Genealogical Information:
A Case Study in Formal Specifications

David Alex Lamb

May, 2012
External Technical Report
ISSN-0836-0227-2012-589

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

Revision: 1.0
Document prepared May 31, 2012
Copyright ©2010-2012 David Alex Lamb

Contents

List of Figures

List of Tables

Abstract

To teach students what really happens while writing formal specifications, I develop one for a simple genealogical system, show my mistakes, and collect the "right" specifications into an appendix. I describe some rhetorical conventions for using formal mathematics along with informal natural-language explanations, and show how formalization fits into the software development process.

1 Introduction

> A question well put is half answered. – John Dewey, *How We Think*, 1910.

Once upon a time I taught an undergraduate course about "formal methods in software engineering." It wasn't particularly fun for either me or the students. A biased summary of the situation is that "the students were highly resistant to the material;" I suspect that the students would have characterized things differently. What I eventually realized was that I shouldn't be teaching "formal methods." What the students and I needed to study was "how to make formalization interesting," or perhaps even fun. Calling it "formal methods" focuses on mathematics; "formalizing stuff you're interested in" focuses on making real-world requirements precise, which is the *motivation* for learning the formal methods.

This paper is an example of what I now consider a better way to teach formalization.

1.1 Pedagogy

Writing formal specifications is an issue of rhetoric: how to convey meaning through language and notation. It is difficult to learn because we must master and integrate three distinct forms of communication: how to read and write mathematics, how to read and write natural language, and, in some cases, how to write programs.[1] With all three, we must cope with the differences between our own assumptions as writers and those of others as readers.

[1] The absence of "read" for programs is a subject for a different discussion about pedagogy.

One central difficulty is that students have no idea how to begin, and get discouraged when they hit roadblocks – a problem common to both programming and mathematics. The earliest formalization process most of us were exposed to is the grade-school *word problem*: things like

> Two people start at opposite ends of a 31-km forest trail; one walks quickly at 5 kph, while the other manages only 4kph. Where in the trail do they meet?

The point of the question is to get students to set up an equation involving distances and speeds and solve it to find a time. The math-talented solve the equations easily, but they hate translating the words into equations. For the math-averse, the equations are boring or scary and the "story" in the problem statement isn't the least bit compelling.

When I started teaching introductory programming in the 1980s, a lot of our assignments were of the same sort:

> Here is a very simple problem you care nothing about that you must solve in a particular way because there is a specific technique we want you to master.

The simple assignments we use to teach basic techniques have their place; we must walk before we run. Initially we must teach a notation unfamiliar to the students, which requires using such very small exercises (the equivalent of words and sentences). The trouble is that, once students learn the basics, they need something more sophisticated. Small questions don't go far enough – or rather, don't start far enough back: The original problem statement has already isolated so much information that, once the student has learned the technique, the formalization is straightforward. When programmers get to the point of having to design a non-trivial program, they find that more realistic problem statements are never so close to a solution.

When we try to introduce slightly larger examples (akin to the "5-paragraph essays" of high-school English), we make two pedagogical mistakes. First, as with small questions, we continue to present well-polished "right answers" instead of showing the messy creative process with guesses, alternatives, refinements, blind alleys, and mistakes.[2] Second, we focus on conveying the nuts and bolts of the notation instead of the problem being solved. Few computing students are pure mathematicians; they are likely to be uninterested in the

[2] I'm sure many people have written whole books about why people have trouble with ambiguity, "mistakes," and the notion that there can be more than one answer.

mathematics for its own sake, and insofar as they're willing to use math at all they want to see its practical application.

One better approach is to present students with a case study: a description of a suitably real problem, and the story of how someone addressed it, mistakes and all (an approach apparently more common in law and economics than in computing). If the case is interesting enough, it provides motivation for learning the analytical material. This paper develops one such study. I was inspired in part by Donald Knuth, who described in fictional form his own exploration of John Conway's definition of numbers.[5]

"Mistakes and all" means that early sections of the paper contain actual *mistakes* – ones I made while developing the specification. This means you must read the case study carefully, because what you first read may well be the wrong way (or at least, not the best way) to do something.[3] I expect that seeing someone else' mistakes will encourage students to persist when they discover their own. Where I noticed such mistakes, I corrected or at least mentioned them later in the paper; any that remain might be especially instructive to readers who discover them.

I didn't record some types of mistakes: those that had nothing to do with the core intellectual task of developing a specification. I made dozens upon dozens of "syntax errors" in the math, but an automated type checker caught them, in the same way the spelling checker found spelling mistakes. I sometimes introduced unnecessary variables. I failed to follow my intended naming conventions.[4] Introducing the summary in Appendix B required rephrasing some of the main text, and I made programming errors in the scripts that generated it.

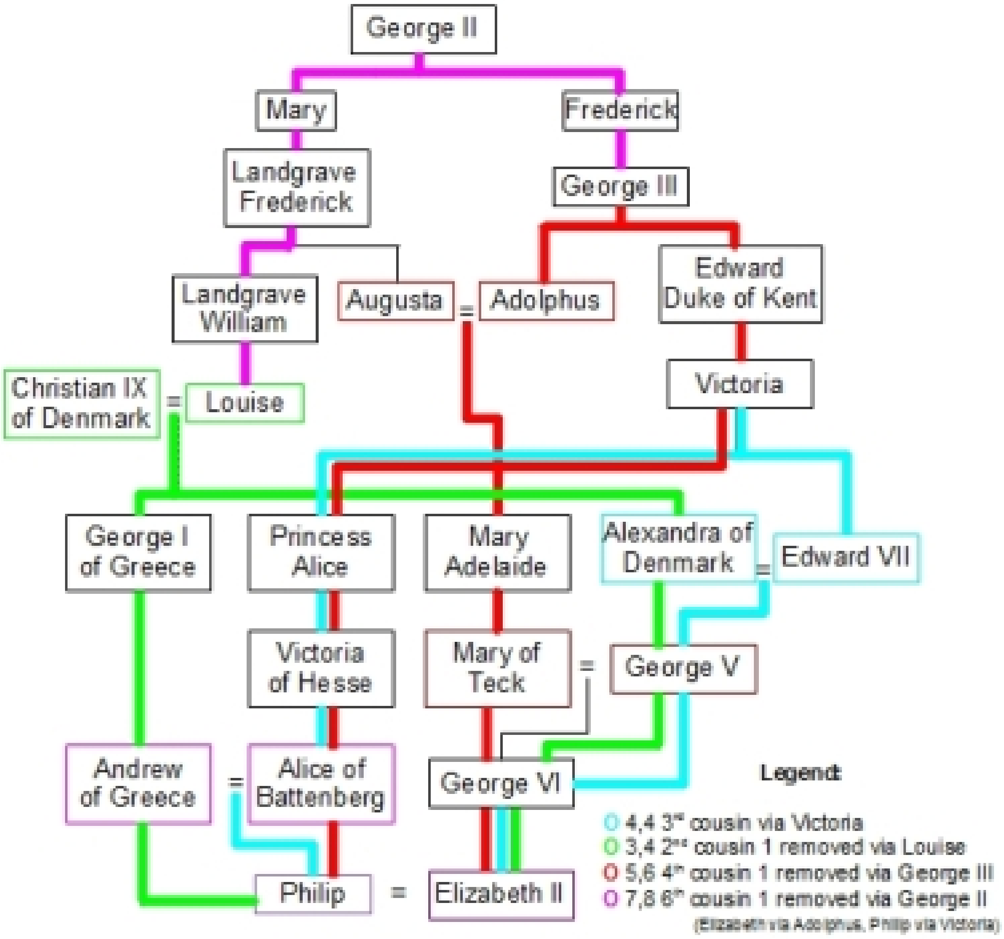
1.2 Problem Statement

Once upon a time, I heard that the British royal family is inbred enough that Queen Elizabeth and Prince Philip are not only husband and wife, but also cousins via two different paths (see Figure 1). I thought it would be fun to write a small program to find and name their relationships. I had a bit of trouble getting the "obvious" implementation to produce the answers I expected. Writing an informal mathematical description (now long lost) helped me set things straight, and wound up being fun in and of itself.

[3] This may be the right way to read almost any non-fiction; mistakes do sometimes make it into publications that didn't mean to include them.

[4] In one case such a misnaming led to a semantic error; see Section 4.2 on page 45.

Figure 1: Relationships Between Queen Elizabeth and Prince Philip



The routes to George II overlap other routes. To avoid one too many sets of crossed lines, the magenta route from Elizabeth to George III (via Adolphus), and from Philip to Louise, are omitted.

The two relationships everyone describes are:

- via Danish king Christian IX, Elizabeth's great-great-grandfather (4 steps) and Philip's great-grandfather (3 steps), which makes them 2nd cousins once removed.
  - Elizabeth: George VI, George V, Alexandra of Denmark, Christian IX of Denmark
  - Philip: Andrew of Greece, George I of Greece, Christian IX of Denmark
- via Queen Victoria, of whom both are great-great grandchildren (4 steps), making them 3rd cousins.
  - Elizabeth: George VI, George V, Edward VII, Queen Victoria.
  - Philip: Alice of Battenberg, Victoria of Hesse, Princess Alice, Queen Victoria

The conventional notion of common ancestor uses the shortest path and might give only the first alternative. Allowing other paths requires generalizing it.

We might or might not want to list other possibilities. For example, the two are related via George III:

- Philip (6 steps): Alice of Battenberg, Victoria of Hesse, Princess Alice, Queen Victoria, Edward Duke of Kent, George III
- Elizabeth (5 steps): George VI, Mary of Teck, Mary Adelaide, Prince Adolphus, George III

Thus they can be considered 4th cousins once removed. These are two paths to a common ancestor different from the earlier ones, but someone using the program might want to reject this path pair because one of the paths includes Queen Victoria.

1.3 Where Formalization Fits Into Software Development

The act of formalization is the focus of this case study, but you need to understand where it fits into a larger context. There is some "overhead" in understanding someone else' specification: you must first learn the culture in which it was written. Culture means more than the technicalities of a notation; it includes conventions about how to use the notation (illustrated throughout this study), the process into which such specifications fit (the subject of this section), and pragmatics.[5]

I'm not entirely fond of definitions, since they often draw sharper boundaries than are appropriate, but we must start somewhere. For the purposes of this paper:[6]

> Formal software specification is a process of analyzing natural language descriptions and translating parts of them into precise mathematical representations suitable for automated analysis or further translation into executable programs.

Software development consists of several intermingled activities, the details and order of which vary depending on the specific methods[7] the developers use.

- The developer interacts with potential users[8] to figure out what the software should do. Sometimes this activity is called "requirements analysis" but that's only part of it. *Requirements elicitation* gets potential users to say, in natural language and within their own worldview, what the software ought to do. *Requirements analysis* applies the developer's analytical skill to find ambiguities, inconsistencies, and omissions in the users' statements. *Requirements specification* turns the initial analysis into something precise. The three processes are inherently intermingled, as analysis suggests new questions to ask the users, and specification reveals difficulties with the analysis.

- The developer decides how to organize the software to meet the requirements: what the major components will be, and how they will interact

[5] Tennent[8] uses "pragmatics" in the context of programming languages to include "language implementation techniques, programming methodology, and language evolution." Translating a specification into code resembles language implementation, rhetorical concerns are part of programming methodology, and reflecting on notation is part of language evolution. Morris[6] defined the term to mean "origins, uses and effects."

[6] Other formalists would have other purposes and thus other definitions.

[7] Software people usually employ the term "methodology" instead of "methods" but to me that ought to mean "the study of methods" rather than a specific collection of methods.

[8] Commonly, actual "end-users" aren't consulted and the software developers talk with their own marketing department or with a customer's higher-level management, who will never use the system or will just use summary reports it produces. This often means the result is not as suited to its intended primary users as it could be.

with each other. Names for these processes often use words like "architecture" and "design."

- The developer implements the design using some specific technologies (a particular programming language, a particular database management system, and so on), reusing existing software components where appropriate.
- Various people verify whether the implemented software meets its requirements – by inspecting the code, by applying mathematical methods, and by testing. In-house testing is usually called "alpha testing." Initial testing by the customer is usually called "acceptance testing" if there's a single customer, or "beta testing" if there's a larger market.
- The software gets delivered to customers. This can sometimes be a lot more complex than just putting something up on a website for downloading.

The first activity is the focus of this study.

1.4 Notes On Usage

I use "Canadian spelling" – a mixture of British for some word classes (such as those ending in *-our* instead of *-or*, and *-tre* instead of *-ter*) and American for others (such as those ending in *-ize* instead of *-ise*).[9] I use the Spivak (1991) gender-neutral pronouns (e, em, eir, emself) when referring to non-specific individual people instead of plural pronouns or awkward constructions like "s/he" or "his or her." I use "I" to describe what I actually did and "we" (instead of "one") to describe what any specification author might do (in addition to its conventional meaning of a plural group to which I happen to belong). In some places I use "you" instead of "the reader."

I use a variant of the Z specification notation[7]. Z has its limitations, including relative obscurity, but is mostly normal math notation.[10] It has four advantages for this paper:

[9] As far as I can tell the informal rule Canadians use is to pick either American or British spelling for any one class, then stick to it (at least in a single document). Thus some might use *-ise*.

[10] I'll explain the strange bits as we go along, often in footnotes; Appendix A summarizes the parts of Z I use.

- Unlike some formal verification systems available when I chose Z, it has a publication format that looks much like regular mathematical notation, which I find aesthetically pleasing.
- There are programs to verify whether a Z specification satisfies what programming language people call "context-free and context-sensitive syntax": structural correctness and type checking.
- It has features, such as "schemas," that make it especially suitable for talking about state changes in computer programs.
- It uses a typed set theory, which makes error checking much easier.

Expanding on the last point: suppose (using an untyped theory) we have two sets X and Y meant to be disjoint, subsets x_1, x_2 from X, and subset y from Y. We could declare them via something like[11]

$$x_1, x_2 : \mathbb{P}\, X$$
$$y : \mathbb{P}\, Y$$

To say that y never overlaps either x_i, we would need to either assert

$$(x_1 \cup x_2) \cap y = \varnothing$$

or prove it. In Z, the declaration

$$[X, Y]$$

guarantees that the two never overlap. The type checkers for Z will tell us that the assertion isn't even legal, since you can't take the intersection of sets of different types.

Standard Z has some technical issues that make specifications a bit harder to read and write than strictly necessary. I've made some changes to the publication form that appears here; Section 3.3 explains the differences.

While writing this paper I kept in mind that different readers might have different intentions and different levels of sophistication, and organized the document to take this into account. Someone who wants to understand every detail can read the manuscript straight through. Someone who just wants to see the correct specification can read Appendix B and possibly Section 2.

[11] $[Y]$ introduces a new type. $y : \mathbb{P}\, Y$ means that y is a set of Ys, or, equivalently, a member of the power set of Y.

Occasionally I divert from the main flow of the case study to discuss a related but somewhat tangential issue that some readers might want to skip. For very short diversions or for explaining a small bit of notation I use parenthetical remarks or footnotes. For moderate diversions that I still expect students to read, I label the section an “interlude.” For greater diversions, especially those explaining a detailed technical concept not specifically needed to understand the case study, I label the section an “aside.”

2 Basic Concepts of Ancestry

This section describes a reconstruction of my requirements analysis of the problem from section 1.2 on page 3: finding and naming relationships between pairs of people.[12] Requirements analysis involves elicitation, analysis, and specification (Section 1.3 on page 5); the three are necessarily intertwined.

The analysis will include some concepts that might seem “obvious” – people (Section 3.1), names (Section 3.5), common ancestors (Section 3.4) – but that is only because it’s likely that you already have some intuitive understanding of the key ideas. Some concepts, such as paths through the graph of ancestry relations (Section 3.6), are a little less obvious. To learn how to analyze more complex things, where the analyst has to learn some unfamiliar concepts, it’s important to see how the process works for simpler cases.

2.1 Requirements Elicitation

I find it useful to view the interaction between a requirements analyst and a client as similar to an annoyingly persistent child trying to figure out rules by questioning a parent.

> Child: Why is Fred my “uncle?”
> Mother: He’s my brother, so he’s your uncle.
> Child: What about Uncle Alan?
> Mother: He’s your father’s brother.
> Child: So two different people’s brothers are both uncles?
> Mother: *ponders how to give a rule that unifies the two* . . . Your uncles are your parents’ brothers.

[12] The passage of time suggests we treat this as historical fiction of the kind that attempts to be reasonably accurate.

> Child: Why is Billy my “cousin?”
> Mother: He’s your Uncle Fred’s son.
> Child: But Grace is my “cousin” and she’s a girl.
> Mother: Gender doesn’t matter. They’re both Uncle Fred’s kids.
> Child: What about Sally?
> Mother: She’s Aunt Mary’s daughter. … *wisely generalizing slightly to short-circuit a lot of other questions* … Anybody with the same grandparent is a cousin.
> Child: What about Andy? We both have the same grandmother but he’s my brother.
> Mother: *sighs*

At this point an appropriate rule might be hard to formulate, but the exasperated parent might have a glimmer that it would have something to do with narrower terms and more specific relationships dominating broader and more general ones – a common pattern.

> Child: Alice says she’s my second cousin. What’s a second cousin?
> Mother: Her mother Sandy is my first cousin, so Alice is your second cousin.
> Child: *ponders briefly* … So what is Alice to you?
> Mother: She’s my first cousin once removed.

I’ve cheated slightly, since the child is likely to have asked some other questions, too, such as:

> Child: What about Uncle Mike?
> Mother: He’s Aunt Mary’s husband, and she’s my sister.
> Child: So what is he to you?
> Mother: My brother-in-law.
> …
> Child: Cinderella had step-sisters …

These introduce other concepts a genealogical system might need to handle, such as relationships changing over time: marriages, divorces, and an initially unrelated person becoming an uncle or step-sister. Given all the things people want to record about their ancestors’ relationships, it can be difficult to decide on appropriate limits. Do godparents count? Honourary aunts and uncles? “Served under General Grant in the Civil War?” Some programs merely use the “catch-all” of textual annotations.

2.2 Requirements Analysis

Analysis turns the elicited sentences into something more organized, precise, and compact, but needn't go all the way to fully mathematical specifications (that being the job of the requirements specification activity). Since someone will have to read and understand a description of the analysis, it's important to find ways of organizing it, such as dividing the problem into smaller parts.

The critical conclusions from analyzing the words of Section 2.1 are:

- Mother named relationships between pairs of people based on the idea of nearest common ancestor, and
- The process for naming relationships depended on the distance (number of steps) between each person and that ancestor, and the genders of some of the people involved.

This immediately lets us divide the problem into two parts: find a common ancestor, and name the relationship based on the paths from each person to that ancestor. A key insight is that changing the definition of common ancestor requires no change in the naming, and changes in the naming (such as via translation to different languages) requires no change in the definition of common ancestor.

Another way of organizing requirements is to find some principle for unifying them. We can use a bit of mathematical notation – variables, subscripts, superscripts, mathematical operators – to make the analysis more precise than what we recorded from the elicitation, but less precise than it will eventually become. Thus we can turn Child and Mother's dialog into a tabular semi-mathematical summary (Table 1). Composing the table required asking a few more questions for disambiguation, which is typical of requirements analysis:

- Without qualifiers, what does "cousin" mean? ("first cousin").[13]
- Whose aunt is Mary? From the answer about cousins, and the assumption that "cousin" means "first cousin" we could deduce she is Child's aunt, but it would be better to ask.

The conventional notion of nearest common ancestor of two people P_1 and P_2 is usually defined by a process: trace back through the ancestors of each, stopping upon finding a person P_0 who is ancestor to both. If there are several

[13] In common English the word "cousin" is actually somewhat ambiguous, sometimes extending to second, third, or even further.

Table 1: Summary of Elicited Relationships

$\mathbf{P_1}$ is	$\mathbf{P_2}$'s	**Relation**	via **Ancestor**	$\mathbf{n_1}$	$\mathbf{n_2}$
Fred	Child	uncle	Mother's parents	1	2
Fred	Mother	brother	Mother's parents	1	1
Alan	Child	uncle	Father's parents	1	2
Alan	Father	brother	Father's parents	1	1
Billy	Child	(1^{st}) cousin	Mother's (and Fred's) parents	2	2
Billy	Fred	son	Fred	1	0
Grace	Child	(1^{st}) cousin	Mother's (and Fred's) parents	2	2
Grace	Fred	daughter	Fred	1	0
Sally	Child	(1^{st}) cousin	some parent's parents	2	2
Sally	Mary	daughter	Mary	1	0
Andy	Child	brother	Mother and Father	1	1
Alice	Child	2^{nd} cousin	Mother's grandparents	3	3
Sandy	Alice	mother	Sandy	0	1
Sandy	Mother	1^{st} cousin	Mother's grandparents	2	2
Alice	Mother	1^{st} cousin once removed	Mother's grandparents	3	2

Relationships are ordered as in the main text. Read each line as:

> P_1 is P_2's *Relation* via *Ancestor*, with n_1 steps from P_1 to *Ancestor* and n_2 steps from P_2 to *Ancestor*.

such people, choose the ancestor whose distance from P_1 and P_2 is smallest.[14] The terminology for describing the relationship between the original two people depends on the number of steps n_1 and n_2 between them and the ancestor. In English the name of the relationship between P_1 to P_2 depends only on the values of n_1 and n_2.

- If $n_1 = 0$, P_1 is the common ancestor: identical to P_2 if $n_2 = 0$, parent if 1, grandparent if 2, and so on. Similarly, when $n_2 = 0$, P_2 is the common ancestor and P_1 is a descendant: self, child, grandchild, and so on.
- If $n_1 = 1$, then P_1 is sibling to P_2 if $n_2 = 1$, aunt or uncle if $n_2 = 2$, great-aunt or great-uncle if 3, and so on.
- When $n_2 = 1$, names work out similarly to $n_1 = 1$ but using "inverse" relationship names: niece or nephew when $n_2 = 2$, great-niece or great-nephew if 3, and so on.
- If both numbers are 2 or greater, the two people are cousins. What kind of cousins are determined by the two numbers, which, since cousinship is symmetric, we can sort "without loss of generality" so that $n_1 \leq n_2$. $n = n_1 - 1$ is the degree (n^{th} cousins) cousins. $r = n_2 - n_1$ is the number of times removed (r removed, or omitted if $r = 0$).

3 The Original Specification

This section describes the specification as I originally developed it. Section 5 discusses how it might change in future. Appendix B collects the "correct" portions of the specification together and omits explanations of the Z language.

3.1 People

The central concept of this formalization is "people." In the Z language, basic concepts become "types." Conventionally, I name types in the singular, and in capital letters.

$$[\mathit{PERSON}]$$

[14] This begs the question about exactly what "smallest" might mean. For example, on general principles (without considering real-world usage about genealogical relationship names) the minimum, maximum, or sum of n_1 and n_2 might each make sense.

Type *PERSON* is the set of all people we'd ever want to represent in the eventual software system. Technically, it's countably infinite; at any time a computer system will only represent a finite subset. This is typical of mathematical specifications: we write definitions with unrealistic assumptions like infinite sets and infinite-capacity integers, then implement a finite subset.

The next critical thing to specify is the notion of mother and father. At this point many of my students would be thinking about a record or class representing people, such as the Java definition:

```
public class Person {
  private Person mother, father;
} // end class Person
```

Part of my job as a teacher is to get them to think in higher-level (more abstract) terms.

The most common mathematical idiom corresponding to a class with instance variables is to consider each variable a function from the class type to the member type:[15]

father_of: PERSON → PERSON
mother_of: PERSON → PERSON

However, genealogists are used to having to deal with either or both being unknown, or at least not represented in their current data. The `person` Java class technically uses a special value, **null**, to represent "unknown." We could do something equivalent: introduce a distinguished value

unknown: PERSON

with the convention that if $father\_of(x) = unknown$ there is no information about x's father, rather than that the father is some specific person we've called *unknown*. Mathematically, though, it is better to think in terms of partial functions.[16] Technically, a partial function is a set of ordered pairs where each possible 1^{st} element of a pair occurs at most once. In a total function, every possible 1^{st} element occurs exactly once. In this case the 1^{st} element is an arbitrary person and the 2^{nd} element is eir father or mother, respectively.[17] If a person *p*'s father is unknown, there is no pair in the *father* function with that person as the 1^{st} element (or "*p* is not in the domain of *father*").

father_of: PERSON ⇸ PERSON
mother_of: PERSON ⇸ PERSON

[15] I commonly give functions names with an *_of* suffix.

[16] ⇸ means "partial function"; → means "total function."

[17] This ignores the issue of distinguishing biological from foster or custodial parents; see Section 5.

It may be “obvious” what the function should mean (how it relates to the natural language requirements), but the explanation should make it specific:

- $\mathit{father\_of}(p)$ and $\mathit{mother\_of}(p)$ are the father and mother of person p, respectively.

Such connections to the natural-language requirements are even more important for multi-parameter functions such as *relation_name_of* on page 36.

3.2 Interlude: “Knowledge,” State, and Schemas

We could continue to develop a specification along these lines. However, ultimately it will help guide a computer implementation, where operations like “add a new person” and “specify that person’s parents” will be necessary. Defining each such operation requires talking about information both before and after the operation, so we can’t use a single global set of unchanging definitions.

The “state” of a software system at a particular time is a representation of what is “known” then. Basically state is a tuple: a list of mathematical values,[18] each corresponding to the representation of one item the software “knows”. To represent state we introduce a special kind of typed tuple Z calls a *schema*:[19]

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{People}} \\
\hline
\mathit{people} : \mathbb{P}\,\mathit{PERSON} \\
\mathit{father\_of} : \mathit{PERSON} \pfun \mathit{PERSON} \\
\mathit{mother\_of} : \mathit{PERSON} \pfun \mathit{PERSON} \\
\hline
\end{array}$$

It shouldn’t be surprising to anyone familiar with software development that this initial specification step is naive, and will need revision.

It is useful to think of the all-capitals sets as being all the people anyone could ever define, while the lower-case sets are the set we’re “currently” talking

[18] “Values” leads non-mathematical programmers to think of scalars such as integers and real numbers, but “mathematical values” includes things like relations and functions and even functions that return functions.

[19] I follow a convention of naming types in all capitals, schemas by capitalizing individual words, and variables in lower case with an underscore (_) between words. Z is case sensitive; it is thus possible to have a type, a schema, and a variable of the same “name” but different typographical conventions – which can be the least confusing thing to do in some circumstances, once one knows the convention.

about – those "known to" (that is, represented within the data of) the intended software package at some particular point in its execution. Initially the package knows nothing about any people; typically the program would read some data file and add people according to what it found there.

A comprehensible formalization needs not only the precise, mathematical portions, but also an informal explanation that connects the formalization to something more understandable to the average technical reader. Thus, if the intervening explanation of schemas were unnecessary, the following explanations should have gone immediately after the *People* schema:

- *people* is the set ($\mathbb{P}$) of currently-known *PERSON*s.

- We know some people's mother or father or both.

The first sentence makes clear that *people* isn't some arbitrary set of people; it is meant to be the set of all *PERSON*s known to the system at a particular point in time. A full software system might have many such sets. For example, it is common to have "undo" operations to reverse prior edits; deleting several people might require a set of recently-deleted *PERSON*s to restore. The 2^{nd} sentence is phrased to avoid simply reading off the function in English.

We'd also want to say something about the consistency of the data: that any *PERSON* mentioned in the father and mother functions is in the set of known people. If I weren't explaining usage conventions piecemeal, I should have said so when I first introduced the functions:

$$\forall\, p, f : PERSON \mid (p, f) \in father\_of$$
$$\bullet\; p \in people \land f \in people$$

A direct translation to English might be:

> For every two *PERSON*s p and f such that (p, f) is in the set *father_of*, it is the case that p and f are both members of set *people*.

We ought instead to write it in a less formal and more reader-friendly way:

- Anyone mentioned in *mother_of* or *father_of*, in either role, is in the set of known *people*, or

- *mother_of* and *father_of* only deal with known *people*.

The two descriptions, mathematical and natural language, complement each other. The natural language gets across the main ideas, and relates the mathematics to the requirements, while the mathematics makes it precise.

Many beginners think about individual elements like this, but it's better to think about entire sets and functions:

$$father\_of \in people \pfun people$$

$$mother\_of \in people \pfun people$$

$people \pfun people$ is the set of all possible (partial) functions from *people* to *people*. *father_of* and *mother_of* are each a single function from this mega-set. Thus a better way to write the *People* schema is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{People}} \\
\hline
people : \mathbb{P}\, PERSON \\
father\_of : PERSON \pfun PERSON \\
mother\_of : PERSON \pfun PERSON \\
\hline
father\_of \in people \pfun people \\
mother\_of \in people \pfun people \\
\hline
\end{array}$$

Everything before the bar is definitions; everything after is assertions. The itemized list of explanations is (usually) in the same order as the parts of the schema. It explains the definitions first and the assertions second. Wherever possible, the explanation uses the terminology of the elicited requirements instead of the mathematics.

3.3 Aside: Taking Liberties with Z

I've found that the official way to write Z is harder to teach than I'd like. It would be much easier to get students started if I could say:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{People}} \\
\hline
people : \mathbb{P}\, PERSON \\
father\_of : people \pfun people \\
mother\_of : people \pfun people \\
\hline
\end{array}$$

- *people* is the set of *PERSON*s currently known to the genealogy system.
- Each person has at most one father recorded.
- Each person has at most one mother recorded.

Z has technical reasons for disallowing this. The definitions (above the bar) can be in any order. Z has a strict definition-before-use semantics, and there's no provision for some convenient topological sorting of the definitions. Furthermore, Z allows a new schema to “include” an older one, which means a adding a literal copy of the old to the new. This combination of features, and a few other constraints, mean that *people* can’t be considered to be defined until the assertion section.

I wrote the first draft of this case study using official Z syntax. Then in a revision pass to improve readability, I realized that some fairly tricky specifications would be shorter and easier to understand if I could write things as in the previous section (Section 3.7.4 on page 31). Hereafter I’ll be using my syntax. To continue to rely on the type checker, I wrote some Unix scripts (using `make`, `bash` and `awk`) to translate my version to the official one.

3.4 Common Ancestors

The first level of defining ancestors is immediate ancestors (parents) and immediate descendants (children).

Parent

$$\begin{array}{l} People \\ parent : people \leftrightarrow people \\ child : people \leftrightarrow people \\ \hline parent = mother\_of \cup father\_of \\ child = parent^{\sim} \end{array}$$

- Each person has have zero or more parents and zero or more children recorded.[20]

- A parent is a mother or a father.

- If one person is another’s parent, the second is the first’s child.[21]

Listing *People* in the first line of *Parent* means that *Parent* includes everything in *People* (definitions plus assertions), plus the newly-defined stuff; it is Z’s way of letting us break up specifications into comprehensible parts. It will turn out also to be essential in defining operations that change program state.

While *mother_of* and *father_of* are each functions, their union is a general relation: if we know both parents, a person occurs as the 1^{st} element in two

[20] Literally, *parent* and *child* are unrestricted binary relations between *people*.

[21] Literally, *child* is the relational inverse of *parent*.

different pairs. Distinguishing functions from general relations is one reason why I name functions with the *_of* suffix and general relations without.

The natural definition of “ancestor” is a parent, or a parent of a parent, and so on for as many steps as the available data provide. Z expresses that as a transitive closure:

$$ancestor = parent^{+}$$

Some specifications are simpler using a zero-step ancestor (“self”). That’s a standard pattern in mathematical descriptions: to allow for a “trivial” or “degenerate” case.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ancestor}} \\ \hline
\textit{Parent} \\
\textit{ancestor} : \textit{people} \leftrightarrow \textit{people} \\
\textit{ancestor}_0 : \textit{people} \leftrightarrow \textit{people} \\ \hline
\textit{ancestor} = \textit{parent}^{+} \\
\textit{ancestor}_0 = \textit{parent}\star \\
\forall\, p : \textit{people} \bullet p \mapsto p \notin \textit{ancestor} \\ \hline
\end{array}
$$

- Someone’s *ancestor* is anyone found by following parent relations repeatedly (at least once).
- Someone’s $ancestor_0$ is either an ancestor or emself.
- No one is eir own ancestor.[22] [23]

It turns out that $ancestor_0$ is used more often than *ancestor* in subsequent specifications,[24]; why use the more complex name for the more often-used concept? Instead of $ancestor_0$ and *ancestor* we could use names *ancestor* and $ancestor_1$ respectively. There’s an important rhetorical issue about whether to define things to be convenient to the specifications or convenient for the users from whom we elicited the requirements. I aim to be reasonably convenient for both audiences but favour the users; it’s hard enough to get students to

[22] $x \mapsto y$ is a notation used for rhetorical purposes. It means the same thing as (x, y) but emphasizes that it is an ordered pair meant to be part of a function rather than a general relation.

[23] This of course eliminates any possibility of representing the relationships in Robert Heinlein’s classic time-travel story, *All You Zombies.*[3]

[24] At a late stage in editing, I had to go through all references to each version of “ancestor,” and definitions derived from them, to verify whether 0 steps was acceptable. I haven’t listed all the places I changed, but there were several errors to fix.

deal with mathematics in the first place, and every step away from natural language makes things harder.

The natural-language definition of a common ancestor is straightforward: given two people, a common ancestor is anyone who is an ancestor to both. To a mathematician a sentence like “given an x and a y, find a z” suggests a function: $f : x \times y \rightarrow z$.

$$
\begin{array}{|l}
\hline
\mathit{CommonAncestor} \\
\mathit{Ancestor} \\
\mathit{common\_ancestors\_of} : \mathit{people} \times \mathit{people} \nrightarrow \mathbb{P}\, \mathit{people} \\
\hline
\forall\, p_1, p_2 : \mathit{people} \bullet \mathit{common\_ancestors\_of}(p_1, p_2) = \\
\qquad \mathit{ancestor}_0 \llparenthesis\, \{p_1\}\, \rrparenthesis \cap \mathit{ancestor}_0 \llparenthesis\, \{p_2\}\, \rrparenthesis \\
\hline
\end{array}
$$

- “Common ancestors” of two people are ancestors of both.[25]

The English phrasing doesn’t say whether we mean *ancestor* or $\mathit{ancestor}_0$. It is sometimes important to *leave out* a clarification because being picky about that detail makes the sentence harder to understand. Formalization involves removing ambiguities, but that’s what the math is for; the accompanying English can remain somewhat ambiguous (though as unambiguous as is reasonable to expect given the rhetorical needs of the prose). This does risk a mismatch between the prose and the mathematics; it is a matter of judgement to decide how far the two should differ in each case.[26]

3.4.1 Aside: Binary Function versus Ternary Relation

There is another way to think about common ancestors: as a ternary relation among people. This is a general principle: whenever we see a function whose result is a set, we should think about relations (sets of tuples) instead of functions. Depending on what other mathematical formulas we must write, it sometimes gives simpler specifications.

[25] Formally speaking, given an arbitrary binary relation r, $r\llparenthesis\, s\, \rrparenthesis$ is the set of 2^{nd} elements from all ordered pairs where any value from set s is the 1^{st} element.

[26] Perhaps a footnote can explain the picky details, to keep the main prose compact.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CommonAncestorRelation}} \\
\hline
\mathit{CommonAncestor} \\
\mathit{ancestry} : \mathbb{P}(\mathit{people} \times \mathit{people} \times \mathit{people}) \\
\hline
\mathit{ancestry} = \{p_0, p_1, p_2 : \mathit{people} \\
\qquad \mid p_0 \in \mathit{common\_ancestors\_of}(p_1, p_2) \\
\qquad \bullet\ (p_1, p_2, p_0)\} \\
\hline
\end{array}$$

- *ancestry* is a set of triples (3-tuple) of *people*
- *ancestry* is defined as the set ({...}) found by
  - considering every combination of three *people* p_0, p_1, p_2 (1^{st} assertion line: the one with '{')
  - for which p_0 is a common ancestor of p_1 and p_2 (2^{nd} line, with "|")
  - and inserting the triple (p_1, p_2, p_0) into the set (3^{rd} line, with "•").

It turns out that later specifications didn't need this particular relation, but in some formalizations the approach might make some assertions simpler.

3.5 Interlude: Dealing with Names

Genealogy involves relationships between people, but research starts with names. Unfortunately the genealogist quickly finds that different people have the same name. Handling this problem requires a great deal of work; the U.S. Social Security Administration had to deal with at least one case where two different people had the same full name, birth date, city of birth, and (if I recall correctly) birth hospital. I'm evading such issues by assuming that each *PERSON* is completely unique and that no two *PERSON*s indicate the same real-world person. To deal with names we introduce

$[\mathit{NAME}]$

- There is a set of *NAME*s, the details of which are (currently) outside the scope of the specification.

Conventionally names are sequences of symbols from some written language, but might be arbitrary glyphs.

Names

$$
\begin{array}{|l}
\hline
\textit{People} \\
\textit{names} : \mathbb{P}\,\textit{NAME} \\
\textit{has\_name} : \textit{people} \leftrightarrow \textit{names} \\
\hline
\end{array}
$$

- The software will record some set of names.[27]
- There is an unconstrained relation, *has_name*, between people and names.[28] Different people may have the same name, the same person might have multiple names (aliases, or changes of name), and some people might (at least briefly) have no name recorded.

This specification derives from *People*, not *CommonAncestor*. Deriving a new schema from one as far up in the hierarchy as possible helps make different parts of the overall software more independent of each other, and makes it easier to talk about what portions of a program's state each operation leaves alone while changing other portions. Later, we'll have to put the pieces together.

3.6 Ancestry Paths

Naming the relationship between two people requires tracing paths through the ancestry relation. Defining nearest common ancestors (and possible generalizations of "nearest") requires knowing at least the lengths of such paths. Unlike computer programs, a mathematical specification has no need to define structures that are as small and efficient as possible, so can describe very large sets if that happens to be convenient.[29] Thus we define the set *ancestry_paths* of all paths from each person to all eir ancestors.

[27] This implies nothing about *how* it will record names. There might be an explicit set, but possibly there would be some indirect representation. For example, if the relation's representation were just a set of ordered pairs, the 2^{nd} elements of all pairs correspond to *names*.

[28] I originally used *names_of* instead of *has_name*, which led me to make the conceptual error discussed in Section 4.2 on page 45.

[29] There is a process called "refinement" which augments a compact but "inefficient" specification into one whose components correspond more directly to typical efficient structures from a chosen programming language. Section 4 on page 42 gives a very brief overview of what that entails.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AncestryPaths}} \\
\hline
\textit{CommonAncestor} \\
\textit{ancestry\_paths} : \mathbb{P}(\mathrm{seq}_1\ \textit{people}) \\
\hline
\forall s : \textit{ancestry\_paths} \bullet \forall i : 2 \mathinner{\ldotp\ldotp} \#s \bullet \\
\qquad s(i) \in \textit{parent}(\!|\ \{s(i-1)\}\ |\!) \\
\forall p_0, p_1 : \textit{people} \mid p_0 \in \textit{ancestor}_0(\!|\ \{p_1\}\ |\!) \\
\qquad \bullet \exists s : \textit{ancestry\_paths} \bullet s(1) = p_1 \land s(\#s) = p_0 \\
\hline
\end{array}$$

- *ancestry_paths* is a set of non-empty sequences of *people*.[30]
- In every ancestry path s, each element $s(i)$[31] is a *parent* of the previous element $s(i-1)$.[32]
- There is an ancestry path from each person p_1 to each of eir ancestors p_0[33].

The first version I wrote of the 1^{st} assertion was excessively complex:

$$\begin{array}{l}
\forall s : \textit{ancestry\_paths};\ n, i : \mathbb{N} \\
\qquad \mid n = \#s - 1 \land i \geq 1 \land i \leq n \\
\qquad \bullet s(i+1) \in \textit{parent}(\!|\ \{s(i)\}\ |\!)
\end{array}$$

I had completely forgotten the $a \mathinner{\ldotp\ldotp} b$ expression, which means the set of numbers between a and b inclusive.[34] I had also introduced the unnecessary variable n; it contributes nothing to either the meaning or the clarity of the assertion.

The first version of the last explanatory item was similarly unnecessarily complex:

[30] seq would mean sequences of any length, including 0. seq_1 excludes empty sequences (those where $\#s$, the size of s, is zero). I eventually realized that, since $parent^+$ has no cycles, it would be technically better to say "iseq," injective sequences, instead of "seq," ordinary ones; injective sequences have no duplicate elements, which ordinary ones might have.

[31] Technically a sequence s of some type T is a function from the integers $1..\#s$ to elements of T.

[32] The English is technically incorrect, since s(1) has no previous element, but that's a common situation: the prose follows intuitive phrasing while the math specifies the precise details.

[33] Initially I used *ancestor* instead of $ancestor_0$; I did not realize my mistake until I happened to review this specification and the *Self* schema on page 37.

[34] It is empty if $a > b$, which means the assertion is trivially true of empty and singleton sequences.

> For every ancestor p_0 of every person p_1, there is an ancestry path s with p_1 as the 1^{st} element and p_0 as the last.

Somewhat better is:

> *ancestry_paths* records a path from each person to each of eir ancestors.

The final version is even better, since it takes one more step towards the informal terminology without losing a connection with the formal specification; the correspondence between the English "ancestry path" and the mathematical *ancestry_paths* is clear enough. It thus took me three tries to write an explanation I now consider to be at the right level of abstraction.

Quite a while after writing the *AncestryPaths* schema, I realized that the assertion and all the versions of its second assertion have a common subtle rhetorical problem. They say that *ancestry_paths* has certain elements, but allow the possibility that there are others.[35] This is a common error of writing specifications of sets (Section C.2). Specifying that *ancestry_paths* is *exactly* this set of sequences requires defining it explicitly via a set comprehension:

$$\begin{array}{l} \mathit{ancestry\_paths} = \{p_0, p_1 : \mathit{people};\ s : \mathrm{seq}_1\, \mathit{people} \\ \quad \mid p_0 \in \mathit{ancestor}_0 (\!|\ \{p_1\}\ |\!) \wedge s(1) = p_1 \wedge s(\#s) = p_0 \\ \quad \bullet\ s\} \end{array}$$

- *ancestry_paths* is the set of all sequences of people where the last element of each sequence is an ancestor of the first.

Read more literally, the set comprehension means:

> Form the set of ancestry paths by considering each pair of people p_0 and p_1 and each possible sequence of people s. Include s in the set if and only if p_1 is the 1^{st} element of the sequence, p_0 is the last, and p_0 is an ancestor of p_1.[36]

Of course, any practical computer program will reify as few of these paths as it needs.

[35] In this specific example it can be proven that there are no others, but in general there might be. In any case it is better to define the set explicitly.

[36] There is a way to write this specification so that p_0 and p_1 are existentially quantified in the "such that" clause (after |) and thus don't appear in the list of variables at the start of the comprehension. Some mathematicians find this more aesthetically pleasing, but I find it harder to read.

3.7 The Genealogical Relationship Problem

Section 2.2 on page 11 identified two sub-problems: finding (generalized) nearest common ancestors (the subject of this section), and naming the resulting relationships (Section 3.8 on page 33). We also have two choices of how to define appropriate common ancestors based on whether to accept George III as a common ancestor of Elizabeth II and Philip:

- Philip (6 steps): Alice of Battenberg, Victoria of Hesse, Princess Alice, Queen Victoria, Edward Duke of Kent, George III
- Elizabeth (5 steps): George VI, Mary of Teck, Mary Adelaide, Prince Adolphus, George III

A priori (before writing a specification) there does not seem to be any technical reason to prefer either alternative; it would be entirely up to the client being interviewed during requirements elicitation. During analysis it is wise to specify both alternatives, and during implementation to permit either based on end-user preferences.

3.7.1 Common Properties, Take 1

Any way of defining nearest common ancestor would correspond to assertions about "acceptable" pairs of ancestry paths. Thus one acceptable path pair for Elizabeth and Philip is

- Elizabeth: George VI, George V, Edward VII, Queen Victoria.
- Philip: Alice of Battenberg, Victoria of Hesse, Princess Alice, Queen Victoria

Adding Edward of Kent to both paths wouldn't be acceptable; it is still a pair of ancestry paths leading to a common ancestor, but it isn't "nearest" in any sense. Traversing past a common ancestor such as Victoria would be appropriate if we want to allow George III. Both possible definitions have some common properties. A pair of ancestry paths define a "nearest common ancestor" relationship only if:

1. Each path begins with one of the two people being compared.
2. Each path ends with the same common ancestor.

3. The paths are distinct.

. This says nothing about what “distinct” should mean. At this point I had very fuzzy ideas about it and wasn’t quite aware of that. I expected that I’d define it more precisely as I wrote the specification.

To express these constraints formally, we must decide on a way to represent “pairs of ancestry paths.” My first thought was to introduce new types:

$$ANCESTRY\_PATH == \mathrm{seq}_1\ PERSON$$

$$PATHPAIR == ANCESTRY\_PATH \times ANCESTRY\_PATH$$

and define a relation saying “for each pair of people, what are the appropriate pairs of ancestry paths?”

$$\begin{array}{|l} ANCESTRY\_PAIRS : PERSON \times PERSON \leftrightarrow PATHPAIR \end{array}$$

While trying to figure out what assertions to write for a schema,[37] it occurred to me that there was no need for such a type. A simple relation is sufficient; from an appropriate pair of paths we could deduce the two people and their common ancestor. At this point I was still using the raw Z syntax:[38]

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ BasicCommonPaths\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
AncestryPaths \\
common\_ancestor\_paths : \\
\qquad ANCESTRY\_PATH \leftrightarrow ANCESTRY\_PATH \\
\hline
\forall s_1, s_2 : ancestry\_paths \mid (s_1, s_2) \in common\_ancestor\_paths \\
\qquad \bullet\ s_1(\# s_1) = s_2(\# s_2) \\
\hline
\end{array}$$

- “Common ancestor paths” is a set of pairs of ancestry paths (which are sequences of people).

- In every ancestry path pair, the last element of each of the sequences is the same person.[39]

[37] This happened before I managed to complete something complicated that I’d have to document given the intent of this study.

[38] I originally made a rhetorical error: *BasicCommonPaths* was called *AcceptablePathPairs* and *common_ancestor_paths* was *acceptable_path_pairs*. The older names were chosen from a technical perspective; the newer are closer to the user’s world-view.

[39] This assertion intentionally doesn’t fully define *common_ancestor_paths*, which is the job of later specifications. Thus it is *not* an example that needs a set comprehension.

As usual with Z specifications, at this point I needed to find an appropriate way to restrict the declaration (which used type *ANCESTRY_PATH*, which in turn uses type *PERSON*) to appropriate "sets of known things." The informal English I had in mind was roughly:

> Everybody mentioned in either sequence of a pair is in *people*, and every sequence is from *ancestry_paths*.

That turned into:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ElementAncestryPath}} \\
\hline
\textit{BasicCommonPaths} \\
\hline
\forall\, p : \textit{people};\ s : \operatorname{dom} \textit{common\_ancestor\_paths} \cup \\
\qquad\qquad \operatorname{ran} \textit{common\_ancestor\_paths} \\
\qquad \mid p \in \operatorname{ran} s \bullet s \in \textit{ancestry\_paths} \\
\hline
\end{array}
$$

The phrase "either sequence of a pair" became variable s[40] while the phrase "everybody mentioned" became variable p.[41] Unfortunately I had forgotten a fundamental guideline for writing readable specifications: It's usually clearer to write assertions about sets rather than elements of sets. I made the mistake in two places: neither s nor p was necessary. In fact the assertion about p was already implied by the assertion about s.

In this case it's far clearer to write:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SetsAncestryPath}} \\
\hline
\textit{BasicCommonPaths} \\
\hline
\textit{common\_ancestor\_paths} \in \textit{ancestry\_paths} \leftrightarrow \textit{ancestry\_paths} \\
\hline
\end{array}
$$

- Both sequences in a "common ancestor path" are ancestry paths.

I then tried to think through what it meant for the paths in a pair to be distinct.

[40] dom and ran are the sets of sequences in the 1^{st} and 2^{nd} positions of all pairs, respectively.

[41] A sequence of people is a function from natural numbers to people, so the "range" (ran) of the sequence is the set of people mentioned.

AncestryPathDistinctness

BasicCommonPaths

$$\begin{array}{l} \forall\, s_1, s_2 : \mathrm{seq}_1\ \mathit{people} \mid (s_1, s_2) \in \mathit{common\_ancestor\_paths} \bullet \\ \qquad s_1(\# s_1) = s_2(\# s_2) \land \\ \qquad (\# s_1 > 1 \lor \# s_2 > 1) \Rightarrow s_1 \neq s_2 \end{array}$$

- The two sequences in every ancestry pair each end[42] with the same person (the common ancestor), and
- The two sequences aren't identical (unless they are each of length 1)

Embarrassingly, my first version of the quantifier ($\forall\, s_1, s_2 \ldots$) failed to note that at least one of the sequences had to have more than one element, and omitted the qualifier ($\# s_1 > 1 \lor \# s_2 > 1$). A program based directly on the specification would have had to say that "David Alex Lamb" is unrelated to "David Alex Lamb" rather than that they are the same person – a problem I originally only discovered in testing the program years ago. More embarrassingly, I forgot that incident when writing this specification; I only realized the error when I reviewed the schema explicitly asking myself "what happens in the trivial case?" (Section 3.7.2) A definition of "acceptable ancestry paths" requires both assertions. I could have included *ElementAncestryPath* in *AncestryPathDistinctness*; however, neither assertion depends on the other, so there is no good reason to privilege one over the other.

3.7.2 Approach 1: No Other Common Ancestor

My first idea for defining "common ancestors" was:

> An appropriate pair of paths has no common ancestors on either path, except the last.

This rules out the relationship based on George III, since Victoria is a common ancestor.

It occurred to me to check if this works for the special case when one person is the ancestor of the other: in this case the ancestor's path has just one element: the ancestor itself. That works: the definition covers both the "normal" case and the "trivial" one (sometimes called the "degenerate" case).[43]

[42] The idiom $s(\# s)$ literally means the sequence s (a function) applied to the size of the sequence, which gives the last element of the sequence.

[43] This is where I realized my mistake in omitting the clause about sequence lengths in schema *AncestryPathDistinctness*.

The rhetorical issue here is finding a readable way to say “all elements but the last” in a sequence. The last element is the one at position $\#s$; the other elements are those from indices 1 to $\#s - 1$. Thus an appropriate quantifier would look something like

$$\forall\, i : 1 \mathrel{..} \#s - 1;\ p : \mathit{people} \mid p = s(i) \ldots$$

which can be read “for every index i from 1 to one less than the size of the sequence s, and for every person p in the sequence at any such index, …” This has the charm that, if the sequence has only one element, the quantifier is trivially satisfied; it isn’t necessary for “…” to be true of the only element in a length 1 sequence.

$$\begin{array}{l}
\mathit{NoOtherAncestor}\\
\hline
\mathit{CommonAncestorPaths}\\
\hline
\forall\, s_1, s_2 : \mathrm{seq}_1\, \mathit{people};\ p_1, p_2 : \mathit{people}\\
\qquad \mid p_1 = s_1(1) \land p_2 = s_2(1) \land\\
\qquad\qquad (s_1, s_2) \in \mathit{common\_ancestor\_paths} \bullet\\
\qquad (\forall\, i : 1 \mathrel{..} \#s_1 - 1 \bullet s_1(i) \notin \mathit{common\_ancestors\_of}(p_1, p_2))\\
\qquad \land\ (\forall\, j : 1 \mathrel{..} \#s_2 - 1 \bullet s_2(j) \notin \mathit{common\_ancestors\_of}(p_1, p_2))\\
\hline
\end{array}$$

The best natural language summary is the quote at the start of the section. Taken line-by-line, a direct reading of the assertion is:

- For every pair of sequences s_1 and s_2
- with 1^{st} elements p_1 and p_2 respectively
- for which s_1 and s_2 are a pair of “common ancestor paths,” it is the case that
  - no element of s_1 (except the last) is a common ancestor of p_1 and p_2
  - and similarly for s_2.

I could have used i in the second assertion instead of j, since the scopes of the two quantifiers don’t overlap, but I judged that using a different name might make the assertion a little clearer.

The parallel structure of the two last clauses (not to mention the excessive length of the schema) eventually caused me to wonder if there were a way to combine them. Eventually I thought of

$$\begin{array}{l}(s_1 \llparenthesis\, 1 \mathrel{..} \#s_1 - 1 \,\rrparenthesis \cup s_2 \llparenthesis\, 1 \mathrel{..} \#s_2 - 1 \,\rrparenthesis) \\ \qquad \cap \mathit{common\_ancestors\_of}(p_1, p_2) = \varnothing\end{array}$$

- . . . it is the case that
  - the elements of both sequences, except the last,
  - are not common ancestors of p_1 and p_2.

Once again this is an example of changing a specification written with elements to one written with sets.

3.7.3 Approach 2: One Child Not a Common Ancestor

Even if we want to generalize "nearest common ancestor" to allow George III, we wouldn't want to allow extending both paths with George II; the longer pair don't give us any additional useful information. One thing that made this example work is that there was a way to get from Elizabeth to George III without going through Victoria; Victoria wasn't on *both* paths. George II would make sense as an "appropriate common ancestor" only if he had some hypothetical child who was an ancestor of one without being an ancestor of the other, which would give yet another distinct pair of paths.

So, perhaps an appropriate definition is that

$$\begin{array}{|l}\hline \mathit{OneChildNotCommon} \\ \mathit{CommonAncestorPaths} \\ \hline \forall\, s_1, s_2 : \mathrm{seq}_1\, \mathit{people};\ p_1, p_2 : \mathit{people} \mid p_1 = s_1(1) \land p_2 = s_2(1) \land \\ \qquad\qquad (s_1, s_2) \in \mathit{common\_ancestor\_paths} \bullet \\ \qquad s_1(\#s_1 - 1) \notin \mathit{common\_ancestors\_of}(p_1, p_2) \lor \\ \qquad s_2(\#s_2 - 1) \notin \mathit{common\_ancestors\_of}(p_1, p_2) \\ \hline \end{array}$$

- In every ancestry path pair, at least one of the children of the common ancestor (2^{nd} last element of the sequence) is not a common ancestor.[44]

[44] I could have broken down this explanation line by line as with *NoOtherAncestor*, but given that schema's textual nearness and the need for an intuitive description at some point, I omitted it.

Long after writing this schema I realized that for the expression $s(\#s - 1)$ to make sense, the set had to be of length 2 or greater. Thus a correct assertion is:[45]

$$
\begin{array}{|l}
\textit{OneChildNotCommon} \\
\hline
\textit{CommonAncestorPaths} \\
\hline
\forall\, s_1, s_2 : \mathrm{seq}_1\, \textit{people};\ \ p_1, p_2 : \textit{people};\ \ ca : \mathbb{P}\, \textit{people} \\
\qquad \mid p_1 = s_1(1) \land p_2 = s_2(1) \land \\
\qquad\qquad (s_1, s_2) \in \textit{common\_ancestor\_paths} \land \\
\qquad\qquad ca = \textit{common\_ancestors\_of}\,(p_1, p_2) \\
\qquad \bullet\ (\ \#s_1 > 1 \Rightarrow s_1(\#s_1 - 1) \notin ca\ ) \lor \\
\qquad\qquad (\ \#s_2 > 1 \Rightarrow s_2(\#s_2 - 1) \notin ca\ ) \\
\hline
\end{array}
$$

- In every ancestry path pair where at least one of the two sequences is of length 2 or greater, at least one of the children of the common ancestor (2^{nd} last element of the sequence) is not itself a common ancestor.

3.7.4 Common Properties, Take 2

At some point a specification meant for other people to read needs an edit for readability. After about the sixth reworking of Section 3.7 I decided the basic Z syntax was less readable than it should be and introduced the variant I described in Section 3.3. I *then* noticed that *SetsAncestryPathAssertion* was redundant, and started shortening Section 3.7.1. That shortening brought *BasicCommonPaths* and *AncestryPathDistinctness* close enough together that I could notice they both contained the assertion

$$s_1(\#s_1) = s_2(\#s_2)$$

That caused me to look at *AncestryPathDistinctness* again, and I noticed that I wanted to allow sequences of length 1 to be exempt. I happened to think about what this might mean for longer sequences and came to yet another embarrassing realization: the $s_1 \neq s_2$ assertion added nothing useful. Any "distinctness" between two paths in a pair is inherently bound up in how we define the nearest common ancestor. Combined with my simplified syntax, all the schemas of Section 3.7.1 could be rewritten as a single schema:

[45] With the assertions becoming textually longer, I introduced ca to shorten them again.

$$
\begin{array}{l}
\textit{AncestryPathPairs} \\
\hline
\textit{AncestryPaths} \\
\textit{common\_ancestor\_paths} : \textit{ancestry\_paths} \leftrightarrow \textit{ancestry\_paths} \\
\hline
\forall\, s_1, s_2 : \textit{ancestry\_paths} \mid (s_1, s_2) \in \textit{common\_ancestor\_paths} \\
\qquad \bullet\; s_1(\# s_1) = s_2(\# s_2)
\end{array}
$$

- “Common ancestor paths” are pairs of ancestry paths.
- In every pair, the last element of both sequences is the same person (the common ancestor).

One of the major advantages of writing specifications in small pieces is that, after making this change, no other parts of the specification needed to be edited. This was especially important given that this version of *CommonAncestorPaths* wasn’t the final one. After many proofreadings of Section 3.7.3, I finally realized that *OneChildNotCommon* as written incorrectly allowed common ancestors to appear *before* the penultimate element of each class. Thus my my initial vague concept of “distinctness” of the two sequences was important; they should contain nothing in common except the last element. This led to what I hope is the final version:

$$
\begin{array}{l}
\textit{CommonAncestorPaths} \\
\hline
\textit{AncestryPaths} \\
\textit{common\_ancestor\_paths} : \textit{ancestry\_paths} \leftrightarrow \textit{ancestry\_paths} \\
\hline
\forall\, s_1, s_2 : \textit{ancestry\_paths} \mid (s_1, s_2) \in \textit{common\_ancestor\_paths} \\
\qquad \bullet\; s_1(\# s_1) = s_2(\# s_2) \\
\qquad \wedge\; s_1 \llparenthesis\, 1 \mathrel{..} \# s_1 - 1 \,\rrparenthesis \cap s_2 \llparenthesis\, 1 \mathrel{..} \# s_2 - 1 \,\rrparenthesis = \varnothing
\end{array}
$$

- “Common ancestor paths” are pairs of ancestry paths.[46]
- In every pair, the last element of both sequences is the same person (the common ancestor).
- The two sequences share no other elements besides the last.

[46] Ancestry paths are defined in Section 3.6 on page 22.

3.7.5 Synthesizing the Result

There were two choices for the distinguishing assertion about ancestry paths. Moreover, the assertions about names were in a separate schema that wasn't included in any of the schemas about acceptable ancestry paths. Thus we could define a complete set of ancestry-concept definitions as either

$$\begin{array}{|l}
\hline
\mathit{AncestryDefinition} \\
\mathit{OneChildNotCommon} \\
\mathit{Names} \\
\hline
\end{array}$$

or the alternative with $\mathit{NoOtherAncestor}$.

3.8 The Relationship Naming Problem

Section 2.2 on page 11 described how relationships are named in English. This section formalizes those requirements. Applying the ideas from previous specification, at this point we expect to define types and schemas for the concepts needed to extend the existing specifications.

3.8.1 Gender

Since some relationship names depend on gender, we need a new type:[47]

$$\mathit{GENDER} == \mathit{male} \mid \mathit{female}$$

- There are two possible genders, male and female.[48]

The natural mathematical specification of finding a person's gender is a function. Appropriate decisions at this point are:

- What are the appropriate assertions about gender? We should at least specify a consistency constraint: fathers are male and mothers are female.

[47] Once upon a time some designers (including me) might have just introduced a variable "$\mathit{male} : \mathit{boolean}$," with the obvious gender bias. Fortunately Z lacks a boolean type, which provides a bit of a push to define types with more appropriate problem-specific names.

[48] To any transgendered or intersex readers: remember what I said about the specification containing mistakes?

- Where in the chain of specifications should the new definitions go? One temptation is to extend the most recent schema, *AncestryDefinition*. However, it's better to make the new definitions include a schema from as high in the hierarchy as possible to make the later schemas as independent of each other as possible.

These considerations lead to making *Gender* depend only on *People*:

$$
\begin{array}{|l}
\textit{Gender} \\
\hline
\textit{People} \\
\textit{gender\_of} : \textit{people} \rightarrow \textit{GENDER} \\
\hline
\forall\, p : \operatorname{ran} \textit{father\_of} \bullet \textit{gender\_of}(p) = \textit{male} \\
\forall\, p : \operatorname{ran} \textit{mother\_of} \bullet \textit{gender\_of}(p) = \textit{female} \\
\hline
\end{array}
$$

- Every recorded person's gender is known.[49]
- Every father ("range" of the *father_of* function) is male.
- Every mother ("range" of the *mother_of* function) is female.

Some later assertions need both ancestry and gender assertions; we can introduce

$$
\begin{array}{|l}
\textit{AncestrySpecification} \\
\hline
\textit{AncestryDefinition} \\
\textit{Gender} \\
\hline
\end{array}
$$

3.8.2 Composing Relationship Names: Take 1

For names of people I decided to avoid details (Section 3.5 on page 21). For naming relationships, the whole point is to specify details; the issue is how much detail is appropriate. An intermediate step would be to introduce a type that abstracts character sequences into "words" or "parts of relationship names." Some relationship names are straightforward: for example, a variable *sister* can represent "sister" or "Sister" or possibly *soeur* (French) or translations to languages with non-Roman alphabets. Z allows for an abbreviation

[49] This was an implicit assertion in my original program. Once noticed it ought to have set off mental alarm bells; I discuss the issue in Section 5.

based on Backus-Naur Form that can represent a type with many specific "constant" instances:

$$\begin{array}{l} \mathit{RELATION\_PART} ::= \mathit{father} \mid \mathit{mother} \mid \mathit{parent} \\ \qquad \mid \mathit{son} \mid \mathit{daughter} \mid \ldots \end{array}$$

This is equivalent to introducing a new type *RELATION_PART* and some specific variables *father*, *mother*, and so on, along with an implicit assertion that all these variables have distinct values.

Considering phrases like "fourth cousin twice removed" steps into territory where the math gets excessively complex. We can at least remove the problem of translating numbers into cardinals and ordinals by defining cousins with simple integers:

$$\mathit{cousin} : \mathbb{N} \times \mathbb{N} \rightarrow \mathit{RELATION\_PART}$$

where the two parameters represent degree and removedness, respectively. Similarly we can partly specify grandparent, grandchild, great grandparent, and so on:

$$\mathit{great} : \mathbb{N} \rightarrow \mathit{RELATION\_PART}$$

with $\mathit{great}(0)$ meaning "grand," $\mathit{great}(1)$ meaning "great grand", and so on. Great-great grandfather would then be the sequence $\langle \mathit{great}(2), \mathit{father} \rangle$. The full declaration of relationship name parts is:

$$[\mathit{RELATION\_PART}]$$

$$\begin{array}{|l} \mathit{grand}, \mathit{self}, \mathit{father}, \mathit{mother}, \mathit{parent}, \\ \qquad \mathit{son}, \mathit{daughter}, \mathit{child}, \mathit{brother}, \mathit{sister}, \\ \qquad \mathit{sibling}, \mathit{aunt}, \mathit{uncle} : \mathit{RELATION\_PART} \\ \mathit{great} : \mathbb{N} \rightarrow \mathit{RELATION\_PART} \\ \mathit{cousin} : \mathbb{N} \times \mathbb{N} \rightarrow \mathit{RELATION\_PART} \\ \hline \mathrm{disjoint} \langle \{\mathit{grand}\}, \{\mathit{self}\}, \{\mathit{father}\}, \{\mathit{mother}\}, \{\mathit{parent}\}, \\ \qquad \{\mathit{son}\}, \{\mathit{daughter}\}, \{\mathit{child}\}, \{\mathit{brother}\}, \{\mathit{sister}\}, \\ \qquad \{\mathit{sibling}\}, \{\mathit{aunt}\}, \{\mathit{uncle}\}, \\ \qquad \mathrm{ran}\ \mathit{great}, \mathrm{ran}\ \mathit{cousin} \rangle \end{array}$$

The disjoint declaration says that the *RELATION_PART*s for individual single names (grand, father, and so on), and the ranges (set of return values) of each of *great* and *cousin*, are all distinct from each other.

Z provides a simpler syntax for such situations. The words that make up relationship names can be represented via a "free type:"

$$\begin{aligned}
&\mathit{RELATION\_PART} ::= \mathit{grand} \mid \mathit{self} \mid \mathit{father} \mid \mathit{mother} \mid \mathit{parent} \\
&\quad \mid \mathit{son} \mid \mathit{daughter} \mid \mathit{child} \mid \mathit{brother} \mid \mathit{sister} \\
&\quad \mid \mathit{sibling} \mid \mathit{aunt} \mid \mathit{uncle} \\
&\quad \mid \mathit{cousin}\langle\!\langle \mathbb{N} \times \mathbb{N} \rangle\!\rangle \\
&\quad \mid \mathit{great}\langle\!\langle \mathbb{N} \rangle\!\rangle
\end{aligned}$$

A relation name is a sequence of such parts:

$$\mathit{RELATION\_NAME} == \mathrm{seq}_1\, \mathit{RELATION\_PART}$$

In Section 2.2 I inferred that relationship names in English depend only on the lengths of the relevant ancestry paths and the genders of the two people and their common ancestor.[50]

$$\begin{array}{|l}
\hline
\mathit{RelationNamingBasics} \\
\mathit{AncestrySpecification} \\
\mathit{relation\_name\_of} : \mathit{ancestry\_paths} \times \mathit{ancestry\_paths} \\
\qquad \rightarrow \mathit{RELATION\_NAME} \\
\mathit{relation\_phrase} : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathit{GENDER} \times \mathit{GENDER} \times \mathit{GENDER} \\
\qquad \nrightarrow \mathit{RELATION\_NAME} \\
\hline
\mathrm{dom}\, \mathit{relation\_name\_of} \subseteq \mathit{common\_ancestor\_paths} \\
\forall s_1, s_2 : \mathrm{seq}_1\, \mathit{people};\ p_1, p_2, \mathit{ca} : \mathit{people} \\
\qquad\qquad \mid (s_1, s_2) \in \mathit{common\_ancestor\_paths} \\
\qquad\qquad \wedge p_1 = s_1(1) \wedge p_2 = s_2(1) \wedge \mathit{ca} = s_1(\# s_1) \\
\qquad \bullet\ \mathit{relation\_name\_of}(s_1, s_2) = \\
\qquad\qquad \mathit{relation\_phrase}(\# s_1, \# s_2, \mathit{gender\_of}(p_1), \mathit{gender\_of}(p_2), \\
\qquad\qquad\qquad \mathit{gender\_of}(\mathit{ca})) \\
\hline
\end{array}$$

- $\mathit{relation\_name\_of}(s_1, s_2)$ means "the name of the relation defined by paths s_1 and s_2" in the direction indicated by the order of the sequences.[51]

[50] But see Section 5 on page 58.

[51] That is, the order of the first person in each sequence. I originally wrote $\mathit{relation\_name\_of}(p_1, p_2)$, applying it to people instead of sequences. The Z type checker can't help with informal, mid-sentence expressions. I happened to notice this during a proofreading pass, but it could easily have lasted into the first draft I showed to other people.

For example, if $s_1 = \langle p_1 \rangle$ and $s_2 = \langle p_2, p_1 \rangle$ and p_1 is male, then p_1 is p_2's father and $\mathit{relation\_name\_of}(s_1, s_2) = \langle \mathit{father} \rangle$ If p_2 is female, $\mathit{relation\_name\_of}(s_2, s_1) = \langle \mathit{daughter} \rangle$.

- The only relation names of interest are those involving ancestry path pairs.

- The name of the relationship between two people is a phrase that depends only[52] on the lengths of the paths ($\#s$) and the genders of the two people and their common ancestor.

3.8.3 Specific Relationship Names

We can specify the names for each relationship in separate schemas, each of which includes *RelationNamingBasics*. Each schema focuses on a particular closely-related set of special cases. A good order in which to tackle things is "simplest first," which, when numbers are involved, often means starting with 1 or 0 and working up. For example, the simplest case is for distance 0, when the "two people" are the same person.

Self

RelationNamingBasics

$\forall g_1, g_2, g : \mathit{GENDER} \bullet \mathit{relation\_phrase}(0, 0, g_1, g_2, g) = \langle \mathit{self} \rangle$

- If the distance between two given people and their common ancestor are both zero, they are the same person.[53]

The phrase about "distance" reminds us that the first two parameters of *relation_phrase* are distances between people in the ancestry graph. We would have needed to say something like this even if the assertion was based directly on *relation_name_of*, but it is even more necessary when using a function that doesn't mention any people. While proofreading this specification, I eventually realized that I was confusing "length of sequence" ($\#s$) with "distance" (number of steps); the latter is one less than the former. I rewrote the *RelationNamingBasics* to subtract one from the length of each sequence (see Section B.6.2 on page 80).

[52] That is, "is a function with parameters ..."

[53] Reviewing this specification led me to realize I needed to use $\mathit{ancestor}_0$ in *AncestryPaths* on page 22.

The next most complex case is direct ancestors and direct descendants, where one distance is zero and the other is nonzero. What occurred to me first was the following monstrosity, which I mistakenly wrote without initially composing a natural-language specification.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{DirectRelation}} \\
\mathit{RelationNamingBasics} \\
\mathit{counting\_names} : \mathbb{N}_1 \rightarrow \mathit{RELATION\_NAME} \\
\hline
\mathit{counting\_names}(1) = \langle\rangle \\
\mathit{counting\_names}(2) = \langle \mathit{grand} \rangle \\
\forall\, n : \mathbb{N};\ \mathit{rn} : \mathit{RELATION\_NAME} \mid n > 2 \bullet \\
\qquad \mathit{counting\_names}(n) = \langle \mathit{great}(n - 2), \mathit{grand} \rangle \\
\forall\, n_1, n_2 : \mathbb{N};\ g_1, g_2, g : \mathit{GENDER};\ \mathit{rn} : \mathit{RELATION\_NAME}; \\
\qquad\qquad\qquad \mathit{rp} : \mathit{RELATION\_PART} \mid \\
\qquad\qquad \mathit{rn} = \mathit{relation\_phrase}(n_1, n_2, g_1, g_2, g) \wedge \mathit{rp} = \mathit{rn}(\#\mathit{rn}) \bullet \\
\quad (\ n_1 = 1 \wedge n_2 > 1 \Rightarrow \mathit{rn} = \mathit{counting\_names}(n_2 - 1) \frown \langle \mathit{rp} \rangle \wedge \\
\qquad (g = \mathit{male} \Rightarrow \mathit{rp} = \mathit{son}) \wedge (g = \mathit{female} \Rightarrow \mathit{rp} = \mathit{daughter})\ ) \wedge \\
\quad (\ n_1 > 1 \wedge n_2 = 1 \Rightarrow \mathit{rn} = \mathit{counting\_names}(n_1 - 1) \frown \langle \mathit{rp} \rangle \wedge \\
\qquad (g = \mathit{male} \Rightarrow \mathit{rp} = \mathit{father}) \wedge (g = \mathit{female} \Rightarrow \mathit{rp} = \mathit{mother})\ ) \\
\hline
\end{array}
$$

You needn't try to puzzle it out; I quickly realized this was too complex – specifically, it tried to do too many things in one large stretch of mathematical notation. Indeed, it was sufficiently complex that I didn't at first notice some outright errors, which may themselves have arisen from the complexity.

3.8.4 Splitting up The Schema

One of the significant benefits of using schemas is being able to split up big specifications into little ones, using schema inclusion to tie them together. There was no need to specify names for ancestor and descendant relationships together. Furthermore, thinking about a split led me to realize that *counting_names*, shared between the two, needed its own schema also.

One benefit of making mistakes is learning how to do better. I *should* have thought of splitting out *counting_names* from the beginning. I knew already that one benefit of separate schemas is that we can introduce auxiliary functions that are only needed for a special case without cluttering the more general case. Direct ancestry potentially involves several occurrences of the word "great;" this suggests using a function that describes how many occurrences of “great” are appropriate, given how many steps there are between the descendant and the ancestor:

- 1 step: no prefix; the ancestor is father or mother.
- 2 steps: prefix “grand”
- 3 steps: prefix “great grand”
- n steps: prefix “great^{n-2} grand”

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CountingNames}} \\
\hline
\textit{RelationNamingBasics} \\
\textit{counting\_names} : \mathbb{N}_1 \rightarrow \textit{RELATION\_NAME} \\
\hline
\textit{counting\_names}(1) = \langle\rangle \\
\textit{counting\_names}(2) = \langle \textit{grand} \rangle \\
\forall n : \mathbb{N} \mid n > 2 \bullet \\
\qquad \textit{counting\_names}(n) = \langle \textit{great}(n-2), \textit{grand} \rangle \\
\hline
\end{array}
$$

In this case we gave the English version before the schema, so there is no need for an itemized list afterwards.

The schemas for direct ancestors and direct descendants are similar:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DirectAncestor}} \\
\hline
\textit{CountingNames} \\
\hline
\forall n_1, n_2 : \mathbb{N};\ g_1, g_2, g : \textit{GENDER};\ \textit{rn} : \textit{RELATION\_NAME}; \\
\qquad\qquad \textit{rp} : \textit{RELATION\_PART} \mid \\
\qquad\qquad \textit{rn} = \textit{relation\_phrase}(n_1, n_2, g_1, g_2, g) \wedge \textit{rp} = \textit{rn}(\#\textit{rn}) \bullet \\
\quad (\ n_1 = 0 \wedge n_2 > 0 \Rightarrow \textit{rn} = \textit{counting\_names}(n_2 - 1) \frown \langle \textit{rp} \rangle \wedge \\
\qquad\quad (g = \textit{male} \Rightarrow \textit{rp} = \textit{father}) \wedge \\
\qquad\quad (g = \textit{female} \Rightarrow \textit{rp} = \textit{mother})\ ) \\
\hline
\end{array}
$$

- When p_1 is the common ancestor ($n_1 = 0$), the relation name ends with “mother or “father,” preceded by an appropriate “great ...” phrase defined by n_2.

I happened to notice at this point that $n_1 = 1$ and $n_2 > 1$ in *DirectRelation* on page 38 were both wrong. The former corresponds to a nephew/niece relation rather than father/mother; the latter omits parents and starts with grandparents. Off-by-one errors are common in programming; this is the same problem in a mathematical context. Unfortunately they are so common that I didn't notice that $n_2 - 1$ as the argument to *counting_names* in *DirectAncestor* was also off by 1; it should be n_2.

Directly copying *DirectAncestor* (thus preserving its mistakes), flipping which person was the ancestor, and editing the relation names led to:

$$
\begin{array}{l}
\hline
\mathit{DirectDescendant} \\
\mathit{CountingNames} \\
\hline
\forall\, n_1, n_2 : \mathbb{N};\ g_1, g_2, g : \mathit{GENDER};\ \mathit{rn} : \mathit{RELATION\_NAME}; \\
\qquad\qquad \mathit{rp} : \mathit{RELATION\_PART} \mid \\
\qquad \mathit{rn} = \mathit{relation\_phrase}(n_1, n_2, g_1, g_2, g) \land \mathit{rp} = \mathit{rn}(\#\mathit{rn}) \bullet \\
\quad (\ n_1 > 0 \land n_2 = 0 \Rightarrow \mathit{rn} = \mathit{counting\_names}(n_1 - 1) \frown \langle \mathit{rp} \rangle \land \\
\qquad (g = \mathit{male} \Rightarrow \mathit{rp} = \mathit{son}) \land \\
\qquad (g = \mathit{female} \Rightarrow \mathit{rp} = \mathit{daughter})\ ) \\
\hline
\end{array}
$$

I also noticed I had to edit g also, changing it to g_1. I suddenly realized that g, the gender of the common ancestor, was irrelevant. A moment's reflection revealed that g_2, the gender of the 2^{nd} person, was also irrelevant: when you say "X is the R of Y," in English only the gender of X matters. *relation_phrase* only needed two distances and one gender as parameters.

The other parameters weren't exactly "wrong;" they were redundant. Irrelevant parameters aren't a matter of correctness; they're a matter of rhetoric. Rhetoric is the study of how to convey meaning effectively; adding irrelevancies makes something harder to understand.

3.8.5 Composing Relationship Names: Take 2

Fixing the problems of Section 3.8.2 requires changing *relation_phrase*, which was defined in *RelationNamingBasics* on page 36. The first issue is that, since there turned out to be several ways to give a meaning to *relation_name_of*, the original *RelationNamingBasics* wasn't quite as basic as it should have been: it should just have defined one function, and left the assertions for later. This allows flexibility for defining names in other languages with different conventions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ \mathit{RelationNamingBasics}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}\\
\mathit{RelationNameOf}\\
\mathit{relation\_phrase} : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathit{GENDER}\\
\qquad \pfun \mathit{RELATION\_NAME}\\
\hline
\forall s_1, s_2 : \mathrm{seq}_1\, \mathit{people};\ p_1, p_2 : \mathit{people}\\
\qquad\qquad \mid (s_1, s_2) \in \mathit{common\_ancestor\_paths}\\
\qquad\qquad \land p_1 = s_1(1) \land p_2 = s_2(1)\\
\qquad \bullet \mathit{relation\_name\_of}(s_1, s_2) =\\
\qquad\qquad \mathit{relation\_phrase}(\# s_1 - 1, \# s_2 - 1, \mathit{gender\_of}(p_1))\\
\hline
\end{array}
$$

- The name of the relationship between two people is a phrase (*relation_phrase*) that depends only on the gender of the first person and the distances between the two and their common ancestor ($\# s_i - 1$).

Schema *CountingNames* from page 39 remains unchanged. *DirectAncestor*, *DirectDescendant*, and *Self* are almost the same as in Section 3.8.3, with the reduced number of parameters to *relation_phrase*. A pair of auxiliary functions can simplify the assertions:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ \mathit{ParentChildNames}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}\\
\mathit{child\_name} : \mathit{GENDER} \rightarrow \mathit{RELATION\_PART}\\
\mathit{parent\_name} : \mathit{GENDER} \rightarrow \mathit{RELATION\_PART}\\
\hline
\mathit{child\_name}(\mathit{male}) = \mathit{son}\\
\mathit{child\_name}(\mathit{female}) = \mathit{daughter}\\
\mathit{parent\_name}(\mathit{male}) = \mathit{father}\\
\mathit{parent\_name}(\mathit{female}) = \mathit{mother}\\
\hline
\end{array}
$$

Direct ancestors are those for which n_1 is zero (and n_2 isn't).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ \mathit{DirectAncestor}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}\\
\mathit{CountingNames}\\
\mathit{ParentChildNames}\\
\hline
\forall n_1, n_2 : \mathbb{N};\ g : \mathit{GENDER};\ \mathit{rn} : \mathit{RELATION\_NAME}\\
\qquad \mid \mathit{rn} = \mathit{relation\_phrase}(n_1, n_2, g)\\
\qquad \bullet n_1 = 0 \land n_2 > 0 \Rightarrow\\
\qquad\qquad \mathit{rn} = \mathit{counting\_names}(n_2) \frown \langle \mathit{parent\_name}(g) \rangle\\
\hline
\end{array}
$$

- When p_1 is the common ancestor ($n_1 = 0$), the relation name ends with "mother or "father," preceded by an appropriate "great ..." phrase defined by n_2.

A "direct descendant" schema would reverse the roles of n_1 and n_2.

The most general ("everything else") case is cousinship.

Cousin

CountingNames

$$\begin{array}{l}
\forall\, n_1, n_2, mn, mx : \mathbb{N};\ g : GENDER;\ rn : RELATION\_NAME \\
\qquad \mid mn = min\{n_1, n_2\} \land mx = max\{n_1, n_2\} \land mn > 1 \\
\qquad \bullet\ relation\_phrase(n_1, n_2, g) = \langle cousin(mn - 1, mx - mn) \rangle
\end{array}$$

- Cousinship is determined by mn and mx, the minimum and maximum of the two numbers.
- If both numbers are 2 or greater ($mn > 1$), the two people are cousins.
- $mn - 1$ is the degree; $mx - mn$ is the number of times removed.

3.8.6 Composing Relationship Names: Take 3

It bothered me that the collection of relation naming schemas (*Self*, *DirectAncestor*/*Descendant*, *Cousin*) was more complex than I wanted it to be. I thought about how to simplify it, and came up with Table 2. A specification is meant to be precise and rigorous, but that doesn't necessarily mean mathematical assertions. The table is rigorous enough; indeed, a fairly straightforward transformation could rewrite each row as a formal assertion.

4 From Specification to Program

The motivation for formalizing something is to clarify what it is, so that we can represent it in a concrete way in the rigid context of computer programs and have them do what you intend. A full description of how to develop a program from a specification is beyond the scope of this already rather long paper, but this section describes a few of the basic ideas.

The specification developed in Section 3 is sufficient to define the meaning of several operations the genealogy program might provide. In general a program maintains some internal state (such as the genealogical database); in

Table 2: Relationship Names in Tabular Form

d_1	d_2	Gender	Name
0	0	any	self
0	1	female	mother
0	1	male	father
0	≥ 2	female	great$^{(d_2-2)}$grandmother
0	≥ 2	male	great$^{(d_2-2)}$grandfather
1	0	female	daughter
1	0	male	son
≥ 2	0	female	great$^{(d_2-2)}$granddaughter
≥ 2	0	male	great$^{(d_2-2)}$grandson
≥ 2	≥ 2	any	$(min(d_1, d_2) - 1)^{th}$ cousin $abs(d_1 - d_2)$ removed

response to "operation" requested from outside, it receives inputs, produces outputs, and changes its state.

Any method of formally representing state changes has to have a way to talk about "the state before the operation" and "the state after the operation." A schema does a good job of representing *one* state; there needs to be a way to talk about two at the same time. Since "state" means "a collection of values for specific variables," we need two versions of every variable, one meaning "before the operation" and one meaning "after." Several specification methods conventionally represent "after" by adding a prime ($'$) after each variable name. An operation-defining schema can use normal inclusion for the "before" copy. Including a schema name ending with a prime means including a copy where every variable gets a prime appended to its name.

SomeOperation

AncestrySpecification
AncestrySpecification$'$

Since this is both redundant very common, it has a special syntax:

Δ*AncestrySpecification*

This gives a way to write a schema that specifies a state change. Talking about inputs and outputs just requires a convention for naming some variables to make clear that they're not part of the overall state, but are just needed to describe the operation. By convention input names end in a question mark (?), and output names end in an exclamation mark (!); these are the only places those special characters can occur.

4.1 Schemas for Basic Operations

An "initialization" operation, such as a constructor in an object-oriented language, leaves the program in some well-defined initial state. For the genealogy program, a suitable initial state might be:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitDatabase}} \\ \hline
\Delta \textit{AncestrySpecification} \\ \hline
\textit{people}' = \varnothing \\
\textit{names}' = \varnothing \\ \hline
\end{array}$$

- Initially, no people or names are known.

Strictly speaking, this means "after *InitDatabase* no people or names are known." The given wording is appropriate for a constructor, but not for a reinitialization operation. In keeping with the principle of writing the English to correspond to the requirements, we must distinguish which of the two was intended rather than describing the mathematics literally.

Initialization need not leave empty state; Section 5.1 shows a state with a set that starts with two specific elements, to which more elements could be added.

Implicitly, because of all the assertions in *AncestrySpecification*, any relation whose domain or range is defined as a subset of either of these sets must itself also be empty. Thus there is not *necessary* to say anything about *has_name*; whether it is *better* to do so is debatable. Deciding on whether to add redundant information is a rhetorical question (in the literal rather than pejorative sense): does it make the meaning clearer to include it? Adding any nontrivial amount of redundant information almost always overwhelms the reader.

It is possible to say that the state, or part of it, doesn't change. Using Ξ instead of Δ introduces the primed and unprimed copies but implicitly asserts, for each variable, that the "before" and "after" versions are equal. Thus for example:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NumberOfPeople}} \\
\hline
\Xi \mathit{AncestrySpecification} \\
\mathit{count}! : \mathbb{N} \\
\hline
\mathit{count}! = \#\mathit{people} \\
\hline
\end{array}
$$

- *NumberOfPeople* leaves the database unchanged.
- Its output, *count*!, is the number of people in the database.

4.2 Adding New People

We need a way to add new people to the database. Since each *PERSON* is meant to be unique, and in an implementation would likely correspond to some unique object, we need to have the program "allocate" one.[54] This corresponds to producing an output for the new *PERSON*. As a first attempt we might write an operation to add a new person with a new name:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddPerson}} \\
\hline
\Delta \mathit{AncestrySpecification} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{person}! : \mathit{PERSON} \\
\hline
\mathit{person}! \notin \mathit{people} \\
\mathit{name}? \notin \mathit{names} \\
\mathit{person}! \in \mathit{people}' \\
\mathit{name}? \in \mathit{names}' \\
\mathit{has\_name}' = \mathit{has\_name} \cup \{\mathit{person}! \mapsto \mathit{name}?\} \\
\hline
\end{array}
$$

- *AddPerson* takes a name as a parameter and returns a person as a result.
- Before the operation neither *person*! nor *name*? can be in the database.[55]
- After the operation both are in the database.

[54] Of course a specification doesn't "allocate" anything; it merely hypothesizes an element from the infinite type that wasn't already present in the finite subset we're dealing with.

[55] Declaring *has_name* as a general relation allows for duplicate names, but the English specifically said that *name*? must be a new name. This ought to prompt us to make sure that's what the users really wanted, and how they expected to deal with duplicate names.

- After the operation $name?$ is the name of $person!$.[56]

A full specification should say what happens if the precondition about $name?$ is false, but that moves further in the direction of specific programming languages (particularly their exception-reporting features).

Strictly speaking we should assert that nothing else in the state changes, except as required by the assertions involving names and people. Thus for example (were we using *RelationNamingBasics* instead of *AncestrySpecification*) we should say

$$gender\_of' = gender\_of$$

Unfortunately people often forget such assertions, which allows an implementor to change *everything* not mentioned. People *also* forget that simply saying

$$person! \in people'$$

says nothing about whatever *else* might change from *people* to $people'$. The assertion should fully define exactly how each state variable changes:

$$\begin{array}{l} people' = people \cup \{person!\} \\ names' = names \cup \{name?\} \end{array}$$

If we are very, very careful (or lucky) about organizing our schemas, we can use a combination of Ξ and Δ to say that some things change while others don't. For example,

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SpecifyGender}} \\
\hline
\Delta \textit{AncestrySpecification} \\
\Xi \textit{AncestryDefinition} \\
p? : people \\
g? : GENDER \\
\hline
p? \notin \operatorname{dom} gender\_of \\
gender\_of' = gender\_of \cup \{p? \mapsto g?\} \\
\hline
\end{array}$$

[56] Originally I used $names\_of$ instead of has_name, which led me to think it was a function instead of a general relation. Thus I wrote

$$names\_of'(person!) = name?$$

It turns out that Z can't detect this semantic error; technically, we'd need to prove (or at least informally check) each function invocation to be sure it is really a function.

The state that could change is everything in *AncestrySpecification*, except things from *AncestryDefinition*; what's left is the components introduced in *Gender*, though we have to refer back to its definition on page 34 to deduce this.[57] In particular, *gender_of* might change. The 1^{st} assertion requires that $p?$'s gender be unknown; the 2^{nd} gives the required addition of gender information.

If we wanted to allow an operation that would change recorded information about gender, we would omit the requirement that gender initially be unknown, and say that the new information overrides the old.

$$gender\_of' = gender\_of \oplus \{p? \mapsto g?\}$$

The $\oplus$ operator applies to any relation, not just functions. It removes from its 1^{st} argument any pair starting with anything in the domain of its 2^{nd} argument ($p?$ in this case) before adding its 2^{nd} argument.

4.3 Creation Versus Naming

A simple genealogy system like the one I was designing typically requires unique names. An extended version would want to allow operations to add multiple names, and even the simple system might want an operation to create an initially nameless person (such as a newborn baby). In this case it is appropriate to split *AddPerson* into two operations: define an initially nameless person, and add a name to a known person.

NewPerson defines a new person.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ NewPerson\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta AncestrySpecification \\
p! : PERSON \\
\hline
p! \notin people \\
people' = people \cup \{p!\} \\
names' = names \\
has\_name' = has\_name \\
gender\_of' = gender\_of \\
father\_of' = father\_of \\
mother\_of' = mother\_of \\
\hline
\end{array}$$

[57] The schema summary in Section D on page 86 might be useful in tracing back through the tree of schema inclusions.

- *NewPerson* returns a new person (one not in the database beforehand, but present afterwards).
- It changes nothing about names or gender.
- It changes nothing else about the set of known people, or mother/father relationships. It thus doesn't change anything from any schema that deals only with ancestry.

Unfortunately I had to explicitly say what doesn't change. We can't include Ξ*Names* or Ξ*Gender* (though my initial incorrect specification did so), since both include *People*, one of whose components (*people*) must change. Section 4.6 discusses alternative ways to say such things.

AddName adds a name for a known person. In this case we *happen* to be able to say that "only variables introduced in *Name* : can change" by combining Ξ and Δ, but any change to the schema inclusions might invalidate this.

$$
\begin{array}{|l}
\mathit{AddName} \\
\hline
\Delta \mathit{AncestrySpecification} \\
\Xi \mathit{Gender} \\
\Xi \mathit{Parent} \\
p? : \mathit{people} \\
n? : \mathit{NAME} \\
\hline
\mathit{has\_name}' = \mathit{has\_name} \cup \{p? \mapsto n?\} \\
\hline
\end{array}
$$

Nothing is said about whether $n? \in \mathit{names}$, so it might or might not duplicate an existing name; this is the opposite of the choice I showed for *AddPerson*, and, as with that choice, it should be verified with users. The assertion about $\mathit{has\_name}'$ in the schema and assertions about *has_name* in *Names* (and thus about *has_name* in Names', which is implicitly included by Δ*AncestrySpecification*) implies that $n? \in \mathit{names}'$. Nothing need be said about *People*, since it was included via Ξ*Parent*, nor anything about common ancestry, since that depends only on *parent*.

4.4 Adding or Changing Relationships

An *AddMother* operation changes information about a person's mother. Once again we are fortunate in being able to implicitly specify that only *mother_of* can change by combining Ξ and Δ.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddMother}} \\
\hline
\Delta \mathit{AncestrySpecification} \\
\Xi \mathit{People} \\
\Xi \mathit{Names} \\
\Xi \mathit{Gender} \\
p? : \mathit{people} \\
m? : \mathit{people} \\
\hline
\mathit{gender\_of}\ m? = \mathit{female} \\
p? \notin \mathit{ancestor}_0 (\!|\ \{m?\}\ |\!) \\
\mathit{mother\_of}' = \mathit{mother\_of} \oplus \{p? \mapsto m?\} \\
\mathit{father\_of}' = \mathit{father\_of} \\
\hline
\end{array}
$$

- The operation takes two people, the 2^{nd} of whom is to be recorded as the mother of the 1^{st}.
- $m?$ must be female.
- $p?$ must not be an ancestor of eir supposed mother $m?$.
- After the operation, $m?$ is known to be the the mother of $p?$, and fatherhood information doesn't change.

The final two assertions are implied by assertions in the included schemas. Were this an ordinary specification it would be a matter of judgement as to whether to include them. I did so because they are things that the implementation would need to explicitly check. A full definition of *AddMother* would be the conjunction of separate schemas for the normal case and each error condition.

4.5 Interlude: Preconditions versus Postconditions

Any assertions that mention only unprimed variables in a schema are *preconditions*: things that must be true before the operation in order for it to execute successfully. Assertions that mention any primed variables are *postconditions*: things the operation must establish as true after it executes, *if* the preconditions are met.

AddMother contained a subtle rhetorical error concerning parent'. It includes $\Delta \mathit{AncestrySpecification}$, which asserts that parent'^{+} contained no cycles. parent implicitly changed because $\mathit{mother\_of}$ changed. Suppose that the database (incorrectly, but unknown to the genealogy program) records $m?$ as descendant of some p_1, who in turn is a descendant of $p?$. There are no cycles in this graph, but introducing $p? \mapsto m?$ introduces one. You might think that the assertion that *parent′* would prevent this – but postconditions are assertions that the operation must *establish* given *only* the preconditions. As written, *AddMother* can't guarantee that this will happen given the state of the database. Thus we need to find a way to say that "adding this pair won't introduce a cycle." We need a way to talk about *parent′* before we can actually assert anything about it.

One way to do this is to introduce a *mu-expression* to give some names to expressions to be evaluated before the operation takes place.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{AcyclicAddMother}} \\
\Delta \mathit{AncestrySpecification} \\
\Xi \mathit{People} \\
\Xi \mathit{Names} \\
\Xi \mathit{Gender} \\
p? : \mathit{people} \\
m? : \mathit{people} \\
\hline
\mathit{gender\_of}\ \ m? = \mathit{female} \\
\mathit{father\_of}' = \mathit{father\_of} \\
\mathit{mother\_of}' = \\
\quad (\mu\, \mathit{newMother} : \mathit{people} \pfun \mathit{people}; \\
\qquad \mathit{newParent} : \mathit{people} \leftrightarrow \mathit{people} \\
\quad\ \mid \mathit{newMother} = \mathit{mother\_of} \oplus \{p? \mapsto m?\} \land \\
\qquad \mathit{newParent} = \mathit{newMother} \cup \mathit{father\_of} \land \\
\qquad (\forall\, p : \mathit{people} \bullet p \mapsto p \notin \mathit{newParent}^{+}) \\
\quad\ \bullet \mathit{newMother}) \\
\hline
\end{array}
$$

- *AcyclicAddMother* takes two parameters, a person $p?$ and eir mother $m?$.

- $m?$ must be female.

- *father_of* doesn't change.

- Adding the new relationship mustn't introduce any cycles. That is, it must be the case that introducing the new $p? \mapsto m?$ relationship to *mother_of* gives a relation *newMother* that, when combined with *father_of*, gives a *newParent* relation that has no cycles. If this precondition is met, *newMother* is the new $mother\_of'$ function.

The mu-expression is a bit like an existential quantifier. It is only defined if all the conditions in the the "such that" clause (introduced by "|") are true, and those mention only unprimed variables. Thus the predicates that are meant to allow the operation to work correctly are true preconditions.

4.6 Interlude: Rhetorical Issues about What Changes

Previous sections pointed out that combining Ξ and Δ on different schemas to specify what changes doesn't always work and, in any case, is difficult to figure out. It would be much simpler to specify what names do change; Z provides an indirect way to do so. First we define a schema that omits certain names.

$$WithoutNames \mathrel{\widehat{=}} AncestrySpecification \setminus (names, has\_name)$$

This means that *WithoutNames* is the same as *AncestrySpecification* without the definitions of *names* and *has_name* and the assertions that mention them. I had to explicitly list all the names newly introduced in *Names*, which is awkward but serves as yet another reason to introduce few new names in any one schema. We can then say that components of *Names* might change but *WithoutNames* doesn't:

$$\begin{array}{|l}
\textit{JustNames} \\
\hline
\Xi\, WithoutNames \\
\Delta Names \\
\hline
\end{array}$$

Since "Δ*Names*" just introduces copies of variables and assertions, it doesn't mean that everything in them changes; the Ξ *WithoutNames* asserts that components of *People* included in *Names* (*people*, *father_of*, *mother_of*, and *parent*) don't change, leaving just *name* and *names*.

Unfortunately Z syntax doesn't permit the \ ("hide") expression as a schema inclusion; this approach requires introducing a new schema every time we want to specify a specific set of components to change.

Another approach is to split *Names* into one schema to introduce component names and another that include it for any assertions that require including other schemas. that combines them. Thus we might say

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NamesComponents}} \\ \hline
\mathit{names} : \mathbb{P}\,\mathit{NAME} \\
\mathit{has\_name} : \mathit{PERSON} \leftrightarrow \mathit{NAME} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Names}1} \\ \hline
\mathit{People} \\
\mathit{NamesComponents} \\ \hline
\mathit{has\_name} \in \mathit{people} \leftrightarrow \mathit{names} \\ \hline
\end{array}$$

along with a similar split for *Parent* and *Gender*. Then *NewPerson* becomes

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NewPerson}1} \\ \hline
\Delta \mathit{AncestrySpecification} \\
\Xi \mathit{NamesComponents} \\
\Xi \mathit{GenderComponents} \\
\Xi \mathit{ParentComponents} \\
p! : \mathit{PERSON} \\ \hline
p! \notin \mathit{people} \\
\mathit{people}' = \mathit{people} \cup \{p!\} \\
\mathit{father\_of}' = \mathit{father\_of} \\
\mathit{mother\_of}' = \mathit{mother\_of} \\ \hline
\end{array}$$

Each approach introduces one new schema for each old schema. The first approach (with $\backslash$) is more flexible, since it allows any list of names to be specified near the place where they are used. However, it introduces what programmers call “coupling:” it requires the same list of names to occur in two widely separated places. The second approach requires a strict discipline when introducing new variables, and and a naming convention that relates the declaration schema to the corresponding assertion schema.

Yet another approach is to avoid thinking of *AncestrySpecification* as the state each operation must deal with:

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{\_\_ } NewPerson2 \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}} \\
\Delta People \\
p! : PERSON \\
\hline
p! \notin people \\
people' = people \cup \{p!\} \\
father\_of' = father\_of \\
mother\_of' = mother\_of \\
\hline
\end{array}$$

Since *People* is the highest schema in the inclusion hierarchy, it happens not to have any changeable state other than the parts we're interested in. The main flaw is that it is not always applicable to schemas lower in the hierarchy unless combined with the separate-component-declaration approach (*NamesComponents* and so on).

A better than any of the three would be some operator like Δ and Ξ that means "only the components introduced in the given schema, excluding anything included from other schemas."

4.7 A Sketch of an Algorithm

The specification so far does not define an algorithm, but does give us terminology for describing a method of discovering common ancestors, and suggests some aspects of what an algorithm needs to do. Since *common_ancestors_of* is defined solely via $ancestor_0$, which is defined via *parent*, finding the set of all known common ancestors involves following *parent* relationships at most until no new parents are found. Even without formally defining the meaning of "nearest common ancestor," we could guess "nearest" would be defined via number of steps taken in tracing parents and stopping upon finding some "first" common ancestor.

This is about the point where I wrote my original program that looked only for conventional nearest ancestors. I had an informal version of the specification in my head, but not a formal definition of the core algorithm's requirements. It is a little dangerous to start programming at this point; it it better to go a little further and write formal requirements for the algorithm (pre- and post-conditions), and a loop invariant for demonstrating that the program meets those requirements. In fact many formalists would argue that, without a precise definition of what the program should do, we can't really claim anything about whether or not it "works correctly."

Informally, it might suffice to follow *parent* relationships until one person's newly-discovered parent were in the set of ancestors of the other parents (including the most-recently-discovered), or we run out of ancestors. Thus:

- For each person keep track of known ancestors and ancestors discovered in the most recent iteration.

- At each step, find the parents of the people discovered in the previous step.

- If the new parents are already in the set of known ancestors of the *other* person, we've found a common ancestor (or set of common ancestors) and can stop.

- Add the newly-discovered ancestors to the corresponding set of known ancestors and continue.

The basic process will work if we start with empty sets of known ancestors and have the initial "newly discovered" ancestors be the two people.

Figure 2 shows pseudocode for this algorithm. Presumably the implementor would try to find an existing representation of sets where the particular operations needed were efficient. It's also appropriate to try to refine the pseudocode to replace potentially expensive set operations, such as unions and intersections, with potentially less expensive operations on individual set elements.[58] Thus a concrete implementation might "optimize" some of this; for example,

```
    while $\text{p1New} \cup \text{p2New} \neq \varnothing$ do
```

might become

```
    while p1New.size() > 0 or p1New.size() > 0 do
```

Similarly, Figure 3 eliminates explicitly forming the relational images $parent(\!|\ \ldots\ |\!)$. In this refinement the program operates on elements instead of sets – exactly the opposite of the approach with mathematical formulations discussed in Section 3.2 on page 16.[59]

The algorithm requires some revisions. Naming the relationships requires knowing how long the paths are from each person to the common ancestor, which requires complicating the algorithm to keep track of paths. As written,

[58] If we *know* the latter are less expensive.

[59] This shouldn't be a surprise; the mathematics should be more abstract, and the code more concrete.

Figure 2: Sketch of a Nearest-Common-Ancestor Algorithm

```
function getCommon(p1, p2: people) returns $\mathbb{P}$ people
declare
   p1Known, p2Known, p1New, p2New: $\mathbb{P}$ people;
begin
   p1Known := p2Known := $\emptyset$;
   p1New := { p1 }; p2New := { p2 };
   while p1New $\cup$ p2New $\neq \emptyset$ do
      p1Known := p1Known $\cup$ p1New;
      p2Known := p2Known $\cup$ p2New;
      common := ( p1New $\cap$ p2Known ) $\cup$
        (p2New $\cap$ p1Known);
      if common $\neq \emptyset$ then return common fi
      p1New := parent$(\!|$ p1New $|\!)$;
      p2New := parent$(\!|$ p2New $|\!)$;
   od
end
```

Set operations are Z symbols; a conventional object-oriented programming language might replace

p1Known := p1Known ∪ p1New;

with

p1Known.add(p1New);

Figure 3: Operating on Elements Instead of Sets

```
P1Temp P2Temp := $\varnothing$;
foundCommon := foundNew := false;
while not foundCommon do
  forall t1:PEOPLE $\in$ P1New do
    f := t1.father_of();
    if f $\neq$ null then
      foundNew := true; P1Temp.add(f);
      if P2Known.hasMember(f) then
        foundCommon := true;
        common.add(f)
      fi
    fi ... -- and similarly for mother_of
  od ... -- and similarly for P2New
od
```

Tests for whether a set is empty become setting of boolean flags when new members are added.

the algorithm stops with the shortest path from a person to a common ancestor; this *happens* to find both of the usually-mentioned relationships between Queen Elizabeth and Prince Philip, since both require 4 steps from the most distant descendant to the common ancestor – (3,4) and (4,4) respectively. It would not find any hypothetical (5,5) relationship (were Edward of Kent the common ancestor instead of Victoria), since it would stop with the length 4 paths, and thus doesn't implement *NoOtherAncestor*. It will *not* find the (6,5) paths to George III, so doesn't implement *OneChildNotCommon* either. It *can* find multiple common ancestors (other than spouses) if paths to those ancestors appear on the same iteration.

4.8 Specifying and Proving the Algorithm

The formal specification of the "find common ancestor paths" operation is straightforward.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{GetCommon}} \\
\hline
\Xi \mathit{AncestrySpecification} \\
p_1? : \mathit{people} \\
p_2? : \mathit{people} \\
ca! : \mathit{ANCESTRY\_PATH} \leftrightarrow \mathit{ANCESTRY\_PATH} \\
\hline
ca! \subseteq \mathit{common\_ancestor\_paths} \\
\forall s_1, s_2 : \mathrm{seq}_1\, \mathit{people} \mid (s_1, s_2) \in ca! \bullet \\
\qquad s_1(1) = p_1? \land s_2(1) = p_2? \\
\exists s_1, s_2 : \mathrm{seq}_1\, \mathit{people} \mid (s_1, s_2) \in \mathit{common\_ancestor\_paths} \land \\
\qquad s_1(1) = p_1? \land s_2(1) = p_2? \\
\quad \bullet\ ca! \neq \varnothing \\
\hline
\end{array}
$$

- The *GetCommon* algorithm changes nothing in the database.
- It takes two people as inputs and returns a set of common ancestor paths.
- Every pair in the result starts with $p_1?$ and $p_2?$, respectively.
- If the two people have any common ancestors, there is at least one path pair in the result.[60]

[60] That is, if there is any sequence pair in the set of common ancestry paths that start with the two people, at least one such pair must be in the output $ca!$.

The definition is straightforward (and the last assertion is minimal) because the "work" of the specification is in *AncestryDefinition* and its included schemas.

What is difficult is discovering whether the algorithm does this – which it doesn't – or, more important, proving that a suitably altered algorithm does. Z semantics are *declarative*: they specify *what* the code must do, but say nothing about *how*. To proved an algorithm correct, we must use an *axiomatic* semantics such as weakest preconditions,[61] (and, of course, translate the pseudo-code into a programming language for which someone has defined such a semantics). Making the transition between the two is possible but beyond the scope of this paper.

At its core the algorithm is a single loop that adds information to some control variables at each step, and terminates when that information runs out. Informally, we can argue that it stops by observing that it follows *parent* relationships, and since *ancestor* has no cycles (is non-reflexive) and *people* is finite, that process ends after at most as many steps as there are people in the database.[62] Arguing that the program works requires finding some mathematical statement of what happens each time around the loop – a *loop invariant* – and proving that when the algorithm stops, the invariant implies the post-condition.

5 Evolving the Package: Domain Analysis

The activity that formulated Section 3 is sometimes called "problem analysis" because it looks at the requirements of a single problem. Faced with a long history of relatively simple problems, programmers are tempted to look too closely at the specific narrow details they're given, and ignore the wider context. It's incredibly common that as soon as you write a program that "solves the problem" you get asked to change it to solve slightly different problems. This sometimes means throwing out much of what you did for the

[61] Often called predicate transformer semantics, which I first encountered in Dijkstra's book.[1]

[62] Formally speaking, for a finite set S of type T and binary relation R on S,

$$
\begin{array}{|l}
S : \mathbb{F}\, T;\ R : T \leftrightarrow T \\
\hline
R \in S \leftrightarrow S \Rightarrow \\
\qquad (\exists\, n : 1 \mathrel{..} \# S \bullet \\
\qquad\qquad \bigcup \{ i : 0 \mathrel{..} n \bullet R^i \} = R^+)
\end{array}
$$

In our case S is *people* and R is *parent*.

first problem, since it embodied assumptions no longer relevant once you have *two* problems to look at.

When things get more complex, we must shift from "problem analysis" to "domain analysis:" not just "how do I solve this one problem?" but "how do I think about a collection of related problems?" The term "analysis" gets used for both the process and the recorded results of the process.

Note the change from "solve" to "think about;" there's a fundamental shift from specific details to general concepts. Both are forms of requirements analysis, but domain analysis has a larger scope. The domain may contain more concepts than the problem; conversely, the problem may omit some concepts from the domain.

The following are a few changes to the requirements that might be appropriate.

- Given a relationship name like "brother" and the data of previous specifications, it should be possible to find all the brothers of a given person. The best way to do this might involve finding some way of associating *RELATION_NAME*s to paths through the ancestry graphs, then defining both "find the relationship between two people" and "find the people in a certain relationship to a given person" via these associations. For example, one finds maternal uncles by the path: "mother, either parent, male children," and all uncles via "parent, parent, male children."

- Normally we expect to know the gender of anyone represented in a genealogical database, which means we expect *gender* to be a total function $people \rightarrow GENDER$ as in Section 3.8.1 on page 34. However, it would be better to define *gender* as a partial function because in some languages some names have ambiguous gender (such as "Leslie" in English) and some genealogical data sources can be incomplete. Thus some old letter might list "Fred X" as having child "Leslie X" without specifying Leslie's gender; in the absence of additional information (such as "Leslie X" being someone's mother) Leslie's gender has to be "recorded as unknown" which normally means a partial function in mathematics.

- The software might perform additional consistency checking, recording each person's gender and verifying that a father is male and a mother is female.[63]

[63] In fact my original software did so, but I left this aspect out of the earlier discussion for simplicity.

- There is a natural bias in a genealogical system towards ancestry relationships, and thus to biological parents and two genders. However, doing a thorough specification of gender would have to recognize that those are not the only two possibilities (Section 5.1). Users typically want to record information about all relatives, not just ancestors; thus they might, for example, want to record infertile XXY children. More speculatively, a chimera (a person with DNA from two merged zygotes) might be capable of both siring and bearing children.[64]

- When I ran the program built from the specifications of the previous section, I immediately noticed that it always reported pairs of relationships, one for a male ancestor and one for a female. For example, siblings would be reported twice, once for their mother and once for their father. Resolving this issue requires recognizing some concept of a mother/father pairing – and, indeed, GEDCOM[2] (a standard for representing genealogical information) does so. Section 5.2 considers this modification.

- Revisions might require handling relationships that violate some of the initial core assumptions. Section 5.3 considers maternal and paternal uncles. Section 5.4 considers half-siblings.

5.1 Gender

Predefining a set of genders might not be appropriate. A "free type" like

$$\mathit{GENDER} ::= \mathit{male} \mid \mathit{female}$$

is not the best approach under these circumstances. A better choice is:

$$[\mathit{GENDER}]$$

- There is a set of genders.

$$\begin{array}{|l} \mathit{male}, \mathit{female} : \mathit{GENDER} \\ \hline \mathit{male} \neq \mathit{female} \end{array}$$

- There are at least two distinct genders, male and female.

[64] The protagonist of a Heinlein science fiction story[3] would have been such a chimera.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Gender}} \\
\hline
\mathit{People} \\
\mathit{genders} : \mathbb{P}\, \mathit{GENDER} \\
\mathit{gender\_of} : \mathit{people} \nrightarrow \mathit{genders} \\
\hline
\{\mathit{male}, \mathit{female}\} \subseteq \mathit{genders} \\
\hline
\end{array}
$$

- There is a set of known genders.
- People have at most one gender recorded.
- Male and female are two of the known genders; they might be the only ones recorded but there might be others.

Rather than pre-specify all possible sets of genders (in an era when the definition of gender has become flexible), the software might provide operations to dynamically add genders. Consistency checking for parenthood might then require specifying which genders can serve what parenting roles.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ParentRole}} \\
\hline
\mathit{Parent} \\
\mathit{Gender} \\
\mathit{mother\_role} : \mathbb{P}\, \mathit{genders} \\
\mathit{father\_role} : \mathbb{P}\, \mathit{genders} \\
\hline
\mathit{male} \in \mathit{father\_role} \\
\mathit{female} \in \mathit{mother\_role} \\
\mathit{gender\_of} \llparenthesis\, \mathrm{dom}\, \mathit{mother\_of} \,\rrparenthesis \subseteq \mathit{mother\_role} \\
\mathit{gender\_of} \llparenthesis\, \mathrm{dom}\, \mathit{father\_of} \,\rrparenthesis \subseteq \mathit{father\_role} \\
\hline
\end{array}
$$

- Some set of genders can serve the role of “father,” and similarly for “mother.”
- Males can be fathers, and females can be mothers.
- Everyone who is recorded as a mother must have a gender capable of playing the role of mother, and similarly for fathers.

5.2 Families

A genealogy program really needs to embody some notion of family. At a minimum, eliminating duplicate output makes this necessary. The software as written from the specification followed parent relationships defined from *mother_of*/*father_of* functions; it could thus (and usually did) find a path from each person to both a male and a female common ancestor. For example, referring to Figure 1 on page 4, it would find that both Christian IX and Louise (husband and wife) were common ancestors, and report the same 2^{nd} cousinship twice.

One solution is to deduce a set of couples from the existing data.

$$
\begin{array}{|l}
\textit{Couples} \\
\hline
\textit{People} \\
\textit{couple} : \textit{people} \leftrightarrow \textit{people} \\
\hline
\textit{couple} = \{ m, f : \textit{people} \\
\qquad \mid (\exists\, p : \textit{people} \bullet m = \textit{mother\_of}(p) \land f = \textit{father\_of}(p)) \\
\qquad \bullet (m, f) \} \\
\hline
\end{array}
$$

- Couples are pairs of people.
- Couples are exactly the pairs of people where the 1^{st} is the mother and the 2^{nd} the father of the same known person.

The software might then combine paths ending in members of a couple, reporting the relationship as “X and Y are Zs via A and B.”

It might at first seem better to introduce a more explicit representation of known couples:

$$
\begin{array}{|l}
\textit{BioParents} \\
\hline
\textit{mother}, \textit{father} : \textit{PERSON} \\
\hline
\end{array}
$$

- Biological parents are a mother and a father.[65]

$$
\begin{array}{|l}
\textit{ParentCouples} \\
\hline
\textit{Parent} \\
\textit{couples} : \mathbb{P}\, \textit{BioParents} \\
\textit{parents\_of} : \textit{people} \pfun \textit{couples} \\
\hline
\forall\, p, m, f : \textit{people} \mid m = \textit{mother\_of}\ p \land f = \textit{father\_of}\ p \\
\qquad \bullet \exists\, c : \textit{couples} \bullet m = c.\textit{mother} \land f = c.\textit{father} \\
\hline
\end{array}
$$

[65] We might eventually decide to represent gestational mothers as well as biological ones, defaulting gestational mother to biological one.

- The software represents a set of couples and, for some people, records what couples are their biological parents.
- Every person whose mother and father are both recorded has a corresponding "biological parent couple" with them as mother and father, respectively.

The algorithm for finding ancestors would then follow *parents_of* relationships instead of *parent* relationships. Unfortunately this representation doesn't handle all cases, since there would be no *BioParent* instance where either parent is unknown; using the *couple* relationship directly might be more appropriate.

A complete representation of "family" would require recording social conventions, marriage relationships, step-relationships, adoption, and fostering.

5.3 Additional Relationship Names

Sections 3.8.2 and 3.8.5 deduced that we could reduce relationship names to distances and gender of the people directly involved. Had Child asked a few more questions, we might have realized this was wrong:

> Child: What's a "paternal uncle?"
> Mother: Your father's brother.
> Child: Like Uncle Alan?
> Mother: Yes. And Fred is your maternal uncle.

Had Mother and Child been speaking Swedish, the issue would have come up immediately with *farbror* and *marbror* (father's brother and mother's brother, respectively). Similarly, in some matrilineal cultural contexts a man's heir was his sister's eldest son, and there might have been a specific word for that relationship. In English only parents' genders are relevant to naming, but once the door is open we can't be sure whether some potential future language requirement might force us to use additional generations. Thus some relationship names might depend on the gender of two or more people. To handle the most general case, *relation_phrase* ceases to be useful and we must base everything on *relation_name_of* directly. Thus, for example:

$$
\begin{array}{|l}
\textit{SwedishUncle} \\
\hline
\textit{RelationNameOf} \\
\textit{uncles} : \textit{ancestry\_paths} \leftrightarrow \textit{ancestry\_paths} \\
\hline
\textit{uncles} = \{s_1, s_2 : \mathrm{seq}_1\, \textit{people} \mid (s_1, s_2) \in \textit{common\_ancestor\_paths} \land \\
\qquad \#s_1 = 2 \land \#s_2 = 3 \land \textit{gender\_of}(s_1(1)) = \textit{male}\} \\
\forall\, s_1, s_2 : \mathrm{seq}_1\, \textit{people};\ \textit{sib} : \textit{people} \mid (s_1, s_2) \in \textit{uncles} \land \textit{sib} = s_2(2) \\
\quad \bullet\ \textit{gender\_of}(\textit{sib}) = \textit{female} \Rightarrow \textit{relation\_name\_of}(s_1, s_2) = \langle \textit{marbror} \rangle \\
\qquad \land\ \textit{gender\_of}(\textit{sib}) = \textit{male} \Rightarrow \textit{relation\_name\_of}(s_1, s_2) = \langle \textit{farbror} \rangle \\
\hline
\end{array}
$$

- Uncle relationships are common ancestor paths with lengths 2 and 3, respectively, where the 1^{st} person (the uncle) is male.
- For every uncle relationship, where *sib* is the uncle's sibling (2^{nd} person in the 2^{nd} path), the relationship is "farbror" if the sibling is male and "marbror" if the sibling is female.

The relationship names would presumably be defined in a revised *RELATION_PART*.

Were we to talk to someone from a traditional Chinese family, we'd find out about needing to distinguish "eldest brother" from other brothers; this might mean recording where each person fit in birth order within their eir family, complicating Section 5.2.

5.4 Half-Siblings

Section 3.8 defined siblings as people with the same parent, which means those with at least one shared parent. A half-sibling is someone who shares one parent but not the other.

$$
\begin{array}{|l}
\textit{HalfSiblingRelation} \\
\hline
\textit{Parent} \\
\textit{half\_sibling} : \textit{people} \leftrightarrow \textit{people} \\
\hline
\textit{half\_sibling} = \textit{half\_sibling}^{\sim} \\
\forall\, p : \mathrm{dom}\, \textit{half\_sibling} \bullet p \in \mathrm{dom}\, \textit{father\_of} \land p \in \mathrm{dom}\, \textit{mother\_of} \\
\hline
\end{array}
$$

- Half-siblinghood is a symmetric relationship between people: if p_1 is p_2's half-sibling then p_2 is p_1's.[66]

[66] I originally wrote this as

$$
\begin{array}{l}
\forall\, p_1, p_2 : \textit{people} \mid p_1 \mapsto p_2 \in \textit{half\_sibling} \bullet \\
\qquad p_2 \mapsto p_1 \in \textit{half\_sibling}
\end{array}
$$

- We know both the mother and the father of every half-sibling.[67]

I put the critical defining assertion in a separate *HalfSibling* schema to break the specification up into easier-to-understand pieces.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{HalfSibling}} \\
\hline
\textit{HalfSiblingRelation} \\
\hline
\forall p_1, p_2 : \textit{people} \mid p_1 \mapsto p_2 \in \textit{half\_sibling} \bullet \\
\qquad \#(\textit{parent}(\!|\ \{p_1, p_2\}\ |\!)) = 1 \\
\hline
\end{array}$$

- Half-siblings have exactly one parent in common.

Because of the second assertion in *HalfSiblingRelation* we can distinguish half-siblinghood from situations where one or both people have only one parent recorded, with nothing known about the other.

It is possible that some users might require that the system be able to record that two people are half-siblings without recording complete information about their parents. In that case we would need to distinguish three relations:

- *deduced_half_sibling*, a renaming of the *half_sibling* already specified.
- *declared_half_sibling*, those declared so explicitly.
- *half_sibling*, the union of the previous two.

This approach (renaming the original *half_sibling* and introducing a new definition for the old name) permits as much as possible of the rest of the specification to remain unchanged.

An operation recording such a declared relationship would have to check if the declaration contradicted anything already known, particularly whether we already knew both parents of both people to be the same. Similarly, an operation to enter parent information would need to check if it contradicted any declarations about half-siblings.

which was another example of writing about elements instead of sets.

[67] The symmetry assertion makes it unnecessary to mention ran *half_sibling* along with dom.

5.5 Vital Statistics

Typical commercial genealogical systems record much more information, such as birth and death dates. It is straightforward to add more (partial) functions such as

$$[DATE]$$

Dates might or might not include times at this point in the analysis. Common speech treats them as different, but computer "dates" usually also represent times of day.

We might need to compare dates. Equality testing is built into Z, but comparisons require explicit definitions:

$$
\begin{array}{|l}
less : DATE \leftrightarrow DATE \\
greater : DATE \leftrightarrow DATE \\
\hline
greater = less^{\sim} \\
\forall\, d_1, d_2, d_3 : DATE \mid (d_1, d_2) \in less \land (d_2, d_3) \in less \\
\bullet\ (d_1, d_3) \in less \\
\forall\, d : DATE \bullet d \mapsto d \notin less^{+}
\end{array}
$$

- *less* and *greater* are relations on dates.
- *greater* is the inverse of *less*.
- *less* (and thus also *greater*) is transitive.
- *less* has no cycles (is irreflexive).

This defines a partial order, which might be necessary if the system had to handle vague dates such as "around 1930."

We can then record information involving dates.

$$
\begin{array}{|l}
\hline
\textit{VitalStatistics} \\
People \\
birth : people \nrightarrow DATE \\
death : people \nrightarrow DATE \\
living : \mathbb{P}\, people \\
\hline
living = people \setminus \operatorname{dom} death \\
\hline
\end{array}
$$

- Living people are those without a recorded death date.

With date information, the system might be able to answer queries such as "who is Y's oldest living relative?"[68]. and "is there a living descendant of P_0 along the female line?" Matrilineal descent can be used to confirm a suspected relationship between someone and a long-dead female ancestor. If P_1 is a direct descendant of a female P_0, and all the people in the ancestral chain are female, and if P_2 (of any gender) shares P_1's mitochondrial DNA, then P_2 is likely a descendant of P_0 also.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Matrilineal}} \\
\mathit{VitalStatistics} \\
\mathit{AncestrySpecification} \\
\mathit{matrilineal} : \mathit{people} \leftrightarrow \mathit{people} \\
\hline
\mathit{matrilineal} = \{ p_0, p_1 : \mathit{people};\ \mathit{path} : \mathit{ancestry\_paths} \\
\qquad \mid p_1 = \mathit{path}(1) \land p_0 = \mathit{path}(\#\mathit{path}) \land \\
\qquad (\forall i : 2 \mathrel{..} \#\mathit{path} \bullet \mathit{gender\_of}(\mathit{path}(i)) = \mathit{female}) \\
\quad \bullet (p_0, p_1) \} \\
\hline
\end{array}$$

- Some people have matrilineal descendants; the 1^{st} person in a pair is the ancestor, and the 2^{nd} the descendant.

- p_1 is a matrilineal descendant of p_0 whenever p_0 and all her descendants between them are female.

For suitable mitochondrial DNA donors for ancestor p_0, we find living matrilineal descendants.

$$\mathit{matrilineal} (\!|\ \{p_0\}\ |\!) \cap \mathit{living}$$

5.6 Additional Features

Some issues are beyond the scope of this study.

With modern reproductive technologies, it is possible to have four different parental roles: biological father, biological mother, gestational mother, and an arbitrary number of custodial and foster parents.[69] Tracing ancestry might care only about biological parents, but a complete genealogical system might

[68] This might presume that the lack of a recorded death date means someone is alive

[69] Genetic engineering might introduce much more information; I recall one science fiction story[4] where someone's body was constructed with chromosome pairs from 23 different people.

want to record all such relationships, which might require a general scheme of binary relations and relational expressions. Representing step-parents requires at least some of this complexity

Every piece of genealogical information has a source, and some sources might contradict other ones. Many of the variables defined in earlier sections might not be primary data, but instead be derived from presumed "facts" which identify their source. For example, information about parents might come from a birth certificate, which could be considered a tuple:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{BirthCertificate} \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{person}, \mathit{mother}, \mathit{father} : \mathit{NAME} \\
\mathit{attendant} : \mathit{NAME} \\
\mathit{birthdate} : \mathit{DATE} \\
\mathit{hospital} : \mathit{NAME} \\
\mathit{uniqueIdentifier} : \mathit{SOURCE} \\
\hline
\end{array}$$

This particular representation requires explicit values for sources, names, and dates, so might require specific values for "unknowns" of each type. An alternative might group a collection of separate functions:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{Sources} \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{People} \\
\mathit{Names} \\
\mathit{sources} : \mathbb{P}\, \mathit{SOURCE} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\, \mathit{BirthData} \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{Sources} \\
\mathit{certificate\_person} : \mathit{sources} \nrightarrow \mathit{names} \\
\mathit{certificate\_birthdate} : \mathit{sources} \nrightarrow \mathit{DATE} \\
\mathit{certificate\_mother} : \mathit{sources} \nrightarrow \mathit{names} \\
\mathit{certificate\_father} : \mathit{sources} \nrightarrow \mathit{names} \\
\mathit{certificate\_hospital} : \mathit{SOURCE} \nrightarrow \mathit{NAME} \\
\mathit{certificate\_attendant} : \mathit{sources} \nrightarrow \mathit{NAME} \\
\hline
\end{array}$$

The last two variables use type *NAME* because hospitals aren't people and attending physicians might not otherwise be in the database.

Other sources might be a conversation or letter about someone mentioning eir father. We might use any or all of the three sources construct a *father_of* function, or check different sources for consistency. Similarly, we might formalize different genealogical file formats as sequences of records defined by a free type, then give assertions that relate the file information to that of earlier sections.

6 Conclusion

I have shown an example of how an experienced teacher develops a formal specification, including mistakes and false paths, accompanied by a polished version of the same. I claimed that this would make it easier to motivate students to learn formal methods.

It is possible to provide evidence for such a claim in two ways, both of which are beyond the scope of this study.

- Appeal to existing pedagogical literature on teaching methods. I have been exposed to some such material through our Centre for Teaching and Learning, and expect to research further.

- Perform an experiment to compare this approach with a "show them the right answer right away" approach. Designing such an experiment requires careful work and consideration for ethics. I don't expect to have the resources for such an experiment in the near future, but I hope that someone reading this study might be able to do so.

There remains one motivational difficulty: Z is a declarative notation, and students in Computing Science might prefer something executable. There are systems based on operational semantics, such as Alloy, that can do so. In future work I expect to investigate to what extent it can help detect errors beyond those the Z type checker can find.

Acknowledgements

I am grateful to Arthur Ryman for teaching me about Z and the rhetoric of specifications through several years of collaboration. Thanks to Margaret Lamb for reviewing the earliest partial draft of this paper, and Jürgen Dingel

for reviewing the first compete draft and suggesting that I investigate Alloy in future work. All remaining errors are “exercises for the reader.”[70]

[70] A euphemism for “my fault.”

A Z Summary

This section describes the subset of Z I used in the paper.

Z includes most basic mathematical notation:

logical operations	$\wedge$ and	$\vee$ or	$\neg$ not	$\Rightarrow$ implies
set operations	$\cap$ intersect	$\cup$ union	$\in$ member	$\notin$ not member
	$\varnothing$ empty set	$\subseteq$ subset	$\subset$ proper subset	
sequences	$\frown$ concatenation	$\langle x_1 \ldots x_n \rangle$ explicit definition		

A.1 Quantifiers

Syntax	**Meaning**
$\forall x : S \mid P(x) \bullet Q(x)$	For all x from set S for which $P(x)$ is true, it is the case that $Q(x)$ is true.
$\exists x : S \mid P(x) \bullet Q(x)$	There exists at least one x from set S for which $P(x)$ is true, such that $Q(x)$ is true.
$\{x_1 : T_1; \cdots;$ $x_n : T_n$ $\mid P(x_1, \cdots, x_n)$ $\bullet Q(x_1, \cdots, x_n)\}$	Set comprehension: for every combination of values for $x_1, \cdots, x_n$ from sets $T_1, \cdots, T_n$ for which predicate $P(x_1, \cdots, x_n)$ is true, the resulting set contains the value of expression $Q(x_1, \cdots, x_n)$.

A.2 Z-Specific Notation

Notation	**Description**
$\mathbb{Z}$	Integers
$\mathbb{N}$	Non-negative integers (natural numbers)
$\mathbb{N}_1$	Positive integers
$[T]$	New type T
$T == \mathit{expr}$	Name for type expression
$\mathbb{P}\, T$	Set of T
$\# T$	Size (cardinality) of set T

Notation	**Description**
$n \mathrel{..} m$	Set of integers from n to m inclusive: $\{i : \mathbb{Z} \mid n \leq i \land i \leq m \bullet i\}$
$X \times Y$	Type consisting of ordered pairs with first element from X, second from Y
$x \mapsto y$	Ordered pair (x,y)
$X \leftrightarrow Y$	Relation on X and Y: $\mathbb{P}(X \times Y)$
$\mathrm{dom}\, z$	Domain of z: set of first elements of the relation: $\forall\, z : X \leftrightarrow Y \bullet \mathrm{dom}\, z =$ $\{x : X;\ y : Y \mid (x, y) \in z \bullet x\}$
$\mathrm{ran}\, z$	Range of z: set of second elements of the relation: $\forall\, z : X \leftrightarrow Y \bullet \mathrm{ran}\, z =$ $\{x : X;\ y : Y \mid (x, y) \in z \bullet y\}$
$z^{\sim}$	Inverse of relation Z (same set of ordered pairs in reverse order): $\{x : X;\ y : Y \mid (x, y) \in z \bullet (y, x)\}$
r^{+}	Transitive closure of relation $r : T \leftrightarrow T$: $r \cup r^2 \cup \ldots$
$r\star$	reflexive transitive closure of relation r
$r(\!\| s \|\!)$	Relational image, the extension of function application to relations: $\forall\, r : X \leftrightarrow Y;\ s : \mathbb{P}\, X \bullet$ $r(\!\| s \|\!) = \{x : s;\ y : Y \mid x \mapsto y \in r \bullet y\}$
$Z == X ⇸ Y$	Partial function from X to Y: relation with no duplicate first element: $\forall\, z : Z;\ x : X \bullet \#z(\!\| \{x\} \|\!) \leq 1$
$Z == X \rightarrow Y$	$\forall\, z : X ⇸ Y;\ x : X \bullet \#z(\!\| \{x\} \|\!) = 1$ Total function from X to Y: partial functio each X occurs exactly once:
$\mathrm{seq}\, X$	Sequence of X: $S == \mathbb{N}_1 ⇸ X$ where $\forall\, s : S \bullet \mathrm{dom}\, s = 1 \mathrel{..} \#s$.
$S == \mathrm{seq}_1\, X$	Nonempty sequence of X: $\forall\, s : S \bullet \#s \geq 1$
$S == \mathrm{iseq}\, X$	injective sequence of X (no duplicate X's): $S == \mathbb{N}_1 ⤔ X$ where $\forall\, s : S \bullet \mathrm{dom}$
$s ⩤ r$	Domain anti-restriction: given $s : \mathbb{P}\, S;\ r : S \leftrightarrow T$, keep only ordered pairs from r whose first element is *not* in s: $\forall\, s : \mathbb{P}\, S;\ r : S \leftrightarrow T \bullet s \lhd r = \{x : S;\ y : T \mid x \mapsto y \in r \land x \notin s \bullet x \mapsto y\}$
$r_1 \oplus r_2$	Override (part of) relation r_1 with r_2. $r_1 \oplus r_2 = ((\mathrm{dom}\, r_2) ⩤ r_1) \cup r_2$

A.3 Variable Definitions and Assertions

Syntax	Meaning
$\ldots x_i : T_i \ldots$ *assertions*	Definitions of new global variables x_i, with assertions about their values.
name $\ldots x_i : T_i \ldots$ *assertions*	New schema *name* with tuple of definitions of variables x_i, with assertions about their values.
$name1$ $name0$	Schema inclusion: new schema *name1* with copies of all definitions and assertions from *name0*
$name1$ $\Delta name0$	New schema *name1* with two copies of all definitions and assertions from *name0*, one identical and one with all variables primed (').
$name1$ $\Xi name0$	As Δ with additional assertions that all variables from *name0* are unchanged (primed version equals unprimed one).

B The Right Stuff: A Full Specification Without Commentary

This appendix gives a version of the specification without all the explanations of Z concepts, introductory material on how to write specifications. and exploration of (sometimes mistaken) alternatives. You could view it as the "final" version of the specification arising from the process detailed in earlier sections. It is (almost entirely) literal copies of earlier material, extracted by an `awk` script; any awkwardness in wording results from the need to use the same text in two slightly different contexts.

B.1 People

The central concept of this formalization is "people."

$$[PERSON]$$

Type *PERSON* is the set of all people we'd ever want to represent in the eventual software system. Schema *People* is the basic state of the software: a representation of a specific set of *PERSON*s and their parents.

People

$$\begin{array}{l} people : \mathbb{P}\, PERSON \\ father\_of : people \pfun people \\ mother\_of : people \pfun people \end{array}$$

- *people* is the set of *PERSON*s currently known to the genealogy system.
- Each person has at most one father recorded.
- Each person has at most one mother recorded.

B.2 Common Ancestors

The first level of defining ancestors is immediate ancestors (parents) and immediate descendants (children).

Parent

$$\begin{array}{l} People \\ parent : people \leftrightarrow people \\ child : people \leftrightarrow people \\ \hline parent = mother\_of \cup father\_of \\ child = parent^{\sim} \end{array}$$

- Each person has have zero or more parents and zero or more children recorded.
- A parent is a mother or a father.
- If one person is another's parent, the second is the first's child.

The natural definition of "ancestor" is a parent, or a parent of a parent, and so on for as many steps as the available data provide. Some specifications are simpler using a zero-step ancestor ("self").

$$\begin{array}{l}
\textit{Ancestor} \\
\hline
\textit{Parent} \\
\textit{ancestor} : \textit{people} \leftrightarrow \textit{people} \\
\textit{ancestor}_0 : \textit{people} \leftrightarrow \textit{people} \\
\hline
\textit{ancestor} = \textit{parent}^+ \\
\textit{ancestor}_0 = \textit{parent}\star \\
\forall\, p : \textit{people} \bullet p \mapsto p \notin \textit{ancestor} \\
\hline
\end{array}$$

- Someone's *ancestor* is anyone found by following parent relations repeatedly (at least once).
- Someone's $\textit{ancestor}_0$ is either an ancestor or emself.
- No one is eir own ancestor.

The natural-language definition of a common ancestor is straightforward: given two people, a common ancestor is anyone who is an ancestor to both.

$$\begin{array}{l}
\textit{CommonAncestor} \\
\hline
\textit{Ancestor} \\
\textit{common\_ancestors\_of} : \textit{people} \times \textit{people} \rightarrow\!\!\!\!\!\!+\; \mathbb{P}\,\textit{people} \\
\hline
\forall\, p_1, p_2 : \textit{people} \bullet \textit{common\_ancestors\_of}(p_1, p_2) = \\
\qquad \textit{ancestor}_0 (\!|\ \{p_1\}\ |\!) \cap \textit{ancestor}_0 (\!|\ \{p_2\}\ |\!) \\
\hline
\end{array}$$

- "Common ancestors" of two people are ancestors of both.

B.3 Interlude: Dealing with Names

Genealogy involves relationships between people, but research starts with names. Unfortunately the genealogist quickly finds that different people have the same name. I'm evading such issues by assuming that each *PERSON* is completely unique and that no two *PERSON*s indicate the same real-world person. To deal with names we introduce

$[\textit{NAME}]$

- There is a set of *NAME*s, the details of which are (currently) outside the scope of the specification.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Names}} \\ \hline
\mathit{People} \\
\mathit{names} : \mathbb{P}\, \mathit{NAME} \\
\mathit{has\_name} : \mathit{people} \leftrightarrow \mathit{names} \\ \hline
\end{array}$$

- The software will record some set of names.

- There is an unconstrained relation, *has_name*, between people and names.

B.4 Ancestry Paths

Naming the relationship between two people requires tracing paths through the ancestry relation. Defining nearest common ancestors (and possible generalizations of "nearest") requires knowing at least the lengths of such paths. Thus we define the set *ancestry_paths* of all paths from each person to all eir ancestors.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AncestryPaths}} \\ \hline
\mathit{CommonAncestor} \\
\mathit{ancestry\_paths} : \mathbb{P}(\mathrm{iseq}_1\, \mathit{people}) \\ \hline
\forall\, s : \mathit{ancestry\_paths};\ i : \mathbb{N} \bullet i \in 2 \mathinner{\ldotp\ldotp} \#s \wedge \\
\qquad s(i) \in \mathit{parent}\llparenthesis\, \{s(i-1)\}\, \rrparenthesis \\
\mathit{ancestry\_paths} = \{p_0, p_1 : \mathit{people};\ s : \mathrm{iseq}_1\, \mathit{people} \\
\qquad \mid p_0 \in \mathit{ancestor}_0\llparenthesis\, \{p_1\}\, \rrparenthesis \wedge s(1) = p_1 \wedge s(\#s) = p_0 \\
\qquad \bullet\ s\} \\ \hline
\end{array}$$

- An ancestry path is a non-empty sequence of people.

- In an ancestry path, each element is a parent of the previous element.

- There is an ancestry path from each person to each of eir "ancestors" (including emself).

B.5 The Genealogical Relationship Problem

Section 2.2 on page 11 identified two sub-problems: finding (generalized) nearest common ancestors (the subject of this section), and naming the resulting relationships (Section B.6 on page 79). We also have two choices of how to define appropriate common ancestors based on whether to accept George III as a common ancestor of Elizabeth II and Philip:

- Philip (6 steps): Alice of Battenberg, Victoria of Hesse, Princess Alice, Queen Victoria, Edward Duke of Kent, George III
- Elizabeth (5 steps): George VI, Mary of Teck, Mary Adelaide, Prince Adolphus, George III

A priori there does not seem to be any technical reason to prefer either alternative; it would be entirely up to the client being interviewed during requirements elicitation. During analysis it is wise to specify both alternatives, and during implementation to permit either based on end-user preferences.

B.5.1 Common Properties

Any way of defining nearest common ancestor would correspond to assertions about "acceptable" pairs of ancestry paths. Thus one acceptable path pair for Elizabeth and Philip is

- Elizabeth: George VI, George V, Edward VII, Queen Victoria.
- Philip: Alice of Battenberg, Victoria of Hesse, Princess Alice, Queen Victoria

Adding Edward of Kent to both paths wouldn't be acceptable; it is still a pair of ancestry paths leading to a common ancestor, but it isn't "nearest" in any sense. Traversing past a common ancestor such as Victoria would be appropriate if we want to allow George III. Both possible definitions have some common properties.

$$
\begin{array}{|l}
\textit{CommonAncestorPaths} \\
\hline
\textit{AncestryPaths} \\
\textit{common\_ancestor\_paths} : \textit{ancestry\_paths} \leftrightarrow \textit{ancestry\_paths} \\
\hline
\forall s_1, s_2 : \textit{ancestry\_paths} \mid (s_1, s_2) \in \textit{common\_ancestor\_paths} \\
\qquad \bullet\ s_1(\# s_1) = s_2(\# s_2) \\
\qquad \land\ s_1 (\!| 1 \mathbin{..} \# s_1 - 1 |\!) \cap s_2 (\!| 1 \mathbin{..} \# s_2 - 1 |\!) = \varnothing \\
\hline
\end{array}
$$

- "Common ancestor paths" are pairs of ancestry paths.[71]
- In every pair, the last element of both sequences is the same person (the common ancestor).
- The two sequences share no other elements besides the last.

[71] Ancestry paths are defined in Section 3.6 on page 22.

B.5.2 Approach 1: No Other Common Ancestor

My first idea for defining “common ancestors” was:

> An appropriate pair of paths has no common ancestors on either path, except the last.

This rules out the relationship based on George III, since Victoria is a common ancestor.

$$
\begin{array}{|l}
\hline
\textit{NoOtherAncestor} \\
\textit{CommonAncestorPaths} \\
\hline
\forall s_1, s_2 : \mathrm{iseq}_1\, \textit{people};\ p_1, p_2 : \textit{people} \\
\quad \mid p_1 = s_1(1) \land p_2 = s_2(1) \land \\
\qquad (s_1, s_2) \in \textit{common\_ancestor\_paths} \\
\quad \bullet\ (s_1(\!|\, 1 \mathrel{..} \# s_1 - 1 \,|\!) \cup s_2(\!|\, 1 \mathrel{..} \# s_2 - 1 \,|\!)) \\
\qquad \cap \textit{common\_ancestors\_of}\,(p_1, p_2) = \varnothing \\
\hline
\end{array}
$$

The best natural language summary is the quote at the start of the section. Taken line-by-line, a direct reading of the assertion is:

- For every pair of sequences s_1 and s_2
- with 1^{st} elements p_1 and p_2 respectively
- for which s_1 and s_2 are a pair of “common ancestor paths,” it is the case that
  - the elements of both sequences, except the last,
  - are not common ancestors of p_1 and p_2.

B.5.3 Approach 2: One Child Not a Common Ancestor

Even if we want to generalize “nearest common ancestor” to allow George III, we wouldn’t want to allow extending both paths with George II; the longer pair don’t give us any additional useful information. One thing that made this example work is that there was a way to get from Elizabeth to George III without going through Victoria; Victoria wasn’t on *both* paths. George II would make sense as an “appropriate common ancestor” only if he had some hypothetical child who was an ancestor of one without being an ancestor of the other, which would give yet another distinct pair of paths.

So, perhaps an appropriate definition is that

$$\begin{array}{|l}
\hline
\mathit{OneChildNotCommon} \\
\mathit{CommonAncestorPaths} \\
\hline
\forall\, s_1, s_2 : \mathrm{iseq}_1\, \mathit{people};\ p_1, p_2 : \mathit{people};\ ca : \mathbb{P}\, \mathit{people} \\
\qquad \mid p_1 = s_1(1) \land p_2 = s_2(1) \land \\
\qquad\qquad (s_1, s_2) \in \mathit{common\_ancestor\_paths} \land \\
\qquad\qquad ca = \mathit{common\_ancestors\_of}\,(p_1, p_2) \\
\qquad \bullet\ (\ \#s_1 > 1 \Rightarrow s_1(\#s_1 - 1) \notin ca\ ) \lor \\
\qquad\qquad (\ \#s_2 > 1 \Rightarrow s_2(\#s_2 - 1) \notin ca\ ) \\
\hline
\end{array}$$

- In every ancestry path pair where at least one of the two sequences is of length 2 or greater, at least one of the children of the common ancestor (2^{nd} last element of the sequence) is not itself a common ancestor.

B.5.4 Synthesizing the Result

There were two choices for the distinguishing assertion about ancestry paths. Moreover, the assertions about names were in a separate schema that wasn't included in any of the schemas about acceptable ancestry paths. Thus we could define a complete set of ancestry-concept definitions as either

$$\begin{array}{|l}
\hline
\mathit{AncestryDefinition} \\
\mathit{OneChildNotCommon} \\
\mathit{Names} \\
\hline
\end{array}$$

or the alternative with *NoOtherAncestor*.

B.6 The Relationship Naming Problem

Section 2.2 on page 11 described how relationships are named in English. This section formalizes those requirements.

B.6.1 Gender

Since some relationship names depend on gender, we need a new type:

$$\mathit{GENDER} == \mathit{male} \mid \mathit{female}$$

- There are two possible genders, male and female.

$$\begin{array}{|l}
\hline
\mathit{Gender} \\
\hline
\mathit{People} \\
\mathit{gender\_of} : \mathit{people} \rightarrow \mathit{GENDER} \\
\hline
\forall\, p : \operatorname{ran} \mathit{father\_of} \bullet \mathit{gender\_of}(p) = \mathit{male} \\
\forall\, p : \operatorname{ran} \mathit{mother\_of} \bullet \mathit{gender\_of}(p) = \mathit{female} \\
\hline
\end{array}$$

- Every recorded person's gender is known.
- Every father ("range" of the *father_of* function) is male.
- Every mother ("range" of the *mother_of* function) is female.

Some later assertions need both ancestry and gender assertions; we can introduce

$$\begin{array}{|l}
\hline
\mathit{AncestrySpecification} \\
\hline
\mathit{AncestryDefinition} \\
\mathit{Gender} \\
\hline
\end{array}$$

B.6.2 Composing Relationship Names

For names of people I decided to avoid details (Section B.3 on page 75). For naming relationships, the whole point is to specify details; the issue is how much detail is appropriate. The words that make up relationship names can be represented via a "free type:"

$$\begin{array}{l}
\mathit{RELATION\_PART} ::= \mathit{grand} \mid \mathit{self} \mid \mathit{father} \mid \mathit{mother} \mid \mathit{parent} \\
\qquad \mid \mathit{son} \mid \mathit{daughter} \mid \mathit{child} \mid \mathit{brother} \mid \mathit{sister} \\
\qquad \mid \mathit{sibling} \mid \mathit{aunt} \mid \mathit{uncle} \\
\qquad \mid \mathit{cousin}\langle\!\langle \mathbb{N} \times \mathbb{N} \rangle\!\rangle \\
\qquad \mid \mathit{great}\langle\!\langle \mathbb{N} \rangle\!\rangle
\end{array}$$

A relation name is a sequence of such parts:

$$\mathit{RELATION\_NAME} == \operatorname{seq}_1 \mathit{RELATION\_PART}$$

B.6.3 Specific Relationship Names

We can specify the names for each relationship in separate schemas, each of which includes *RelationNamingBasics*. Each schema focuses on a particular closely-related set of special cases. A good order in which to tackle things is "simplest first," which, when numbers are involved, often means starting with 1 or 0 and working up. For example, the simplest case is for distance 0, when the "two people" are the same person.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Self}}\\
\textit{RelationNamingBasics}\\
\hline
\forall g : \textit{GENDER};\; p : \textit{people} \bullet \textit{relation\_phrase}(0, 0, g) = \langle \textit{self} \rangle\\
\hline
\end{array}$$

- If the distances between two given people and their common ancestor are both zero, they are the same person.

Direct ancestry potentially involves several occurrences of the word "great;" this suggests using a function that describes how many occurrences of "great" are appropriate, given how many steps there are between the descendant and the ancestor:

- 1 step: no prefix; the ancestor is father or mother.
- 2 steps: prefix "grand"
- 3 steps: prefix "great grand"
- n steps: prefix "great^{n-2} grand"

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CountingNames}}\\
\textit{RelationNamingBasics}\\
\textit{counting\_names} : \mathbb{N}_1 \rightarrow \textit{RELATION\_NAME}\\
\hline
\textit{counting\_names}(1) = \langle\rangle\\
\textit{counting\_names}(2) = \langle \textit{grand} \rangle\\
\forall n : \mathbb{N} \mid n > 2 \bullet\\
\qquad \textit{counting\_names}(n) = \langle \textit{great}(n-2), \textit{grand} \rangle\\
\hline
\end{array}$$

A pair of auxiliary functions can simplify the assertions:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ParentChildNames}} \\
\hline
\mathit{child\_name} : \mathit{GENDER} \rightarrow \mathit{RELATION\_PART} \\
\mathit{parent\_name} : \mathit{GENDER} \rightarrow \mathit{RELATION\_PART} \\
\hline
\mathit{child\_name}(\mathit{male}) = \mathit{son} \\
\mathit{child\_name}(\mathit{female}) = \mathit{daughter} \\
\mathit{parent\_name}(\mathit{male}) = \mathit{father} \\
\mathit{parent\_name}(\mathit{female}) = \mathit{mother} \\
\hline
\end{array}$$

Direct ancestors are those for which n_1 is zero (and n_2 isn't).

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DirectAncestor}} \\
\hline
\mathit{CountingNames} \\
\mathit{ParentChildNames} \\
\hline
\forall\, n_1, n_2 : \mathbb{N};\ g : \mathit{GENDER};\ rn : \mathit{RELATION\_NAME} \\
\qquad \mid rn = \mathit{relation\_phrase}(n_1, n_2, g) \\
\qquad \bullet\ n_1 = 0 \land n_2 > 0 \Rightarrow \\
\qquad\qquad rn = \mathit{counting\_names}(n_2) \frown \langle \mathit{parent\_name}(g) \rangle \\
\hline
\end{array}$$

- When p_1 is the common ancestor ($n_1 = 0$), the relation name ends with "mother or "father," preceded by an appropriate "great ..." phrase defined by n_2.

A "direct descendant" schema would reverse the roles of n_1 and n_2.

The most general ("everything else") case is cousinship.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Cousin}} \\
\hline
\mathit{CountingNames} \\
\hline
\forall\, n_1, n_2, mn, mx : \mathbb{N};\ g : \mathit{GENDER};\ rn : \mathit{RELATION\_NAME} \\
\qquad \mid mn = \mathit{min}\{n_1, n_2\} \land mx = \mathit{max}\{n_1, n_2\} \land mn > 1 \\
\qquad \bullet\ \mathit{relation\_phrase}(n_1, n_2, g) = \langle \mathit{cousin}(mn - 1, mx - mn) \rangle \\
\hline
\end{array}$$

- Cousinship is determined by mn and mx, the minimum and maximum of the two numbers.

- If both numbers are 2 or greater ($mn > 1$), the two people are cousins.

- $mn - 1$ is the degree; $mx - mn$ is the number of times removed.

C Common Mathematical Mistakes

Everyone makes unconscious assumptions: we presume something is true without being aware of it. For example, English-speakers once assumed "doctor" meant "male doctor." People writing mathematical specifications make unconscious (and wrong) assumptions, too. This section summarizes some common ones.

C.1 Translation Errors

Expressions in English sometimes mean different things from similar-seeming expressions in mathematics, so in formalizing the English we must be careful to match the two.

In common English, if two things of the same kind are mentioned, they are assumed distinct. Thus if we write the English equivalent of

$$\forall\, p_1, \ldots p_n : \mathit{person} \ldots$$

every p_i is distinct from every other; in mathematics, some might equal others. Either write the … so that it is true with duplicate *person*s, or specify that they are distinct. For a pair we write $p_1 \neq p_2$. For a long list, we use a Z idiom:

$$\mathrm{disjoint}\langle \{p_1\}, \ldots, \{p_n\} \rangle$$

"Or" in English is mutually exclusive: the phrase "A or B or C," if spoken by a user during requirements elicitation, likely means exactly one is true. "Or" in mathematics is inclusive: at least one of A, B, or C must be true, but they might all be true, too. For two possibilities, the proper translation of the English is

$$(A \lor B) \land \neg\, (A \land B)$$

This rapidly becomes unreadable for more than two possibilities. However, if the predicates $P_1, \ldots P_n$ correspond to sets $S_1, \ldots S_n$ with intuitive meanings (that is, their *characteristic sets* mean something natural to the user), in Z we can say

$$\mathrm{disjoint}\langle S_1, \ldots S_n \rangle$$

People sometimes think two conditions exclude each other when they actually don’t. Thus if $A \vee B$ is mathematically correct, the English explanation should read “A or B or both’ or “at least one of A or B or . . ..”’[72]

The word “some” implies a subset, and in common English a subset is smaller than the set. Thus “Some X are Y” should be translated $X \subset Y$. On the other hand, the imprecision of natural language usage means that perhaps all X could be Y: $X \subseteq Y$; during elicitation the analyst should have the user clarify this detail.

C.2 Technical Errors

An *off-by-one* error uses a number n when $n \pm 1$ is correct. Examples include $\mathbb{N}$ versus $\mathbb{N}_1$ or writing an assertion about a sequence s with indices from $1 \mathinner{\ldotp\ldotp} n$ instead of $2 \mathinner{\ldotp\ldotp} n$ or $1 \mathinner{\ldotp\ldotp} n-1$; another is assuming that $s(0)$ makes sense, when by definition only positive indices are legal.

A function is a kind of relation, a set of ordered pairs; writing $f(x) = y$ assumes there is exactly one pair in f whose first element is x. If f were in fact a general relation, we must write $f(\!|\ \{x\}\ |\!)$. I made this mistake with *name_of* in Section 4.2.

A subtle mistake is to define a set by saying each element has a certain property when we really should have said it is all the elements with that property. Thus instead of

$$\begin{array}{|l} S : \mathbb{P}\,PERSON \\ \hline \forall x : S \bullet P(x) \end{array}$$

it might be appropriate to say

$$S = \{y : PERSON \bullet P(x)\}$$

An example is the definition of *ancestry_paths* in Section 3.6 on page 24; it should have said that it was exactly the set of pairs of paths where each pair ended with the same person.

[72] This suggests that in the “mutually exclusive” case we might rephrase the English in the specification as “at most one of A or B or C” despite this seeming a little awkward to native speakers.

C.3 Formalization Errors

By “formalization error” I mean mistakes of trusting the elicited requirements to be accurate, ignoring possible ambiguities or rare possibilities. Technically “some” and “or” from previous sections are examples, albeit minor ones.

Elicited requirements may mistake the majority for the universe, with serious effects on the evolution of a program. One example was the number of genders (Section 5.1); another was the assumption that only gender of the first person was relevant to a relationship name. If requirements say “people are either male or female,” perhaps the formalization process could discover that “most people are either male or female.” Similar examples include lists of specific political parties (such as Republican and Democrat in the US,[73]), specific races in a survey, or specific units of measure. The formalization might wind up defining an initial set of possibilities and allowing for its expansion.

“Most” isn’t directly representable in the formalization, nor should it usually be:[74] a particular invocation of the final system might involve primarily (one might say “mostly”) the minority. It might be appropriate to add a footnote telling implementers that, given a choice of representations, they should pick one that is fastest for the majority case.

A formalization might fail to deal with absent information. For example, everyone has a biological mother or father, so we might write

$$mother\_of : people \rightarrow people$$

However, no genealogy system can represent more than a finite chain of parents, so $\rightarrow$ should be $\nrightarrow$. A more complex example was “half-sibling” (Section 5.4). We can define what it means to be a half-sibling – having exactly one parent in common – but given that some parental information might be absent, real-world data might say that two people are half-siblings without identifying their parents. This requires introducing *declared_half_sibling* versus *deduced_half_sibling*, with *half_sibling* as their union.

[73] The list of parties with seats in Parliament in Canada in 2012 (Bloc Québécois, Conservative, Green, Independent, Liberal, and New Democrat) might be long enough to suggest the need for a general scheme.

[74] If elicitation and analysis can discover a quantitative way to clarify the qualitative “most” or “some” the specification should of course record it somehow.

D Schema Summary

Tables 3 summarizes where to find each schema, what schemas (if any) included them, and whether inclusion was via Ξ or Δ or neither.

Table 3: Schema Summary

Schema	**Page**	**Included in**
AddMother	48	
AddName	48	
AddPerson	45	
Ancestor	19	CommonAncestor
AncestryDefinition	33	SpecifyGender (via Ξ)
AncestryPathDistinctness	27	
AncestryPaths	76	CommonAncestorPaths
AncestrySpecification	34	SpecifyGender (via Δ)
BasicCommonPaths	26	
BioParents	62	
BirthCertificate	68	
BirthData	68	
CommonAncestor	20	AncestryPaths
CommonAncestorPaths	77	OneChildNotCommon
CommonAncestorRelation	20	
CountingNames	81	DirectAncestor
Couples	62	
Cousin	42	
DirectAncestor	41	
ElementAncestryPath	27	
Gender	34	AncestrySpecification
GenderComponents		NewPerson1 (via Ξ)
GetCommon	57	
HalfSibling	65	
HalfSiblingRelation	64	
InitDatabase	44	
JustNames	51	
Matrilineal	67	
Names	21	JustNames (via Δ)
NamesComponents		Names1, NewPerson1 (via Ξ)

Table 3: Schema Summary (cont'd)

Schema	**Page**	**Included in**
NewPerson	47	
NoOtherAncestor	78	
NumberOfPeople	44	
OneChildNotCommon	31	AncestryDefinition
Parent	18	Ancestor
ParentChildNames	41	DirectAncestor
ParentComponents		NewPerson1 (via Ξ)
ParentCouples	62	
People	17	Parent
RelationNameOf	80	RelationNamingBasics
RelationNamingBasics	36	Self
Self	37	
SetsAncestryPath	27	
Sources	68	
SpecifyGender	46	
SwedishUncle	63	
VitalStatistics	66	
WithoutNames	51	JustNames (via Ξ)

References

[1] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. ISBN 0-13-215871-X.

[2] *The GEDCOM Standard Release 5.5.* Family History Department, Church of Jesus Christ of Later-Day Saints, 1996. `http://www.math.clemson.edu/~simms/genealogy/ll/gedcom55.pdf`.

[3] Robert A. Heinlein. All you zombies. *Magazine of Fantasy and Science Fiction*, March 1959.

[4] Robert A. Heinlein. *Time Enough for Love.* Ace, 1973. ISBN 0-7394-1944-7.

[5] Donald E. Knuth. *Surreal Numbers*. Addison-Wesley, 1974. ISBN 0-201-03812-9.

[6] C. W. Morris. Foundations of the theory of signs. *International Encyclopedia of Unified Science*, 1(2), 1938.

[7] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992. Now available on the Internet at http://spivey.oriel.ox.ac.uk/ mike/zrm/.

[8] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981. ISBN 0-13-709873-1.