiv [ \text{declarations} \mid \text{predicates} ]
  \]

```
Schema
---
declarations
predicates
```
Schemas as types

- Schemas can be used in the place of types.
- This provides the only way of making constrained types.
- Elements are identified by name, not position (as in tuples)

\[
Month ::= \text{jan} \mid \text{feb} \mid \text{mar} \mid \text{apr} \mid \text{may} \mid \text{jun} \mid \text{jul} \mid \text{aug} \mid \text{sep} \mid \text{oct} \mid \text{nov} \mid \text{dec}
\]

\[
\begin{array}{l}
\text{Date} \\
\hline
\text{month} : \text{Month} \\
\text{day} : 1 \ldots 31 \\
\hline
\text{month} \in \{\text{apr}, \text{jun}, \text{sep}, \text{nov}\} \implies \text{day} \leq 30 \\
\text{month} = \text{feb} \implies \text{day} \leq 29
\end{array}
\]
Schemas as declarations

- Schemas may be used everywhere declarations are expected
- This includes binding positions in quantifiers and lambda abstractions
- The effect is to introduce bindings for all variables declared in the schema
- The predicate of the schema is taken into account

\{Date \mid day = 31 \land month\} is the set of all months with 31 days.

- One can also bind to existing variables
- This *characteristic binding* binds all variables in the schema to existing variables of the same name
- It is written $\theta Schema$ and does not enforce the predicate
Schemas as predicates

- Schemas may also be used as predicates
- The predicates in the schema are applied to existing variables

\[ \forall \, \text{month} : \text{Month}, \, \text{day} : \mathbb{Z} \cdot \text{Date} \Rightarrow \text{day} \in 1 \ldots 31 \]

- The declarations of a schema may contain hidden predicates
- To avoid confusion, normalize schemas

<table>
<thead>
<tr>
<th>DateNormalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>month : Month</td>
</tr>
<tr>
<td>day : \mathbb{Z}</td>
</tr>
<tr>
<td>day \in 1 \ldots 31</td>
</tr>
<tr>
<td>month \in {apr, jun, sep, nov} \Rightarrow day \leq 30</td>
</tr>
<tr>
<td>month = feb \Rightarrow day \leq 29</td>
</tr>
</tbody>
</table>
A useful operation on schemas is renaming
This allows cleaner documentation
Renaming renames the variables declared and all their occurrences in the predicates
Renaming does not touch the types of the declarations
Renaming is also used in defining change

If we define $StartDate \triangleq Date[startmonth/month, startday/day]$ and $FinishDate \triangleq Date[finishmonth/month, finishday/day]$, then $s : StartDate$ and $f : FinishDate$ cannot be compared. However, $s.startday$ and $f.finishday$ have identical types and can be compared.
Generic schemas

- Schemas may also be parameterized, just as definitions
- The schema is annotated with formal parameters
- These may then be used within the schema
- The schema is then used with actual parameters

\[
\text{System} \left[ \text{Items, Occupants} \right]
\]

\[
\begin{array}{l}
\text{position} : \mathbb{P} \text{Items} \\
\text{occupied} : \text{Items} \rightarrow \text{Occupants} \\
\text{dom} \text{occupied} \subseteq \text{position}
\end{array}
\]

Compare \( \text{System} \left[ \text{Seat, Student} \right] \) and \( \text{System} \left[ \text{Base, Player} \right] \).
Schema conjunction

- Two schemas can be combined with *schema conjunction*.
- This creates one new schema containing
  1. The variables of both schemas, all occurring just once
  2. The predicates of both schemas, joined with a logical conjunction

- Conjunction is undefined if the schemas contain variables with equal name but differing type

\[ Year \cong [\text{year} : \mathbb{Z}] \]

\[ \text{YearDate} \cong \text{Year} / \text{zandDate} \]

\[
\begin{array}{c}
\text{YearDate} \\
\hline \\
\text{Date} \\
\text{Year}
\end{array}
\]
Schema decoration

- The schemas and models thusfar were static
- Adding dynamic behaviour requires distinguishing between a before and an after
- This is done through *decoration*

\[
Date' \equiv Date[month'/month, day'/day]
\]

\[
NextDay \equiv [Date, Date' \mid day' = day + 1]
\]

- This pattern is very common
- Abbreviation for \([Schema, Schema']\) is \(\Delta Schema\)
- Another common pattern is no change, 
  \([\Delta Schema \mid \theta Schema = \theta Schema']\), abbreviated as \(\Xi Schema\)
Schema disjunction

- Schema *disjunction* allows adding alternatives
- This creates one new schema containing
  1. The variables of both schemas, all occurring just once
  2. The predicates of both schemas, joined with a logical disjunction

```
InvalidDate

month : Month
day : ℤ
year : num
result! : Response

¬ Date ⇒ result! = ‘Please enter a valid date’
```

```
DateEntry ⊑ YearDate ∨ InvalidDate
```
As seen on the previous slide, negation of schemas is also allowed.

Negating a schema introduces a new schema containing:

1. The variables of the schema, with their least constrained types
2. The predicates of the schema, negated

Note that the implicit predicates constraining the types of the variables are also taken into account and negated.

This is therefore only easily understood with normalized schemas.
Quantification and hiding

- We may also quantify over some variables in a schema.
- The effect is to create a new schema with:
  1. The variables quantified over removed from the schema.
  2. The quantifications added surrounding the predicates of the schema.

\[ \exists \text{month} : \text{Month} \bullet \text{Date} \text{ is the schema} \]

\[
\begin{align*}
\text{day} : & 1 \ldots 31 \\
\exists \text{month} : \text{Month} \bullet (\text{month} \in \{\text{apr, jun, sep, nov}\} \Rightarrow \text{day} \leq 30 \land \\
& \text{month} = \text{feb} \Rightarrow \text{day} \leq 29)
\end{align*}
\]
Composition

- Decoration introduces the concept of before and after
- This only allows specification of a single stop
- *Composition* allows putting steps after each other

\[ \text{NextDay} \circ \text{NextDay} \text{ increases the day by two and is equal to} \]
\[ (\text{NextDay}[\text{day''} / \text{day'}, \text{month''} / \text{month'}] \land \]
\[ \text{NextDay}[\text{day''} / \text{day}, \text{month''} / \text{month}]) \ (\text{day''}, \text{month''}) \]
A worked example

- Let us specify Noughts and Crosses in Z
- The game board is $3 \times 3$ grid
- Play alternates between the players X and O, with O playing first
- At each turn, a player marks an empty cell his/her symbol
- Each player aims to fill a line of 3 cells with his/her symbol
- A line may be a row, column or diagonal
- Play ends when a player succeeds or the board is full
The game board is a $3 \times 3$ grid. How to represent this?

- We will represent each cell as a tuple ($row$, $column$)
- The entire board can be represented as below

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>1</td>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>2</td>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
</tr>
</tbody>
</table>
Modeling the game board (2)

- How do we specify this in $\mathbb{Z}$?
- We begin by defining the row and column indexes:

$$Index = \{ x : \mathbb{N} \mid x < 3 \}$$

- Then we define the cells of the board:

$$OXOCe ll = Index \times Index$$

- Check for yourself that this specifies all cells!
A few useful functions

- We define \( \text{rowOf} \ c \) and \( \text{columnOf} \ c \) to tell us which row or column a cell \( c \) lies on:

\[
\begin{align*}
\text{rowOf}, \text{columnOf} : \text{OXOCell} & \rightarrow \text{Index} \\
\forall x, y : \text{Index} \Rightarrow \text{rowOf} \ (x, y) = x \\
\forall x, y : \text{Index} \Rightarrow \text{columnOf} \ (x, y) = y
\end{align*}
\]

- This tells us that \( \text{rowOf} \ (1, 2) = 1 \) and \( \text{columnOf} \ (1, 2) = 2 \)
Describing the lines: rows and columns

**Row**

\[
\begin{align*}
\text{area} & : \mathbb{P} \text{OXOCell} \\
\#\text{area} & = 3 \\
\forall pos_1, pos_2 : \text{area} \land \text{rowOf pos}_1 & = \text{rowOf pos}_2
\end{align*}
\]

**Column**

\[
\begin{align*}
\text{area} & : \mathbb{P} \text{OXOCell} \\
\#\text{area} & = 3 \\
\text{Figure this out yourself}
\end{align*}
\]
Describing the lines: the diagonals

**TopRightToBottomLeft**

\[ \text{area} : \mathbb{P} \text{OXOCell} \]

\[ \#\text{area} = 3 \]

\[ \forall \text{pos} : \text{area} \bullet \text{rowOf pos} = \text{columnOf pos} \]

**BottomRightToTopLeft**

\[ \text{area} : \mathbb{P} \text{OXOCell} \]

\[ \#\text{area} = 3 \]

What would this be?
Describing the lines: solutions

**Column**

\[
\text{area} : \mathbb{P} \text{OXOC}ell
\]

\[
\#\text{area} = 3
\]

\[
\forall \text{pos}_1, \text{pos}_2 : \text{area} \bullet \text{columnOf} \ pos_1 = \text{columnOf} \ pos_2
\]

**BottomRightToTopLeft**

\[
\text{area} : \mathbb{P} \text{OXOC}ell
\]

\[
\#\text{area} = 3
\]

\[
\forall \text{pos} : \text{area} \bullet \text{rowOf} \ pos + \text{columnOf} \ pos = 2
\]
Given the definitions for the different lines, we can use disjunction to create a schema containing all lines

\[ \text{Line} \equiv \text{Row} \lor \text{Column} \lor \text{TopLeftToBottomRight} \lor \text{BottomLeftToTopRight} \]
More sets

- We need to specify marks on the board:

  \[ Marks ::= \text{blank} \mid X \mid O \]

- We also need some messages:

  \[ Report ::= \text{OK} \mid \text{InvalidMove} \mid \text{OWins} \mid \text{XWins} \mid \text{Draw} \]
The state of the game

- We need to specify what the board looks like at any moment in the game
- We also need to specify whose turn it is

\[
\begin{aligned}
\text{OXOGame} &:\quad \text{OXOCell} \rightarrow \text{Mark} \\
\text{currPlayer} &:\quad \{X, O\}
\end{aligned}
\]

- We use a total function from board cell to mark. Could it have been partial? How? And why would we have chosen this?
- We have no constraints on the game state. Do we need any?
The start of the game

At the start all fields are blank, and O is the current player.

\[
\text{InitOXOGame} \quad \begin{array}{c}
\text{OXOGame'} \\
\text{ran contents'} = \{ \text{blank} \} \\
\text{currPlayer'} = O \\
\end{array}
\]

(We could also have written \( \text{contents'} (\mid \text{OXOCell} \mid) = \{ \text{blank} \} \))
The game operations

- I will not fully specify these, as it would give away parts of the solution to the assignment
- We need to specify valid moves and invalid moves for each player
- We also need to specify end conditions
- Given these we may specify the game as the conjunction

\[ \text{OXOPlay} \equiv \text{OMoves} \lor \text{XMoves} \lor \text{EndCondition} \]
### Example of operation

\[ OPlays \]
\[ \triangle OXOGame \]
\[ pos? : OXOCell \]
\[ message! : Report \]

<table>
<thead>
<tr>
<th>currPlayer = O</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents pos? = blank</td>
</tr>
<tr>
<td>contents' = contents ( \oplus { pos? \mapsto O } )</td>
</tr>
<tr>
<td>currPlayer' = X</td>
</tr>
<tr>
<td>message! = OK</td>
</tr>
</tbody>
</table>

This would have looked differently with \textit{contents} a partial function. How?

\[ OMoves \equiv OPlays \lor OMakesAnError \]
Example of end condition

\[
\begin{align*}
&OWinsGame \quad \Xi \quad OXOGame \\
&message! : Report \\
&\exists l : Line \bullet contents (\mid l.area \mid) = \{O\} \\
&message! = OWins
\end{align*}
\]

Note: this could not have been specified like this if contents had been a partial function. Think about a solution.

\[
EndCondition \equiv OWinsGame \lor XWinsGame \lor GamesADraw
\]
Questions?
Your assignment: Specifying *Rubik’s Illusion*

- The assignment is online at
  http://www.cs.uu.nl/wiki/Swe/Assignments
- You have to specify the rules of the game *Rubik’s Illusion*
- You may work in pairs