Antti-Juhani Kaijanaho

The formal method known as B and a sketch for its implementation

Master's Thesis
in Information Technology (Software Engineering)
20th December 2002

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Abstract

Kaijanaho, Antti-Juhani
The formal method known as B and a sketch for its implementation / Antti-Juhani Kaijanaho
Jyväskylä: University of Jyväskylä, 2002
154 p.
Master's Thesis

This thesis provides a reconstruction of the B-method and sketches an implementation of its tool support.

For background, this work investigates the field of formal methods in general and the relevance of formal methods to software engineering in particular. Formal (first-order) logic is also considered: both its development and important points relevant to formal methods. Automated reasoning, particularly its theoretical limits as well as unification and resolution, is discussed.

The main part of this thesis is a systematic reconstruction of the B-method, starting from its version of untyped predicate calculus and typed set theory, continuing with the Generalized Substitution Language (GSL) and finishing with the Abstract Machine Notation (AMN). Specification, refinement and implementation of a simple example using the B-method is presented. Both validation and verification of specifications, refinements and implementations using the B-method is discussed.

The thesis concludes with a report of the current state of the effort (by the author) to implement the tool support of the B-method, as the Ebba Toolset. The main design decisions are discussed. The use of Unicode as the primary input encoding of AMN and GSL in Ebba is described.

ACM Categories and Subject Descriptors: F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs — mechanical verification, specification techniques, and B-method; D.2.4 [Software Engineering]: Software/Program Verification — formal methods; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic — set theory, and mechanical theorem proving;
Keywords: formal logic — history

Tiivistelmä

Kaijanaho, Antti-Juhani
Formaali menetelmä nimeltä B ja sen toteutuksen hahmotelma / Antti-Juhani Kaijanaho
Jyväskylä: Jyväskylän yliopisto, 2002
154 s.
pro gradu -tutkielma

Tässä pro gradu -työssä tarkastellaan B-menetelmää ja hahmotellaan sen työkalutuen toteutusta.

Työn taustaksi tarkastellaan formaalien menetelmien kokonaisuutta sekä relevanssia ohjelmistotekniikassa, muodollisen logiikan kehitystä ja formaaleille menetelmille tärkeitä tuloksia sekä automaattisen päättelyn teoreettisia rajoituksia ja samaistus- ja resoluutiotekniikoita.

Työn pääasiallinen sisältö on B-menetelmän rekonstruointi alkaen sen käyttämästä logiikan ja joukko-opin versiosta, jatkaen korvausten yleistämisellä ja päättäen abstraktien koneiden määrittelyyn, tarkennukseen ja toteutukseen. B-menetelmällä tehtyjen määrittelyjen, tarkennusten ja toteutusten validointia ja verifiointia tarkastellaan. Esimerkkinä käydään läpi yksinkertaisen ongelman ratkaisun määrittely, tarkennus ja toteutus B-menetelmällä.

Työn lopuksi esitellään tekijän yritystä toteuttaa B-menetelmän tarvitsemat työkalut ohjelmistona nimeltä The Ebba Toolset. Tärkeimmät suunnittelupäätökset käydään läpi. Unicoden käyttö pääasiallisena syötemenetelmänä esitellään.

Asiasanat (YSA): kuvauskielet — Abstract Machine Notation, kuvauskielet — Generalized Substitution Language, formaalinen logiikka — historia, ohjelmistotekniikka — matemaattiset menetelmät — B, atk-ohjelmat — Ebba
Avainsanat: formaalit menetelmät — B, automaattinen päättely

Acknowledgements

I would like to thank my thesis supervisor, Tommi Kärkkäinen, for his advice and guidance throughout my studies of formal methods, including the production of this thesis. I am grateful to my friends and coworkers, Tuukka Hastrup, Janne V. Kujala and Jonne Itkonen for their comments on the manuscript of this thesis in its various forms, and for interesting discussions on and off topic.

Several acquaintances from Internet Relay Chat — especially Moshe Zadka, Ian Lynagh and Ganesh Sittampalam — have given me invaluable feedback on various (partial and complete) drafts of this thesis.

I would also like to thank Lauri Kahanpää, from the Department of Mathematics, for guiding my self-studies in mathematics before my entering the University and also supporting (with kind words and advice) my studies in University, first in Mathematics, and now later in Information Technology. He also deserves extra mention for introducing me into the world of formal logic.

I would like to thank the rest of the Department staff (especially certain people — you know who you are) for believing in me.

My teachers in Tampere Rudolf Steiner School, especially Timo Poranen (mathematics) and Esko Tallgren (physics), and my teachers in Jyväskylä Rudolf Steiner School, especially Lea Blåfield, deserve thanks for guiding me during my formative years (1984–1993 in Jyväskylä and 1993-1997 in Tampere).

My mother Maija Tuomaala and my father Kari Kaijanaho have always supported me in whatever I have been doing. Mother, Father — *thank you*.

Finally, I'd like to express my gratitude to Jean-Raymond Abrial — who I have never met except through his book [1] — for inventing the B method and thus saving me the work of doing that myself.

Jyväskylä, December 2002

Antti-Juhani Kaijanaho

Contents

List of Tables

1 Introduction

> *The modern magic, like the old, has its boastful practitioners: "I can write programs that control air traffic, intercept ballistic missiles, reconcile bank accounts, control production lines." To which the answer comes: "So can I, and so can any man, but do they work when you do write them?"*
>
> *— Frederick P. Brooks [16]*

This thesis documents research done by the present author in the past year. The original aim was to develop a full toolset for the formal method known as B for software engineering.

Formal methods apply mathematical methods and formalisms to software construction, in different phases of the development process. They can be used in the specification phase, where they help in making the specification precise and in identifying areas where the development team lacks understanding of the problem to be solved. Sometimes they are used in the code generation phase to guarantee the correctness of the code with respect to the specification. Generally, the use of formal methods, however insignificant, does have a positive effect on the reliability of the software in question, if other software engineering practices are not abandoned. Formal methods research has also helped mainstream software engineering to effect many changes for the better (structured programming being one example).

The B method is a formal method designed to be used in the specification and code generation phases of software development, and it has extensive tool support. It was originally developed by Jean-Raymond Abrial. Eventually, a company called B-Core was formed to commercialize the method and the original B toolkit [6]. Later, a French group developed another commercial toolset, Atelier B [113], again in close collaboration with Abrial. Both toolsets are proprietary, which makes them very bad starting points for independent research on the B method. Both are scarcely documented in scientific literature and in other freely available media. Without applying for a commercial or academic license for these products, it is nearly impossible to find out how they work. For the research described in this thesis, obtaining such a license was not an option.

The present author has a strong ethical commitment to free software[1]. According to this position, it is unethical to use or produce proprietary software. In practice, if

there is no free implementation of something (which is the case with the B method), one's options are essentially limited to abandoning it altogether, or writing one's own implementation. When doing that, it will be necessary to avoid referring to any proprietary materials, especially programs, in order to keep the legal situation clear. Otherwise, the program could be construed as being a derivative work in the sense of copyright law, and it would not be free software. This kind of an independent reimplementation is often called a *cleanroom* reimplementation (not to be confused with Cleanroom software engineering).

The original goal of a full reimplementation of B proved to be too ambitious, but a starting point for one has been realized in the Ebba toolset, which is described in the penultimate chapter. It is a cleanroom reimplementation: the only sources used were publically available documentation in the form of books (such as [1, 107]) and public web sites (such as [5]).

There were three incomplete attempts to write Ebba. The first implements an incomplete lexer and parser based on a Unicode representation of the B notation, along with a simple editor for this representation designed to run under a UTF-8-capable XTerm. The second includes an X-based rewrite of the editor but little of the other functionality. Both were written in C++. The third and final implementation is written in Haskell, and includes much more of the core functionality (typechecking, automated reasoning) than the previous ones but includes no editor.

The scientific contributions of this thesis are the following. First, it describes an independent reimplementation of the tools for the B method based on public sources; the problems encountered draw a picture of the completeness (or lack thereof) of the documentation of the B method. Second, as far as the present author knows, the early prototypes of Ebba were the first to employ a Unicode-based input format for B. Thus, this thesis contributes a design for this format. Finally, and less importantly, it provides an anecdotal analysis of the strengths and weaknesses of C++ compared to Haskell.

The thesis is roughly organized in two parts. The first part, Chapters 2–4, gives the general backround for the research discussed in this thesis: Chapter 2 is an introduction to formal methods in general and to their relationship to conventional software engineering; Chapter 3 is a historically motivated review of formal logic and set theory, culminating in two important results in the feasibility of mechanized logic; Chapter 4 introduces the basic mechanisms used in automated reasoning. The second part, Chapters 5 and 6, discusses the research itself: Chapter 5 is a sketch of a reconstruction of the B method from public sources, and Chapter 6 documents the

Ebba toolset and the process of its development.

The thesis contains lots of logical and set-theoretical formulae. The notation used in the first part is a variant of established notation. It is built rigorously in Appendix A, but the appendix is not a prerequisite for the rest of the thesis. The second part uses another variant, that which Abrial uses in his B-Book [1]; it is described in Chapter 5.

Due to the vastness of the subject matter (Chapter 5 alone covers much of the material of the 800-page B-Book [1]), it was necessary to omit a lot of material. Even though large parts of this thesis are mathematical in nature, no proofs are presented. Most of the mathematical propositions expressed or implied have been given proofs in the cited sources. The huge disproportionality between the available space and the size of the material has also had an adverse effect on the readability of the thesis: the whole thesis, but especially Chapter 5, introduces lots of new notation and concepts, and this may give a reader the feeling of being shot at with a machine gun. This is, unfortunately, unavoidable given the constraints of this thesis.

Notes

[1]The word "free" refers to freedom, not zero cost. Specifically, a piece of software is free, if the following freedoms are guaranteed to the user with respect to the software [112]:

0 the freedom to run the program, for any purpose;

1 the freedom to study how the program works, and adapt it to one's needs;

2 the freedom to redistribute copies (this is necessary so that one can fulfill the fundamental ethical maxim of helping those one is near: neighbours and relatives being the obvious examples); and

3 the freedom to improve the program, and release the improvements to the public (this is necessary so that one's work can benefit the whole community).

Access to source code is necessary to guarantee the freedoms 1 and 3.

Free software is sometimes called “Open Source Software”, but that term fails to communicate the ethical aspects of free software, which the current author considers essential.

2 Formal methods in software engineering

> *Professional Engineers are expected to use discipline, science, and mathematics to assure that their products are reliable and robust. We should expect no less of anyone who produces programs professionally.*
>
> — *David Lorge Parnas [91]*

This chapter discusses formal methods in general: first, the problems that they try to solve, and then what formal methods are and the range of existing formal methods.

2.1 The chronic software crisis

Software is in a crisis. This fact was acknowledged in the first NATO conference on software engineering [85] more than thirty years ago, and it is still said to be true (cf. eg. [37, 49, 73]). Software projects finish late, they run over budget and their products are unreliable [37]. The conference was held in 1968 as an attempt to address these problems. The two most lasting contributions of that conference are the term "software engineering" itself, which was deliberately chosen as provocative, and the definition of the term provided by Fritz Bauer:

> [Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.[1]

The understanding was that there is a "need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering." [85]

Now, thirty-four years since the conference, the software crisis, especially software unreliability, is still an acute problem (cf. eg. [54]). Several "silver bullets" have been proposed in the mean time, promising to end the crisis, but none have succeeded. Structured programming [33], literate programming [71], CASE tools (cf. [54]), object-orientation [13] and open source software [99] have all been said to deliver reliable software. Many of them have already contributed enormously to the state of

the art in software engineering — for example, Hoare [60] credits structured programming for the current relative clarity of contemporary source code. It is clear, however, that software failures are commonplace today [54].

2.2 Formal methods

Formal methods is one of those subdisciplines of software engineering that are claimed to be the ultimate silver bullet, a cure for all diseases in software production [54]. In the words of Frederick Brooks [16], *there is no silver bullet*. However, a study [96] suggests that the use of formal methods, when combined with effective testing, does actually have a dramatic effect on the number of bugs in a delivered software product, although a similar effect was not observed before testing. The most likely reason for this is that the formal specification led into relatively simple and independent components, which allowed for straightforward unit testing.

According to Rushby [102], the use of mathematics in design and construction to ensure product quality is common practice in established engineering disciplines, such as bridge or aircraft building, and even computer (hardware) construction, where one applies mathematically expressed physical and other natural laws to model a problem that deals with the behaviour of concrete systems in the physical world. These models are continuous in nature. In particular, small changes in the parameters usually generate relatively small changes in the models.

The problems of software engineering are, in contrast, discrete. The theoretical foundation on which all software development is laid is that of the random access machine (RAM)[2]. The state of the machine changes abruptly at each instruction, so there is no smoothness like in the changes of most physical systems. This creates a similar effect in software to that which is popularly known as the "butterfly effect" from chaos theory: small changes can cause large effects [121, p. 9]. A striking example of this is that a small typo in a program or in the specification of a program can cause a catastrophic failure, which was what happened to the Mariner 1 mission to Venus [50, 81, 82].

Since the domain of software engineering is discrete, it is apt that discrete mathematics, as well as modern logic, forms the basis of the mathematics of software engineering. Programs can be described as predicate transformers (see eg. [34] and Chapter 5), or as functions (see eg. [10]).

Formal methods are the mathematics of software engineering. The NASA formal methods guidebook [84] characterizes formal methods thus:

> Formal Methods cionsist of a set of techniques and tools based on mathematical modeling and formal logic that are used to specify and verify requirements and designs for computer systems and software.

It is important to note the two distinct meanings of the word "formal" in software engineering parlance: first, it is used to describe the nature of the software process and certain meetings common in proper software processes (eg. formal inspections); second, it is used to describe a manner of computation, where the meaning of symbols is intentionally ignored and all computation is done using specified computational rules. In this thesis, we will distinguish these two meanings by reserving the word "formal" for the second meaning; when we speak of the first meaning, we use the term "formalized".

John Rushby [102] suggests a four-level classification of the usage of formal methods in software development processes:

Level 0 Formal methods are not used. Most software development is on Level 0, but this need not be a bad thing. Even if formal methods are not used at all, the development processes themselves can be very formalized (see eg. [97]). Typical of this level of development are specifications written in natural languages such as English, the use of formalized meetings for code inspection, and extensive and well-planned testing.

Level 1 Concepts and notation from discrete mathematics is used as a substitute of natural language in specifications and other documents. All proofs and checks are done manually in a style reminiscent of the way mathematics is conventionally done, and also of the way mathematics is used in traditional engineering. Many formal methods (including Z [65]) started out at Level 1 and were then formalized for the benefit of tool support.

The advantage of this level of usage is that the notation is flexible: it is possible to invent on the spot new notation for concepts that cannot be easily expressed using existing notation. A disadvantage is that there is really no guarantee that the mathematical parts of specifications and other documents are any more correct (or even meaningful) than their natural language counterparts, since there are no tool support for checking syntactic or semantic errors in the documents. However, as Hoare [60] points out, people who are familiar with mathematical notation and methodology are usually capable of creating correct proofs — apart from trivial typos — efficiently by hand.

The Cleanroom process model (see eg. [76]) is an example of a Level 1 formal method.

Level 2 Formal specification languages with some automated tools are used. Usually, there are tools for checking the syntax and some semantic properties of formal documents written in some formal notation. These tools require committing to one concrete syntax and semantics for the notation, which sacrifices flexibility for partial guarantees of correctness.

Most proofs in a Level 2 formal method are done by hand without the aid of automation. However, the notation usually is accompanied by a fully-developed logic that can be used in proofs.

The Z notation [65] is a well-known example of a Level 2 formal method, although it is slowly working its way up to Level 3. Tools for it include CaDi$\mathbb{Z}$ [116] and ZETA [53], and many more. Both parse and typecheck documents written in Z. CaDi$\mathbb{Z}$ also contains some support for automated reasoning, and ZETA is able to animate specifications.

Level 3 Formal specification languages with extensive automated tools are used, including advanced tools for automated reasoning. Tool support for this level is sufficiently advanced so that all or almost all proofs can be conducted mechanically, or at least can be mechanically verified. This level of software development can be called "fully formal".

An example of a Level 3 formal method is the refinement calculus [9]. Tools for it include The Refinement Calculator [20] and PRT [26].

As the level of formality rises in a software development project, one gets better and better guarantees for correctness. However, no unconditional guarantees are possible. Formal software development at all levels is handicapped by having to translate informal requirements into a (more or less) formal specification, and it is never possible to prove the correctness of this translation. Additionally, reliance on automated tools in the higher levels of formality induces a new risk: that of accidental or intentional incorrectness of the tools. The tools themselves are going to have bugs, and trying to ensure their correctness using formal methods will not eliminate that risk completely.

There are also no guarantees that the tools used with a formal method are free of malicious misfeatures. As Ken Thompson [115] has pointed out, malicious misfeatures can be intentionally planted in any software development toolchain so that

their presence is hard to detect and their absence cannot be proven mechanically. In practice, however, most of them will be detectable and removable by software similar to standard antivirus software once the misfeatures become widely circulated and their existence becomes known.

According to Rushby [102], higher levels of formality can cause increased costs, since one will have to formalize things that one takes for granted in lower levels. This may become a lesser issue once formal methods become more widely accepted and verified libraries of formalizations become generally available.

2.3 The Capability Maturity Model

The level of formality in software development is independent of the level of formalization in the processes of software development. Most formal methods can be used in any kind of environment, but, as always, the best results can be expected from an organization which has very formalized processes. A common measure of formalization is the Capability Maturity Model (CMM) [92]), which classifies organizations into five levels:

1. On the *initial* level, the organization produces software in an ad hoc way, and the processes can occasionally be even chaotic. Success on this level depends on individual effort.

2. On the *repeatable* level, basic project management processes exist and track cost, schedule and functionality. There is enough discipline to repeat prior successes in similar projects.

3. On the *defined* level, management and engineering processes are documented, standardized, integrated and used in all projects.

4. On the *managed* level, detailed measures of process and product quality are collected and the processes are understood and controlled quantitatively.

5. On the *optimizing* level, quantitative feedback from processes and innovative pilot projects is used to improve the processes.

Naturally, all levels require the successful adoption of previous levels.

Although the cleanroom software engineering methodology is classified at Level 1 of formality, it is quite possible for an organization to achieve level 4 of the CMM

measure of formalization using cleanroom. Thus, the levels of formality and formalization are two independent dimensions. Ideally, we would have tools that support methods which take part in a process, which would lead into a situation where an organization can be at Level 5 of CMM and on Level 3 in formality.

2.4 Classification of formal methods

Formal methods can be classified into two categories. The first category is *property-based methods*, which are based on the indirect specification of properties. Property-based methods are, in turn, divided into two subcategories: *axiomatic* and *algebraic methods*. The second category is *model-based methods*, which are based on forming a formal model of the software system.

2.4.1 Axiomatic methods Axiomatic treatment of software may well be the oldest kind of formal methods. The earliest paper on it seems to be by Floyd in 1967 [39], on which Hoare built the now famous Floyd-Hoare logic [58]. The basic construct in the logic is the Hoare triple

$$\{Q\}P\{R\},$$

which consists of a program statement P preceded by a precondition assertion Q and followed by a postcondition assertion R. The triple itself is an assertion that, if Q is true at the initiation of P, then R will be true at the end of P. Programs are specified by defining what shall hold at the end of the program being specified (ie. what is its postcondition) given certain assumptions about the input and the initial state of the program (ie. its precondition). Programs are proven correct by proving a Hoare triple consisting of the program, its specified precondition and its specified postcondition, using the axioms and inference rules of Floyd-Hoare logic. An example of an inference rule in Floyd-Hoare logic is the rule of iteration

$$\{I \wedge E\}\ P\ \{I\} \vdash \{I\}\ \textbf{while}\ E\ \textbf{do}\ P\ \textbf{od}\ \{\neg E \wedge I\},$$

which states that in order to prove that the loop **while** E **do** P **od** preserves the *loop invariant* I and makes the loop condition E false, it suffices to prove that the loop body P preserves the invariant I as long as the loop condition E is true at the beginning of the loop body.

The hard part of the axiomatic method lies in the fact that lots of assertions need

to be generated. For example, loop invariants need to be come up with. Fortunately, there have been attempts at automating this process since the 1970's (cf. eg. [38]).

In the 1970's Dijkstra developed his guarded commands and weakest precondion style for formal software development and programming language semantics [34]. The axiomatic style evolved and spawned also such things as Hoare's communicating sequential processes [59] and the refinement calculus [9]. All three have had an important effect on later formal methods.

2.4.2 Algebraic methods[3] Algebraic methods emerged in the 1970's. The basic idea, informally speaking, is to define a mathematical set of things in the following manner. First, some elements of the set are named as *constants*: for example, one could name two constants, **true** and **false**, in a set of truth values. Then, one names one or more *functions* (of different arities) between elements of this set: for example, for a set of truth values, it makes sense to name the function **nand** (also denoted by $|$ and called the Sheffer stroke), which maps pairs of truth values to truth values. Finally, one gives one or more *equations* that relate the functions and constants to each other. For example, in the set of truth values, one would have the following equations:

$$\begin{aligned} \mathbf{true} \mid \mathbf{true} &= \mathbf{false} \\ \mathbf{true} \mid \mathbf{false} &= \mathbf{true} \\ \mathbf{false} \mid \mathbf{true} &= \mathbf{true} \\ \mathbf{false} \mid \mathbf{false} &= \mathbf{true} \end{aligned}$$

Now we sever the connection between the set we are describing with these constants, functions and equations, and call the system which consists of

- the names of the constants,
- the names and the signature of the functions, and
- the equations

an *abstract data type* or an *ADT*. In principle, any set of things complemented with a set of functions between its elements and a mapping (called an *interpretation*) of the ADT names to the actual elements and functions, satisfying certain requirements, is a valid implementation of the ADT. The requirement is that the constants must

be mapped to distinct elements of the set and the ADT function names must be mapped to the actual functions in such a way that the equations hold.

For example, the set $\{0, 1\}$ complemented by the function $(a, b) \mapsto 1-ab$ does form an implementation of the ADT of truth values described above under the interpretation where **true** maps to 1, **false** maps to 0 and **nand** maps to the given function, since the equations hold:

$$1 - 1 \cdot 1 = 0$$
$$1 - 1 \cdot 0 = 1$$
$$1 - 0 \cdot 1 = 1$$
$$1 - 0 \cdot 0 = 1$$

Another valid implementation is the set of integers $\mathbb{Z}$ complemented by the above-mentioned function.

In algebraic methods, the ADTs are specifications and their implementations are programs. Program construction in algebraic methods is about finding executable implementations to the ADTs of the specification. Program verification is proving that the proposed implementations are really implementations of the ADTs.

An example of the algebraic method is the Extended ML language [67], which allows the specification of programs and their refinement, leading to an implementation in Standard ML.

2.4.3 Model-based methods The basic idea of model-based specification of ADTs is to build an abstract mathematical structure that models the ADT together with a set of operations whose behaviour with respect to the model is described, for example, using preconditions and postconditions.

Typically, a model-based specification of an ADT contains zero or more abstract variables, an invariant which constrains the possible values of the variables (more generally than the equations used in algebraic methods), an initialization operation and other operations. The specifics vary depending on the actual method.

The best known model-based methods are Z [65] and the Vienna Development Method (VDM) [62]. The B method discussed in Chapters 5 and 6 is also model-based.

Notes

[1]Curiously, this quote does not seem to be in the conference report [85], although several textbooks (at least Pressman [97] and Vliet [121]) cite the report as the source for this definition.

[2]A RAM has countably infinite number of registers X_i ($i \geq 0$), any of which can store any integer whatsoever. The instruction set of a RAM consists of $X_i \leftarrow c$ (constant store), $X_i \leftarrow X_j + X_k$ (summation), $X_i \leftarrow X_j - X_k$ (subtraction), $X_i \leftarrow X_{X_j}$ (indirect access), $X_{X_i} \leftarrow X_j$ (indirect store), **if** $X_i \geq 0$ **goto** m (conditional jump to line m), **read** X_i (input) and **write** X_i (output). Negative addresses in indirect access and store halt the machine. RAMs are computationally equivalent to Turing machines. [28, 109]

[3]This exposition of algebraic methods is a simplified synthesis by the present author. A more rigorous treatment is given, for example, in Ehrig and Mahr's book [36].

3 A historically motivated review of formal logic and set theory

> *Hardly anything more unfortunate can befall a scientific writer than to have one of the foundations of his edifice shaken after the work is finished.*
>
> *— Gottlob Frege on Russell's paradox [45]*

The B method, and formal methods in general, are based on formal logic and set theory. This chapter traces the important points of development of classical modern formal logic. When we talk about "classical" logic, we exclude modal and deontic logic and other such more philosophical logics. By modern, we mean logic as it has been developed since the 1850s. By formal, we mean logic where all deductions can be mechanically verified and sometimes even mechanically generated.

Until the twentieth century, logic and the theory of classes (sets) have been understood as pretty much the same thing. Consequently, this chapter also traces the development of set theory.

We will conclude with an overview of two important results in effective computability: Gödel's result on the undecidability of arithmetic and Church's result on the undecidability of first-order logic.

In composing this chapter up to Frege, von Wright's treatises on analytical philosophy [127, 129] have been greatly useful. Jech's book on set theory [66] helped in tracing the history of the ZF theory, and a term paper by Dirk Schlimm [106] was helpful in tracing the development of primitive recursion.

We assume that the reader is familiar with the basic notions of modern logic. The variant of modern formal notation used is discussed at length in Appendix A.

3.1 Aristotle's syllogisms

It could be said that Aristotle founded formal logic. He wrote six texts which were later collectively named "Organon", instrument, based on the idea that they together formed a common thinking tool for philosophers working on diverse subjects. The organon consists of "Categories", "On interpretation", "Prior analytics", "Posterior analytics", "Topics" and "On sophistical refutations".

In Prior analytics [4][1], Aristotle introduced the concept of a syllogism. Syllogisms deal with propositions of the forms "A is predicated of every B", "A is predicated of no B", "A is predicated of some B" and "A is not predicated of some B". In these propositions, the "A" is called a predicate and "B" is called a subject. For example, in the proposition "Mortality is predicated of every man", "mortality" is the predicate and "man" is the subject. The phrase "is predicated of" can be understood as class membership or subclassness, and the subjects and the predicates as classes; thus, the example above could be understood as "the class of all men is a subclass of the class of mortals", or equivalently, "all men are mortal".

A syllogism is a form of inference consisting of three propositions: two premisses[2] and a conclusion. An example would be

Mortality is predicated of every man	(ie. men are mortal)
Manhood is predicated of some Athenians	(ie. some Athenians are men)
Mortality is predicated of some Athenians	(ie. some Athenians are mortal)

Medieval logicians started the practice of labelling syllogisms systematically. Each proposition is labelled with a three-letter acronym, where the first letter (uppercase) denotes the predicate and the third letter (also uppercase) denotes the subject. The middle letter (lowercase) is either "a", "e", "i" or "o", and it denotes the logical constants ("is predicated of", "every" and so on) of the proposition. The acronym "AaB" refers to a proposition of the form "A is predicated of every B", the acronym "AeB" refers to "A is predicated of no B", the acronym "AiB" refers to "A is predicated of some B", and the acronym "AoB" refers to "A is not predicated of B". The syllogisms are represented by schemata where the premisses (denoted by three-letter acronyms) are separated by commas and followed by a semicolon and the conclusion (also denoted by a three-letter acronym). Thus, the above syllogism would be rendered as the schema "AaB, BiC; AiC".

Medieval logicians also gave mnemonic names to syllogisms. The vowels of a name are the lowercase letters of the syllogism schema, and the rest of the name is chosen to aid memory. For example, the syllogism above is called Darii.

What makes syllogistic logic *formal* is the fact that valid syllogisms are valid regardless of which particular predicates and subjects are used in them, as long as the *form* of the syllogism is explicit and obeyed. In the case of the Darii syllogism, the form states that the first premiss should have the form "A is predicated of every B", the second premiss should have the form "B is predicated of some C", and the conclusion should have the form "A is predicated of some C". As indicated by

Name	Form	Name	Form
Barbara	AaB, BaC; AaC	Baroco	BaA, BoC; AoC
Celarent	AeB, BaC; AeC	Darapti	AaB, CaB; AiC
Darii	AaB, BiC; AiC	Felapton	AeB, CaB; AoC
Ferio	AeB, BiC; AoC	Datisi	AaB, CiB; AiC
Cesare	BeA, BaC; AeC	Disamis	AoB, CaB; AiC
Camestres	BaA, BeC; AeC	Bocardo	AoB, CaB; AoC
Festino	BeA, BiC; AoC	Ferison	AeB, CiB; AoC

Table 3.1: Aristotle's syllogisms.

the choice of the uppercase letters, the predicate of the first premiss should be the predicate of the conclusion and the subject of the first premiss should be the predicate of the second premiss, for the syllogism to qualify as Darii. Altogether, Aristotle listed fourteen valid syllogisms; they are listed in Table 3.1.

Aristotle's Organon remained as the definitive books on logic for more than two millennia, until the nineteenth century. There were some work on logic after Aristotle, but nothing spectacular. In fact, Immanuel Kant wrote about logic in 1787 [70]: "...since Aristotle, it has been unable to advance a step and, thus, to all appearance has reached its completion."

3.2 Logic as a calculus

The first hints of the coming revolution in logic were given by Leibniz, who contributed a new perspective on it in the latter part of the seventeenth century. Aristotle's logic is, according to von Wright [127], an instance of the axiomatic method, which is exemplified by Euclid's geometry. The other corner stone of mathematics is calculation, the mechanical manipulation of symbols to produce a desired result, of which algebra is the prime example. Leibniz's idea was to make logic calculable, or in other words, create a calculus of logic.

For example, in *Rules from which a decision can be made, by means of numbers, about the validity of inferences and about the forms and moods of categorical syllogisms* [75], Leibniz proposes a kind of arithmetic of logic, where subjects and predicates of syllogistic propositions are denoted by pairs of relatively prime integers, one positive and one negative. The idea was to calculate the truth of a universal affirmative propo-

sition (such as "Every wise man is pious") by dividing the positive number of the subject by the positive number of the predicate, and respectively for the negative numbers. If there are no remainders, then the proposition was to be true. Leibniz developed also other numerical and symbolic calculi for logic.

Unfortunately, Leibniz's work on logic was mostly unknown to his contemporaries and early successors, although he was recognized as one of the great geniuses of philosophy and mathematics even while he lived. The reason seems to be that most of his written works were published posthumously. Only in the twentieth century was it finally recognized that Leibniz was perhaps the greatest logician after Aristotle [127]. This explains why Kant could give his remarks on nothing having happened in logic since Aristotle, some one hundred years after Leibniz.

In 1847, George Boole published the essay *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning* [14], where he discussed a calculus of logic (not to be confused with modern Boolean algebra[3]). He developed it further in the next years, and published a more mature version of it in 1854 as the book *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities* [15].

Boole's basic idea was to try and map logic to (high-school) algebra. In his logical calculus, letters such as x and y denote classes of things (such as "(all) men" or "(all) good (things)"), operation signs such as $+$, $-$ and $\times$ (which is often omitted in formulae) represent operations that combine things to form new things, and the identity sign $=$ represents equality. From these building blocks formulae are constructed in the familiar way of high-school algebra. Multiplication (xy or $x \times y$) denotes the class of those things where all the formulas apply; for example, the formula xy denotes the class of things where both x and y apply. In modern terms, then, juxtaposition denotes class intersection. For example, if x denotes "(all) good (things)" and y denotes "(all) men", then xy denotes "(all) good men". As a special case, xx (or "(all) good (things) (that are) good (things)") denotes the same thing as x, and so we may as well write $x^2 = x$. Thus, we can start building an algebra of classes.

Boole noted that this algebra of logic or classes (which, for him, was the same thing) behaves identically to the algebra of numbers: addition (denoting "and" or "or", does not matter which) is commutable and associative. The multiplication (juxtaposition) described above and addition are also distributive: if x denotes "men", y denotes "women" and z denotes "European", then $z(x + y)$ would denote "European men and women", which naturally is the same thing as "European men

Modern name	Modern notation	Boole's notation
negation	$\neg x$	$1 - x$
conjunction	$x \wedge y$	xy
(inclusive) disjunction	$x \vee y$	$x + y - xy$
exclusive disjunction	$(x \vee y) \wedge \neg(x \wedge y)$	$x(1-y) + y(1-x)$
implication	$x \rightarrow y$	$1 - x + xy$
equivalence	$x \leftrightarrow y$	$1 - x(1-y) - y(1-x)$

Table 3.2: Summary (see [127]) of Boole's logical calculus [14, 15].

and European women", or $zx + zy$. In the same manner, one can continue building this algebra. The algebra would still work like normal algebra of numbers, except that one peculiar thing: $x^n = x$ for all natural numbers n. It will be possible to calculate logic using this algebra in the same manner as one calculates with numbers.

Von Wright summarized the relation of Boole's algebra to modern logical terminology [127]; the summary is reproduced here in Table 3.2[4].

According to Burris [19], Boole's discovery was not well received. Most of his contemporaries did not understand it — it seemed to give correct results but nobody knew why. For a long time, authors mistreated Boole's work, and as a consequence, what is currently known as Boolean algebra bears little resemblance to Boole's actual work.

3.3 Frege's successful failure

The first truly modern writing on logic was Gottlob Frege's *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought* [43], published in 1879. Frege's notation is idiosyncratic from the modern point of view (see Figure 3.1 for an example). Nevertheless, it is the first logical system that could be called modern. *Begriffsschrift* includes notational elements for implication, negation and universal quantification, which together are enough to form the full system of predicate logic. Frege's aim was to develop a logical theory of mathematics, in essence show that mathematics is based on logic. He later developed this idea in his two books, *Grundlagen der Arithmetik* [41] and *Grundgesetze der Arithmetik begriffsschriftlich abgeleitet* [42].

In *Grundgesetze* (and in some degrees, in his earlier works), Frege attempted to construct an Euclidean theory of arithmetic: to create a work where mathematics is

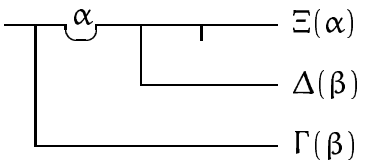

Figure 3.1: Frege's *Begriffsschrift*-notation for the predicate which would be denoted by $\Gamma(\beta) \rightarrow \forall\alpha\colon (\Delta(\beta) \rightarrow \neg\Xi(\alpha))$ in modern notation.

derived out of a couple of axioms and explicitly given rules of inference. However, when the second volume of *Grundgesetze* was already in press, Bertrand Russell wrote to him about a fundamental flaw in his system.

In *Begriffsschrift* [43], Frege abandons the traditional Aristotelian terminology of *subject* and *predicate*, preferring instead *argument* and *function*. For example, in the Fregian assertion $\vdash \Phi(A)$[5], we call $\Phi(\xi)$ the *function* (the Aristotelian *predicate*) and A the *argument* (the Aristotelian *subject*). He even goes so far as to allow the function part (Φ in the example) to be variable — in Frege's words, "we can also regard $\Phi(A)$ as a function of the argument Φ" [43, p. 24]. It should be noted that in modern terms, Frege's *function* is a *predicate*, while his *argument* is still called *argument*. Thus, his functions are truth-functions[6].

The famous Russell's antinomy (also known as Russell's paradox) is the following [103]. Let Φ be the function described as "$\Phi(\xi)$ is true if $\xi(\xi)$ is false".[7] Now, is $\Phi(\Phi)$ true or false? An alternative formulation given by Russell in his letter to Frege is more familiar to modern people: "Likewise, there is no class (as a totality) of those classes which, each taken as a totality, do not belong in themselves."[8]

The discovery of Russell's antinomy shook the basis of modern logic as it was about to be formed. New theories of classes and logic would have to take it into account. Frege attempts to salvage his work in an appendix that he added at the last moment to the second volume of his *Grundgesetze*. Russell himself later proposed type theory as a solution in *Principia mathematica* [122] and other writings (such as *Mathematical logic as based on the theory of types* [104]). His basic idea is that objects are grouped into a hierarchy of types: individual things are of the zeroth type, classes of individual types are of the first type, and, in general, classes of type n contain objects of type $n-1$. Russell allows only objects of type n to be members of objects of type $n+1$. The usual solution today is to use first-order logic, where this cannot happen, and then give axioms for set theory in it, usually using Zermelo and Fraenkel's axiom set, but other axiomatizations have been proposed (see Appendix A for an

example).

3.4 Set theory emerges

Giuseppe Peano started to work on founding mathematics on the basis of logic independently of Frege in the 1880s. His first work on the subject is his *Arithmetices principia, nova methodo exposita* [93], where he states his axioms. He does not handle them well from the formal point of view, though. He gives no rules of inference, and so his proofs are just lists of formulae, with nothing that connects them logically. He does introduce the separation of logic and classes (set theory), and he introduces much notation that is still used, such as the "membership" operator ε, which later evolved into $\in$.

Peano's original axioms were the following (written in modern notation):

1. $1 \in \mathbb{N}$
2. $a \in \mathbb{N} \to a = a$
3. $(a \in \mathbb{N} \land b \in \mathbb{N}) \to (a = b \leftrightarrow b = a)$
4. $(a \in \mathbb{N} \land b \in \mathbb{N}) \land c \in \mathbb{N} \to ((a = b \land b = c) \to a = c)$
5. $(a = b \land b \in \mathbb{N}) \to a \in \mathbb{N}$
6. $a \in \mathbb{N} \to a + 1 \in \mathbb{N}$
7. $(a \in \mathbb{N} \land b \in \mathbb{N}) \to (a = b \leftrightarrow a + 1 = b + 1)$
8. $a \in \mathbb{N} \to a + 1 \neq 1$
9. $1 \in k \land (\forall x \in \mathbb{N}\colon x \in k \to x + 1 \in k) \to \mathbb{N} \subset k$

Nowadays, axioms 2–5 would be omitted from the list on the grounds that they more properly belong to the underlying logical system. Note that $0 \notin \mathbb{N}$ according to this axiomatixation.

Earlier, in the 1870s, Georg Cantor was developing his ideas of infinity. Until the publication of his 1874 paper *Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen* [24], all infinities were regarded as being of the same size. Cantor proved that algebraic real numbers (real roots of polynomial equations) can be put into one-to-one correspondence with natural numbers, and vice versa, and also

that real numbers cannot be put into one-to-one correspondence with algebraic real numbers. Thus, there seemed to be different magnitudes of infinity. Cantor published in 1878 the paper *Ein Betrag zur Mannigfaltigkeitslehre* [25], where he suggested the now current terminology of the power of a set and the equipotence of sets: in this terminology, the set of algebraic real numbers is equipotent to the set of natural numbers but less in power than the set of real numbers. Eventually it would lead to the theory of aggregates (sets), and as an important part of it, to the concept of transfinite numbers.

Cantor defines the *cardinality* or the *cardinal number* of a multiplicity[9] as "the general notion that applies to it and to all multiplicities equivalent [equipotent] to it" [23]. Natural numbers comprise the finite cardinals, and the smallest infinite cardinal is $\aleph_0$. An ordered set has a *type*, which is "the general notion that applies to it and all ordered sets similar to it, and to these alone" [23], where similarity of sets means that the sets can be brought to a one-to-one correspondence where the order of the elements is preserved. Now, if all subsets of an ordered set have a smallest element, then the set is *well-ordered* and its type is called an *ordinal number*. Ordinal numbers are simply ordered [23].

Cantor's set theory is naïve in the sense that its fundamentals are not precise. Cantor defined an aggregate (set) as "any collection into a whole M of definite and separate objects m of our intuition or our thought" [22]. Antinomies, of which Russell's antinomy described above is a famous example, began to appear. Cesare Buriali-Forti published in 1897 two notes [18, 17] describing one that is perhaps the earliest of the modern antinomies. The crux of the antinomy is this: *What is the ordinal number of the set of all ordinal numbers?*

Cantor has proved that the set of ordinal numbers Ω is well-ordered. Thus, it has an ordinal number; we'll call it β. Now, since β is an ordinal number, it is a member of the set of ordinal numbers. Cantor has also proved that each ordinal number α is the ordinal number of the set of all ordinal numbers strictly smaller than α (including zero, which Cantor does not regard as an ordinal number). Now, due to this β is the ordinal number of the set Ω' of all ordinal numbers strictly smaller than β, which, naturally, does not include β itself. Thus, β is the cardinal number of both Ω and Ω', which is impossible since Ω' is not similar to Ω. We arrive thus into a contradiction [23].

Cantor's solution to the problem was to talk of *multiplicities* and *sets*. Every collection of things is a multiplicity, but only those multiplicities that can, without contradiction, be treated as objects in their own right, are sets. Thus, the multiplici-

ties Ω and Ω' of the previous paragraph are not actually sets. A similar distinction was made by von Neumann in 1925, when he distinguished between sets and classes [87]. The distinction is also made in Appendix A.

3.5 Interlude: Modern notation and terminology

The formal basis of logic and set theory was developed in the first decades of the twentieth century. Today, we talk about propositional (or sentential) logic, (first-order) predicate logic and set theory. A formal reconstruction of predicate logic and set theory is included as Appendix A. We will introduce all three here informally.

3.5.1 Propositional logic Propositional logic deals with the form of (certain kind of) declarative sentences called propositions. To denote arbitrary propositions (whose structure is not under discussion) we use *metasyntactic variables* (we will usually abbreviate that to *metavariable*). They are not part of the logic we are considering; rather they are part of the metalanguage, in this case English, that we use to describe the logic. When discussing propositional logic, metavariables denote arbitrary propositions. We will distinguish metavariables from the constructs of logic by typesetting them in boldface.

Propositions have one distinctive feature: in any possible world, each of them is either true or false (noth both at the same time). This is called its truth value. For any proposition $\mathbf{P}$, we denote its truth value as $[\![\mathbf{P}]\!]$. This is, of course, a function of the possible world under discussion; unless the context implies otherwise, an arbitrary possible world is assumed.

We set aside notation two special propositions: $\top$ (pronounced *verum*) denotes some proposition that is true in all possible worlds and $\bot$ (pronounced *falsum*) denotes some proposition that is false in all possible worlds (thus, $[\![\top]\!]$ denotes truth and $[\![\bot]\!]$ denotes falsity regardless of the possible world under discussion).

Given propositions $\mathbf{P}$ and $\mathbf{Q}$, we can form the compound propositions

- $\neg\mathbf{P}$ (pronounced *not* $\mathbf{P}$),
- $\mathbf{P} \wedge \mathbf{Q}$ (pronounced $\mathbf{P}$ *and* $\mathbf{Q}$),
- $\mathbf{P} \vee \mathbf{Q}$ (pronounced $\mathbf{P}$ *or* $\mathbf{Q}$),
- $\mathbf{P} \veebar \mathbf{Q}$ (pronounced $\mathbf{P}$ *xor (ex-or)* $\mathbf{Q}$),

$[\![\mathbf{P}]\!]$	$[\![\mathbf{Q}]\!]$	$[\![\mathbf{P} \to \mathbf{Q}]\!]$
true	true	true
true	false	false
false	true	true
false	false	true

Table 3.3: Truth table for implication.

- $\mathbf{P} \to \mathbf{Q}$ (pronounced **P** *only if* **Q**),
- $\mathbf{P} \leftrightarrow \mathbf{Q}$ (pronounced **P** *if and only if* **Q**),
- $\mathbf{P} \downarrow \mathbf{Q}$ (pronounced *neither* **P** *nor* **Q**) and
- $\mathbf{P} \mid \mathbf{Q}$ (pronounced *not both* **P** *and* **Q**).

A proposition which is not a compound proposition is called *atomic*.

The connectives $\neg$ (pronounced *negation*), $\wedge$ (pronounced *conjunction*), $\vee$ (pronounced *(inclusive) disjunction*), $\veebar$ (pronounced *exclusive disjunction*), $\to$ (pronounced *implication*), $\leftrightarrow$ (pronounced *equivalence*), $\downarrow$ (pronounced *Peirce's arrow* or *nor*) and $\mid$ (pronounced *Sheffer stroke* or *nand*) are *truth functions* in that they map a pair of truth values (or in the case of negation, a single truth value) into a truth value.

For most informal uses, the pronunciation guides are enough to describe the semantics of the compound propositions, but there are some caveats. First, disjunction is inclusive: If $[\![\mathbf{P}]\!] = [\![\top]\!]$ and $[\![\mathbf{Q}]\!] = [\![\top]\!]$, then $[\![\mathbf{P} \vee \mathbf{Q}]\!] = [\![\top]\!]$. In contrast, exclusive disjunction is, well, exclusive: If $[\![\mathbf{P}]\!] = [\![\top]\!]$ and $[\![\mathbf{Q}]\!] = [\![\top]\!]$, then $[\![\mathbf{P} \veebar \mathbf{Q}]\!] = [\![\bot]\!]$. In all other respects, inclusive and exclusive disjunction are identical. Second, a false proposition implies everything: if $[\![\mathbf{P}]\!] = [\![\bot]\!]$, then regardless of which proposition **Q** is, $[\![\mathbf{P} \to \mathbf{Q}]\!] = [\![\top]\!]$.

A precise definition of the semantics of connectives is usually given using truth tables[10]. Table 3.3 gives as an example the truth table of implication. In a truth table, each row gives one combination of truth values for the independent (unknown) component propositions (denoted by metavariables) and the corresponding truth value for the compound proposition. There is a row for each possible combination, 2^n rows in all if there are n independent unknown components.

There are five classes of propositions:

If P is ...	then $\neg P$ is...
tautological	contradictory
contradictory	tautological
contingent	contingent
satisfiable	refutable
refutable	satisfiable

Table 3.4: Relationships between different classes of propositions.

1. The class of *tautologies* consists of those propositions that are true in every possible world. Note that $\mathbf{P} \leftrightarrow \top$ if and only if **P** is a tautology. Tautologies are sometimes called *valid* propositions.

2. The class of *contradictions* consists of those propositions that are false in every possible world. Note that $\mathbf{P} \leftrightarrow \bot$ if and only if **P** is a contradiction.

3. The class of *contingencies* consists of propositions that are true in at least one possible world but not in all of them. Note that a proposition is contingent if and only if it is neither a tautology nor a contradiction. Most propositions fall into this category.

4. Classes 1 (the class of tautologies) and 3 (the class of contingencies) together form the class of *satisfiable* propositions. Satisfiable propositions are true in at least one possible world.

5. Classes 2 (the class of contradictions) and 3 (the class of contingencies) together form the class of *refutable* propositions. Refutable propositions are false in at least one possible world. Refutable propositions are sometimes called *invalid* propositions.

Tables 3.4 and 3.5 show how these five classes of propositions are related and how they complement each other.

Proof in propositional logic is a list of propositions that follows certain inference rules. A complete set of inference rules for (predicate and hence propositional) logic is given in Appendix A; here we skip them. What is worth to note here, however, is that tautologies and only tautologies can be proven (this follows from the consistency and completeness results for propositional logic). For this reason, the class of theorems (which is a synonym for provable propositions) is the same as the class of

If P is ...	then P is not...
tautological	refutable
contradictory	satisfiable
satisfiable	contradictory
refutable	tautological

Table 3.5: Complementary pairs of different classes of propositions.

tautologies. Conditional proof of a proposition **P** given the assumptions **Q** (a conjunction of the assumed propositions) is the (unconditional) proof of the proposition $\mathbf{Q} \rightarrow \mathbf{P}$.

3.5.2 Predicate logic Predicate logic can be seen as an extension of propositional logic. The extension takes place in four parts: first, expressions are defined; second, the form of atomary propositions is defined as relations between expressions; third, propositions are renamed (well-formed) formulae; and fourth, two new forms of nonatomary predicates is defined.

An *expression* is a description of a thing. In predicate logic, there are three kinds of expressions: *variables*, *constants* and *functions*. We usually typeset variables as lowercase greek letters and constants as lowercase latin letters. Functions map one or more things into another thing; we will write function expressions as follows: the name of the function (typeset like a constant) is followed in parentheses by the argument expressions separated by commas. The number of arguments required by a function (its *arity*) is, together with its name, part of its identity: if two functions have the same name but different arity, they are different functions. Constants can be regarded as nullary functions (functions whose arity is zero).

Sometimes binary functions are written differently: instead of $\mathbf{f}(\mathbf{a}, \mathbf{b})$ (prefix notation), one writes $(\mathbf{afb})$ (infix notation). Parentheses can be omitted if it does not endanger understandability. The difference is purely notational and has no effect on the meaning of an expression.

Atomic formulae are predicates. A predicate has a name and it takes zero or more expression arguments, like a function does. Unlike with functions, the name of a predicate is typeset as an uppercase latin letter. Otherwise, predicates have the same appearance as functions. Unary predicates (predicates whose arity is one) are sometimes called *properties*. There are two nullary predicates: $\top$ and $\bot$. Binary

predicates, like binary functions, are sometimes written in infix notation.

The semantics of a predicate is that of a relation: a predicate denotes a relation between things in a possible world. A predicate's truth value is true if and only if the things that its argument expressions denote are in the particular relation with each other that the predicate denotes. Thus, if P is a binary predicate denoting the relation *is the father of*, a is a constant denoting the current author, f is a unary function mapping people to their fathers, then in any possible world where the current author has been fathered, $[\![P(f(a), a)]\!]$ is true. Like with functions, the arity of a predicate is part of its identity.

It should be noted that the notion of semantics is more complicated in predicate logic than in propositional logic. For each possible world, there are many ways one can interpret constants, functions and predicates. But once the possible world and the interpretation are fixed, every construct of predicate logic has a fixed meaning: constants denote particular things, functions denote particular mappings of things to things and predicates denote relations between things. Like in propositional logic, the possible world and the interpretation are usually treated as arbitrary.

The concept of a metavariable needs to be reconsidered. In propositional logic every metavariable denotes a proposition. In predicate logic such simplicity won't do. Metavariables are needed to denote constants, function names, predicate names and variables. We will continue the convention that metavariables are typeset in boldface. Other properties of the metavariable decides what it denotes: a boldface uppercase latin letter denotes an arbitrary well-formed formula, a bolface lowercase latin letter denotes an arbitrary expression or an arbitrary function name (it will be clear from the context)

Sometimes, however, full arbitrarity is too much, and for that reason, a side condition given in plain English is sometimes attached to the use of a metavariable to constrain the possible denotations of the metavariable. However, within that constraint the choice of denotation is arbitrary. We call formulae and expressions that contain metavariables *formula schemata* and *expression schemata*, respectively.

It is customary to allow definitions. A definition is a metalinguistic mapping of a particular constant or a function (where its argument are expression metavariables) to an expression containing zero or more instances of the argument metavariables. A definition can also be a metalinguistic mapping of a particular predicate (where its arguments are expression metavariables) to a well-formed formula containing the argument metavariables. A constant, function or predicate for which there is a definition is called a *derived* construct. Constructs are not derived are called *primitive*

constructs.

In principle, whenever one sees a derived construct, one should substitute the expression or formula that it is mapped to in the definition for the derived constant, function or predicate. This allows, in principle, the complete removal of derived constructs from formulae. In practice, this is rarely done in full. Usually one just expands the definitions as necessary, leaving as many derived constructs as possible alone.

It should be noted that most of the connectives can be seen as derived constructs. For example,

- $\mathbf{P} \rightarrow \mathbf{Q}$ can be defined as $\neg\mathbf{P} \vee \mathbf{Q}$ and
- $\mathbf{P} \vee \mathbf{Q}$ can be defined as $\neg\mathbf{P} \downarrow \neg\mathbf{Q}$.

In fact, it suffices to leave only either Peirce's arrow or Sheffer stroke as primitive.

Two new kinds of compound formulae are added:

- $\forall\, \boldsymbol{\alpha}$: **P** (pronounced *for all* $\boldsymbol{\alpha}$, **P** *holds*) and
- $\exists\, \boldsymbol{\alpha}$: **P** (pronounced *there is an* $\boldsymbol{\alpha}$ *for which* **P** *holds*).

In the context of both formulae, the variable $\boldsymbol{\alpha}$ is said to be *bound* throughout **P**. A variable is said to have a *free* occurrence in a formula if it occurs in a formula and at least in one of its occurrences it is not bound.

The symbol $\forall$ is called the *universal quantifier*, and the symbol $\exists$ is called the *existential quantifier*. Both can be defined in terms of the other.

The five categories of propositions (tautological, contradictory, contingent, satisfiable, refutable) extend easily to formulae of predicate logic. There is only one caveat: when we talked about "possible worlds" before, we must now talk of "possible worlds and interpretations". For example, a tautology is true in all possible worlds and in all interpretations.

There are two additional types of compound formulae which are not commonly used in predicate logic but which we use freely in this thesis:

- $\lambda\, \boldsymbol{\alpha}$: **e** (pronounced *the function that maps* $\boldsymbol{\alpha}$ *to* **e**) and
- $\iota\boldsymbol{\alpha}$: **P** (pronounced *the unique* $\boldsymbol{\alpha}$ *for which* **P** *holds*).

In predicate logic these two would have to be primitive notions, but they can be defined in set theory as derived constructs (this has been done in Appendix A).

3.5.3 Sets First-order set theory is an extension of predicate logic much in the same way as predicate logic is an extension of propositional logic. The extension proceeds as follows.

We will perform some naming:

- Ω is a constant.
- $=$ and $\in$ are binary predicates.
- All expressions are called classes.
- Those classes $\mathbf{c}$ for which $\mathbf{c} \in \Omega$ is a tautology, are called sets. In other words Ω is the class of all sets.

We will define a new kind of an expression, called *class comprehension*. A class comprehension has the form $\{\, \boldsymbol{\alpha} \colon \mathbf{P} \,\}$.

We will restrict ourselves to consider only those possible worlds and interpretations where the following statements hold:

- $[\![\mathbf{e} = \mathbf{f}]\!]$ is true if and only if $[\![\mathbf{e}]\!]$ and $[\![\mathbf{f}]\!]$ contain the same elements.
- $[\![\mathbf{e} \in \mathbf{f}]\!]$ is true if and only if $[\![\mathbf{e}]\!]$ is an element of $[\![\mathbf{f}]\!]$.
- $[\![\mathbf{e}]\!]$ is an element of $[\![\{\, \boldsymbol{\alpha} \colon \mathbf{P} \,\}]\!]$ if and only if $[\![\mathbf{P}]\!]$ is true when all free occurrences of $\boldsymbol{\alpha}$ in $\mathbf{P}$ are replaced by $\mathbf{e}$.

There are several new (auxiliary) constants, functions and predicates:

- $\varnothing$ (*the empty class*), the class that contains no things,
- $\mathbf{e} \cup \mathbf{f}$ (*union*), the class of all things that are in $\mathbf{e}$ or $\mathbf{f}$,
- $\mathbf{e} \cap \mathbf{f}$ (*intersection*), the class of all things that are in both $\mathbf{e}$ and $\mathbf{f}$,
- $\mathbf{e} \setminus \mathbf{f}$ (*class difference*), the class of all things that are in $\mathbf{e}$ but not in $\mathbf{f}$,
- $\mathbf{e} \times \mathbf{f}$ (*Cartesian product*), the class of all pairs whose first element is in $\mathbf{e}$ and whose second element is in $\mathbf{f}$, and
- $\mathbf{e} \subset \mathbf{f}$ (*subclassness*), pronounced *all elements of* $\mathbf{e}$ *are in* $\mathbf{f}$ or $\mathbf{e}$ *is a subclass of* $\mathbf{f}$, and
- $2^{\mathbf{e}}$ (*power class*) the class of all subsets of $\mathbf{e}$.

Sets can encode arithmetic, so also the familiar numerals and arithmetic operators are available.

We will now actively forget the meaning of "function" and "predicate" as used in predicate logic and, until now, in this subsection. We will replace them with new meanings. A n-ary *predicate*, also known as a n-ary *relation* is a set of n-tuples. If $\mathbf{r}$ is a n-ary relation, then the notation $\mathbf{r}(\mathbf{e}_1, \ldots, \mathbf{e}_n)$ is a derived construct (meta)denoting the construct $(\mathbf{e}_1, \ldots, \mathbf{e}_n) \in \mathbf{r}$. We will reintroduce the convention of writing binary predicates in infix as well as prefix form.

A n-ary *function* is a $(n+1)$-ary predicate for which it holds that for any two $(n+1)$-tuples $(\mathbf{e}_1, \ldots, \mathbf{e}_{n+1})$ and $(\mathbf{f}_1, \ldots, \mathbf{f}_{n+1})$ it holds that $(\mathbf{e}_1, \ldots, \mathbf{e}_n) = (\mathbf{f}_1, \ldots, \mathbf{f}_n)$ only if $\mathbf{e}_{n+1} = \mathbf{f}_{n+1}$. If $\mathbf{f}$ is a n-ary function, then the notation $\mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_n)$ is a derived construct (meta)denoting the construct $\iota\boldsymbol{\alpha}\colon \mathbf{f}(\mathbf{e}_1, \ldots, \mathbf{e}_n, \boldsymbol{\alpha})$. We will reintroduce the convention of writing binary functions in infix as well as prefix form.

Note that the functions and predicates defined above (union and others) are not functions in this new sense.

3.6 Axiomatizing set theory

The need for a solid basis for set theory produced several axiomatic systems. The best known of them is nowadays called the Zermelo–Fraenkel set theory (sometimes with the Axiom of Choice), abbreviated as ZF (ZFC).

Ernst Zermelo described his axioms in 1908 [130]. He lists seven axioms (reproduced here in approximately the same form as in the original, alongside with a reformulation in modern notation):

Axiom I (Axiom of extensionality) If every element of a set M is also an element of N and vice versa, then always $N = M$.

Axiom II (Axiom of elementary sets) There exists a set, the null set $\varnothing$, that contains no elements at all. If a is any object of the domain, there exists a set $\{a\}$ containing a and only a as element; if a and b are any two elements of the domain, there always exists a set $\{a, b\}$ containing as elements a and b but no object x distinct from both.

Axiom III (Axiom of separation) Whenever the propositional function $f(x)$ is definite for all elements of a set M, M possesses a subset M_f containing as elements precisely those elements x of M for which $f(x)$ is true.

Axiom IV (Axiom of the powerset) To every set T there corresponds another set $\mathfrak{U}T$, the *power set* of T, that contains as elements precisely all subsets of T.

Axiom V (Axiom of the union) To every set T there corresponds a set $\bigcup T$, the *union* of T, that contains as elements precisely all elements of elements of T.

Axiom VI (Axiom of choice) If T is a set whose elements all are sets that are different from $\varnothing$ and mutually disjoint, its union $\bigcup T$ includes at least one subset S_1 having one and only one element in common with each element of T.

Axiom VII (Axiom of infinity) There exists in the domain at least one set Z that contains the null set as an element and is so constituted that to each of its elements a there corresponds a further element of the form $\{a\}$.

There were some problems with this formulation. Skolem [111] criticized in a conference of Scandinavian mathematicians in Helsinki in 1922 the idea of a "domain" of objects (some of which are sets) as the basis of the axiomatization. Skolem pointed out that such an axiomatization cannot be privileged among axiomatic theories, it cannot form the basis of all other mathematics.

The concept of *definiteness* of a propositional function used by Zermelo in the axiom of separation is vague, and it was justly criticized by for example Abraham Fraenkel [40] and Skolem [111]. The now-standard formulation resembles the one given by Skolem: definite propositional functions are formulae of first-order logic with the additional predicates "$\in$" (set membership) and "=" (identity).

Fraenkel[11] and Skolem [111] formulated also a new axiom, the axiom of replacement, which is reproduced here in the form given by Skolem:

Axiom VIII (Axiom of replacement) Let U be a definite proposition that holds for certain pairs (a, b) in the domain B; assume further, that for every a there exists at most one b such that U is true. Then, as a ranges over the elements of a set M_a, b ranges over all elements of a set M_b.

The final axiom of what is now commonly called the Zermelo-Fraenkel axiomatization of set theory[12] was formulated by John von Neumann [87]. Von Neumann's axiomatization talked about functions, not sets, so the axiom is not very understandable in its orgininal form (axiom IV 2 of [87]). Therefore, we give a modern formulation (after Jech [66] but using our notation):

Axiom IX (Axiom of regularity) $\forall S \in \Omega \colon S \neq \varnothing \rightarrow \exists x \in S \colon S \cap x = \varnothing$

Here Ω denotes the class of all sets.

3.7 The dream torn asunder

An important part of the program of the logicians of twentieth century was to find the perfect axiomatization that will dictate the universal truth (tautologicity) or universal falsity (contradictionality), as appropriate, of all mathematical propositions. One reason was the will to demonstrate that mathematics is universal, as philosophers had maintained for millennia. The other was the will to manufacture an ultimate mechanical decision procedure for all mathematics, much in the spirit of Leibniz. One hoped that with the right axiomatization, a decision procedure could be manufactured.

In this section, we will describe how this dream was shown impossible. First, however, we need some preliminaries.

3.7.1 Primitive recursion A problem with the Zermelo-Fraenkel axiomatization of set theory is that its axiom set is not finite. The axioms of replacement and separation are really axiom schemata: when one plugs in any formula, one gets an axiom of the system. Von Neumann [87] developed a different system of axioms for sets, and this set is finite.

An important requirement for any set of axioms is that it should be expressible using finite means. More generally, it should be possible to mechanically enumerate all axioms. In essence, there should not be "too many" axioms. Fortunately, ZFC does fulfill this condition.

A model of mechanically enumerable sets is called *primitive recursive functions*. In 1887 Richard Dedekind investigated, among other things, the recursive definition of number-theoretic functions in his book *Was sind und was sollen die Zahlen* [31], essentially defining the class of primitive-recursive functions. Skolem [110] and Hilbert [57] used the concept of primitive recursive functions (although they didn't have the name for it) for certain developments in arithmetic. The name "primitive recursive" was coined by Rósza Péter in 1934, who also essentially founded their theory.

A function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be defined by *primitive recursion* by the functions $g : \mathbb{N}^{n-1} \to \mathbb{N}$ and $h : \mathbb{N}^{n+1} \to \mathbb{N}$ if for all $k, x_2, x_3, \ldots, x_n \in \mathbb{N}$ the following hold:

$$f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n),$$
$$f(k+1, x_2, \ldots, x_n) = h(k, f(k, x_2, \ldots, x_n), x_2, \ldots x_n).$$

A function $f : \mathbb{N}^n \to \mathbb{N}$ is called *primitive recursive* if it is a constant function or

the successor function ($\lambda\,\alpha\colon \alpha + 1$), or it is derived by substitution from a primitive recursive function, or by primitive recursion from two primitive recursive functions. A relation $R \subset \mathbb{N}^n \times \mathbb{N}^m$ is primitive recursive if there is a primitive recursive function $f : N^n \times N^m \to \mathbb{N}$ such that $R(a, b)$ is true iff $f(a, b) = 0$ is true. A set $S \subset \mathbb{N}^n$ is primitive recursive if there is a primitive recursive function $f : \mathbb{N}^n \to N$ such that $\forall\,\alpha \in \mathbb{N}^n\colon \alpha \in S \leftrightarrow f(x) = 0$.

Since formulae are strings which in turn are (when the alphabet is fixed) essentially natural numbers (see Appendix A), then sets of formulae are essentially sets of natural numbers. Thus, it is fairly easy to define a primitive recursive set of formulae: a set S of formulae is primitive recursive if gödel$[S]$ is primitive recursive.

3.7.2 Gödel Kurt Gödel showed in 1931 that there is no perfect axiomatization for any formal system that includes arithmetic. Before we can state his result formally, we need to introduce one more concept.

A set of formulae is ω-consistent, if there is no well-formed formula $\mathcal{F}$ containing exactly one free variable ($\boldsymbol{\alpha}$) for which $\mathcal{F}[\boldsymbol{\alpha}/\mathbf{n}]$ is a theorem for every natural number $\mathbf{n}$ but for which the well-formed formula $\forall\,\boldsymbol{\alpha} \in \mathbb{N}\colon \mathcal{F}$ is not a theorem. In other words, in an ω-consistent axiom system, if we can prove something individually for each natural number, we can also prove a generalization of it stating that it holds for all natural numbers.

We will now state Gödel's result.

Theorem 3.1 (Kurt Gödel) *For each ω-consistent, primitive recursive set of formulae (in a formal logic capable of expressing the natural numbers) there is a well-formed formula $\mathcal{F}$ containing no free variables for which neither $\mathcal{F}$ nor $\neg\mathcal{F}$ follows from the set of formulae.*

We skip the proof (see eg. [48]), noting that the proof actually constructs such a formula.

To put it succinctly but imprecisely[13], Gödel showed that

> Every interesting formal theory contains contingent formulae.

Here, interesting obviously excludes everything that is incapable of encoding the natural numbers and their elementary operations.

Another imprecise characterization of Gödel's result is the following:

> Every interesting formal theory has a nonstandard model.

Here, a standard model means the intended possible world and interpretation — the one one has in mind when constructing the formal theory. In contrast, a nonstandard model is a different possible world or interpretation, one where the axioms all hold but where something (namely, one or more of the contingent formulae) is different from the standard model. Nonstandard models for well-known theories include non-Euclidean geometries and nonstandard analysis.

3.7.3 Entscheidungsproblem A related problem was posed in 1928 by Hilbert and Ackermann [56] as the decision problem of predicate logic (commonly called the *Entscheidungsproblem*): find a mechanical procedure that will determine whether or not any given formula of predicate logic is a tautology.

To solve the problem, one needs to first define "a mechanical procedure". Alonzo Church defined λ-calculus [27]. A little later Alan Turing came up independently with his well-known machine, and after becoming aware of Church's work, he proved that λ-calculus and Turing machines are able to solve the same set of problems [117]. They both, independently of each other, postulated that their respective definitions characterize the class of mechanically solvable problems; this is known as the *Church–Turing thesis*.

Both Church and Turing solved the Entscheidungsproblem. Their results can be summarized as follows:

Theorem 3.2 (Alonzo Church) *There is no λ-expression D for which, the λ-expression $\{D\}(P)$ has a normal form that indicates whether P is a tautology or not for all formulae P of first-order logic (expressed as a λ-expression).*

Theorem 3.3 (Alan Turing) *There is no Turing machine that will halt for all formulae of first-order logic (encoded suitably for consumption of the machine), leaving an indicator of whether the formula is a tautology or not on the tape.*

These (equivalent) theorems showed that logic cannot be mechanized perfectly in the more interesting cases. This gives a theoretical limit of what a Level 3 formal method can do.

Notes

[1] This exposition of Aristotle's logic is partly based on Simo Knuuttila's notes on Prior analytics printed as an appendix to Finnish edition [4].

[2] *Premisses* is the plural of *premiss*, and not a typo. *Premiss* is an alternative spelling of *premise* but with a narrower meaning — *premises* can mean also "land and buildings together considered as a place of business" [125], but *premisses* has no such second meaning.

[3] Burris [19] believes that the first reference to "Boolean algebras" was by Sheffer in 1913 [108]; Sheffer's use of the term refers to several symbolic, or algebra-like, treatments of logic, including that of Whitehead and Russell in *Principia mathematica* [122], all of which are quite different from Boole's original. Nowadays the term refers, if one is allowed to simplify it a bit, to modulo 2 arithmetic, interpreted as propositional logic (cf. eg. [72]).

[4] The modern names were rewritten to make them familiar to non-philosophers, and some inessential lines were left out.

[5] Frege considers the sign $\vdash$ to be composite: he calls the vertical bar the *judgment stroke* and the horizontal bar the *content stroke*. If we remove the judgment stroke, the formula denotes a thought, but when the judgment stroke is present, the formula is an assertion of truth.

[6] The term "truth-function" was apparently introduced by Wittgenstein in his *Tractatus logico-philosophicus* [124].

[7] We deviate slightly here from Frege's notation in order to be more understandable.

[8] Actually, as Frege noted in his reply to Russell [44], this formulation of the antinomy is not correct. Frege's notation does not allow a predicate to be predicated of itself. However, this does not invalidate the problem. Frege calls the class of things that are predicated of a predicate $\Phi(\xi)$ the *extension* of the predicate, denoted by $\grave{\epsilon}\Phi(\epsilon)$. Now, we can redefine the function $\Phi(\xi)$ above as "$\Phi(\xi)$ is true if $\Xi(\grave{\epsilon}\Xi(\epsilon))$".

[9] To Cantor, a multiplicity can be a set but not all of them are; this will become apparent when we discuss Buriali-Forti's antinomy.

[10] Truth tables were apparently first used by Ludwig Wittgenstein in his *Tractatus logico-philosophicus* [124, 4.31].

[11] According to Jech [66], Fraenkel published his version of the axiom in an article

in *Mathematische Annalen* in 1922; I have not been able to find either a copy or a reprint of the article.

[12]It is unfortunate that this name fails to mention the contributions of Thoralf Skolem and von Neumann.

[13]These characterizations of Gödel's work are due to the current author (though probably not unique to him). The current author lacks expertise on model theory and logic to evaluate the uniqueness or correctness of these characterizations, but Putnam's [98] presentation of the Gödel theorem and of a remarkably simple proof by Kripke does seem to validate the characterizations.

4 Automated reasoning

47. *As Will Rogers would have said, "There is no such thing as a free variable."*
— Alan J. Perlis [94]

Level 3 formal methods require tool support for automated reasoning. In this chapter, we'll review the basic techniques. This chapter is not intended to be a comprehensive treatment of the subject.

The basic methods of automated reasoning are covered by standard university textbooks on artificial intelligence, such as Nils Nilsson's [88] and George Luger and William Stubblefield's [78]. More thoroghly it is described in Wos, Overbeek, Lusk and Boyle's book [126]. A thoroughly theoretical but a little outdated account is given by Loveland [77]. The recent North-Holland handbook [100] is an advanced treatment.

4.1 The limits of automated reasoning

The ultimate goal of research in the field of automated reasoning is to find an efficient decision procedure for each interesting logical system. In practice, this is an unattainable goal: In the case of propositional calculus, a decision procedure does exist but it is efficient only if $P = NP$ (this is the SAT problem which is NP complete [90]), and predicate calculus is undecidable, as was mentioned in the previous chapter.

Fortunately, there exist proof procedures for predicate calculus, ie. algorithms which are able to accurately recognize tautologies but which do not terminate for some refutable formulae. Similarly, there exist refutation procedures, ie. algorithms which are able to accurately recognize contradictory formulae but which do not terminate for some satisfiable formulae. Here is an important point: Although we have these two equivalent[1] definitions of proof procedures, this does not allow us to combine them to create a decision procedure by running a proof procedure on a formula and a refutation procedure on its negation.

A proof procedure may fail to terminate if the formula it works on is a refutable formula. Some refutable formulae are contradictory, but some are contingent. If it is

contradictory, then a refutation procedure will notice this. If it is contingent, then its negation is also contingent, and a refutation procedure, too, may fail to terminate on it. Thus there can exist well-formed formulae that cause both a proof procedure and a refutation procedure to fail to terminate; some contingent formulae are such. The fact that they indeed do exist follows from Church and Turing's theorem discussed in the previous chapter.

Note that this terminology (uniformly used in publications in automated reasoning, such as [88, 78, 126, 77, 100]) may be confusing. A "refutable formula" is one that is not true in all possible worlds and interpretations. However, a "refutation procedure" does not detect refutable formulae but contradictions!

Higher-order logic does not even have a proof procedure; this follows straightforwardly from Gödel's incompleteness results discussed in the previous chapter.

We will only consider refutation procedures for first-order logic in this chapter.

4.2 Normal forms for first-order formulae

We call a term or a well-formed formula of first-order logic *propositional* if it contains no quantifiers and no variables.

For the purposes of automated reasoning, a propositional formula is usually written in a *conjunctive normal form* (CNF). We call a well-formed formula that contains no quantifiers, variables or connectives a *literal*, and we call a set of literals a *clause*. A set of clauses is a *CNF-formula*. Now, let $\{C_1, \ldots, C_n\}$ be a CNF-formula, where each $C_i = \{T_{i,0}, \ldots, T_{i,m_i}\}$ is a term and each $T_{i,j}$ is a CNF-term. Then that CNF-formula is understood to denote the propositional formula $(T_{1,1} \vee \ldots \vee T_{1,m_1}) \wedge \ldots \wedge (T_{n,1} \vee \ldots \vee T_{n,m_n})$. For every propositional formula there is a logically equivalent CNF-formula (cf. eg. [105]).

A quantifier-free formula may be converted to CNF using the following procedure ($\mathbf{P}$, $\mathbf{Q}$ and $\mathbf{R}$ are well-formed formulae):

1. Remove any shorthands from the formula except $\vee$ and $\wedge$.
2. Recursively transform all occurrences of $\mathbf{P} \rightarrow \mathbf{Q}$ in the formula to $\neg\mathbf{P} \vee \mathbf{Q}$.
3. Recursively transform all occurrences of $\neg(\mathbf{P} \wedge \mathbf{Q})$ to $\neg\mathbf{P} \vee \neg\mathbf{Q}$, all occurrences of $\neg(\mathbf{P} \vee \mathbf{Q})$ to $\neg\mathbf{P} \wedge \neg\mathbf{Q}$ and all occurrences of $\neg\neg\mathbf{P}$ to $\mathbf{P}$.
4. Recursively transform all occurrences of $\mathbf{P} \vee (\mathbf{Q} \wedge \mathbf{R})$ to $(\mathbf{P} \vee \mathbf{Q}) \wedge (\mathbf{P} \vee \mathbf{R})$, and transform all occurrences of $(\mathbf{P} \wedge \mathbf{Q}) \vee \mathbf{R})$ to $(\mathbf{P} \vee \mathbf{R}) \wedge (\mathbf{Q} \vee \mathbf{R})$.

For full first-order logic, CNF is not sufficient; we need *Skolem conjunctive form* (SCF). This form is reminiscent of CNF in that it has the same basic form. The differences are that variables are allowed in literals, and all quantifiers are universal and are located at the very beginning of the formula. For each formula there is an unsatisfiable SCF-formula if and only if it itself is unsatisfiable [77].

A formula can be converted to Skolem conjunctive form using the following procedure [77]:

Precondition: Input is a first-order formula.

1. For each variable $\boldsymbol{\alpha}$ free in the formula, prepend "$\exists\, \boldsymbol{\alpha}\colon$ " to the formula.[2]
2. Remove any shorthands from the formula except $\vee$ and $\wedge$.
3. Recursively transform all occurrences of $\mathbf{P} \rightarrow \mathbf{Q}$ in the formula to $\neg\mathbf{P} \vee \mathbf{Q}$.
4. Recursively transform all occurrences of $\neg\, \forall\, \boldsymbol{\alpha}\colon \mathbf{P}$ to $\exists\, \boldsymbol{\alpha}\colon \neg\mathbf{P}$, all occurrences of $\neg\, \exists\, \boldsymbol{\alpha}\colon \mathbf{P}$ to $\forall\, \boldsymbol{\alpha}\colon \neg\mathbf{P}$, all occurrences of $\neg(\mathbf{P} \wedge \mathbf{Q})$ to $\neg\mathbf{P} \vee \neg\mathbf{Q}$, all occurrences of $\neg(\mathbf{P} \vee \mathbf{Q})$ to $\neg\mathbf{P} \wedge \neg\mathbf{Q}$ and all occurrences of $\neg\neg\mathbf{P}$ to $\mathbf{P}$.
5. If two quantifiers in the formula quantify over the same variable, rename one of the variables in the following manner. Let us have a subformula generated by $\forall\, \boldsymbol{\alpha}\colon \mathbf{P}$ (or $\exists\, \boldsymbol{\alpha}\colon \mathbf{P}$), where $\boldsymbol{\alpha}$ needs to be renamed to $\boldsymbol{\beta}$; then the formula generated by $\forall\, \boldsymbol{\beta}\colon \mathbf{P}[\boldsymbol{\alpha}/\boldsymbol{\beta}]$ (or, respectively, $\exists\, \boldsymbol{\beta}\colon \mathbf{P}[\boldsymbol{\alpha}/\boldsymbol{\beta}]$) is the result of the rename.
6. Each subformula generated by $\exists\, \boldsymbol{\alpha}\colon \mathbf{P}$ is replaced by $\mathbf{P}[\boldsymbol{\alpha}/\mathbf{t}]$, where $\mathbf{t}$ is a *Skolem function* generated by $\mathbf{u}(\boldsymbol{\beta}_1, \ldots \boldsymbol{\beta}_n)$, where $\mathbf{u}$ is an individual constant that does not occur at this time in the formula being Skolemized, and each $\boldsymbol{\beta}_i$ is a variable for which it holds that the formula being Skolemized contains a subformula $\forall\, \boldsymbol{\beta}_i\colon \mathbf{Q}$, where the subformula being considered is a subformula of $\mathbf{Q}$. If there are no such variables, $\mathbf{t}$ is generated by $\mathbf{u}$.
7. Transform all occurrences of $\forall\, \alpha\colon \mathbf{P}$ to $\mathbf{P}$.
8. For each free variable $\boldsymbol{\alpha}$ in the formula, prepend $\forall\, \boldsymbol{\alpha}\colon$ to the formula.

Postcondition: The output is in Skolem conjunctive form.

If we omit the last step of the algorithm, the result is called a matrix. In a matrix there are no quantifiers and all variables are universally quantified implicitly.

4.3 Unification

At the core of any theorem proving algorithm is a unification algorithm. Unification is a process which, by specializing variables, makes two terms identical, if it is at all possible.

First we need a couple of definitions. A substitution σ is a unifier of two formula schemata $\mathcal{F}$ and $\mathcal{F}'$, if $\mathcal{F}\sigma = \mathcal{F}'\sigma$ holds. It is the most general unifier (mgu) of the formula schemata, if for all unifiers ρ, there exists a substitution μ such that $\sigma\,\mu = \rho$. We will denote the most general unifier of $\mathcal{F}$ and $\mathcal{F}'$ by $\mathrm{mgu}(\mathcal{F}, \mathcal{F}')$.

Thus formally, unification is the process of finding out if there is a unifier of two formula schemata given as inputs, and if there is, constructing the mgu.

The standard recursive descent algorithm for unification [7] proceeds as follows:

Algorithm 4.1 (Recursive descent unification for first-order terms)
Precondition: *Input is two formula schemata $\mathcal{F}$ and $\mathcal{F}'$ (order is significant), both (SCF-)literals, and a substitution σ (which is nonempty only in self-recursive invocations).*

1. *If $\mathcal{F}$ is a variable, let $\mathcal{F} \leftarrow \mathcal{F}\sigma$.*
2. *If $\mathcal{F}'$ is a variable, let $\mathcal{F}' \leftarrow \mathcal{F}'\sigma$.*
3. *If $\mathcal{F}$ is a variable and $\mathcal{F} = \mathcal{F}'$, output σ and halt with success.*
4. *If $\mathcal{F} = \mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\mathcal{F}' = \mathbf{g}(\mathbf{t}'_1, \ldots, \mathbf{t}'_m)$ for some function symbols $\mathbf{f}$ and $\mathbf{g}$ and some terms $\mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{t'}_m, \ldots, \mathbf{t'}_m$ ($n = 0$ and $m = 0$ are allowed), go to step 8.*
5. *If $\mathcal{F}$ is not a variable, apply this algorithm to the inputs $\mathcal{F}'$, $\mathcal{F}$ and σ (note the order), output what it outputs and halt with success if it was successful and with failure otherwise.*
6. *If $\mathcal{F}$ occurs as a subformula in $\mathcal{F}'$, halt with failure.*
7. *Output $\sigma \cup \{(\mathcal{F}, \mathcal{F}')\}$ and halt with success.*
8. *If $n \neq m$ or $\mathbf{f} \neq \mathbf{g}$, halt with failure.*
9. *Let $i \leftarrow 1$.*
10. *If $i \geq n$, output σ and halt with success.*
11. *Apply this algorithm to the inputs t_i, t'_i and σ. If it fails, halt with failure. Otherwise let σ be its result (destructively updating it).*

12. *Let* $i \leftarrow i + 1$.

13. *Go to step 10.*

Postcondition: *The output is* $\{ (x,f) : (x \in dom(\mathrm{mgu}(\mathcal{F}, \mathcal{F}')) \rightarrow f = \mathrm{mgu}(\mathcal{F}, \mathcal{F}')(x)) \wedge (x \notin dom(\mathrm{mgu}(\mathcal{F}, \mathcal{F}')) \rightarrow f = \sigma(x)) \}$.

4.4 Resolution

Resolution is a simple refutation procedure for propositional and predicate calculi. It was introduced by J. A. Robinson [101] in 1960's. The basic idea is to use of the following two inference rules [8] repeatedly until neither can be used:

$$\text{Binary resolution} \quad \frac{\mathbf{P} \vee \mathbf{R} \quad \mathbf{Q} \vee \neg \mathbf{S}}{(\mathbf{P} \wedge \mathbf{Q})\, \mathrm{mgu}(\mathbf{R}, \mathbf{S})}$$

$$\text{Positive factoring} \quad \frac{\mathbf{P} \vee \mathbf{Q} \vee \mathbf{R}}{(\mathbf{P} \vee \mathbf{Q})\, \mathrm{mgu}(\mathbf{Q}, \mathbf{R})}$$

The input formula is unsatisfiable if a contradiction (a clause of the form $(\mathbf{P} \vee \neg\mathbf{Q})\, \mathrm{mgu}(\mathbf{P}, \mathbf{Q})$) is produced. Otherwise it is satisfiable.

Resolution, when applied to Skolem conjunctive forms, is refutationally complete, that is, it is a refutation procedure for first-order logic. The current author has previously implemented a resolution-based theorem prover for a restricted (Prolog-like) input language [68].

Notes

[1] Proof procedures and refutation procedures are equivalent in the sense that one can turn a proof procedure into a refutation procedure, and vice versa, by negating every input formula.

[2] If the SCF-formula would be given to a proof procedure, we'd prepend a universal quantification, but since we are assuming a refutation procedure, existential quantifier is better. If an unsatisfiable formula has a free variable, this means that in no model there are any objects that qualify as a value for the variable; if we were to prepend a universal quantifier, we'd be saying that in no model does every object qualify.

5 The B method

B is a method for specifying, designing, and coding software systems.
— Jean-Raymond Abrial [1]

This chapter sketches a reconstruction of the B method. We follow mainly Abrial [1], but we deviate from his presentation in certain important points.

Syntax descriptions in this chapter are all for abstract syntax. Extended BNF as standardized by ISO and IEC [63] is used[1].

5.1 Overview

The B method is a Level 3 formal method in the model-based tradition, aiming to cover the project lifecycle from design to code generation. The specification of a program takes the form of an *abstract machine*, which roughly corresponds to the modules in many programming languages such as Ada [61] and Haskell [95]. Abstract machines can be composed of other abstract machines. Abstract machines can be refined into other abstract machines. Ultimately, the chain of refinement ends at a concrete machine, the *implementation*, which is still written in more or less the same notation as the original abstract machine but which is mechanically translatable into executable code. Implementations can be self-sufficient or they can depend on the services of other abstract machines (which are separately refined into implementations).

The language for specification-in-the-large in B is *Abstract Machine Notation*, abbreviated as AMN. The language for specification-in-the-small is *Generalized Substitution Language*, abbreviated as GSL. In specification, the two are intervowen with the standard language of first-order logic and a restricted kind of set theory. It should be noted that there is a B can be understood as being a composition on layers: at the bottom, we have first-order logic with equality; on top of it is constructed a typed set theory; the next layer is the GSL; and the final layer is the AMN.

5.2 The logic of B

The B method is founded on a version of first-order logic and set theory specifically tailored for it. For example, the set theory is weaker than the usual ZFC set theory. This section is completely, apart from certain exceptions, based on Abrial's book [1] and its structure is loosely followed. Unfortunately, he only gives an abstract syntax for his notation, and only gives faint hints on the concrete syntax. The concrete syntax used by the B-Toolkit or Atelier B is not described at all by Abrial.

5.2.1 Schemata and metavariables We will often use schemata in place of actual strings of the formal system. As in Appendix A, schemata are a way to generate a set of valid strings with a common structure. A schema properly lives outside the object language (the formal system) in the metalevel language (the language used to describe the formal system). A schema is a string of the formal system with one exception: it may include metavariables (roughly, variables of the metalevel language). We typeset metavariables in boldface.

A schema is said to generate a given string in the formal system, if the schema becomes that string when all its metavariables are systematically replaced by strings of the formal system. For example, the schema $\mathbf{s} \in \mathbf{t}$ generates the string $x \cap y \in x \cup y$, but does not generate the string $x \cap y$ (the membership operator is missing). A schema may also include natural language directions. Often there are also natural-language side conditions constraining the metavariables.

Generally, if we say that something holds for a schema (for example, that a certain schema is an inference rule), we mean that it holds for every valid string that the schema generates.

5.2.2 Inference Abrial [1] starts his presentation by giving a generic formal model of inference. At this point, the syntactic category *Predicate* is left abstract; no instances are given. The syntactic category *Sequent* contains assertions of (syntactic) consequence:

Sequent = [*Predicate*, {',', *Predicate* }], '$\vdash$', *Predicate*

The syntactic category *Inference rule* is then given:

$$\textit{Inference rule} = \frac{\{\textit{Sequent}\}}{\textit{Sequent}}$$

The sequents above the horizontal line are called *antecedents* of the rule, and the sequent below the horizontal line is called the *consequent* of the rule. Note that there might be no sequents above the horizontal rule, in which case the inference rule is called an *axiom*.

Abrial gives the following general inference rule schemata[2]:

Entailment axiom $$\frac{}{\mathbf{p} \vdash \mathbf{p}}$$

Monotonicity rule $$\frac{\mathbf{H} \vdash \mathbf{p} \quad \mathbf{H} \text{ is included in } \mathbf{H}'}{\mathbf{H}' \vdash \mathbf{p}}$$

Entailment rule (derived) $$\frac{\mathbf{p} \text{ occurs in } \mathbf{H}}{\mathbf{H} \vdash \mathbf{p}}$$

Lemma rule $$\frac{\mathbf{H} \vdash \mathbf{p} \quad \mathbf{H}, \mathbf{p} \vdash \mathbf{q}}{\mathbf{H} \vdash \mathbf{q}}$$

From now on, the metavariables $\mathbf{p}$ and $\mathbf{q}$ are restricted to strings that belong to the syntactic category *Predicate* and the metavariables $\mathbf{H}$ and $\mathbf{H}'$ are restricted to strings that are comma-separated (empty or nonempty) lists of *Predicate*s.

5.2.3 Propositional calculus Conjunction, implication and negation are used as the primitive connectives:

Predicate = *Predicate*, '$\wedge$', *Predicate*
| *Predicate*, '$\Rightarrow$', *Predicate*
| '$\neg$', *Predicate*

In concrete representations of this abstract syntax, parentheses are used in the usual way to control grouping. Here $\neg$ has the highest precedence, $\wedge$ the next highest and $\Rightarrow$ the lowest. All three connectives associate from the left.

Table 5.1 gives the inference rules that govern the propositional subsystem. The $\Rightarrow$-introduction rule is also called the *deduction rule*.

There are also two derived connectives:

Predicate = *Predicate*, '$\vee$', *Predicate*
| *Predicate*, '$\Leftrightarrow$', *Predicate*

Their meaning is defined using two rewrite rules, which are to be applied repeatedly to all subformulae of any formula until no rule can be applied before the formula is submitted to any formal scrutiny.

$$\mathbf{p} \vee \mathbf{q} :\Rightarrow \neg \mathbf{p} \Rightarrow \mathbf{q}$$
$$\mathbf{p} \Leftrightarrow \mathbf{q} :\Rightarrow (\mathbf{p} \Rightarrow \mathbf{q}) \wedge (\mathbf{q} \Rightarrow \mathbf{p})$$

$\wedge$-introduction	$\dfrac{\mathbf{H} \vdash \mathbf{p} \quad \mathbf{H} \vdash \mathbf{q}}{\mathbf{H} \vdash \mathbf{p} \wedge \mathbf{q}}$
$\wedge$-elimination L	$\dfrac{\mathbf{H} \vdash \mathbf{p} \wedge \mathbf{q}}{\mathbf{H} \vdash \mathbf{p}}$
$\wedge$-elimination R	$\dfrac{\mathbf{H} \vdash \mathbf{p} \wedge \mathbf{q}}{\mathbf{H} \vdash \mathbf{q}}$
$\Rightarrow$-introduction	$\dfrac{\mathbf{H}, \mathbf{p} \vdash \mathbf{q}}{\mathbf{H} \vdash \mathbf{p} \Rightarrow \mathbf{q}}$
$\Rightarrow$-elimination	$\dfrac{\mathbf{H} \vdash \mathbf{p} \Rightarrow \mathbf{q}}{\mathbf{H}, \mathbf{p} \vdash \mathbf{q}}$
Modus ponens (derived)	$\dfrac{\mathbf{H} \vdash \mathbf{p} \quad \mathbf{H} \vdash \mathbf{p} \Rightarrow \mathbf{q}}{\mathbf{H} \vdash \mathbf{q}}$
Reductio ad absurdum I	$\dfrac{\mathbf{H}, \neg\mathbf{q} \vdash \mathbf{p} \quad \mathbf{H}, \neg\mathbf{q} \vdash \neg\mathbf{p}}{\mathbf{H} \vdash \mathbf{q}}$
Reductio ad absurdum II	$\dfrac{\mathbf{H}, \mathbf{q} \vdash \mathbf{p} \quad \mathbf{H}, \mathbf{q} \vdash \neg\mathbf{p}}{\mathbf{H} \vdash \neg\mathbf{q}}$

Table 5.1: Inference rules for propositional calculus in B.

From now on, the symbol $:\Rightarrow$ denotes the phrase "is to be rewritten as". In concrete representations, the precedence of $\vee$ is the same as the precedence of $\wedge$ and the precedence of $\Leftrightarrow$ is lower than the precedence of $\Rightarrow$.

5.2.4 Predicate calculus with equality We now introduce several new syntactic categories. One of the syntactic categories, *Identifier*, is left abstract[3]. We will assume that all nonempty strings consisting of letters that are not reserved words are identifiers.

For now, variables are identifiers. This will be expanded later on.

Variable = *Identifier*

Expressions will later denote mathematical objects such as sets and natural numbers. For now, we stay with a minimal definition.

Expression = *Variable*
| '[', *Variable*, ':=', *Expression*, ']', *Expression*

The latter expression type denotes substitution of free occurrences of the variable with the first expression in the second expression. The precedence of the second production is higher than the precedence of the $\neg$ connective. The second production associates to the right

	Non-freeness	Condition
NF 1	$\mathbf{x} \setminus \mathbf{y}$	the variables that $\mathbf{x}$ and $\mathbf{y}$ stand for are distinct
NF 2	$\mathbf{x} \setminus (\mathbf{p} \wedge \mathbf{q})$	$\mathbf{x} \setminus \mathbf{p}$ and $\mathbf{x} \setminus \mathbf{q}$
NF 3	$\mathbf{x} \setminus (\mathbf{p} \Rightarrow \mathbf{q})$	$\mathbf{x} \setminus \mathbf{p}$ and $\mathbf{x} \setminus \mathbf{q}$
NF 4	$\mathbf{x} \setminus \neg\mathbf{p}$	$\mathbf{x} \setminus \mathbf{p}$
NF 5	$\mathbf{x} \setminus \forall \mathbf{x} \cdot \mathbf{p}$	
NF 6	$\mathbf{x} \setminus \forall \mathbf{y} \cdot \mathbf{p}$	$\mathbf{x} \setminus \mathbf{y}$ and $\mathbf{x} \setminus \mathbf{p}$
NF 7	$\mathbf{x} \setminus [\mathbf{x} := \mathbf{e}]\mathbf{f}$	$\mathbf{x} \setminus \mathbf{e}$
NF 8	$\mathbf{x} \setminus [\mathbf{y} := \mathbf{e}]\mathbf{f}$	$\mathbf{x} \setminus \mathbf{y}$ and $\mathbf{x} \setminus \mathbf{e}$ and $x \setminus \mathbf{f}$
NF 9	$\mathbf{x} \setminus (\mathbf{e} = \mathbf{f})$	$\mathbf{x} \setminus \mathbf{e}$ and $\mathbf{x} \setminus \mathbf{f}$

Table 5.2: Non-freeness rules for predicate calculus with equality.

We add universal quantification, substitution in a predicate and equality to the *Predicate* syntactic category:

Predicate = '$\forall$', *Variable*, '$\cdot$', *Predicate*
| '[', *Substitution*, ']', *Predicate*
| *Expression*, '=', *Expression*

The $\cdot$ operator present in quantification has precedence equal to the precedence of substitution and associates to the right. Abrial does not specify the precedence of equality.

Note that here we use a new syntactic category, *Substitution*. This is because there will be more than one kind of substitution in predicates. But more on that later.

Substitution = *Variable*, ':=', *Expression*

Non-freeness is an important kind of side condition constraining metavariables. Informally speaking, a variable has a free occurrence in a formula if it occurs in the formula and at least one of the occurrences is outside the scope of a quantifier where it is the quantification variable. If a variable $\mathbf{x}$ has no free occurrences in a formula $\mathbf{f}$, it is said to be *non-free* in that formula, written $\mathbf{x} \setminus \mathbf{f}$. The rules in Table 5.2 determine whether a variable is non-free in a formula of predicate calculus with equality.

From now on, the metavariables $\mathbf{e}$, $\mathbf{f}$ and $\mathbf{g}$ are restricted to strings that belong to the syntactic category *Expression*, and the metavariables $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are restricted to strings that belong to the syntactic category *Variable*.

$$[\mathbf{x} := \mathbf{e}]\mathbf{x} :\Rightarrow \mathbf{e} \qquad \text{(SUB 1)}$$

$$[\mathbf{x} := \mathbf{e}]\mathbf{y} :\Rightarrow \mathbf{y} \qquad \text{if } x \setminus y \qquad \text{(SUB 2)}$$

$$[\mathbf{x} := \mathbf{e}](\mathbf{p} \wedge \mathbf{q}) :\Rightarrow [\mathbf{x} := \mathbf{e}]\mathbf{p} \wedge [\mathbf{x} := \mathbf{e}]\mathbf{q} \qquad \text{(SUB 3)}$$

$$[\mathbf{x} := \mathbf{e}](\mathbf{p} \Rightarrow \mathbf{q}) :\Rightarrow [\mathbf{x} := \mathbf{e}]\mathbf{p} \Rightarrow [\mathbf{x} := \mathbf{e}]\mathbf{q} \qquad \text{(SUB 4)}$$

$$[\mathbf{x} := \mathbf{e}]\neg\mathbf{p} :\Rightarrow \neg[\mathbf{x} := \mathbf{e}]\mathbf{p} \qquad \text{(SUB 5)}$$

$$[\mathbf{x} := \mathbf{e}]\,\forall\,\mathbf{y} \cdot \mathbf{p} :\Rightarrow \forall\,\mathbf{z} \cdot [\mathbf{x} := \mathbf{e}][\mathbf{y} := \mathbf{z}]\mathbf{p} \qquad \text{where } \mathbf{z} \setminus \mathbf{e} \text{ and } \mathbf{z} \setminus \mathbf{p} \qquad \text{(SUB 6')}$$

$$[\mathbf{x} := \mathbf{e}](\mathbf{f} = \mathbf{g}) :\Rightarrow [\mathbf{x} := \mathbf{e}]\mathbf{f} = [\mathbf{x} := \mathbf{e}]\mathbf{g} \qquad \text{(SUB 12)}$$

Table 5.3: Rewrite rules for substitutions in predicate calculus with equality.

The meaning of substitutions is defined formally by the rewrite rules given in Table 5.3. They must be applied every time they can be applied, during any formal manipulation of any formula. The idea of SUB 6' is that **z** can be chosen freely as long as the side condition is honoured; in some cases it can be identical to **y**. Abrial lists two rules (called SUB 6 and SUB 7 but not reproduced in this thesis) instead of our SUB 6', dealing separately with the cases where the substituted and the quantified variables are the same and distinct, respectively. The problem with his rules is that they assume implicitly that a quantified variable can be renamed at will; such an assumption degrades the formality of the presentation in the opinion of the current author. Abrial's rules SUB 8 through SUB 11 are derived and hence inessential.

These substitutions are designed to mirror the customary metaoperation of substitution of an expression for a variable in a predicate, usually denoted by $P(e/c)$ (note: this syntax is not valid in B). In B, substitutions are moved into the formal system itself, which allows their generalization, which will be done later in this chapter. An operational intuititon for substitutions of the form $[\mathbf{x} := \mathbf{e}]\mathbf{p}$ (or $[\mathbf{x} := \mathbf{e}]\mathbf{f}$) is that **p** (or **f**) is evaluated in an environment where **x** is bound to **e**.

There are four new rules of inference, given in Table 5.4. They govern the use of quantification and equality.

There are two derived constructs. As before with the derived connectives, we'll augment the syntax and give a rewrite rule.

Predicate = '$\exists$', *Variable*, '$\cdot$', *Predicate*
| *Expression*, '$\neq$', *Expression*

$\forall$-introduction	$\dfrac{\mathbf{h} \vdash \mathbf{p} \quad \mathbf{x} \setminus \mathbf{q} \text{ for every } \mathbf{q} \text{ in } \mathbf{h}}{\mathbf{h} \vdash \forall\, \mathbf{x} \cdot \mathbf{p}}$
$\forall$-elimination	$\dfrac{\mathbf{h} \vdash \forall\, \mathbf{x} \cdot \mathbf{p}}{\mathbf{h} \vdash [\mathbf{x} := \mathbf{e}]\mathbf{p}}$
Leibniz's law	$\dfrac{\mathbf{h} \vdash \mathbf{e} = \mathbf{f} \quad \mathbf{h} \vdash [\mathbf{x} := \mathbf{e}]\mathbf{p}}{\mathbf{h} \vdash [\mathbf{x} := \mathbf{f}]\mathbf{p}}$
Reflexivity	$\dfrac{}{\mathbf{p} \vdash \mathbf{e} = \mathbf{e}}$

Table 5.4: Additional inference rules for predicate calculus with equality.

$$\exists\, \mathbf{x} \cdot \mathbf{p} :\Rightarrow \neg\, \forall\, \mathbf{x} \cdot \neg \mathbf{p}$$
$$\mathbf{e} \neq \mathbf{f} :\Rightarrow \neg \mathbf{e} = \mathbf{f}$$

5.2.5 Proof procedure Abrial sketches a proof procedure for the logic of B in Sections 1.2.4 and 1.3.7 of the B-Book [1]. Unfortunately, this remains just a sketch: although the predicate calculus proof procedure is easy enough to transcribe into a working program, there is no help from Abrial for extending this to cover equality or set theory.

The proof procedure consists of several derived rules together with certain fundamental rules, to be used backward in a certain order. The rules are, in order,[4]

1. Conditional entailment rule (BS1, derived)

$$\frac{\mathbf{r} \text{ occurs in } \mathbf{h}}{\mathbf{h} \vdash \mathbf{p} \Rightarrow \mathbf{r}}$$

2. Entailment rule (BS2, derived)

$$\frac{\mathbf{p} \text{ occurs in } \mathbf{h}}{\mathbf{h} \vdash \mathbf{p}}$$

3. Simplified reductio ad absurdum I (DB1, derived)

$$\frac{\mathbf{p} \text{ occurs in } \mathbf{h}}{\mathbf{h} \vdash \neg \mathbf{p} \Rightarrow \mathbf{r}}$$

4. Simplified reductio ad absurdum II (DB2, derived)

$$\frac{\neg \mathbf{p} \text{ occurs in } \mathbf{h}}{\mathbf{h} \vdash \mathbf{p} \Rightarrow \mathbf{r}}$$

5. Double negation rule (DR1, derived)

$$\frac{\mathbf{h} \vdash \mathbf{p}}{\mathbf{h} \vdash \neg\neg\mathbf{p}}$$

6. Negated implication rule (DR2, derived)

$$\frac{\mathbf{h} \vdash \mathbf{p} \quad \mathbf{h} \vdash \neg\mathbf{q}}{\mathbf{h} \vdash \neg(\mathbf{p} \Rightarrow \mathbf{q})}$$

7. Negated conjunction rule (DR3, derived)

$$\frac{\mathbf{h} \vdash \mathbf{p} \Rightarrow \neg\mathbf{q}}{\mathbf{h} \vdash \neg(\mathbf{p} \wedge \mathbf{q})}$$

8. Double-negated condition rule (DR 4, derived)

$$\frac{\mathbf{h} \vdash \mathbf{p} \Rightarrow \mathbf{q}}{\mathbf{h} \vdash \neg\neg\mathbf{p} \Rightarrow \mathbf{q}}$$

9. Negated implication as condition rule (DR 5, derived)

$$\frac{\mathbf{h} \vdash \mathbf{p} \Rightarrow (\neg\mathbf{q} \Rightarrow \mathbf{r})}{\mathbf{h} \vdash \neg(\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow \mathbf{r}}$$

10. Negated conjunction as condition rule (DR6, derived)

$$\frac{\mathbf{h} \vdash \neg\mathbf{p} \Rightarrow \mathbf{r} \quad \mathbf{h} \vdash \neg\mathbf{q} \Rightarrow \mathbf{r}}{\mathbf{h} \vdash \neg(\mathbf{p} \wedge \mathbf{q}) \Rightarrow \mathbf{r}}$$

11. Implication as condition rule (DR7, derived)

$$\frac{\mathbf{h} \vdash \neg\mathbf{p} \Rightarrow \mathbf{r} \quad \mathbf{h} \vdash \mathbf{q} \Rightarrow \mathbf{r}}{\mathbf{h} \vdash (\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow \mathbf{r}}$$

12. Conjunction as condition rule (DR8, derived)

$$\frac{\mathbf{h} \vdash (\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow \mathbf{r}}{\mathbf{h} \vdash \mathbf{p} \wedge \mathbf{q} \Rightarrow \mathbf{r}}$$

13. Existential condition rule (DR9, derived)

$$\frac{\mathbf{x} \setminus \mathbf{r} \quad \mathbf{x} \setminus \mathbf{q} \text{ for each } \mathbf{q} \text{ in } \mathbf{h} \quad \mathbf{h} \vdash \neg \mathbf{p} \Rightarrow \mathbf{r}}{\mathbf{h} \vdash \neg \forall \mathbf{x} \cdot \mathbf{p} \Rightarrow \mathbf{r}}$$

14. $\exists$-introduction rule (DR10, derived)

$$\frac{\mathbf{h} \vdash [\mathbf{x} := \mathbf{e}] \neg \mathbf{p}}{\mathbf{h} \vdash \neg \forall \mathbf{x} \cdot \mathbf{p}}$$

15. Universal hypothesis particularization rule (DR11, derived)

$$\frac{\forall \mathbf{x} \cdot \mathbf{p} \text{ occurs in } \mathbf{h} \quad \mathbf{h} \vdash [\mathbf{x} := \mathbf{e}] \mathbf{p} \Rightarrow \mathbf{r}}{\mathbf{h} \vdash \mathbf{r}}$$

16. $\wedge$-introduction rule (CNJ)

$$\frac{\mathbf{h} \vdash \mathbf{p} \quad \mathbf{h} \vdash \mathbf{q}}{\mathbf{h} \vdash \mathbf{p} \wedge \mathbf{q}}$$

17. $\Rightarrow$-introduction rule (DED)

$$\frac{\mathbf{h}, \mathbf{p} \vdash \mathbf{q}}{\mathbf{h} \vdash \mathbf{p} \Rightarrow \mathbf{q}}$$

5.3 Set notation

In this section we will extend the logic to cover ordered pairs and sets.

5.3.1 Ordered pairs The logic is extended to cover ordered pairs. Defining them this way instead of in the set theory has the advantage that quantification over variable pairs is possible. The standard definition of a pair $(\mathbf{e}, \mathbf{f})$ as the set $\{\{\mathbf{e}\}, \{\mathbf{e}, \mathbf{f}\}\}$ is counterintuitive and can even degrade to a unit set (when $\mathbf{e} = \mathbf{f}$).

There are two new productions for both *Expression* and *Variable*:

Expression = *Expression*, ',', *Expression*
| *Expression*, '$\mapsto$', *Expression*
Variable = *Variable*, ',', *Variable*
| *Variable*, '$\mapsto$', *Variable*

	Non-freeness	Condition
NF 10	$(\mathbf{x}, \mathbf{y}) \setminus \mathbf{e}$	$\mathbf{x} \setminus \mathbf{e}$ and $\mathbf{y} \setminus \mathbf{e}$
NF 11	$\mathbf{x} \setminus (\mathbf{e}, \mathbf{f})$	$\mathbf{x} \setminus \mathbf{e}$ and $\mathbf{x} \setminus \mathbf{f}$
NF 12	$\mathbf{x} \setminus \forall(\mathbf{y}, \mathbf{z}) \cdot \mathbf{p}$	$\mathbf{x} \setminus \forall\, \mathbf{y} \cdot \forall\, \mathbf{z} \cdot \mathbf{p}$

Table 5.5: Additional non-freeness rules for predicate calculus with equality and pairs.

The precedence of the comma and $\mapsto$ is left unspecified by Abrial, but he notes that they associate to the left. The $\mapsto$ operator is derived, and we give the following rewrite rules for it:

$$\mathbf{x} \mapsto \mathbf{y} :\Rightarrow \mathbf{x}, \mathbf{y}$$
$$\mathbf{e} \mapsto \mathbf{f} :\Rightarrow \mathbf{e}, \mathbf{f}$$

The set of non-freeness rules need to be enlarged as given in Table 5.5. Similarly, there needs to be one new substitution rule, dealing with simultaneous independent substitution:

$$[\mathbf{x}, \mathbf{y} := \mathbf{e}, \mathbf{f}]\mathbf{g} :\Rightarrow [\mathbf{z} := \mathbf{f}][\mathbf{x} := \mathbf{e}][\mathbf{y} := \mathbf{z}]\mathbf{g} \qquad \text{if } \mathbf{x} \setminus \mathbf{y} \text{ and where } \mathbf{z} \setminus (\mathbf{x}, \mathbf{y}, \mathbf{e}, \mathbf{f}, \mathbf{g}) \quad \text{(SUB 13)}$$

The operational intuition for $[\mathbf{x}, \mathbf{y} := \mathbf{e}, \mathbf{f}]\mathbf{g}$ is the independent evaluation of $\mathbf{e}$ and $\mathbf{f}$ in the initial environment and the evaluation of $\mathbf{g}$ in the environment obtained from the initial environment by binding $\mathbf{x}$ to the resulting value of $\mathbf{e}$ and $\mathbf{y}$ to the resulting value of $\mathbf{f}$. Note that for this to make sense, $\mathbf{x}$ and $\mathbf{y}$ must be distinct (or the evaluation results of $\mathbf{e}$ and $\mathbf{f}$ must be equal, but this is not allowed by Abrial's definition). As an example, consider $[x, y := y, x](x = y)$:

$[x, y := y, x](x = y)$	apply SUB 13 (choose z as $\mathbf{z}$)
$[z := x][x := y][y := z](x = y)$	apply SUB 12
$[z := x][x := y]([y := z]x = [y := z]y)$	apply SUB 2
$[z := x][x := y](x = [y := z]y)$	apply SUB 1
$[z := x][x := y](x = z)$	apply SUB 12
$[z := x]([x := y]x = [x := y]z)$	apply SUB 1

	Non-freeness	Condition
NF 13	$\mathbf{x} \setminus (\mathbf{e} \in \mathbf{s})$	$\mathbf{x} \setminus \mathbf{e}$ and $\mathbf{x} \setminus \mathbf{s}$
NF 14	$\mathbf{x} \setminus \mathsf{choice}(\mathbf{s})$	$\mathbf{x} \setminus \mathbf{s}$
NF 15	$\mathbf{x} \setminus (\mathbf{s} \times \mathbf{t})$	$\mathbf{x} \setminus \mathbf{s}$ and $\mathbf{x} \setminus \mathbf{t}$
NF 16	$\mathbf{x} \setminus \mathbb{P}(\mathbf{s})$	$\mathbf{x} \setminus \mathbf{s}$
NF 17	$\mathbf{x} \setminus \{\mathbf{y} \mid \mathbf{p}\}$	$\mathbf{x} \setminus \forall \mathbf{y} \cdot \mathbf{p}$
NF 18	$\mathbf{x} \setminus \mathsf{BIG}$	

Table 5.6: Non-freeness rules for set constructs.

$[z := x](y = [x := y]z)$	apply SUB 2
$[z := x](y = z)$	apply SUB 12
$([z := x]y = [z := x]z)$	apply SUB 2
$(y = [z := x]z)$	apply SUB 1
$(y = x)$.	

No new inference rules are needed.

5.3.2 Sets We add two new predicate productions and six new expression productions:

Predicate = *Expression*, '$\in$', *Expression*
| 'infinite', '(', *Expression*, ')'
Expression = 'choice', '(', *Expression*, ')'
| *Expression*, '$\times$', *Expression*
| '$\mathbb{P}$','(', *Expression*, ')'
| '{', *Variable*, '|', *Predicate*, '}'
| 'BIG'

The infinite predicate will be defined as a derived predicate using a rewrite rule in Subsection 5.4.2. Abrial uses a separate syntactic category *Set*, but his usage of the notation reveals that no such syntactic category really exists.

The rules for non-freeness need to be enlarged as given in Table 5.6. From now on, the metavariables **s**, **t** and **u** are restricted to strings that belong to the syntactic category *Expression*.

$$[\mathbf{x} := \mathbf{e}](\mathbf{f} \in \mathbf{s}) :\Rightarrow [\mathbf{x} := \mathbf{e}]\mathbf{f} \in [\mathbf{x} := \mathbf{e}]\mathbf{s} \quad \text{(SUB 14)}$$

$$[\mathbf{x} := \mathbf{e}]\,\mathsf{choice}(\mathbf{s}) :\Rightarrow \mathsf{choice}([\mathbf{x} := \mathbf{e}]\mathbf{s}) \quad \text{(SUB 15)}$$

$$[\mathbf{x} := \mathbf{e}](\mathbf{s} \times \mathbf{t}) :\Rightarrow [\mathbf{x} := \mathbf{e}]\mathbf{s} \times [\mathbf{x} := \mathbf{e}]\mathbf{t} \quad \text{(SUB 16)}$$

$$[\mathbf{x} := \mathbf{e}]\,\mathbb{P}(\mathbf{s}) :\Rightarrow \mathbb{P}([\mathbf{x} := \mathbf{e}]\mathbf{s}) \quad \text{(SUB 17)}$$

$$[\mathbf{x} := \mathbf{e}]\{\,\mathbf{y} \mid \mathbf{p}\,\} :\Rightarrow \{\,\mathbf{y} \mid \mathbf{p}\,\} \qquad \text{if } \mathbf{x} \text{ occurs in } \mathbf{y} \quad \text{(SUB 18)}$$

$$[\mathbf{x} := \mathbf{e}]\{\,\mathbf{y} \mid \mathbf{p}\,\} :\Rightarrow \{\,\mathbf{z} \mid [\mathbf{x} := \mathbf{e}][\mathbf{y} := \mathbf{z}]\mathbf{p}\,\} \quad \text{if } \mathbf{y} \setminus \mathbf{x} \text{ and where } \mathbf{z} \setminus (\mathbf{x}, \mathbf{e}) \quad \text{(SUB 19')}$$

$$[\mathbf{x} := \mathbf{e}]\,\mathsf{BIG} :\Rightarrow \mathsf{BIG} \quad \text{(SUB 20)}$$

Table 5.7: Rewrite rules for substitution in set notation.

SET 1: Cartesian product	$\overline{\mathbf{h} \vdash (\mathbf{e}, \mathbf{f}) \in (\mathbf{s} \times \mathbf{t}) \Leftrightarrow (\mathbf{e} \in \mathbf{s} \wedge \mathbf{f} \in \mathbf{t})}$	
SET 2: Power set	$\overline{\mathbf{h} \vdash \mathbf{s} \in \mathbb{P}(\mathbf{t}) \Leftrightarrow \forall \mathbf{x}\colon (\mathbf{x} \in \mathbf{s} \Rightarrow \mathbf{x} \in \mathbf{t})}$	where $\mathbf{x} \setminus (\mathbf{s}, \mathbf{t})$
SET 3: Comprehension	$\overline{\mathbf{h} \vdash \mathbf{e} \in \{\,\mathbf{x} \mid \mathbf{p}\,\} \Leftrightarrow [\mathbf{x} := \mathbf{e}]\mathbf{p}}$	where $\mathbf{x} \setminus \mathbf{s}$
SET 4: Extensionality	$\overline{\mathbf{h} \vdash \forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{s} \Leftrightarrow \mathbf{x} \in \mathbf{t}) \Rightarrow \mathbf{s} = \mathbf{t}}$	where $\mathbf{x} \setminus (\mathbf{s}, \mathbf{t})$
SET 5: Choice	$\overline{\mathbf{h} \vdash \exists \mathbf{x}\colon (\mathbf{x} \in \mathbf{s}) \Rightarrow \mathsf{choice}(\mathbf{s}) \in \mathbf{s}}$	where $\mathbf{x} \setminus \mathbf{s}$
SET 6: Infinity	$\overline{\mathbf{h} \vdash \mathsf{infinite}(\mathsf{BIG})}$	

Table 5.8: Axioms of set notation in B.

Substitution rules need enlargement too as given in Table 5.7. Rule SUB 19′ is modified from the Abrial original for the same reasons rule SUB 6′ was.

Formulae using set notation are constrained by the requirement that they must pass typechecking, which will be introduced in the next section. It is typechecking which will disallow the paradoxes of naïve set theory.

Six new inference rule schemata (or rather, axiom schemata) are added as given in Table 5.8. These are similar to the standard ZFC axioms presented in Chapter 3, but they are simpler and weaker, since the full power of ZFC is not needed. Abrial writes the comprehension axiom as $\mathbf{e} \in \{\,\mathbf{x} \mid \mathbf{x} \in \mathbf{s} \wedge \mathbf{p}'\,\} \Leftrightarrow \mathbf{x} \in \mathbf{s} \wedge [\mathbf{x} := \mathbf{e}]\mathbf{p}'$, where $\mathbf{x} \setminus \mathbf{s}$. However, in the presence of typechecking, the additional constraint for the form of the predicate $\mathbf{p}$ is redundant.

Axioms of pairing, replacement, union or foundation are not included. The axiom of pairing is no longer necessary, as ordered pairs are part of the underlying logic. The axiom of replacement is unnecessary as we are not interested in ordinal numbers. Typechecking alleviates the need for the axiom of foundation. Abrial notes that the axiom of union is "not indispensable".

5.3.3 Typechecking As mentioned earlier, well-formed formulae of set notation are restricted by typechecking as well as the syntax. The aim of types is to rule out all forms of paradox in the formal system by restricting which formulae are well-formed.

Typechecking operates with *Type*s, *Type predicate*s, *Type assumption*s and *Type sequents*. *Type rule*s are also needed.

Type = 'type', '(', *Expression*, ')'
| 'super', '(', *Expression*, ')'
| *Type*, '×', *Type*
| 'ℙ', '(', *Type*, ')'

Type predicate = 'check', '(', *Predicate*, ')'
| *Type*, '≡', *Type*

Type assumption = 'given', '(', (*Identifier* | 'BIG'), ')'
| *Identifier*, '∈', *Expression*

Type sequent = [*Type assumption*, {*Type assumption*}], ⊢, *Type predicate*

$$\textit{Type rule} = \frac{\{\textit{Type sequent}\}}{\textit{Type sequent}}$$

Typechecking a predicate **p** containing no free variables consists of applying Algorithm 5.1 to the type sequent given(BIG) ⊢ check(**p**). Typechecking an expression **e** can be done by typechecking the predicate **e** = **e**. If there are free variables, then appropriate type assumptions have to be prepended to the type sequent before typechecking.

From now on, the metavariable **i** is restricted to those strings that belong to the syntactic category *Identifier*. In this section, the metavariable **E** is restricted to those strings that are comma-separated lists of strings that belong to the syntactic category *Type assumption*, and the metavariables **T** and **U** are restricted to those strings that belong to the syntactic category *Type*.

The typechecking algorithm given by Abrial proceeds as follows:

Algorithm 5.1 (Typechecking B according to Abrial) *In this algorithm and in the names of the typechecking rules, the apostrophe is a decoration of numbers, and i' denotes the number i decorated with an apostrophe. The algorithm uses unification (cf. Algorithm 4.1) as tailored for the type language.*

Precondition: *Input is a string that belongs to the syntactic category Type sequent.*

1. *Use the rewrite rules to eliminate all derived constructs from the input.*
2. *Let $i \leftarrow 1$.*
3. *Unify the input with the consequent of rule T i, and assign the unifier to σ. If unsuccessful, go to step 7.*
4. *For each antedecent of rule T i, instantiate its variables using σ.*
5. *Verify each non-type-sequent antedecent. (This may require further unification; if so, instantiate the unified variables in all antecedents.) If at least one of them fail, go to step 7.*
6. *Apply this algorithm recursively to each type sequent antedecent. If all succeed, halt with success.*
7. *If there exists a rule T i', let $i \leftarrow i'$ and go to step 3.*
8. *If $i = j'$ for some j, let $i \leftarrow j$ and repeat this step.*
9. *If there is no number n strictly greater than i for which there exists a rule T n, halt with failure.*
10. *Let $i \leftarrow j$, where j is the least number strictly greater than i for which there exists a rule T j, and go to step 3.*

Postcondition: *Output is an indication of success or failure.*

The typing rules for Algorithm 5.1 are given in Table 5.9. Composite predicates are decomposed using rules T 1–T 3. Quantification is removed by enlarging the environment in rules T 4–T 6. Rules T 7 and T 8 transform the typechecking of primitive predicates ("$\in$" and "$=$") to appropriate type equivalence conjectures. Rules T 9–T 18 and their primed variants decompose the expressions involved in a type equivalence conjecture (T 16 and T 16′ enlarge the environment in the process). Rules T 19–T 20 decompose power set and cartesian product expressions. Rule T 21 acts as the base case.

T 1 $$\frac{\mathbf{E} \vdash \mathsf{check}(\mathbf{p}) \quad \mathbf{E} \vdash \mathsf{check}(\mathbf{q})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{p} \wedge \mathbf{q})}$$

T 2 $$\frac{\mathbf{E} \vdash \mathsf{check}(\mathbf{p}) \quad \mathbf{E} \vdash \mathsf{check}(\mathbf{q})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{p} \Rightarrow \mathbf{q})}$$

T 3 $$\frac{\mathbf{E} \vdash \mathsf{check}(\mathbf{p})}{\mathbf{E} \vdash \mathsf{check}(\neg \mathbf{p})}$$

T 4 $$\frac{\mathbf{i} \setminus \mathbf{s} \quad \mathbf{i} \setminus \mathbf{q} \text{ for each } \mathbf{q} \text{ in } \mathbf{E} \quad \mathbf{E}, \mathbf{i} \in \mathbf{s} \vdash \mathsf{check}(\mathbf{p})}{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{i} \cdot (\mathbf{i} \in \mathbf{s} \Rightarrow \mathbf{p}))}$$

T 5 $$\frac{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x} \cdot (\mathbf{x} \in \mathbf{s} \Rightarrow \forall\, \mathbf{y} \cdot (\mathbf{y} \in \mathbf{t} \Rightarrow \mathbf{p})))}{\mathbf{E} \vdash \mathsf{check}(\forall (\mathbf{x}, \mathbf{y}) \cdot (\mathbf{x}, \mathbf{y} \in \mathbf{s} \times \mathbf{t} \Rightarrow \mathbf{p}))}$$

T 6 $$\frac{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q} \wedge \mathbf{r}))}{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x} \cdot (\mathbf{p} \wedge \mathbf{q} \Rightarrow \mathbf{r}))}$$

T 7 $$\frac{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}) \equiv \mathsf{type}(\mathbf{f})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{e} = \mathbf{f})}$$

T 8 $$\frac{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}) \equiv \mathsf{super}(\mathbf{f})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{e} \in \mathbf{f})}$$

T 9 $$\frac{\mathbf{i} \in \mathbf{s} \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathbf{i}) \equiv \mathbf{U}}$$

T 10 $$\frac{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}) \times \mathsf{type}(\mathbf{f}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}, \mathbf{f}) \equiv \mathbf{U}}$$

T 11 $$\frac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathsf{choice}(\mathbf{s})) \equiv \mathbf{U}}$$

T 12 $$\frac{\mathbf{E} \vdash \mathbb{P}(\mathsf{super}(\mathbf{s})) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathbf{s}) \equiv \mathbf{U}}$$

T 13 $$\frac{\mathbf{i} \in \mathbf{s} \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbb{P}(\mathbf{U})}{\mathbf{E} \vdash \mathsf{super}(\mathbf{i}) \equiv \mathbf{U}}$$

T 14 $$\frac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \times \mathsf{super}(\mathbf{t}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\mathbf{s} \times \mathbf{t}) \equiv \mathbf{U}}$$

T 15 $$\frac{\mathbf{E} \vdash \mathbb{P}(\mathsf{super}(\mathbf{s})) \equiv U}{\mathbf{E} \vdash \mathsf{super}(\mathbb{P}(\mathbf{s})) \equiv \mathbf{U}}$$

T 16 $$\frac{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x} \cdot (\mathbf{x} \in \mathbf{s} \Rightarrow \mathbf{p}) \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{s} \wedge \mathbf{p} \,\}) \equiv \mathbf{U}}$$

T 17 $$\frac{\mathsf{given}\ \mathbf{i} \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathbf{i} \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\mathbf{i}) \equiv \mathbf{U}}$$

T 18 $$\frac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbb{P}(\mathbf{U})}{\mathbf{E} \vdash \mathsf{super}(\mathsf{choice}(\mathbf{s})) \equiv \mathbf{U}}$$

T 19 $$\frac{\mathbf{T} \equiv \mathbf{U}}{\mathbf{E} \vdash \mathbb{P}(\mathbf{T}) \equiv \mathbb{P}(\mathbf{U})}$$

T 20 $$\frac{\mathbf{E} \vdash \mathbf{T} \equiv \mathbf{T}' \quad \mathbf{E} \vdash \mathbf{T}' \equiv \mathbf{U}'}{\mathbf{E} \vdash \mathbf{T} \times \mathbf{U} \equiv \mathbf{T}' \times U'}$$

T 21 $$\frac{\mathsf{given}(\mathbf{i}) \text{ occurs in } \mathbf{E}}{\mathbf{E} \vdash \mathbf{i} \equiv \mathbf{i}}$$

Rules 9′–18′ can be obtained from the corresponding rules 9–18 by applying to their consequents the rewrite rule $\mathbf{E} \vdash \mathbf{T} \equiv \mathbf{U} :\Rightarrow \mathbf{E} \vdash \mathbf{U} \equiv \mathbf{T}$.

Table 5.9: Typechecking rules for Algorithm 5.1.

5.3.4 Better typechecking algorithms There are a couple of problems with the typechecking procedure as reproduced above. For example, it halts with failure for the type sequent $\mathsf{given}(s), \mathsf{given}(t) \vdash \mathsf{check}(\forall x, y \cdot (x \in s \land y \in t \Rightarrow x = y))$, although by the cartesian product axiom the predicate being checked is equivalent to $\forall x, y \cdot (x, y \in s \times t \Rightarrow x = y)$, which does typecheck under the same environment. Abrial himself uses predicates similar to the first one quite liberally. Another example is $\mathsf{given}(s) \vdash \mathsf{check}(\exists x \cdot (x = s \Rightarrow x = x))$, for which the algorithm halts with failure but where it (intuitively speaking) should halt with success. Even with $\mathsf{given}(s) \vdash \mathsf{check}(\exists x \cdot x \in s)$, the algorithm halts with failure, since it expects to see a compound predicate inside a quantification.

A solution to these problems was devised by the present author in the summer of 2002[5]. The basic idea is to introduce a preprocessing phase that rewrites the universal quantified predicates to a *type-specifying normal form*. To specify this normal form, we'll assume that there is a total ordering of identifiers, a canonical order. A predicate or an expression is in this normal form if it contains no derived constructs and all universal quantifications and all set comprehensions that it contains are in this normal form. A universal quantification is in this normal form, if it is of the form $\forall \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q})$, and all of the following conditions hold:

1. The variable $\mathbf{x}$ is sorted. A variable is sorted, if it is an identifier or it is of the form $(\mathbf{y}, \mathbf{i})$, where $\mathbf{y}$ is sorted and $\mathbf{i}$ is an identifier.

2. The predicate $\mathbf{p}$ is type-specifying. A predicate is type-specifying if it is in one of the following forms:

   (a) $\mathbf{i} \in \mathbf{s}$ (in which case $\mathbf{i}$ *is specified* by the predicate),

   (b) $\mathbf{i} = \mathbf{s}$ (in which case $\mathbf{i}$ *is specified* by the predicate), or

   (c) $\mathbf{r} \land \mathbf{r}'$, where $\mathbf{r}$ and $\mathbf{r}'$ are type-specifying and where all identifiers that $\mathbf{r}$ specifies are strictly less than all identifiers that $\mathbf{r}'$ specifies in the total ordering of identifiers (in which case the identifiers that are specified by either $\mathbf{r}$ or $\mathbf{r}'$ *are specified* also by this predicate).

3. The predicate $\mathbf{q}$ is in this normal form.

The following algorithm will convert a predicate to the type-specifying normal form. It is complicated, as it needs to consider many cases.

Algorithm 5.2 (Convert a predicate to the type-specifying normal form) *In this algorithm, statements in parentheses at the beginning of an item are assertions. If they contain metavariables, then unification is used to make all the assertions true; the resulting bindings for the metavariables are valid until the end of the item. If an assertion is false, then the algorithm is broken.*

Precondition: *Input is a string that belongs to the syntactic category Predicate or Expression and contains no derived constructs.*

1. *If the input is of the form* $\mathbf{p} \wedge \mathbf{q}$*, apply this algorithm to* $\mathbf{p}$ *to obtain* $\mathbf{p}'$ *and to* $\mathbf{q}$ *to obtain* $\mathbf{q}'$*, output* $\mathbf{p}' \wedge \mathbf{q}'$ *and halt with success. If any of this fails, halt with failure.*

2. *If the input is of the form* $\mathbf{p} \Rightarrow \mathbf{q}$*, apply this algorithm to* $\mathbf{p}$ *to obtain* $\mathbf{p}'$ *and to* $\mathbf{q}$ *to obtain* $\mathbf{q}'$*, and output* $\mathbf{p}' \Rightarrow \mathbf{q}'$ *and halt with success. If any of this fails, halt with failure.*

3. *If the input is of the form* $\neg\mathbf{p}$*, apply this algorithm to* $\mathbf{p}$ *to obtain* $\mathbf{p}'$*, and output* $\neg\mathbf{p}'$ *and halt with success. If any of this fails, halt with failure.*

4. *If the input is of the form* $\forall\, \mathbf{x} \cdot \mathbf{p}$*, go to step 14.*

5. *If the input is of the form* $\mathbf{e} = \mathbf{f}$*, apply this algorithm to* $\mathbf{e}$ *to obtain* $\mathbf{e}'$ *and to* $\mathbf{f}$ *to obtain* $\mathbf{f}'$*, and output* $\mathbf{e}' = \mathbf{f}'$ *and halt with success. If any of this fails, halt with failure.*

6. *If the input is of the form* $\mathbf{e} \in \mathbf{s}$*, apply this algorithm to* $\mathbf{e}$ *to obtain* $\mathbf{e}'$ *and to* $\mathbf{s}$ *to obtain* $\mathbf{s}'$*, and output* $\mathbf{e}' \in \mathbf{s}'$ *and halt with success. If any of this fails, halt with failure.*

7. *If the input is of the form* $[\mathbf{x} := \mathbf{e}]\mathbf{p}$*, apply this algorithm to* $\mathbf{e}$ *to obtain* $\mathbf{e}'$ *and to* $\mathbf{p}$ *to obtain* $\mathbf{p}'$*, and output* $[\mathbf{x} := \mathbf{e}']\mathbf{p}'$*, and halt with success. If any of this fails, halt with failure.*

8. *If the input is of the form* $[\mathbf{x} := \mathbf{e}]\mathbf{f}$*, apply this algorithm to* $\mathbf{e}$ *to obtain* $\mathbf{e}'$ *and to* $\mathbf{f}$ *to obtain* $\mathbf{f}'$*, and output* $[\mathbf{x} := \mathbf{e}']\mathbf{f}'$*, and halt with success. If any of this fails, halt with failure.*

9. *If the input is of the form* $\mathsf{choice}(\mathbf{s})$*, apply this algorithm to* $\mathbf{s}$ *to obtain* $\mathbf{s}'$ *and output* $\mathsf{choice}(\mathbf{s}')$ *and halt with success. If any of this fails, halt with failure.*

10. *If the input is of the form* $\mathbb{P}(\mathbf{s})$*, apply this algorithm to* $\mathbf{s}$ *to obtain* $\mathbf{s}'$ *and output* $\mathbb{P}(\mathbf{s}')$ *and halt with success. If any of this fails, halt with failure.*

11. *If the input is of the form* $\mathbf{s} \times \mathbf{t}$*, apply this algorithm to* $\mathbf{s}$ *to obtain* $\mathbf{s}'$ *and to* $\mathbf{t}$ *to obtain* $\mathbf{t}'$ *and output* $\mathbf{s}' \times \mathbf{t}'$ *and halt with success. If any of this fails, halt with failure.*

12. *If the input is of the form* $\{\, \mathbf{x} \mid \mathbf{p} \,\}$*, halt with failure (there is no sensible way to apply this algorithm to such input).*

13. *If the input is of the form* BIG*, output* BIG *and halt with success. If any of this fails, halt with failure.*

14. *(Input is of the form* $\forall\, \mathbf{x} \cdot \mathbf{p}$*.) Replace input by* $\forall\, \mathbf{x}' \cdot \mathbf{p}$*, where* $\mathbf{x}'$ *is* $\mathbf{x}$ *sorted.*

15. *(Input is still of the form* $\forall\, \mathbf{x} \cdot \mathbf{p}$ *and* $\mathbf{x}$ *is sorted.) If* $\mathbf{p}$ *is not of the form* $\mathbf{q} \Rightarrow \mathbf{r}$*, replace input by* $\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{x} = \mathbf{x})$*.*

16. *(Input is of the form* $\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q})$ *and* $\mathbf{x}$ *is sorted.) Use Algorithm 5.3 with* $\mathbf{p}$ *and* $\mathbf{q}$ *with* $\Rightarrow$ *as inputs. Let the output be* $\mathbf{p}'$ *and* $\mathbf{q}'$ *(both possibly empty).*

17. *If* $\mathbf{p}'$ *is empty, halt with failure.*

18. *If* $\mathbf{q}'$ *is empty, go to step 23.*

19. *(Input is of the form* $\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q})$ *and* $\mathbf{x}$ *is sorted.) Apply this algorithm to* $\mathbf{p}'$*. Let* $\mathbf{p}''$ *be the result.*

20. *Apply this algorithm to* $\mathbf{q}'$*. Let* $\mathbf{q}''$ *be the result.*

21. *Rearrange* $\mathbf{p}''$ *so that identifiers on the left-hand side of a "*$\in$*" are in the canonical order. Let* $\mathbf{p}'''$ *be the result.*

22. *(Input is of the form* $\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q})$ *and* $\mathbf{x}$ *is sorted.) Output* $\forall\, \mathbf{x} \cdot (\mathbf{p}''' \Rightarrow \mathbf{q}'')$ *and halt successfully.*

23. *(Input is of the form* $\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q})$ *and* $\mathbf{x}$ *is sorted and* $\mathbf{p}'$ *and* $\mathbf{q}'$ *are the output of Algorithm 5.3 with* $\mathbf{p}$ *and* $\mathbf{q}$ *with* $\Rightarrow$ *as the inputs, and* $\mathbf{p}$ *is nonempty and* $\mathbf{q}$ *is empty.) Apply this algorithm to* $\mathbf{p}'$*. Let* $\mathbf{p}''$ *be the result.*

24. *Rearrange* $\mathbf{p}''$ *so that identifiers on the left-hand side of a "*$\in$*" are in the canonical order. Let* $\mathbf{p}'''$ *be the result.*

25. *(Input is of the form* $\forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q})$ *and* $\mathbf{x}$ *is sorted.) Output* $\forall\, \mathbf{x} \cdot (\mathbf{p}''' \Rightarrow \mathbf{x} = \mathbf{x})$ *and halt successfully.*

Postcondition: *Output is a string that belongs to the same syntactic category as the input, contains no derived constructs and is in the type-specifying normal form, or an indication of failure.*

A helper algorithm is needed to work the predicates within quantifications.

Algorithm 5.3 *In this algorithm, statements in parentheses at the beginning of an item are assertions. If they contain metavariables, then unification is used to make all the assertions true; the resulting bindings for the metavariables are valid until the end of the item. If an assertion is false, then the algorithm is broken.*

Precondition: *Input is two strings that belong to the syntactic category Predicate. The second string can also be empty. Order matters. Additionally, a logical constant, either $\wedge$ or $\Rightarrow$, is given as input (denoted here by $\circ$).*

1. *If the first input string is of the form $\mathbf{i} \in \mathbf{s}$, output the input and halt.*
2. *If the first input string is of the form $\mathbf{e}, \mathbf{f} \in \mathbf{s} \times \mathbf{t}$, apply this algorithm to $\mathbf{e} \in \mathbf{s} \wedge \mathbf{f} \in \mathbf{t}$, and output whatever it outputs, then halt.*
3. *If the first input string is of the form $\mathbf{p} \wedge \mathbf{q}$, go to step 6.*
4. *If the second string is empty, output the empty string and the first string and halt.*
5. *(The first input string is $\mathbf{p}$ and the second input string is $\mathbf{q}$.) Output the empty string and the string $\mathbf{p} \circ \mathbf{q}$ and halt.*
6. *(First input string is of the form $\mathbf{p} \wedge \mathbf{q}$ and the second input string is $\mathbf{r}$.) Apply this algorithm to $\mathbf{p}$ and $\mathbf{r}$ with $\circ$. Let $\mathbf{p}'$ be the first output string (possibly empty) and let $\mathbf{r_p}$ (possibly empty) be the second output string.*
7. *(First input string is of the form $\mathbf{p} \wedge \mathbf{q}$ and the second input string is $\mathbf{r}$.) Apply this algorithm to $\mathbf{q}$ and $\mathbf{r_p}$ with $\circ$. Let $\mathbf{q}'$ be the first output string (possibly empty) and let $\mathbf{r_q}$ be the second output string (possibly empty).*
8. *If $\mathbf{p}'$ is empty, output $\mathbf{q}'$ and $\mathbf{q_r}$ and halt.*
9. *If $\mathbf{q}'$ is empty, output $\mathbf{p}'$ and $\mathbf{q_r}$ and halt.*
10. *Output $\mathbf{p}' \wedge \mathbf{q}'$ and $\mathbf{q_r}$ and halt.*

Postcondition: *Output is two strings that belong to the syntactic category Predicate. Both can also be empty. Order matters.*

Now, the typechecking algorithm can be modified to read as follows:

Algorithm 5.4 (Typechecking B better) *As before, in this algorithm and in the names of the typechecking rules, the apostrophe is a decoration of numbers, and i' denotes the number i decorated with an apostrophe.*

Precondition: *Input is a string that belongs to the syntactic category Type sequent.*

1. *Use the rewrite rules to eliminate all derived constructs from the input.*
2. *If the input is of the form $\mathbf{E} \vdash \mathsf{check}(\mathbf{p})$, apply Algorithm 5.2 to $\mathbf{p}$ to obtain $\mathbf{p}'$ and replace the input by $\mathbf{E} \vdash \mathsf{check}(\mathbf{p}')$. If this fails, halt with failure.*
3. *Let $i \leftarrow 1$.*
4. *Unify the input with the consequent of rule Tm i, and assign the unifier to σ. If unsuccessful, go to step 8.*
5. *For each antedecent of rule Tm i, instantiate its variables using σ.*
6. *Verify each non-type-sequent antedecent. (This may require further unification; if so, instantiate the unified variables in all antecedents.) If at least one of them fail, go to step 8.*
7. *Apply this algorithm recursively to each type sequent antedecent. If all succeed, halt with success.*
8. *If there exists a rule Tm i', let $i \leftarrow i'$ and go to step 4.*
9. *If $i = j'$ for some j, let $i \leftarrow j$ and repeat this step.*
10. *If there is no number n strictly greater than i for which there exists a rule Tm n, halt with failure.*
11. *Let $i \leftarrow j$, where j is the least number strictly greater than i for which there exists a rule Tm j, and go to step 4.*

Typechecking rules stay largely the same, so for most rules T i, the corresponding rule Tm i is identical. The exceptions are those rules that deal with quantification; they are listed in Table 5.10.

This still does not solve the problem of $\forall\, x \cdot (x = e \Rightarrow \mathbf{p})$, but to solve that, a new production for *Type assumption* would have to be added, and many rules would need to be changed.

$$\text{Tm 5}\quad \frac{\mathbf{i} \text{ and } \mathbf{i}' \text{ are distinct} \quad \mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x}, \mathbf{i} \cdot (\mathbf{p} \Rightarrow \mathbf{q})) \quad \mathbf{E} \vdash \mathsf{check}(\mathbf{i}' \in \mathbf{s})}{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x}, \mathbf{i} \cdot (\mathbf{p} \wedge \mathbf{i}' \in \mathbf{s} \Rightarrow \mathbf{q}))}$$

$$\text{Tm 6}\quad \frac{\mathbf{E} \vdash \mathsf{check}\, \forall\, \mathbf{i} \cdot (\mathbf{i} \in \mathbf{s} \Rightarrow \forall\, \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q}))}{\mathbf{E} \vdash \mathsf{check}(\forall\, \mathbf{x}, \mathbf{i} \cdot (\mathbf{p} \wedge \mathbf{i} \in \mathbf{s} \Rightarrow \mathbf{q}))}$$

Table 5.10: Modified typechecking rules for Algorithm 5.4.

Unfortunately, there is a huge problem with this approach. It is not possible to rearrange the variable of a set comprehension, as the type of the set comprehension itself is dependent on the structure of the variable. Another approach seems to be going through the existing variable and searching the abstract syntax tree of the type-constraining predicate for set membership or equality predicates from which a type for each identifier in the type can be extracted, and then constructing a type for the whole variable from those bits. This idea has been implemented in Ebba.

5.3.5 Derived formulae for set notation We define two derived predicates and several derived expressions.

Predicate = *Expression*, '$\subseteq$', *Expression*
| *Expression*, '$\subset$', *Expression*
Expression = *Expression*, '$\cup$', *Expression*
| *Expression*, '$\cap$', *Expression*
| *Expression*, '$-$', *Expression*
| '{', *Expression*, '}'
| '$\varnothing$'
| '$\mathbb{P}_1$', '(', *Expression*, ')'
| 'inter', '(', *Expression*, ')'
| 'union', '(', *Expression*, ')'
| '$\bigcup$', *Variable*, '$\cdot$', '(', *Predicate*, '|', *Expression*, ')'
| '$\bigcap$', *Variable*, '$\cdot$', '(', *Predicate*, '|', *Expression*, ')'

$$\mathbf{s} \subseteq \mathbf{t} :\Rightarrow \mathbf{s} \in \mathbb{P}(\mathbf{t})$$
$$\mathbf{s} \subset \mathbf{t} :\Rightarrow \mathbf{s} \subseteq \mathbf{t} \wedge \neg \mathbf{s} = \mathbf{t}$$
$$\mathbf{s} \cup \mathbf{t} :\Rightarrow \{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{u} \wedge (\mathbf{x} \in \mathbf{s} \vee \mathbf{x} \in \mathbf{t}) \,\} \qquad \text{where } \mathbf{s} \subseteq \mathbf{u} \text{ and } \mathbf{t} \subseteq \mathbf{u}$$
$$\mathbf{s} \cap \mathbf{t} :\Rightarrow \{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{u} \wedge (\mathbf{x} \in \mathbf{s} \wedge \mathbf{x} \in \mathbf{t}) \,\} \qquad \text{where } \mathbf{s} \subseteq \mathbf{u} \text{ and } \mathbf{t} \subseteq \mathbf{u}$$
$$\mathbf{s} - \mathbf{t} :\Rightarrow \{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{u} \wedge (\mathbf{x} \in \mathbf{s} \wedge \neg \mathbf{x} \in \mathbf{t}) \,\} \qquad \text{where } \mathbf{s} \subseteq \mathbf{u} \text{ and } \mathbf{t} \subseteq \mathbf{u}$$
$$\{\mathbf{s}\} :\Rightarrow \{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{u} \wedge \mathbf{x} = \mathbf{s} \,\} \qquad \text{where } \mathbf{s} \subseteq \mathbf{u}$$

$$
\begin{aligned}
\{\mathbf{s},\mathbf{t}\} &:\Rightarrow \{\mathbf{s}\}\cup\{\mathbf{t}\} \\
\varnothing &:\Rightarrow \mathbf{u}-\mathbf{u} \\
\mathbb{P}_1(\mathbf{s}) &:\Rightarrow \mathbb{P}(\mathbf{s})-\{\varnothing\} \\
\mathrm{union}(\mathbf{u}) &:\Rightarrow \{\,\mathbf{x}\in\mathbf{s}\wedge\forall\,\mathbf{y}\cdot(\mathbf{y}\in\mathbf{u}\Rightarrow\mathbf{x}\in\mathbf{y})\,\} && \text{where } \mathbf{u}\in\mathbb{P}_1(\mathbb{P}(\mathbf{s})) \\
\mathrm{inter}(\mathbf{u}) &:\Rightarrow \{\,\mathbf{x}\in\mathbf{s}\wedge\forall\,\mathbf{y}\cdot(\mathbf{y}\in\mathbf{u}\wedge\mathbf{x}\in\mathbf{y})\,\} && \text{where } \mathbf{u}\in\mathbb{P}_1(\mathbb{P}(\mathbf{s})) \\
\bigcup\mathbf{x}\cdot(\mathbf{p}\mid\mathbf{e}) &:\Rightarrow \mathrm{union}(\{\,\mathbf{x}\mid\mathbf{p}\wedge\mathbf{x}=\mathbf{e}\,\}) \\
\bigcap\mathbf{x}\cdot(\mathbf{p}\mid\mathbf{e}) &:\Rightarrow \mathrm{inter}(\{\,\mathbf{x}\mid\mathbf{p}\wedge\mathbf{x}=\mathbf{e}\,\})
\end{aligned}
$$

We deviate from Abrial in one point here: Abrial defines $\varnothing$ as $\mathsf{BIG}-\mathsf{BIG}$. He notes that by extensionality, $\varnothing$ is equal to $\mathbf{u}-\mathbf{u}$ for whatever set $\mathbf{u}$. However, typechecking is done before any proofs, so this cannot be used as an argument for Abrial's definition. Also, our definition of $\bigcup$ and $\bigcap$ are slightly simpler but equivalent to Abrial's.

Abrial's definitions for $\mathbf{s}\cup\mathbf{t}$, $\mathbf{s}\cap\mathbf{t}$, $\mathbf{s}-\mathbf{t}$ and $\{\mathbf{s}\}$ given here have one big disadvantage. They require typechecking to find a $\mathbf{u}$ such that $\mathbf{s}\subseteq\mathbf{u}$ and $\mathbf{t}\subseteq\mathbf{u}$ both become true. Clearly such a $\mathbf{u}$ can be found if the predicate $\mathbf{s}=\mathbf{t}$ can be successfully typechecked, and one candidate for $\mathbf{u}$ is the set counterpart of $\mathsf{type}(\mathbf{s})$. In other words, it requires typechecking to instantiate a metavariable permanently (which is not what Algorithms 5.1 and 5.4 are designed to do). Also we need a fresh metavariable for each rewrite, so to properly describe the rewrite rules, one would need meta-metavariables!

A solution is inspired by Abrial and Mussat's article [2] on conditional definitions. The rewrite rules for set intersection and set difference could be given as follows, too:

$$
\begin{aligned}
\mathbf{s}\cap\mathbf{t} &:\Rightarrow \{\,x\mid\mathbf{x}\in\mathbf{s}\cup\mathbf{t}\wedge(\mathbf{x}\in\mathbf{s}\wedge\mathbf{x}\in\mathbf{t})\,\} \\
\mathbf{s}-\mathbf{t} &:\Rightarrow \{\,x\mid\mathbf{x}\in\mathbf{s}\cup\mathbf{t}\wedge(\mathbf{x}\in\mathbf{s}\wedge\neg\mathbf{x}\in\mathbf{t})\,\}
\end{aligned}
$$

These definitions have the advantage that they don't mention a mysterious new $\mathbf{u}$ metavariable. However, unless set union is cleverly redefined in a way that does not require $\mathbf{u}$, we have made things more complicated. Fortunately, there is a way to do this.

The important step is to make set union an undefined notion and use an axiom to characterize it:

SET 7: Union $\dfrac{}{\mathbf{h} \vdash \mathbf{e} \in \mathbf{s} \cup \mathbf{t} \Leftrightarrow \mathbf{e} \in \mathbf{s} \vee \mathbf{e} \in \mathbf{t}}$

Naturally, new typing rules are needed, too:

T 22 $\dfrac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U} \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{t}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\mathbf{s} \cup \mathbf{t}) \equiv \mathbf{U}}$

T 22′ $\dfrac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U} \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{t}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathbf{U} \equiv \mathsf{super}(\mathbf{s} \cup \mathbf{t})}$

Similarly, the singular set can be made an undefined notion. It too needs an axiom and typing rules. Here **f** must not be a pair.

SET 8: Singular set $\dfrac{}{\mathbf{h} \vdash \mathbf{e} \in \{\mathbf{f}\} \Leftrightarrow \mathbf{e} = \mathbf{f}}$

T 25 $\dfrac{\mathbf{E} \vdash \mathsf{type}(\mathbf{f}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\{\mathbf{f}\}) \equiv \mathbf{U}}$

T 25′ $\dfrac{\mathbf{E} \vdash \mathsf{type}(\mathbf{f}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathbf{U} \equiv \mathsf{super}(\{\mathbf{f}\})}$

After this, we can fix the typechecking to handle typing through equality by rewriting, for the purposes of type checking, every equality predicate $\mathbf{e} = \mathbf{f}$ as $\mathbf{e} \in \{\mathbf{f}\} \wedge \mathbf{f} \in \{\mathbf{e}\}$. We will assume later on that this has been done.

Binary relations, functions and their operations are derived constructs, too. Their abstract syntax productions are given in Table 5.11 and the corresponding rewrite rules are given in Table 5.12.

5.4 Recursively defined sets

Many fundamental mathematical structures, such as the set of natural numbers, the sets of finite sequences and the sets of finite trees are most aptly defined using a recursive process. In this section, we will introduce definitions and results that allow us to do this. Additionally, we will give a definition for some of the abovementioned constructs.

5.4.1 Fixpoints The fundamental idea of recursive definition of a set $s \subseteq t$ is to find a set transformer $f \in \mathbb{P}(t) \to \mathbb{P}(t)$ for which s is a fixpoint, $f(s) = s$. To use this device, we will need a functional which will find us this fixpoint given a set transformer.

Note first that $f(s) = s$ can be decomposed to $f(s) \subseteq s$ and $s \subseteq f(s)$. This suggests that one or more of the following sets might be fixpoints:

$$\mathrm{inter}(\{\, x \mid x \in \mathbb{P}(t) \wedge f(x) \subseteq x \,\}) \tag{5.1}$$

Expression =	*Expression*, '$\leftrightarrow$', *Expression*	(* binary relation *)
	Expression, '$^{-1}$'	(* inverse *)
	'dom', '(', *Expression*, ')'	(* domain set *)
	'ran', '(', *Expression*, ')'	(* range set *)
	Expression, '; ', *Expression*	(* forward composition *)
	Expression, '$\circ$', *Expression*	(* composition *)
	'id', '(', *Expression*, ')'	(* identity relation *)
	Expression, '$\lhd$', *Expression*	(* domain restriction *)
	Expression, '$\rhd$', *Expression*	(* range restriction *)
	Expression, '$\hookleftarrow$', *Expression*	(* domain subtraction *)
	Expression, '$\hookrightarrow$', *Expression*	(* range subtraction *)
	Expression, '[', *Expression*, ']'	(* image of a set *)
	Expression, '$<+$', *Expression*	(* overriding *)
	Expression, '$\otimes$', *Expression*	(* direct product *)
	'prj_1', '(', *Expression*, ',', *Expression*, ')'	(* left projection *)
	'prj_2', '(', *Expression*, ',', *Expression*, ')'	(* right projection *)
	Expression, ' $\parallel$ ', *Expression*	(* parallel product *)
	Expression, '$\nrightarrow$', *Expression*	(* partial function *)
	Expression, '$\rightarrow$', *Expression*	(* total function *)
	Expression, '$\rightarrowtail$', *Expression*	(* partial injection *)
	Expression, '$\hookrightarrow$', *Expression*	(* total injection *)
	Expression, '$\twoheadrightarrow$', *Expression*	(* partial surjection *)
	Expression, '$\twoheadrightarrow$', *Expression*	(* total surjection *)
	Expression, '$\rightarrowtail\!\!\!\!\twoheadrightarrow$', *Expression*	(* partial bijection *)
	Expression, '$\hookrightarrow\!\!\!\!\twoheadrightarrow$', *Expression*	(* total bijection *)
	'λ', *Variable*, '$\cdot$', '(', *Predicate*, '$\mid$', *Expression*, ')'	(* λ-abstraction *)
	Expression, '(', *Expression*, ')'	(* function application *)

Table 5.11: Abstract syntax productions for relation and function constructs.

$$
\begin{array}{rll}
\mathbf{s} \leftrightarrow \mathbf{t} & :\Rightarrow \mathbb{P}(\mathbf{s} \times \mathbf{t}) & \\
\mathbf{r}^{-1} & :\Rightarrow \{\, \mathbf{y}, \mathbf{x} \mid (\mathbf{y}, \mathbf{x}) \in \mathbf{t} \times \mathbf{s} \wedge (\mathbf{x}, \mathbf{y}) \in \mathbf{r} \,\} & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \\
\mathrm{dom}(\mathbf{r}) & :\Rightarrow \{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{s} \wedge \exists\, \mathbf{y} \cdot (\mathbf{y} \in \mathbf{t} \wedge (\mathbf{x}, \mathbf{y}) \in \mathbf{r}) \,\} & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \\
\mathrm{ran}(\mathbf{r}) & :\Rightarrow \mathrm{dom}(\mathbf{r}^{-1}) & \\
\mathbf{q};\ \mathbf{r} & :\Rightarrow \{\, \mathbf{x}, \mathbf{z} \mid (\mathbf{x}, \mathbf{z}) \in \mathbf{s} \times \mathbf{u} \wedge & \\
 & \qquad \exists\, \mathbf{y} \cdot (\mathbf{y} \in \mathbf{t} \wedge (\mathbf{x}, \mathbf{y}) \in \mathbf{q} \wedge (\mathbf{y}, \mathbf{z}) \in \mathbf{r}) \,\} & \text{where } \mathbf{q} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{r} \in \mathbf{t} \leftrightarrow \mathbf{u} \\
\mathbf{r} \circ \mathbf{q} & :\Rightarrow \mathbf{q};\ \mathbf{r} & \text{where } \mathbf{q} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{r} \in \mathbf{t} \leftrightarrow \mathbf{u} \\
\mathrm{id}(\mathbf{s}) & :\Rightarrow \{\, \mathbf{x}, \mathbf{y} \mid (\mathbf{x}, \mathbf{y}) \in \mathbf{s} \times \mathbf{s} \wedge \mathbf{x} = \mathbf{y} \,\} & \\
\mathbf{u} \lhd \mathbf{r} & :\Rightarrow \mathrm{id}(\mathbf{u});\ \mathbf{r} & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{u} \subseteq s \\
\mathbf{r} \rhd \mathbf{v} & :\Rightarrow \mathbf{r};\ \mathrm{id}(\mathbf{v}) & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{v} \subseteq t \\
\mathbf{u} \lhd\!\!\!\!- \mathbf{r} & :\Rightarrow (\mathrm{dom}(\mathbf{r}) - \mathbf{u}) \lhd \mathbf{r} & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{u} \subseteq s \\
\mathbf{r} \rhd\!\!\!\!- \mathbf{v} & :\Rightarrow \mathbf{r} \rhd (\mathrm{ran}(\mathbf{r}) - \mathbf{v}) & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{v} \subseteq t \\
\mathbf{r}[\mathbf{w}] & :\Rightarrow \mathrm{ran}(\mathbf{w} \lhd \mathbf{r}) & \text{where } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{w} \subseteq s \\
\mathbf{q} <\!\!+ \mathbf{r} & :\Rightarrow (\mathrm{dom}(\mathbf{r}) \lhd\!\!\!\!- \mathbf{q}) \cup r & \text{where } \mathbf{q} \in \mathbf{s} \leftrightarrow \mathbf{t} \text{ and } \mathbf{r} \in \mathbf{s} \leftrightarrow \mathbf{t} \\
\mathbf{f} \otimes \mathbf{g} & :\Rightarrow \{\, \mathbf{x}, (\mathbf{y}, \mathbf{z}) \mid (\mathbf{x}, (\mathbf{y}, \mathbf{z})) \in \mathbf{s} \times (\mathbf{u} \times \mathbf{v}) \wedge & \\
 & \qquad (\mathbf{x}, \mathbf{y}) \in \mathbf{f} \wedge (\mathbf{x}, \mathbf{z}) \in \mathbf{g} \,\} & \text{where } \mathbf{f} \in \mathbf{s} \leftrightarrow \mathbf{u} \text{ and } \mathbf{g} \in \mathbf{s} \leftrightarrow \mathbf{v} \\
\mathrm{prj}_1(\mathbf{s}, \mathbf{t}) & :\Rightarrow \{\, \mathbf{x}, \mathbf{y}, \mathbf{z} \mid (\mathbf{x}, \mathbf{y}, \mathbf{z}) \in \mathbf{x} \times \mathbf{t} \times \mathbf{s} \wedge \mathbf{z} = \mathbf{x} \,\} & \\
\mathrm{prj}_2(\mathbf{s}, \mathbf{t}) & :\Rightarrow \{\, \mathbf{x}, \mathbf{y}, \mathbf{z} \mid (\mathbf{x}, \mathbf{y}, \mathbf{z}) \in \mathbf{x} \times \mathbf{t} \times \mathbf{s} \wedge \mathbf{z} = \mathbf{y} \,\} & \\
\mathbf{h} \parallel \mathbf{k} & :\Rightarrow \{\, (\mathbf{x}, \mathbf{y}), (\mathbf{z}, \mathbf{w}) \mid ((\mathbf{x}, \mathbf{y}), (\mathbf{z}, \mathbf{w})) \in (\mathbf{s} \times \mathbf{t}) \times (\mathbf{u} \times \mathbf{v}) \wedge & \\
 & \qquad (\mathbf{x}, \mathbf{z}) \in \mathbf{h} \wedge (\mathbf{y}, \mathbf{w}) \in \mathbf{k} \,\} & \text{where } \mathbf{h} \in \mathbf{s} \leftrightarrow \mathbf{u} \text{ and } \mathbf{k} \in \mathbf{t} \leftrightarrow \mathbf{v} \\
\mathbf{s} \nrightarrow \mathbf{t} & :\Rightarrow \{\, \mathbf{r} \mid \mathbf{r} \in \mathbf{s} \times \mathbf{t} \wedge (\mathbf{r}^{-1};\ \mathbf{r}) \subseteq \mathrm{id}(\mathbf{t}) \,\} & \\
\mathbf{s} \rightarrow \mathbf{t} & :\Rightarrow \{\, \mathbf{f} \mid \mathbf{f} \in \mathbf{s} \nrightarrow \mathbf{t} \wedge \mathrm{dom}(\mathbf{f}) = \mathbf{s} \,\} & \\
\mathbf{s} \rightarrowtail \mathbf{t} & :\Rightarrow \{\, \mathbf{f} \mid \mathbf{f} \in \mathbf{s} \nrightarrow \mathbf{t} \wedge \mathbf{f}^{-1} \in \mathbf{t} \nrightarrow s \,\} & \\
\mathbf{s} \rightarrowtail \mathbf{t} & :\Rightarrow \mathbf{s} \rightarrowtail \mathbf{t} \cap \mathbf{s} \rightarrow \mathbf{t} & \\
\mathbf{s} \twoheadrightarrow \mathbf{t} & :\Rightarrow \{\, \mathbf{f} \mid \mathbf{f} \in \mathbf{s} \nrightarrow \mathbf{t} \wedge \mathrm{ran}(\mathbf{f}) = \mathbf{t} \,\} & \\
\mathbf{s} \twoheadrightarrow \mathbf{t} & :\Rightarrow \mathbf{s} \twoheadrightarrow \mathbf{t} \cap \mathbf{s} \rightarrow \mathbf{t} & \\
\mathbf{s} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbf{t} & :\Rightarrow \mathbf{s} \rightarrowtail \mathbf{t} \cap \mathbf{s} \twoheadrightarrow \mathbf{t} & \\
\mathbf{s} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbf{t} & :\Rightarrow \mathbf{s} \rightarrowtail \mathbf{t} \cap \mathbf{s} \twoheadrightarrow \mathbf{t} & \\
\lambda\, \mathbf{x} \cdot (\mathbf{p} \mid \mathbf{e}) & :\Rightarrow \{\, \mathbf{x}, \mathbf{y} \mid \mathbf{y} = \mathbf{e} \wedge \mathbf{p} \,\} & \\
\mathbf{f}(\mathbf{e}) & :\Rightarrow \mathsf{choice}(\mathbf{f}[\{\mathbf{e}\}]) &
\end{array}
$$

Table 5.12: Rewrite rules for relation and function constructs.

$$\text{union}(\{\, x \mid x \in \mathbb{P}(t) \wedge f(x) \subseteq x \,\}) \tag{5.2}$$
$$\text{inter}(\{\, x \mid x \in \mathbb{P}(t) \wedge x \subseteq f(x) \,\}) \tag{5.3}$$
$$\text{union}(\{\, x \mid x \in \mathbb{P}(t) \wedge x \subseteq f(x) \,\}) \tag{5.4}$$

Clearly, the set 5.2 is t and the set 5.3 is $\varnothing$. They are thus of no interest. Theorem 5.1 will show that the other two are indeed fixpoints (but not equal!) under a certain assumption. To anticipate this, and to allow a more concise statement of the Theorem, we will introduce two new derived constructs.

Expression = 'fix', '(', *Expression*, ')'
| 'FIX', '(', *Expression*, ')'

$$\begin{aligned} \text{fix}(\mathbf{f}) &:\Rightarrow \text{inter}(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbb{P}(\mathbf{s}) \wedge \mathbf{f}(\mathbf{x}) \subseteq \mathbf{x} \,\}) && \text{if } \mathbf{f} \in \mathbb{P}(\mathbf{s}) \to \mathbb{P}(\mathbf{s}) \\ \text{FIX}(\mathbf{f}) &:\Rightarrow \text{union}(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbb{P}(\mathbf{s}) \wedge \mathbf{x} \subseteq \mathbf{f}(\mathbf{x}) \,\}) && \text{if } \mathbf{f} \in \mathbb{P}(\mathbf{s}) \to \mathbb{P}(\mathbf{s}) \end{aligned}$$

Theorem 5.1 (Knaster and Tarski) *Let* $\mathbf{f}$ *and* $\mathbf{s}$ *be sets such that* $\mathbf{f} \in \mathbb{P}(\mathbf{s}) \to \mathbb{P}(\mathbf{s})$ *and* $\forall(\mathbf{x}, \mathbf{y}) \cdot ((\mathbf{x}, \mathbf{y}) \in \mathbb{P}(\mathbf{s}) \times \mathbb{P}(\mathbf{s}) \wedge \mathbf{x} \subseteq \mathbf{y} \Rightarrow \mathbf{f}(\mathbf{x}) \subseteq \mathbf{f}(\mathbf{y}))$ *(in other words,* $\mathbf{f}$ *preserves set containment), then* $\text{fix}(\mathbf{f})$ *and* $\text{FIX}(\mathbf{f})$ *are fixpoints of* $\mathbf{f}$*, that is,* $\mathbf{f}(\text{fix}(\mathbf{f})) = \text{fix}(\mathbf{f})$ *and* $\mathbf{f}(\text{FIX}(\mathbf{f})) = \text{FIX}(\mathbf{f})$.

A proof for this theorem can be found in [1] under the theorems 3.2.5 and 3.2.6.

5.4.2 Finiteness and infiniteness A nonempty finite set can be characterized as a set which can be built from another finite set by adding one new element. If we denote the set of finite subsets of a set s by $\mathbb{F}(s)$ (we will define this notation soon), we can write a characterization of that set as the following two formulae:

$$\varnothing \in \mathbb{F}(s) \tag{5.5}$$
$$\forall(u, x) \cdot (u \in s \wedge x \in \mathbb{F}(s) \Rightarrow \{u\} \cup x \in \mathbb{F}(s)) \tag{5.6}$$

Let's go on a limb and declare a definition for $\mathbb{F}$[6]:

Expression = '$\mathbb{F}$', '(', *Expression*, ')'
| '$\mathbb{F}_1$', '(', *Expression*, ')'

$$\begin{aligned} \mathbb{F}(\mathbf{s}) :\Rightarrow \text{fix}(\lambda\, \mathbf{z} \cdot (\mathbf{z} \subseteq \mathbb{P}(\mathbf{s}) \mid \{\varnothing\} \cup \mathbf{z} \cup \\ \{\, \mathbf{x} \mid \mathbf{x} \subseteq \mathbb{P}(\mathbf{s}) \wedge \exists\, \mathbf{y}, \mathbf{u} \cdot (\mathbf{y} \in \mathbf{z} \wedge \mathbf{u} \in \mathbf{s} \wedge \mathbf{y} \cup \{\mathbf{u}\} \in \mathbf{x}) \,\})) \\ \mathbb{F}_1(\mathbf{s}) :\Rightarrow \mathbb{F}(\mathbf{s}) - \{\varnothing\} \end{aligned}$$

It is clear after a period of consideration that this definition satisfies the conditions (5.5) and (5.6) and that $\mathbb{F}$ is indeed a fixpoint (the function fix is applied to is monotonic). Thus, $\mathbb{F}$ is well-defined.

It is possible to use induction to prove properties of all finite subsets of a set, as given in the next theorem.

Theorem 5.2 (Induction principle for $\mathbb{F}$) *Let* $\mathbf{s}$ *be a set. Let* $\mathbf{p}$ *be a predicate and let* $\mathbf{x}$ *be a variable such that* $\mathbf{x} \in \mathbb{F}(\mathbf{s}) \vdash \mathsf{check}(\mathbf{p})$. *Also, let* $\mathbf{u}$ *be a variable distinct from* $\mathbf{x}$. *Now, if* $[\mathbf{x} := \varnothing]\mathbf{p}$ *and* $\forall\, \mathbf{x} \cdot (\mathbf{x} \in \mathbb{F}(\mathbf{s}) \wedge \mathbf{p} \Rightarrow \forall\, \mathbf{u} \cdot (\mathbf{u} \in \mathbf{s} \Rightarrow [\mathbf{x} := \{\mathbf{u}\} \cup \mathbf{x}]\mathbf{p}))$ *are formal theorems, then* $\forall\, \mathbf{x} \cdot (\mathbf{x} \in \mathbb{F}(\mathbf{s}) \Rightarrow \mathbf{p})$ *is a formal theorem.*

A proof for this theorem can be found in [1] under the theorem 3.3.1.

Now, we can define the finiteness and infiniteness of a set using the $\mathbb{F}$ construct:

Expression = 'finite', '(', *Expression*, ')'
| 'infinite', '(', *Expression*, ')'

$$\begin{aligned} \mathrm{finite}(\mathbf{s}) &:\Rightarrow \mathbf{s} \in \mathbb{F}(\mathbf{s}) \\ \mathrm{infinite}(\mathbf{s}) &:\Rightarrow \neg\, \mathrm{finite}(\mathbf{s}) \end{aligned}$$

5.4.3 Numbers We will be defining the set of natural numbers:

Expression = '$\mathbb{N}$'
| '0'
| 'succ'

The set of natural numbers is characterized by induction: zero is in the set and every element of that set has a successor in the set. Formally we may express this as a fixpoint construct as follows:

$$\begin{aligned} 0 &:\Rightarrow \mathsf{BIG} - \mathsf{BIG} \\ \mathrm{succ} &:\Rightarrow \lambda\, n \cdot (n \in \mathbb{F}(\mathsf{BIG}) \mid \{\mathsf{choice}(\mathsf{BIG} - n)\} \cup n) \\ \mathbb{N} &:\Rightarrow \mathrm{fix}(\lambda\, s \cdot (s \subseteq \mathbb{F}(\mathsf{BIG}) \mid \{0\} \cup \mathrm{succ}[s])) \end{aligned}$$

Table 5.13 lists the abstract syntax productions for several conventional constructs involving natural numbers. Table 5.14 gives the corresponding rewrite rules.

Abrial proved in his book [1] that this definition of natural numbers satisfies the Peano axioms.

We will now state without proof the two well-known induction theorems for natural numbers.

Expression = '1'
| '$\mathbb{N}_1$'
| 'pred'
| 'gtr'
| 'geq'
| 'lss'
| 'leq'
| 'min'
| 'genf'
| *Expression*, '..', *Expression*

Predicate = *Expression*, '$\leq$', *Expression*
| *Expression*, '$<$', *Expression*
| *Expression*, '$\geq$', *Expression*
| *Expression*, '$>$', *Expression*

Table 5.13: Abstract syntax productions for natural number constructs.

Theorem 5.3 (Induction principle in $\mathbb{N}$) *Let* $\mathbf{p}$ *be a predicate and let* $\mathbf{n}$ *be a variable such that* $\mathbf{n} \in \mathbb{N} \vdash \mathsf{check}(\mathbf{p})$. *Now, if* $[\mathbf{n} := 0]\mathbf{p}$ *and* $\forall\, \mathbf{n} \cdot (\mathbf{n} \in \mathbb{N} \land \mathbf{p} \Rightarrow [\mathbf{n} := \mathrm{succ}(n)]\mathbf{p})$ *are formal theorems, then* $\forall\, \mathbf{n} \cdot (\mathbf{n} \in \mathbb{N} \Rightarrow \mathbf{p})$ *is a formal theorem.*

Theorem 5.4 (Strong induction principle in $\mathbb{N}$) *Let* $\mathbf{p}$ *be a predicate and let* $\mathbf{n}$ *be a variable such that* $\mathbf{n} \in \mathbb{N} \vdash \mathsf{check}(\mathbf{p})$. *Also, let* $\mathbf{m}$ *be a variable distinct from* $\mathbf{n}$. *Now, if* $\forall\, \mathbf{n} \cdot (\mathbf{n} \in \mathbb{N} \land \forall\, \mathbf{m} \cdot (\mathbf{m} \in \mathbb{N} \land \mathbf{m} < \mathbf{n} \Rightarrow [\mathbf{n} := \mathbf{m}]\mathbf{p}) \Rightarrow \mathbf{p})$ *is a formal theorem, then* $\forall\, \mathbf{n} \cdot (\mathbf{n} \in \mathbb{N} \Rightarrow \mathbf{p})$ *is a formal theorem.*

Many number-theoretic functions (including addition and multiplication) are defined recursively. To this end, we defined in Tables 5.13 and 5.14 the functional construct genf. The idea is that a function $f \in \mathbf{s} \leftrightarrow \mathbb{N}$ which is defined by the recursive equations

$$\begin{cases} f(0) = a \\ \forall\, n \cdot (n \in \mathbb{N} \Rightarrow f(\mathrm{succ}(n)) = g(f(n))) \end{cases}$$

where $a \in s$ and $g \in s \to s$, is $\mathrm{fix}(\mathrm{genf}(a, g))$. However, $\mathrm{fix}(\mathrm{genf}(a, g))$ is not a function for all possible a and g, so when this is used for real, its functionness should be separately proved.

$$
\begin{array}{rll}
1 & :\Rightarrow \mathrm{succ}(0) & \\
\mathbb{N}_1 & :\Rightarrow \mathbb{N} - \{0\} & \\
\mathrm{pred} & :\Rightarrow \mathrm{succ}^{-1} \rhd \mathbb{N} & \\
\mathbf{n} \leq \mathbf{m} & :\Rightarrow \mathbf{n} \subseteq \mathbf{m} & \text{if } \mathbf{n} \in \mathbb{N} \text{ and } \mathbf{m} \in \mathbb{N} \\
\mathbf{n} < \mathbf{m} & :\Rightarrow \mathbf{n} \neq \mathbf{m} \wedge \mathbf{n} \leq \mathbf{m} & \text{if } \mathbf{n} \in \mathbb{N} \text{ and } \mathbf{m} \in \mathbb{N} \\
\mathbf{n} \geq \mathbf{m} & :\Rightarrow \mathbf{m} \leq \mathbf{n} & \text{if } \mathbf{n} \in \mathbb{N} \text{ and } \mathbf{m} \in \mathbb{N} \\
\mathbf{n} > \mathbf{m} & :\Rightarrow \mathbf{m} < \mathbf{n} & \text{if } \mathbf{n} \in \mathbb{N} \text{ and } \mathbf{m} \in \mathbb{N} \\
\mathrm{gtr} & :\Rightarrow \{ m, n \mid m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge m < n \} & \\
\mathrm{geq} & :\Rightarrow \{ m, n \mid m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge m \leq n \} & \\
\mathrm{lss} & :\Rightarrow \{ m, n \mid m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge m > n \} & \\
\mathrm{leq} & :\Rightarrow \{ m, n \mid m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge m \geq n \} & \\
\mathrm{min} & :\Rightarrow \lambda s \cdot (s \in \mathbb{P}_1(\mathbb{N}) \mid \mathrm{inter}(s)) & \\
\mathrm{max} & :\Rightarrow \lambda s \cdot (s \in \mathbb{F}_1(\mathbb{N}) \mid \mathrm{union}(s)) & \\
\mathrm{genf}(\mathbf{a}, \mathbf{g}) & :\Rightarrow \lambda \mathbf{h} \cdot (\mathbf{h} \subseteq \mathbb{N} \times \mathbf{s} \mid \{0 \mapsto \mathbf{a}\} \cup (\mathrm{pred};\ \mathbf{h};\ \mathbf{g})) & \\
\mathbf{m}..\mathbf{n} & :\Rightarrow \{ \mathbf{x} \mid \mathbf{x} \in \mathbb{N} \wedge \mathbf{m} \leq \mathbf{x} \wedge \mathbf{x} \leq \mathbf{n} \} & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N}
\end{array}
$$

Table 5.14: Rewrite rules for natural number constructs.

$$
\begin{aligned}
\mathit{Expression} = {} & \text{'plus'} \\
& \mid \text{'mult'} \\
& \mid \text{'exp'} \\
& \mid \mathit{Expression}, \text{'}+\text{'}, \mathit{Expression} \\
& \mid \mathit{Expression}, \text{'}-\text{'}, \mathit{Expression} \\
& \mid \mathit{Expression}, \text{'}\times\text{'}, \mathit{Expression} \\
& \mid \mathit{Expression}, \text{'}/\text{'}, \mathit{Expression} \\
& \mid \mathit{Expression}, \text{'mod'}, \mathit{Expression} \\
& \mid \mathit{Expression}, {}^{\mathit{Expression}} \\
& \mid \text{'log'}, {}_{\mathit{Expression}}, \text{'('}, \mathit{Expression}, \text{')'} \\
& \mid \text{'LOG'}, {}_{\mathit{Expression}}, \text{'('}, \mathit{Expression}, \text{')'}
\end{aligned}
$$

Table 5.15: Abstract syntax productions for arithmetical constructs.

The definitions of arithmetic are given in Tables 5.15 (abstract syntax productions) and 5.16 (rewrite rules). Note that the abstract syntax productions for an exponentation expression and a logarithm are not valid EBNF; it is, however, hoped that the slight abuse of notation is understandable enough. Also note that the symbol $\times$ is used for both cartesian products and natural number products; Abrial never notes this problem and thus does not give a solution. It would be tempting to use the type information to determine which meaning is to be used, but this would make it impossible to use cartesian products on those sets that happen to be also natural numbers. We will, for now, choose the path set up for us by Abrial: we will ignore it and let the context decide.

Abrial also defines iteration of a relation, cardinal number of a finite set, transitive closures of a relation, sequences and trees. He also extends informally the notation defined here for the set of integers ($\mathbb{Z}$). We will skip these for reasons of space.

5.5 Generalized substitution language

The core of specification of operations in B is a generalization of syntactic substitutions, modelled after Edsger W. Dijkstra's weakest precondition calculus [34] and its later generalizations by for example Nelson [86]. The presentation of generalized

$$
\begin{array}{rll}
\text{plus} & :\Rightarrow \lambda\, m \cdot (m \in \mathbb{N} \mid \text{fix}(\text{genf}(m, \text{succ}))) & \\
\text{mult} & :\Rightarrow \lambda\, m \cdot (m \in \mathbb{N} \mid \text{fix}(\text{genf}(0, \text{plus}(m)))) & \\
\text{exp} & :\Rightarrow \lambda\, m \cdot (m \in \mathbb{N} \mid \text{fix}(\text{genf}(1, \text{mult}(m)))) & \\
\mathbf{m} + \mathbf{n} & :\Rightarrow \text{plus}(\mathbf{m})(\mathbf{n}) & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N} \\
\mathbf{m} - \mathbf{n} & :\Rightarrow \text{plus}(\mathbf{m})^{-1}(\mathbf{n}) & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N} \text{ and } \mathbf{m} < \mathbf{n} \\
\mathbf{m} \times \mathbf{n} & :\Rightarrow \text{mult}(\mathbf{m})(\mathbf{n}) & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N} \\
\mathbf{n}/\mathbf{m} & :\Rightarrow \min(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbb{N} \wedge \mathbf{n} < \mathbf{m} \times \text{succ}(\mathbf{x}) \,\}) & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N}_1 \\
\mathbf{n} \bmod \mathbf{m} & :\Rightarrow \mathbf{n} - (\mathbf{m} \times (\mathbf{n}/\mathbf{m})) & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N}_1 \\
\mathbf{m}^{\mathbf{n}} & :\Rightarrow \text{exp}(\mathbf{m})(\mathbf{n}) & \text{if } \mathbf{m} \in \mathbb{N} \text{ and } \mathbf{n} \in \mathbb{N} \\
\log_{\mathbf{m}}(\mathbf{n}) & :\Rightarrow \min(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbb{N} \wedge \mathbf{n} < \mathbf{m}^{\text{succ}(\mathbf{x})} \,\}) & \text{if } \mathbf{m} \in \mathbb{N} - \{0, 1\} \text{ and } \mathbf{n} \in \mathbb{N} \\
\text{LOG}_{\mathbf{m}}(\mathbf{n}) & :\Rightarrow \min(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbb{N} \wedge \mathbf{n} < \mathbf{m}^{\mathbf{x}} \,\}) & \text{if } \mathbf{m} \in \mathbb{N} - \{0, 1\} \text{ and } \mathbf{n} \in \mathbb{N}
\end{array}
$$

Table 5.16: Rewrite rules for arithmetical constructs.

substitutions given by Abrial is complicated. A simplified and more powerful theory of generalized substitutions is given by Steve Dunne [35], and we will mainly follow his presentation.

5.5.1 Basic constructs The specificational intuition for the syntactic substitution in a predicate $[\mathbf{x} := \mathbf{e}]\mathbf{p}$ is the following:

> The predicate $[\mathbf{x} := \mathbf{e}]\mathbf{p}$ is the weakest precondition under which the operation $\mathbf{x} := \mathbf{e}$ establishes the postcondition $\mathbf{p}$.

Edsger W. Dijkstra [34] based his definition of an abstract programming language on this intuition. His basic idea was to *define* the semantics of a programming construct by giving its weakest precondition as a function of the postcondition. Thus, for example, a *guarded command* $\mathbf{p} \mid \mathbf{s}$, where $\mathbf{s}$ is a command, can be defined by stating that its weakest precondition given the postcondition $\mathbf{q}$ is the logical conjunction of $\mathbf{p}$ and the weakest precondition of $\mathbf{s}$ given the postcondition $\mathbf{q}$.

In this section, the metavariables $\mathbf{s}$ and $\mathbf{t}$ will exceptionally stand for substitutions.

Substitution = *Predicate*, '|', *Substitution*
| *Predicate*, '$\Longrightarrow$', *Substitution*
| *Substitution*, '[]', *Subsitution*
| '@', *Variable*, '$\cdot$', *Substitution*
| *Substitution*, '; ', *Substitution*
| 'skip'

Variable = 'frame', '(', *Substitution*, ')'
| 'normalizeframe', '(', *Variable*, ')'
| 'subtractframe', '(', *Variable*, ',', *Variable* ')'
| 'mergeframes', '(', *Variable*, ',', *Variable* ')'
| 'emptyframe'

Table 5.17: Abstract syntax productions for basic generalized substitutions.

This *wp-calculus* ("wp" standing for "weakest precondition") was later amended by Carroll Morgan [83] and others to allow underspecification of programs. It is this specificational wp-calculus that Abrial used as the basis for his generalized substitutions.

We will extend the abstract syntax of the B notation as given in Table 5.17. If we assume that the production *Substitution* is never parsed in isolation, but instead its strings will always occur in the substituted predicate construct $[\mathbf{s}]\mathbf{p}$, then we may consider these derived constructs and give them rewrite rules in Table 5.18. This is contrast with Abrial's definition; he considers these constructs primitive and gives them typechecking rules and axioms.

You'll notice that we have also added several new derived variable constructs and one new primitive variable construct, emptyframe. These comprise a formal definition of the *frame* of a substitution, originally introduced to the B notation by Steve Dunne [35] and formalized by the present author. Intuitively, the frame of a substitution is a set of those identifiers that the substitution operates on. It is defined here using a set of rewrite rules and represented as a sorted list variable containing no duplicates. Frames allow us to define a metalevel equivalence of substitutions: if two substitutions have the same frame and equivalent weakest preconditions given equivalent postconditions, then the substitutions are equivalent.

Non-freeness of a variable in a substitution needs an additional rule:

$$
\begin{aligned}
[\mathbf{p} \mid \mathbf{s}]\mathbf{q} &:\Rightarrow \mathbf{p} \wedge [\mathbf{s}]\mathbf{q} \\
[\mathbf{p} \Longrightarrow \mathbf{s}]\mathbf{q} &:\Rightarrow \mathbf{p} \Rightarrow [\mathbf{s}]\mathbf{q} \\
[\mathbf{s} \,[]\, \mathbf{t}]\mathbf{q} &:\Rightarrow [\mathbf{s}]\mathbf{q} \wedge [\mathbf{t}]\mathbf{q} \\
[@\mathbf{x} \cdot \mathbf{s}]\mathbf{q} &:\Rightarrow \forall\, \mathbf{x} \cdot [\mathbf{s}]\mathbf{q} \quad \text{where } \mathbf{x} \setminus \mathbf{q} \\
[\mathbf{s};\ \mathbf{t}]\mathbf{q} &:\Rightarrow [\mathbf{s}][\mathbf{t}]\mathbf{q} \\
[\mathrm{skip}]\mathbf{q} &:\Rightarrow \mathbf{q} \\
\mathrm{frame}(\mathbf{x} := \mathbf{e}) &:\Rightarrow \mathrm{normalizeframe}(\mathbf{x}) \\
\mathrm{frame}(\mathbf{p} \mid \mathbf{s}) &:\Rightarrow \mathrm{frame}(\mathbf{s}) \\
\mathrm{frame}(\mathbf{p} \Longrightarrow \mathbf{s}) &:\Rightarrow \mathrm{frame}(\mathbf{s}) \\
\mathrm{frame}(\mathbf{s} \,[]\, \mathbf{t}) &:\Rightarrow \mathrm{mergeframes}(\mathrm{frame}(\mathbf{s}), \mathrm{frame}(\mathbf{t})) \\
\mathrm{frame}(@\mathbf{x} \cdot \mathbf{s}) &:\Rightarrow \mathrm{subtractframe}(\mathrm{frame}(\mathbf{s}, \mathrm{normalizeframe}(\mathbf{x}))) \\
\mathrm{frame}(\mathbf{s};\ \mathbf{t}) &:\Rightarrow \mathrm{mergeframes}(\mathrm{frame}(\mathbf{s}), \mathrm{frame}(\mathbf{t})) \\
\mathrm{frame}(\mathrm{skip}) &:\Rightarrow \mathrm{emptyframe} \\
\mathrm{emptyframe}, \mathbf{x} &:\Rightarrow \mathbf{x} \\
\mathbf{x}, \mathrm{emptyframe} &:\Rightarrow \mathbf{x} \\
\forall\, \mathrm{emptyframe} \cdot \mathbf{p} &:\Rightarrow \mathbf{p} \\
[\mathrm{emptyframe} := \mathbf{e}]\mathbf{p} &:\Rightarrow \mathbf{p} \\
\mathrm{normalizeframe}(\mathrm{emptyframe}) &:\Rightarrow \mathrm{emptyframe} \\
\mathrm{normalizeframe}(\mathbf{i}) &:\Rightarrow \mathbf{i} \\
\mathrm{normalizeframe}(\mathbf{x}, \mathbf{y}) &:\Rightarrow \mathrm{mergeframes}(\mathrm{normalizeframe}(\mathbf{x}), \mathrm{normalizeframe}(\mathbf{y}))
\end{aligned}
$$

Table 5.18: Rewrite rules for basic generalized substitutions *(cont.)*

$$
\begin{array}{rll}
\text{mergeframes}(\mathbf{i},\mathbf{i}) & :\Rightarrow \mathbf{i} & \\
\text{mergeframes}(\mathbf{i},\mathbf{j}) & :\Rightarrow \mathbf{i},\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{mergeframes}(\mathbf{i},\mathbf{j}) & :\Rightarrow \mathbf{j},\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{mergeframes}((\mathbf{x},\mathbf{i}),\mathbf{i}) & :\Rightarrow \mathbf{x},\mathbf{i} & \\
\text{mergeframes}((\mathbf{x},\mathbf{i}),\mathbf{j}) & :\Rightarrow \mathbf{x},\mathbf{i},\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{mergeframes}((\mathbf{x},\mathbf{i}),\mathbf{j}) & :\Rightarrow \text{mergeframes}(\mathbf{x},\mathbf{j}),\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{mergeframes}(\mathbf{i},(\mathbf{y},\mathbf{i})) & :\Rightarrow \mathbf{y},\mathbf{i} & \\
\text{mergeframes}(\mathbf{i},(\mathbf{y},\mathbf{j})) & :\Rightarrow \text{mergeframes}(\mathbf{i},\mathbf{y}),\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{mergeframes}(\mathbf{i},(\mathbf{y},\mathbf{j})) & :\Rightarrow \mathbf{y},\mathbf{j},\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{mergeframes}((\mathbf{x},\mathbf{i}),(\mathbf{y},\mathbf{i})) & :\Rightarrow \text{mergeframes}(\mathbf{x},\mathbf{y}),\mathbf{i} & \\
\text{mergeframes}((\mathbf{x},\mathbf{i}),(\mathbf{y},\mathbf{j})) & :\Rightarrow \text{mergeframes}((\mathbf{x},\mathbf{i}),\mathbf{y}),\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{mergeframes}((\mathbf{x},\mathbf{i}),(\mathbf{y},\mathbf{j})) & :\Rightarrow \text{mergeframes}(\mathbf{x},(\mathbf{y},\mathbf{j})),\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{subtractframe}(\mathbf{i},\mathbf{i}) & :\Rightarrow \text{emptyframe} & \\
\text{subtractframe}(\mathbf{i},\mathbf{j}) & :\Rightarrow \mathbf{i},\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{subtractframe}(\mathbf{i},\mathbf{j}) & :\Rightarrow \mathbf{j},\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{subtractframe}(\mathbf{i},(\mathbf{y},\mathbf{i})) & :\Rightarrow \mathbf{y} & \\
\text{subtractframe}(\mathbf{i},(\mathbf{y},\mathbf{j})) & :\Rightarrow \text{subtractframe}(\mathbf{i},\mathbf{y}),\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{subtractframe}(\mathbf{i},(\mathbf{y},\mathbf{j})) & :\Rightarrow \mathbf{y},\mathbf{j},\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{subtractframe}((\mathbf{x},\mathbf{i}),\mathbf{i}) & :\Rightarrow \mathbf{x} & \\
\text{subtractframe}((\mathbf{x},\mathbf{i}),\mathbf{j}) & :\Rightarrow \mathbf{x},\mathbf{i},\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{subtractframe}((\mathbf{x},\mathbf{i}),\mathbf{j}) & :\Rightarrow \text{subtractframe}(\mathbf{x},\mathbf{j}),\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i} \\
\text{subtractframe}((\mathbf{x},\mathbf{i}),(\mathbf{y},\mathbf{j})) & :\Rightarrow \text{subtractframe}((\mathbf{x},\mathbf{i}),\mathbf{y}),\mathbf{j} & \text{if } \mathbf{i} \prec \mathbf{j} \\
\text{subtractframe}((\mathbf{x},\mathbf{i}),(\mathbf{y},\mathbf{j})) & :\Rightarrow \text{subtractframe}(\mathbf{x},(\mathbf{y},\mathbf{j})),\mathbf{i} & \text{if } \mathbf{j} \prec \mathbf{i}
\end{array}
$$

Here, the metavariables $\mathbf{i}$ and $\mathbf{j}$ stand for distinct identifiers, and the operator $\prec$ stands for lexical ordering of identifiers.

Table 5.18: *(cont.)* Rewrite rules for basic generalized substitutions.

Non-freeness	Condition
$\mathbf{x} \setminus \mathbf{s}$	$\mathbf{x} \setminus [\mathbf{s}](\mathrm{frame}(\mathbf{s}) = \mathrm{frame}(\mathbf{s}))$ and $\mathbf{x} \setminus \mathrm{frame}(\mathbf{s})$

The substitution $\mathbf{p} \mid \mathbf{s}$ is a *preconditioned substitution*. Its operational intuition is "if **p** is true, do **s**, otherwise abort". In practice, users of this substitution will prove that **p** is true before invoking the substitution. As a special case, the substitution $\neg\, \mathsf{BIG} = \mathsf{BIG} \mid \mathrm{skip}$ will always abort.

The substitution $\mathbf{p} \Longrightarrow \mathbf{s}$ is a *guarded substitution*. Its operational intuition is "if **p** is true, do **s**, otherwise do whatever you like, even miracles". When **p** is false, the weakest precondition of this substitution is true regardless of the postcondition; thus, when **p** is false, the substitution promises to find a state which satisfies the impossible postcondition — hence it is potentially miraculous. The problem with miracles is that we don't know how to implement them, and thus any miraculous substitution is unimplementable, also called *infeasible*.

The substitution $\mathbf{s} \,[]\, \mathbf{t}$ is a *bounded choice substitution*, also known as *demonic choice*. Its operational intuition is "do either **s** or **t**; choose freely but don't choose so that the whole substitution becomes infeasible". Here "the whole substitution" refers to the maximal substitution that the bounded choice substitution being considered is part of. We can view the bounded choice as being controlled by a demon who makes the choice for us. The demon is constrained by feasibility, and in essence, he will, before making the choice, look into the future and see if any of the choices lead to infeasibility. If one or more do, then those choices are not considered by the demon. An example is the substitution $(x := 1 \,[]\, x := 2);\ (x := 1 \Longrightarrow \mathrm{skip})$, which is equivalent to the substitution $x := 1$. For this reason, Dunne [35] describes the demon's behaviour as "angelic with respect to feasibility".

Essentially, the bounded choice substitution allows specificational nondeterminism. Later, when the specification is refined into an implementation, the person making the refinement takes the role of the demon, selecting some feasible alternative among the bounded choices.

Another nondeterminism construct is *unbounded choice*, $@\mathbf{x} \cdot \mathbf{s}$. Its operational intuition is "choose a value for **x** freely but don't choose so that the whole substitution becomes infeasible". Here again the demon shows himself, and here, too, he is constrained by feasibility. For this reason, and to satisfy typechecking, **s** invariably takes the form of a guarded substitution.

The substitution $\mathbf{s};\ \mathbf{t}$ is *sequential composition* of substitutions. Its operational intuition is obvious: "do **s** and then do **t**".

Predicate = 'trm', '(', *Substitution*, ')'
| 'abt', '(', *Substitution*, ')'
| 'mir', '(', *Substitution*, ')'
| 'fis', '(', *Substitution*, ')'
| 'prd', '(', *Substitution*, ')'
Substitution = *Substitution*, '||', *Substitution*

Table 5.19: Abstract syntax productions for additional substitution constructs.

$$
\begin{aligned}
\mathrm{trm}(\mathbf{s}) &:\Rightarrow [\mathbf{s}](\mathsf{BIG} = \mathsf{BIG}) \\
\mathrm{abt}(\mathbf{s}) &:\Rightarrow \neg\,\mathrm{trm}(\mathbf{s}) \\
\mathrm{mir}(\mathbf{s}) &:\Rightarrow [\mathbf{s}](\mathsf{BIG} \neq \mathsf{BIG}) \\
\mathrm{fis}(\mathbf{s}) &:\Rightarrow \neg\,\mathrm{mir}(\mathbf{s}) \\
\mathrm{prd}(\mathbf{s}) &:\Rightarrow \neg[\mathbf{s}](\mathbf{x} \neq \mathbf{x}') && \text{if } \mathbf{x}' \setminus \mathbf{s} \\
\mathbf{s} \parallel \mathbf{t} &:\Rightarrow \mathrm{trm}(\mathbf{s}) \wedge \mathrm{trm}(\mathbf{t}) \mid @\mathbf{x}'', \mathbf{y}'' \cdot \\
&\qquad \neg[\mathbf{s}](\mathbf{x} \neq \mathbf{x}'') \wedge \neg[\mathbf{t}](\mathbf{y} \neq \mathbf{y}'') \Longrightarrow \mathbf{x}, \mathbf{y} := \mathbf{x}'', \mathbf{y}''
\end{aligned}
$$

Here, the metavariable $\mathbf{x}$ stands for frame($\mathbf{s}$), the metavariable $\mathbf{x}'$ stands for $\mathbf{x}$ where a prime has been appended to each identifier and the metavariable $\mathbf{x}''$ stands for a variable that has the same structure as $\mathbf{x}$ and is non-free in $\mathbf{s}$. Similarly, the metavariable $\mathbf{y}$ stands for frame($\mathbf{t}$), the metavariable $\mathbf{y}'$ stands for $\mathbf{y}$ where a prime has been appended to each identifier and the metavariable $\mathbf{y}''$ stands for a variable that has the same structure as $\mathbf{y}$ and is non-free in $\mathbf{t}$.

Table 5.20: Rewrite rules for additional substitution constructs.

5.5.2 Characteristic predicates and parallel composition Substitutions can be characterized using the predicates trm, abt, mir, fis and prd defined in Tables 5.19 and 5.20. The predicate trm($\mathbf{s}$) is true for all initial states for which $\mathbf{s}$ terminates successfully. The predicate abt($\mathbf{s}$) is true for all initial states for which $\mathbf{s}$ aborts. The predicate mir($\mathbf{s}$) is true for all initial states for which $\mathbf{s}$ is miraculous (infeasible). The predicate fis($\mathbf{s}$) is true for all initial states for which $\mathbf{s}$ is feasible. The predicate prd($\mathbf{s}$) is the *before-after predicate* of the substitution $\mathbf{s}$: it relates the initial values of the frame of $\mathbf{s}$ to the corresponding final values of the frame (denoted by a variable obtained from the frame by appending a prime to each identifier) under the substitution $\mathbf{s}$.

Using these new predicates, we can define a derived substitution construct: a liberalized *parallel composition* of substitutions, $\mathbf{s} \parallel \mathbf{t}$. The operational intuition of this substitution is that, if the frame of $\mathbf{s}$ is non-free in the frame of $\mathbf{t}$, then $\mathbf{s}$ and $\mathbf{t}$ are independently performed in parallel. If the frames contain common identifiers, then the substitution is feasible if and only if $\mathbf{s}$ and $\mathbf{t}$ yield the same value for the common identifiers. We call this *liberalized parallel substitution* after Dunne [35], since Abrial's (rather clumsy) definition allows parallel composition only when the frames contain no common identifiers.

5.5.3 Healthiness conditions and a normal form Dunne [35] gives three meta-level healthiness conditions for constructs in the syntactic category *Substitution*. The first is a version of Leibniz's law of equality:

$$\forall \mathbf{x} \cdot (\mathbf{p} \Leftrightarrow \mathbf{q}) \Rightarrow \forall \mathbf{x} \cdot ([\mathbf{s}]\mathbf{p} \Leftrightarrow [\mathbf{s}]\mathbf{q}) \tag{GS1}$$

for all predicates $\mathbf{p}$ and $\mathbf{q}$, and all substitutions $\mathbf{s}$, where $\vdash \mathsf{check}(\forall \mathbf{x} \cdot \mathbf{p})$ and $\vdash \mathsf{check}(\forall \mathbf{x} \cdot \mathbf{q})$. The second is the law of positive conjunctivity:

$$[\mathbf{s}](\mathbf{p} \wedge \mathbf{q}) \Leftrightarrow [\mathbf{s}]\mathbf{p} \wedge [\mathbf{s}]\mathbf{q} \tag{GS2}$$

for all predicates $\mathbf{p}$ and $\mathbf{q}$ and all substitutions $\mathbf{s}$. The third is the law of frame circumscription:

$$\forall \mathbf{x} \cdot ([\mathbf{s}]\mathbf{p} \Leftrightarrow \mathrm{trm}(\mathbf{s}) \wedge (\mathrm{fis}(\mathbf{s}) \Rightarrow \mathbf{p})) \tag{GS3}$$

for all substitutions $\mathbf{s}$ and all predicates $\mathbf{p}$ such that $\mathrm{frame}(\mathbf{s}) \setminus \mathbf{p}$ and $\vdash \mathsf{check}(\forall \mathbf{x} \cdot \mathbf{p})$. The fact that the currently defined constructs obey these laws is easy to prove; we will skip the proofs.

We will state the following two theorems without proof (which can be found in Dunne [35]):

Theorem 5.5 (Normal form of generalized substitutions) *Let* $\mathbf{s}$ *be a substitution and let* $\mathbf{x}$ *be a variable whose structure is the same as the structure of* $\mathrm{frame}(\mathbf{s})$*, and which has no free occurrences in* $\mathrm{frame}(\mathbf{s})$*. Then* $\mathbf{s}$ *is equivalent to*

$$\mathrm{trm}(\mathbf{s}) \mid @\mathbf{x} \cdot \mathrm{prd}(\mathbf{s}) \Longrightarrow \mathrm{frame}(\mathbf{s}) := \mathbf{x}.$$

Theorem 5.6 (Monotonicity of generalized substitutions) *Let* $\mathbf{s}$ *be a substitution and let* $\mathbf{p}$ *and* $\mathbf{q}$ *be predicates, where* $\vdash \mathsf{check}(\forall \mathbf{x} \cdot \mathbf{p})$ *and* $\vdash \mathsf{check}(\forall \mathbf{x} \cdot \mathbf{q})$*. Then*

$$\forall \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow \forall \mathbf{x} \cdot ([\mathbf{s}]\mathbf{p} \Rightarrow [\mathbf{s}]\mathbf{q})$$

is a formal theorem.

Substitution = *Substitution*, '^'

Table 5.21: Abstract syntax production for the opening of a substitution.

$$\mathbf{s}^{\wedge} :\Rightarrow (\mathbf{s};\ \mathbf{s}^{\wedge}) \,[]\, \text{skip}$$

Table 5.22: Rewrite rule for the opening of a substitution.

5.5.4 Iteration The fundamental iteration construction in GSL is the *opening* of a substitution $\mathbf{s}^{\wedge}$. Its operational intuition is "either do nothing or do **s** one or more times, at your choice". It is the third construct to use demonic nondeterminism, and like with the others, the demon is constrained by feasibility. For example, the substitution $(\mathbf{p} \implies \mathbf{s})^{\wedge};\ (\neg\mathbf{p} \implies \text{skip})$ is a traditional while-loop with **p** as the condition and **s** as the loop body.

The formal definition of opening is tricky. Back and von Wright [9] use in their higher-order logic framework a generalized form of the Knaster-Tarski theorem (Theorem 5.1) to define a fixpoint operator for (their equivalent of) substitutions, and then use that to define the opening. Abrial defines a "set-theoretic counterpart" for substitutions and then proves that it is unique modulo variable typing — essentially, to each substitution and each possible typing of its frame, there is a unique function that maps sets of after-values to sets of before-values. Then he proceeds by defining the set-theoretic counterpart of the opening as a fixpoint of a set transformer. Neither option is really satisfying, as the goal is to find a rewrite rule that would remove openings from the formulae.

The problem with Abrial's method is that it presupposes a context where the frame is given a type that is invariant under the substitution. There is no problem giving that type when a generalized substitution is used in the context of an abstract machine (see Section 5.6), but we'd like to develop GSL independently of AMN, to preserve the nice clean separation of the layers in B.

Tables 5.21 and 5.22 give a partial definition which does not satisfy typechecking but does give a meaning to the construct, if the rewrite rule is applied lazily (that is, only when needed). A rewrite rule based on fixpoints would be preferable, but no satisfying version seems to be available.

5.6 Abstract machine notation

Software systems are specified in B as abstract machines: things that have a state and one or more operations. The actual notation used is *abstract machine notation* (AMN). It allows the complete specification of abstract machines, their refinement and finally, their implementation.

5.6.1 Abstract machines The operational intuition for an abstract machine lies somewhere between a traditional module and an abstract class in the sense of object-orientation. Like abstract classes, abstract machines define the structure of the state of their instances, and the signatures of its operations. Unlike normal abstract classes, abstract machines specify the semantics of all their operations as substitutions. Like a module, an abstract machine is not a type.

The abstract syntax for abstract machines is given in Table 5.23.

The state of a machine is specified by specifying a frame (a set of named, distinct *variables*), which gives us a rough structure of the state as a cartesian product of things, and an *invariant*, which is a predicate whose free variables are all in the frame, giving a type to the frame and constraining further the values that it can get. Furthermore, an *initialization* substitution that initializes the state must be provided.

Operations modify the state under the constraints of the invariant. They can have zero or more parameters and return values. These are identifiers distinct from the state variables of the machine. The operations are specified as substitutions whose frames contain no variables apart from the state variables of the machine and the parameters and the return values.

Operations are not usually written in the plain generalized substitution language; rather certain derived constructs, more readable by programmers, are used. Tables 5.24 and 5.25 specify these constructs formally. However, sequencing and opening of substitutions (and any construct derived from these) may not be used in abstract machines.

Abstract machines can also name *given sets* (new types). They are nonempty finite sets, and their elements (represented by unique, specifier-chosen identifiers) can be enumerated. If the elements are not enumerated, then the set is deferred, and the cardinality of the set is indeterminate. The specifier can constrain the given sets by giving a predicate, the *properties* predicate, whose free variables are the given sets.

Concrete *constants* can also be specified in an abstract machine. Constants must

Machine = '**MACHINE**', *Machine header*, {*Machine clause*}, '**END**'
Machine header = *Identifier*
| *Identifier*, '(', *Identifier list*, ')'
Machine clause = '**CONSTRAINTS**', *Predicate*
| '**SETS**', *Set declaration*, {';', *Set declaration*}
| '**CONSTANTS**', *Identifier list*
| '**SEES**', *Identifier list*
| '**USES**', *Identifier list*
| '**ABSTRACT_CONSTANTS**', *Identifier list*
| '**PROPERTIES**', *Predicate*
| '**INCLUDES**', *Machine instantiation list*
| '**PROMOTES**', *Identifier list*
| '**EXTENDS**', *Machine instantiation list*
| '**VARIABLES**', *Identifier list*
| '**CONCRETE_VARIABLES**', *Identifier list*
| '**INVARIANT**', *Predicate*
| '**ASSERTIONS**', *Predicate*
| '**DEFINITIONS**', *Definition declaration*, {';', *Definition declaration*}
| '**INITIALIZATION**', *Substitution*
| '**OPERATIONS**', *Operation declaration*, {';', *Operation declaration*}
Set declaration = *Identifier*
| *Identifier*, '=', '{', *Identifier list*, '}'
Definition declaration = *Identifier*, '$\widehat{=}$', *Formal text*
| *Identifier*, '(', *Identifier list*, ')', '$\widehat{=}$', *Formal text*
Operation declaration = *Operation header*, '$\widehat{=}$', *Substitution*
Operation header = *Identifier list*, '$\longleftarrow$', *Identifier*, '(', *Identifier list*, ')'
| *Identifier*, '(', *Identifier list*, ')'
| *Identifier list*, '$\longleftarrow$', *Identifier*
| *Identifier*
Identifier list = *Identifier*, {',', *Identifier*}
Machine instantiation list = *Machine instantiation*, {',', *Machine instantiation*}
Machine instantiation = *Identifier*
| *Identifier*, '(', *Expression list*, ')'
Expression list = *Expression*, {',', *Expression*}

The nonterminal *Formal text* refers to anything that can be parsed according to the B grammar when taking the union of all nonterminals as the start symbol.

Table 5.23: Abstract syntax productions for abstract machines.

Substitution = '**BEGIN**', *Substitution*, '**END**'
| '**PRE**', *Predicate*, '**THEN**', *Substitution*, '**END**'
| '**IF**', *Predicate*, '**THEN**', *Substitution*,
{'**ELSIF**', *Predicate*, '**THEN**', *Substitution*},
['**ELSE**', *Substitution*],
'**END**'
| '**CHOICE**', *Substitution*, {'**OR**', *Substitution*}, '**END**'
| '**ANY**', *Variable*, '**WHERE**', *Predicate*, '**THEN**', *Substitution*, '**END**'
| '**SELECT**', *Predicate*, '**THEN**', *Substitution*,
{'**WHEN**', *Predicate*, '**THEN**', *Substitution*},
['**ELSE**', *Substitution*],
'**END**'
| '**CASE**', *Expression*, '**OF**',
'**EITHER**', *Identifier or literal list*, '**THEN**', *Substitution*,
{'**OR**', *Identifier or literal list*, '**THEN**', *Substitution*},
['**ELSE**', *Substitution*],
'**END**', '**END**'
| '**VAR**', *Variable*, '**IN**', *Substitution*, '**END**'
| '**LET**', *Variable*, '**BE**',
{*Variable*, '=', *Expression*, '∧'},
Variable, '=', *Expression*,
'**IN**', *Substitution*,
'**END**'
| *Variable*, ':=', 'bool', '(', *Predicate*, ')'
| *Variable*, ':∈', *Expression*
| *Variable*, '(', *Expression*, ')', ':=', *Expression*
| *Variable*, ':', *Predicate*
| *Identifier* (* invocation of an operation *)
| *Identifier*, '(', *Expression list*, ')' (* invocation of an operation *)
| *Variable*, '⟵', *Identifier* (* invocation of an operation *)
| *Variable*, '⟵', *Identifier*, '(', *Expression list*, ')' (* invocation of an operation *)
Identifier or literal list = *Identifier list*
| *Literal number*, {',', *Literal number*}

The members of an *Identifier or literal list* must be distinct. The double **END** in the **CASE** construct is not a typo.

Table 5.24: Abstract syntax productions for the AMN counterparts for GSL constructs.

$$
\begin{aligned}
\textbf{BEGIN s END} &:\Rightarrow \textbf{s} \\
\textbf{PRE p THEN s END} &:\Rightarrow \textbf{p} \mid \textbf{s} \\
\textbf{IF } \mathbf{p_0} \textbf{ THEN } \mathbf{s_0} \textbf{ ELSIF} \cdots & \\
\cdots \textbf{ ELSIF } \mathbf{p_{n-1}} \textbf{ THEN } \mathbf{s_{n-1}} \textbf{ ELSE } \mathbf{s_n} \textbf{ END} &:\Rightarrow (\mathbf{p_0} \Longrightarrow \mathbf{s_0}) \,[]\, (\mathbf{p_1} \Longrightarrow \mathbf{s_1}) \,[]\, \cdots \\
& \qquad \cdots \,[]\, (\neg(\mathbf{p_0} \wedge \cdots \wedge \mathbf{p_{n-1}}) \Longrightarrow \mathbf{s_n}) \\
\textbf{IF p THEN s END} &:\Rightarrow \textbf{IF p THEN s ELSE } \mathrm{skip} \textbf{ END} \\
\textbf{CHOICE } \mathbf{s_0} \textbf{ OR} \cdots \textbf{OR } \mathbf{s_n} \textbf{ END} &:\Rightarrow \mathbf{s_0} \,[]\, \cdots \,[]\, \mathbf{s_n} \\
\textbf{ANY x WHERE p THEN s END} &:\Rightarrow @\mathbf{x} \cdot (\mathbf{p} \Longrightarrow \mathbf{s}) \\
\textbf{SELECT } \mathbf{p_0} \textbf{ THEN } \mathbf{s_0} \cdots & \\
\cdots \textbf{WHEN } \mathbf{p_n} \textbf{ THEN } \mathbf{s_n} \textbf{ END} &:\Rightarrow \textbf{CHOICE } \mathbf{p_0} \Longrightarrow \mathbf{s_0} \textbf{ OR} \cdots \textbf{OR } \mathbf{p_n} \Longrightarrow \mathbf{s_n} \textbf{ END} \\
\textbf{CASE e OF EITHER } \mathbf{l_0} \textbf{ THEN } \mathbf{s_0} \textbf{ OR} \cdots & \\
\cdots \textbf{OR } \mathbf{l_n} \textbf{ THENs}_n \textbf{ END} &:\Rightarrow \textbf{SELECT } \mathbf{e} \in \{\mathbf{l_0}\} \textbf{ THEN } \mathbf{s_0} \cdots \\
& \qquad \cdots \textbf{WHEN } \mathbf{e} \in \{\mathbf{l_n}\} \textbf{ THEN } \mathbf{s_n} \textbf{ ELSE } \mathrm{skip} \textbf{ END} \\
\textbf{VAR x IN s END} &:\Rightarrow @\mathbf{x} \cdot \mathbf{s} \\
\textbf{LET x BE p IN s END} &:\Rightarrow @\mathbf{x} \cdot (\mathbf{p} \Longrightarrow \mathbf{s}) \\
\mathbf{x} := \mathrm{bool}(\mathbf{p}) &:\Rightarrow \textbf{IF p THEN x} := \mathrm{true} \textbf{ ELSE x} := \mathrm{false} \textbf{ END} \\
\mathbf{x} :\in \mathbf{e} &:\Rightarrow @\mathbf{y} \cdot (\mathbf{y} \in \mathbf{e} \Longrightarrow \mathbf{x} := \mathbf{y}) \quad \text{where } \mathbf{y} \setminus \mathbf{x} \text{ and } \mathbf{y} \setminus \mathbf{e} \\
\mathbf{x}(\mathbf{e}) := \mathbf{f} &:\Rightarrow \mathbf{x} := \mathbf{x} <\!\!+\!\!\{\mathbf{x} \mapsto \mathbf{f}\} \\
\mathbf{x} : \mathbf{p} &:\Rightarrow @\mathbf{x}' \cdot ([\mathbf{x_0}, \mathbf{x}' := \mathbf{x}, \mathbf{x}']\mathbf{p} \Longrightarrow \mathbf{x} := \mathbf{x}')
\end{aligned}
$$

Some of the rules are difficult to write down as textual rewrite rules (due to technical difficulties, not due to any fundamental difficulty), but the abstract syntax tree rewrite algorithms should be fairly obvious.
Here $\mathbf{x_0}$ (note that the nought is *not* typeset in boldface) and $\mathbf{x}'$ are obtained from $\mathbf{x}$ by suffixing each identifier in $\mathbf{x}$ with a nought subscript or a prime superscript, respectively.

Table 5.25: Rewrite rules for the AMN counterparts for GSL constructs.

be invariant over the operations (in essence, the constants may not occur in the frames of the operations) and their types are constrained. Constants must be of one of the following types: $\mathbf{s}$, $\mathbf{s} \rightarrow \mathbf{t}$ or $\mathbf{s}_1 \times \cdots \times \mathbf{s}_n \rightarrow \mathbf{t}$, where $\mathbf{s}$, $\mathbf{s}_1, \ldots \mathbf{s}_n$ and $\mathbf{t}$ are either given sets of the machine or sets of the form $\mathbf{m}..\mathbf{n}$, where $\mathbf{m}$ and $\mathbf{n}$ are numeric constants or literals. The reason for this restriction is that concrete constants will need to be implemented in the final program as initialized scalar constants or as initialized array constants. The types of constants and any additional constraints for them are given in the *properties* predicate.

Machines can contain *abstract constants*, too. They are useful to defining constants that are not meant to be accessible to the implementation of the machine but are useful in specifying the machine. The type of an abstract constant is not restricted, but it needs to be specified in the *properties* predicate. Similarly, *concrete variables* can be specified. Their types are restricted like the types of (concrete) constants, and must be specified in the *invariant* predicate.

Machines may be parametrized over a finite set or a natural number. If the parameter name is in uppercase, the parameter is a set, and if it is in lowercase, the parameter is a natural number[7]. Further constraints to the parameters can be specified in the *constraints* section.

There is an additional clause that can be used in abstract machines. The *assertions* clause allows giving lemmas that may help in discharging the proof obligations for an abstract machine (see Subsection 5.6.3).

It is possible to compose an abstract machine from several machines. This is done by listing the component machines in the *includes* clause, giving values to their parameters, if any. The composite machine's clauses are formed by concatenation, with a few exceptions. Properties, invariants and assertions are composed using conjunction, and the initializations are composed using parallel composition. The parameters of the component machines cannot be seen from the composite machine; however, the sets and constants are visible. The invariant of the composite can see the variables of the components. The inclusion relationship is transitive.

Operations are not composed by default; if some operations from the component machines are to be used verbatim for the composite machine, they can be *promoted*. The composite machine can contain additional operations — they will usually invoke the component machines' operations. The operations have read-only access to the component machines' variables (that is, they can be used in the operations but they cannot be used in their frames). If there is a component machine all of whose operations need to be promoted, it can be mentioned in the *extends* clause of the

composite machine.

An operation invocation is treated as a macro invocation: parameters are replaced with the arguments in the operation text (we treat the return parametrs as parameters), and the resulting text is used to replace the invocation.

Note that there are restrictions on the component machines — these can be deduced from the way the composite is formed and the restrictions on a machine in general. For example, two components may not name the same variable, and an operation of the composite may call at most one operation of a given component machine[8].

There is another way of using an abstract machine in another. The using machine can name the machine to be used in the *using* clause. The effect is that all clauses of the used machine except the operations are combined like in inclusion with those in the using machine. The operations of the used machine are not visible at all in the using machine. The intent is that the using and the used machine are both included in another machine. The used machine is then shared by the using machine and the including machine. The uses relationship is not transitive.

A third way is to *see* another machine. Seeing a machine means that the seen machine will be *imported* (see Subsection 5.6.4) in a refinement or implementation of the seeing machine. The seeing machine can see the given sets and (concrete and abstract) constants of the seen machine in its *includes*, *properties*, *invariants* and *operations* clauses. Additionally, the (concrete and abstract) variables of the seen machine can be seen in a read-only fashion by the seeing machine's operations. The parameters of the seen machine are not instantiated by the seeing machine, and thus it cannot see them.

5.6.2 Typechecking an abstract machine We add a new type assumption form that records the parameter and return types of an operation.

Type assumption = *Type list*, '⟵', *Identifier*
| *Type list*, '⟵', *Identifier*, '(', *Type list*, ')'
Type list = *Type*, {',', *Type*}

We add a typing rule for handling operation invocations:

$$\frac{\mathbf{s} \longleftarrow \mathbf{o}(\mathbf{t}) \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathsf{check}(\mathbf{x} \in \mathbf{s} \wedge \mathbf{e} \in \mathbf{t})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{x} \longleftarrow \mathbf{o}(\mathbf{e}))}$$

The following algorithm, formalized by the current author based on Abrial's [1] semiformal description, typechecks abstract machines that do not contain a *uses*

clause. We will skip the case of a machine with a *uses* clause, as it would only increase the notational clutter.

Algorithm 5.5 *Typechecking an abstract machine*
Precondition: *Input is a Machine that does not contain an* uses *clause, where*

- $\mathbf{M}_1(\mathbf{A}_{1,1}, \ldots, \mathbf{A}_{\mathbf{h}_1,1}, \mathbf{a}_{1,1}, \ldots, \mathbf{a}_{\mathbf{g}_1,1}), \ldots, \mathbf{M}_\mathbf{s}(\mathbf{A}_{1,\mathbf{s}}, \ldots, \mathbf{A}_{\mathbf{h}_\mathbf{s},\mathbf{s}}, \mathbf{a}_{1,\mathbf{s}}, \ldots, \mathbf{a}_{\mathbf{g}_\mathbf{s},\mathbf{s}})$ *are the included machines and their parameters;*
- $\mathbf{X}_1, \ldots, \mathbf{X}_\mathbf{k}$ *are the uppercase parameters of the machine;*
- $\mathbf{S}_1, \ldots, \mathbf{S}_\mathbf{l}$ *are those given sets whose elements are not enumerated;*
- $\mathbf{T}_1, \ldots, \mathbf{T}_\mathbf{n}$ *are those given sets whose elements are enumerated;*
- $\mathbf{t}_{1,1}, \ldots, \mathbf{t}_{1,\mathbf{m}_1}$ *are the enumerated elements of* $\mathbf{T}_1$*;*
- $\mathbf{t}_{2,1}, \ldots, \mathbf{t}_{2,\mathbf{m}_2}$ *are the enumerated elements of* $\mathbf{T}_2$*; and so on until*
- $\mathbf{t}_{\mathbf{n},1}, \ldots, \mathbf{t}_{\mathbf{n},\mathbf{m}_\mathbf{n}}$ *that are the enumerated elements of* $\mathbf{T}_\mathbf{n}$*;*
- $\mathbf{c}_1, \ldots, \mathbf{c}_\mathbf{p}$ *are the constants and abstract constants of the machine;*
- $\mathbf{x}_1, \ldots, \mathbf{x}_\mathbf{q}$ *are the lowercase parameters of the machine;*
- $\mathbf{v}_1, \ldots, \mathbf{v}_\mathbf{r}$ *are the variables and concrete variables of the machine;*
- $\mathbf{C}$ *is the constraints of the machine;*
- $\mathbf{P}$ *is the properties predicate of the machine;*
- $\mathbf{I}$ *is the invariant of the machine;*
- $\mathbf{U}$ *is the initialization substitution of the machine; and*
- $\mathbf{O}$ *is the operations of the machine.*

For each included machine $\mathbf{M}_\mathbf{i}$*, we denote its respective components by suffixing the appropriate notation with the superscript* $(\mathbf{i})$*. For example,* $\mathbf{C}^{(2)}$ *is the constraints predicate of* $\mathbf{M}_2$*.*

1. *Typecheck all included machines* $\mathbf{M}_\mathbf{i}$*, and let* $\mathbf{sig}(\mathbf{M}_\mathbf{i})$ *be the output. If this fails, halt with failure.*

2. *If the parameters, the given sets, the enumerated elements of given sets, (concrete and abstract) constants and (concrete and abstract) variables of the machine and all included machines are all not distinct identifiers, halt with failure.*

3. *If the operation names of the machine and all included machines are all not distinct, halt with failure.*

4. *If the given sets, the enumerated elements of given sets, (concrete and abstract) constants or (concrete and abstract) variables have free occurrences in the constraints predicate, halt with failure.*

5. *If the parameters or the (concrete and abstract) variables have free occurrences in the properties predicate, halt with failure.*

6. *For each* $\mathbf{i} = 1, \ldots, \mathbf{s}$*, use one of the earlier algorithms for typechecking B with the input*

$$
\begin{array}{c}
\mathit{given}(\mathbb{N}), \mathit{given}(\mathbb{Z}), \mathit{given}(\mathrm{BOOL}), \mathit{maxint} \in \mathbb{N}, \mathit{minint} \in \mathbb{Z}, \\
\mathit{true} \in \mathrm{BOOL}, \mathit{false} \in \mathrm{BOOL}, \mathit{INT} \in \mathbb{P}(\mathbb{Z}), \\
\mathit{INT}_1 \in \mathbb{P}(\mathbb{Z}), \mathit{NAT} \in \mathbb{P}(\mathbb{N}), \mathit{NAT}_1 \in \mathbb{P}(\mathbb{N}), \\
\mathit{given}(\mathbf{X}_1), \ldots, \mathit{given}(\mathbf{X_k}), \mathit{given}(\mathbf{S}_1), \ldots, \mathit{given}(\mathbf{S}_l), \mathit{given}(\mathbf{T}_1), \ldots, \mathit{given}(\mathbf{T_n}), \\
\mathbf{t}_{1,1} \in \mathbf{T}_1, \ldots \mathbf{t}_{1,\mathbf{m}_1} \in \mathbf{T}_1, \ldots, \mathbf{t}_{\mathbf{n},1} \in \mathbf{T_n}, \ldots, \mathbf{t}_{\mathbf{n},\mathbf{m_n}} \in \mathbf{T_n}, \\
\mathit{given}(\mathbf{S}_1^{(1)}), \ldots, \mathit{given}(\mathbf{S}_{\mathbf{l}^{(\mathbf{s})}}^{(1)}), \ldots, \mathit{given}(\mathbf{S}_1^{(\mathbf{s})}), \ldots, \mathit{given}(\mathbf{S}_{\mathbf{l}^{(\mathbf{s})}}^{(\mathbf{s})}), \\
\mathit{given}(\mathbf{T}_1^{(1)}), \ldots, \mathit{given}(\mathbf{T}_{\mathbf{n}^{(1)}}^{(1)}), \ldots \mathit{given}(\mathbf{T}_1^{(\mathbf{s})}), \ldots, \mathit{given}(\mathbf{T}_{\mathbf{n}^{(\mathbf{s})}}^{(\mathbf{s})}) \\
\vdash \\
\forall \mathbf{x}_1, \ldots, \mathbf{x}_q \cdot (\mathbf{C} \Rightarrow \forall \mathbf{c}_1, \ldots, \mathbf{c}_p \cdot (\mathbf{P} \Rightarrow [\mathbf{X}_1^{(\mathbf{i})}, \ldots, \mathbf{X}_{\mathbf{k}^{(\mathbf{i})}}^{(\mathbf{i})}, \ldots \mathbf{x}_1^{(\mathbf{i})}, \ldots, \mathbf{x}_{\mathbf{q}^{(\mathbf{i})}}^{(\mathbf{i})}, := \\
\mathbf{A}_{1,\mathbf{i}}, \ldots, \mathbf{A}_{\mathbf{h_i},\mathbf{i}}, \ldots \mathbf{a}_{1,\mathbf{i}}, \ldots, \mathbf{s}_{\mathbf{g_i},\mathbf{i}}]\mathbf{C}^{(\mathbf{i})}))
\end{array}
$$

   If not all succeed, halt with failure.

7. *Use one of the earlier algorithms for typechecking B with the input*

$$
\begin{array}{c}
\mathit{given}(\mathbb{N}), \mathit{given}(\mathbb{Z}), \mathit{given}(\mathrm{BOOL}), \mathit{maxint} \in \mathbb{N}, \mathit{minint} \in \mathbb{Z}, \\
\mathit{true} \in \mathrm{BOOL}, \mathit{false} \in \mathrm{BOOL}, \mathit{INT} \in \mathbb{P}(\mathbb{Z}), \\
\mathit{INT}_1 \in \mathbb{P}(\mathbb{Z}), \mathit{NAT} \in \mathbb{P}(\mathbb{N}), \mathit{NAT}_1 \in \mathbb{P}(\mathbb{N}), \\
\mathit{given}(\mathbf{X}_1), \ldots, \mathit{given}(\mathbf{X_k}), \mathit{given}(\mathbf{S}_1), \ldots, \mathit{given}(\mathbf{S_l}), \mathit{given}(\mathbf{T}_1), \ldots, \mathit{given}(\mathbf{T_n}),
\end{array}
$$

$$\mathbf{t_{1,1}} \in \mathbf{T_1}, \ldots \mathbf{t_{1,m_1}} \in \mathbf{T_1}, \ldots, \mathbf{t_{n,1}} \in \mathbf{T_n}, \ldots, \mathbf{t_{n,m_n}} \in \mathbf{T_n}$$
$$\mathit{given}(\mathbf{S_1^{(1)}}), \ldots, \mathit{given}(\mathbf{S_{l^{(s)}}^{(1)}}), \ldots \mathit{given}(\mathbf{S_1^{(1)}}), \ldots, \mathit{given}(\mathbf{S_{l^{(s)}}^{(s)}}), \ldots$$
$$\mathit{given}(\mathbf{T_1^{(1)}}), \ldots, \mathit{given}(\mathbf{T_{n^{(1)}}^{(1)}}), \ldots \mathit{given}(\mathbf{T_1^{(s)}}), \ldots, \mathit{given}(\mathbf{T_{n^{(s)}}^{(s)}})$$
$$[\mathbf{X_1^{(1)}}, \ldots, \mathbf{X_{k^{(1)}}^{(1)}}, \ldots \mathbf{x_1^{(1)}}, \ldots, \mathbf{x_{q^{(1)}}^{(1)}}, := \mathbf{A_{1,1}}, \ldots, \mathbf{A_{h_1,1}}, \ldots \mathbf{a_{1,1}}, \ldots, \mathbf{s_{g_1,1}}] \ \mathbf{sig}(\mathbf{M_1}), \ldots$$
$$[\mathbf{X_1^{(s)}}, \ldots, \mathbf{X_{k^{(s)}}^{(i)}}, \ldots \mathbf{x_1^{(s)}}, \ldots, \mathbf{x_{q^{(s)}}^{(s)}}, := \mathbf{A_{1,s}}, \ldots, \mathbf{A_{h_s,s}}, \ldots \mathbf{a_{1,s}}, \ldots, \mathbf{s_{g_s,s}}] \ \mathbf{sig}(\mathbf{M_s})$$
$$\vdash$$
$$\forall \mathbf{x_1}, \ldots, \mathbf{x_q} \cdot (\mathbf{C} \Rightarrow \forall \mathbf{c_1}, \ldots, \mathbf{c_p} \cdot (\mathbf{P} \Rightarrow \forall \mathbf{v_1}, \ldots, \mathbf{v_r} \cdot (\mathbf{I} \Rightarrow [\mathbf{U};\ \mathbf{O}](\text{frame}(\mathbf{U};\ \mathbf{O}) = \text{frame}(\mathbf{U};\ \mathbf{O})))))$$

with the modification that it keeps track of and outputs the types inferred for $\mathbf{c_i}$, $\mathbf{x_i}$, $\mathbf{v_i}$ *and the signature of each operation in* $\mathbf{O}$ *(which can be obtained by replacing each iidentifier in the operation header with their inferred type); we'll denote them by* $\mathit{type}(\mathbf{c_i})$, $\mathit{type}(\mathbf{x_i})$, $\mathit{type}(\mathbf{v_i})$ *and* $\mathbf{sig}(\mathbf{O})$. *If it fails, halt with failure.*

8. *Output*

$$\mathbf{x_1} \in \mathit{type}(\mathbf{x_1}), \ldots, \mathbf{x_q} \in \mathit{type}(\mathbf{x_q}), \mathbf{t_{1,1}} \in \mathbf{T_1}, \ldots, \mathbf{t_{n,m_n}} \in \mathbf{T_n},$$
$$\mathbf{c_1} \in \mathit{type}(\mathbf{c_1}), \ldots, \mathbf{c_p} \in \mathit{type}(\mathbf{c_p}), \mathbf{v_1} \in \mathit{type}(\mathbf{v_1}), \ldots, \mathbf{v_r} \in \mathit{type}(\mathbf{v_r}), \mathbf{sig}(\mathbf{O})$$

and halt with success.

Postcondition: *Output is an environment (a list of Type assumptions) or an indication of failure.*

We deviate from Abrial [1] in that we allow $\mathbb{N}$ and $\mathbb{Z}$ in abstract machines. Abrial's reasoning is that machines should be implementable, and infinite sets are not implementable, but in the opinion of the current author, this should be ensured with proof obligations, not by artificially restricting the range of numbers that can be expressed. Schneider [107], too, allows the infinite number sets in abstract machines.

5.6.3 Verification of an abstract machine An abstract machine can be verified by proving certain propositions about it. The propositions are traditionally called *proof obligations* and proving them is traditionally called *discharging* them. To avoid the notational baggage noticed above in typechecking, we describe them in words only.

There are three proof obligations for abstract machines.

Initialization establishes the invariant Prove the following:

> **Assuming** that the uppercase parameters are finite sets of integers, and **assuming** that the given sets are finite sets of integers, and **assuming** that the enumerated given sets comprise exactly the enumerated elements, which are distinct, and **assuming** that the constraint predicate holds, and **assuming** that the properties predicate holds, **then** for the initialization substitution **s** and the invariant **p**, the predicate [**s**]**p** holds.

Assertions hold Prove the following:

> **Assuming** that the uppercase parameters are finite sets of integers, and **assuming** that the given sets are finite sets of integers, and **assuming** that the enumerated given sets comprise exactly the enumerated elements, which are distinct, and **assuming** that the constraint predicate holds, and **assuming** that the properties predicate holds, and **assuming** that the invariant holds, **then** the assertions hold.

Operations preserve the invariant Prove the following for each operation **q** | **v**:

> **Assuming** that the uppercase parameters are finite sets of integers, and **assuming** that the given sets are finite sets of integers, and **assuming** that the enumerated given sets comprise exactly the enumerated elements, which are distinct, and **assuming** that the constraint predicate holds, and **assuming** that the properties predicate holds, and **assuming** that the invariant **p** holds, and **assuming** the assertions hold, and **assuming** that the precondition **q** holds, **then** the predicate [**s**]**p** holds.

Existence proofs for actual machine parameters, concrete constants, variables or actual parameters for operations are not required. This is to make proofs at this stage feasible. This does not sacrifice correctness, however, since constructive proofs for these matters will be provided in the implementation stage at the latest.

Additional proof obligations (as well as modifications to the ones presented above) are needed for machines that *include*, *use* or *see* another machine. We skip them here for reasons of space.

5.6.4 Refinement In B, specifications are refined in one or more steps, to reach an implementation of the specification that can be mechanically translated to machine language. The final result of this process, the *implementation machine* is discussed later. The object of this subsection is to introduce the key concepts of refinement and the intermediate result of the process, the *refinement machine*.

A *refinement* of a specification is a specification that, for all inputs for which the original specification specifies normal termination, allows only behaviour that the original specification allows. There are three ways it can differ from the original specification: it can be more deterministic, it can specify normal termination in more cases than the original, and it can be more algorithmic.

There is a metalevel relation between generalized substitutions, called *refinement*. Informally speaking, a generalized substitution $\mathbf{s}$ refines another $\mathbf{t}$ when the frame of $\mathbf{t}$ contains every variable contained in the frame of $\mathbf{s}$ and for every postcondition $\mathbf{p}$, it holds that $\forall \boldsymbol{\alpha} \cdot [\mathbf{t}]\mathbf{p} \Rightarrow [\mathbf{s}]\mathbf{p}$, where $\boldsymbol{\alpha}$ contains every free variable of the quantified predicate. Refinement of substitutions is a partial ordering, and all thus far introduced generalized substitution constructs are monotonic with respect to refinement. This means that refinement of a large substitution can be done by refining its components.

Refinement of an abstract machine consists of writing a different machine, written in a special *refinement* syntax, which declares its refinement relationship to the original machine. The refinement machine's invariant is required to relate the original machine's (abstract) state to the state of the refinement machine. The refinement machine is a "patch" to the original machine for reaching the real refinement; essentially, the refinement machine is combined with the original machine to reach a different machine that is the actual refined machine.

We skip the formal development of refinements.

5.6.5 Implementation Implementation is the final stage of development of an abstract machine. It is a refinement specification that is directly implementable by a translation automaton. Its behaviour is specified fully algorithmically, it has no indeterminism and all its variables and constants are concrete.

Like refinement, implementation is specified as an *implementation machine* that has similar structure to a refinement machine. The implementation machine is combined with the original abstract machine and its other refinements, if any.

Implementation machines can *import* abstract machines. The idea of importation of abstract machines is similar to the idea of using the interfaces of other modules

in conventional programming.

We skip the formal development of implementations, too.

5.6.6 A parallel between AMN and Object-Orientation There is an obvious parallel between object-orientation and the Abstract Machine Notation. We can identify a rough partial mapping between them[9]:

AMN		OO
abstract machine	$\longrightarrow$	abstract class
concrete variables	$\longrightarrow$	attributes
operation	$\longrightarrow$	method
refinement machine	$\longrightarrow$	derived abstract class
refinement	$\longrightarrow$	inheritance
implementation machine	$\longrightarrow$	derived concrete class
importation	$\longrightarrow$	aggregation

However, there are also concepts in AMN that do not have an obvious parallel in OO. Abstract variables and abstract constants are such — even in object-oriented systems that support programming by contract (such as Eiffel [80]) there is no way to specify abstract state that will not be represented directly in the final program. Also, there are deficiencies even in the mapping shown above. For example, refinement and implementation machines do not have independent status: one cannot use a refinement or implementation directly in other machines. Likewise, abstract machines are not types.

Notes

[1] We will deviate from the standard in two points. First, we allow incremental specification, ie. multiple definitions of the same nonterminal are concatenated. Second, we do not require a terminating semicolon.

[2] Abrial gives names to only some inference rules. The rest of the names are either customary or coined up by the present author.

[3] In fact, Abrial never defines the *Identifier* syntactic category in his B-Book [1].

[4]The names have mostly been coined by the present author. The code names in parentheses are from Abrial [1].

[5]It is clear that existing implementations solve these problems in some way, but the algorithms are not publically available as far as the current author knows.

[6]We give here a different, and hopefully clearer, definition than Abrial.

[7]So which one a mixed-case identifier is? Abrial does not tell.

[8]Abrial [1] states the latter as a rule, without giving it any justification except an example where not heeding the restriction will break the machine. However, Dunne's theory of general substitutions [35], which was used earlier in this chapter, gives an intuitive explanation: parallel composition of substitutions with overlapping frames is invalid unless the substitutions are equivalent with respect to the overlap.

[9]The similarity of abstract machines and classes has been noted by Abrial [1]. However, the other parallels are due to the current author but they are not necessarily unique to him.

6 The Ebba Toolset

The Ebba Toolset, also known simply as Ebba, is intended to be an independent reimplementation of the B method based on public sources. The intent is to produce a free implementation for research (and other) use. The purpose of this chapter is to describe its development and current state.

6.1 Development history

The development of Ebba started in late 2001. The current author had studied the B method for many months, and the lack of a free software implementation made it harder than it has to be. This provided the motivation for building Ebba.

It was decided early on that no specific project model or software process was to be used. In a one-person project, they would merely be unnecessary baggage, and since the present author had no experience with writing such software, their benefits were seen negligible, as software processes are generally meant to ensure repeating earlier successes and not generating the first success. Experimentation was seen as the best route.

There were three main choices for the principal programming language in which Ebba was to be written. Haskell [95] was the author's pet language, but his lack of experience in it was deemed too big a risk factor at that time. Java [51], however, looked promising. The author had recently worked in Java for one and a half years. Also, it was clear that the front end of Ebba would strongly resemble the front end of a compiler (cf. eg. [3]), and there was a promising compiler front end generator available (SableCC, [46, 47]). However, using Java would have meant sacrificing some of the freeness of the result, as the free implementations of Java have many deficiences. The only remaining candidate, C++ [64], looked very good, as the author has experience in it and also has a rather deep understanding of it. Therefore, C++ was initially chosen.

The first plans for Ebba included grandiose ideas of a metasystem: a system where the B method could be embedded and dynamically extended. However, this proved to be too hard.

The first implementation attempt, dubbed Ebba I[1], consists of an editor for Unicode text and a lexer and a parser and inference engine for Unicode-encoded inference rule collections (written in Prolog-like notation) and a subset of the abstact machine notation. At that point, much of the effort was directed at making the use of Unicode input as unobtrusive as possible. Unfortunately, the editor was written for character-cell terminals, and thus its ability to show or read Unicode characters is dependent not only on the correct fonts being available but also on correct configuration of the terminal (or the terminal emulator). Its compile times were also intolerable[2].

The second attempt (dubbed Ebba II) tried to ease the use of the editor by taking care of input and output of Unicode characters itself. For this, it includes a rewrite of the editor, which runs under the X Windowing System, Version 11.

From the experience of Ebba I, it became increasingly clear that C++ was clumsy for writing programs that operate on trees with many kinds of nodes. The final attempt, dubbed Ebba H, was written in Haskell. The author felt that Haskell's support for algebraic data types would make it easier to write those parts of the program that operate on abstract syntax.

Unlike its predecessors, Ebba H's development focused on actually producing a typechecker and prover for the B method. Unicode input was seen as a secondary issue, and Ebba H used an ASCII notation for months. Recently, even Ebba H was changed to support Unicode input.

All three implementation attempts of Ebba are available on the Ebba home page, http://www.nongnu.org/ebba/. The rest of this chapter will describe Ebba H; any occasional reference to Ebba I or Ebba II will be clearly marked.

6.2 Abstract syntax

Haskell programs are organized into modules. The central module in Ebba H is AbstractSyntax which declares the types of the abstract syntax tree data structure for the logic, set theory and GSL parts of the language. Appendix B shows (a simplified version of) the module. It declares four new mutually recursive data types: Predicate, Variable, Expression and Substitution. Each corresponds to a nonterminal in the abstract syntax given in Chapter 5. The module also implements functions that perform unification (over metavariables), removal of derived constructs and other support functions.

There are certain points that should be noted:

- A value constructed using MetavariablePredicate, MetavariableVariable or MetavariableExpression denotes a metavariable that denotes a predicate, a variable or an expression, respectively. They all list side conditions that constrain possible instantiations of the metavariables (for example, that the instantiation of the metavariable should contain no free instances of a certain variable).

- The nullary constructor NoPredicate denotes the lack of a predicate in the context. In all context, this is interpreted as the identity element of whatever construct it is part of. For example, in the excerpt

  result = *simplify* (*p* ‘LogicalAnd‘ NoPredicate) == *p*

  the value of *result* is True for all values of *p* in the type Predicate (here *simplify* is a function that removes all derived constructs from its arguments).

- Similarly, EmptyVariable is meant to represent the empty frame and is treated appropriately by quantification constructs. Values constructed using the constructors IllFormedPredicate and IllFormedExpression denote the simplifications of ill-formed formulae (such as EmptyVariable used in an expression context). The parameter to the two constructors is supposed to be a human-readable error message describing the nature of the ill-formedness.

- The constructors PredicateApplication and FunctionApplication encode common patterns of behaviour of many derived constructs. For example, set union and set intersection are encoded as FunctionApplications. The benefit is that there is no need to add a separate case for all derived constructs to all functions that operate on the abstract syntax tree by cases.

- Values constructed using PSubst denote the (delayed) substitution of a predicate for a metavariable.

The abstract syntax of abstract machines is defined elsewhere and is not discussed in this thesis.

6.3 The frontend

The frontend of Ebba H is responsible for reading the Unicode characters, scanning the input for tokens and parsing the stream of tokens into an abstract syntax tree.

Operators	Associativity
.	$\Longleftarrow$
$[\,](\,)$ (substitution)	
$(\,)(\,)$ (function application)	$\Longrightarrow$
$=$	$\Longrightarrow$
$,\quad \mapsto$	$\Longrightarrow$
$\times$	$\Longrightarrow$
$\cup \quad \cap \quad -$	$\Longrightarrow$
$\in \quad \subseteq$	
$\neg$	$\Longleftarrow$
$\wedge \quad \vee$	$\Longrightarrow$
$\Rightarrow$	$\Longrightarrow$
$\Leftrightarrow$	$\Longrightarrow$

Higher precedence is depicted as position nearer the top.

Table 6.1: Operator precedence and associativity for Ebba H.

The Haskell 98 (revised) report [95] specifies that the built-in character subsystem implements a version of Unicode. However, none of the Haskell implementations considered (Hugs, The Glorious Glasgow Haskell Compiler (GHC) and the Nearly a Haskell Compiler (NHC)) implement Unicode in sufficient detail. For this reason, a library of Unicode support modules (`ebba-unicode`) was written for Ebba H. It is summarized in Appendix D.

The Unicode library converts any octet stream into a stream of Unicode characters, following the UTF-8 transformation format. The characters are then passed on to `EbbaLexer`, which turns the stream of characters into a stream of tokens. The tokens are given to the parser, which was written using the Happy [79] parser generator. The parser generates an abstract syntax tree of the input based on the token stream.

Writing the parser required fixing certain aspects of the concrete syntax left unspecified by Abrial (and by ourselves in Chapter 5). The most important of these is operator precedence and associativity, which is summarized in Table 6.1 for those operators that Ebba H currently supports. The assignment of precedence levels and associativity for operators was basically arbitrary, given the constraints specified by Abrial. The assignment is hoped to be useful, but there has been very little testing.

6.4 Unicode input

Ebba's frontend processes all input as Unicode [120] text. This gives it the ability to use the original publication language as the tool input instead of a hardcoded "hardware" language which is limited by the abilities of decades-old 7-bit ASCII encoding. There are currently very few real reasons for such a restricted input method, and as the Unicode support of systems software get better in the future, even the remaining reasons vanish. If the available input devices do have such a limitation, the use of Unicode as the primary input language allows the use of a preprocessor tailored to the specific limitations of the input device and the needs of the specific user.

The use of Unicode was inspired primarily by the present author's previous work on CatDVI [69], a translator from TEX Device Indepedent (DVI) files to plain text, which uses Unicode internally and as one of the output character sets. Secondarily it was inspired by the use of Unicode in programming languages such as Java, and the use of ISO 10646 in specifying the primary input language of ISO Z [65].

The biggest problem with using Unicode is its poor support in systems software. For example, Microsoft Windows 2000 supports Unicode version 2.0 [123]. A big problem with this is that Unicode versions below 3.1 specified Unicode code points to be representable in 16 bits (with a provision for extension using a baroque technique called "surrogates"), which requirement was lifted in 3.1 and made obsolete by assigning code units beyond that range in 3.2. The documentation of version 2.0 of the GTK+ widget set for the X Windowing System does not specify the level of Unicode support, but it is clear that it too is limited to supporting 16-bit characters. This is a particularly bad problem for Ebba, as many of the characters it requires were assigned in Unicode version 3.2 beyond the 16-bit range.

Early versions of Ebba (Ebba I and Ebba II) concentrated in working around deficiences in systems software. Two simplistic editors were prototyped, one using the Curses library [32] (in Ebba I), which allows writing full-screen software for a wide variety of character-cell terminals, and one using Xt Intrinsics and the Athena widgets [89] (in Ebba II), which are simple libraries that allow the writing of software for the X Windowing System. The third and current version (Ebba H) concentrated first on other issues, but Unicode support was recently implemented. This version assumes proper support from systems software — it does not attempt to duplicate their functionality — and failing that, it allows the use of compatibility preprocessors for using legacy input devices and deficient systems software.

Notation	Code unit	Character name
$\vdash$	U+22A2	RIGHT TACK
$\wedge$	U+2227	LOGICAL AND
$\Rightarrow$	U+21D2	RIGHTWARDS DOUBLE ARROW
$\neg$	U+00AC	NOT SIGN
$\vee$	U+2228	LOGICAL OR
$\Leftrightarrow$	U+21D4	LEFT RIGHT DOUBLE ARROW
[	U+005B	LEFT SQUARE BRACKET
]	U+005D	RIGHT SQUARE BRACKET
$\forall$	U+2200	FOR ALL
$\cdot$	U+22C5	DOT OPERATOR
$=$	U+003D	EQUALS SIGN
$:=$	U+2254	COLON EQUALS
$\exists$	U+2203	THERE EXISTS
$\neq$	U+2260	NOT EQUAL TO
,	U+002C	COMMA
$\mapsto$	U+21A6	RIGHTWARDS ARROW FROM BAR
$\in$	U+2208	ELEMENT OF
(	U+0028	LEFT PARENTHESIS
)	U+0029	RIGHT PARENTHESIS
$\times$	U+00D7	MULTIPLICATION SIGN (cartesian product)
$\mathbb{P}$	U+2119	DOUBLE-STRUCK CAPITAL P
{	U+007B	LEFT CURLY BRACKET
}	U+007D	RIGHT CURLY BRACKET
$\mid$	U+007C	VERTICAL LINE (comprehension separator, preconditioning)
$\equiv$	U+2261	IDENTICAL TO
$\subseteq$	U+2286	SUBSET OF OR EQUAL TO
$\subset$	U+2282	SUBSET OF
$\cup$	U+222A	UNION
$\cap$	U+2229	INTERSECTION
$-$	U+2212	MINUS SIGN (set difference)
$\varnothing$	U+2205	EMPTY SET
$\bigcup$	U+22C3	N-ARY UNION
$\bigcap$	U+22C2	N-ARY INTERSECTION
$\leftrightarrow$	U+2194	LEFT RIGHT ARROW
;	U+003B	SEMICOLON (forward composition, sequencing)
$<\!\!+$	U+003C U+002B	LESS-THAN SIGN, PLUS SIGN

Table 6.2: Unicode code unit assignments for Ebba input characters *(cont.)*

Notation	Code unit	Character name
∘	U+2218	RING OPERATOR
◁	U+25C1	WHITE LEFT-POINTING TRIANGLE
▷	U+25B7	WHITE RIGHT-POINTING TRIANGLE
⩤	U+2A64	Z NOTATION DOMAIN ANTIRESTRICTION
⩥	U+2A65	Z NOTATION RANGE ANTIRESTRICTION
⊗	U+2297	CIRCLED TIMES
∥	U+2016	DOUBLE VERTICAL LINE (parallel product)
⇸	U+21F8	RIGHTWARDS ARROW WITH VERTICAL STROKE
→	U+2192	RIGHTWARDS ARROW
⤔	U+2914	RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
↣	U+21A3	RIGHTWARDS ARROW WITH TAIL
⤖	U+2900	RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
↠	U+21A0	RIGHTWARDS TWO HEADED ARROW
⤕	U+2917	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE
⤀	U+2916	RIGHTWARDS TWO-HEADED ARROW WITH TAIL
λ	U+03BB	GREEK SMALL LETTER LAMDA
𝔽	U+1D53D	MATHEMATICAL DOUBLE-STRUCK CAPITAL F
ℕ	U+2115	DOUBLE-STRUCK CAPITAL N
.	U+002E	FULL STOP
+	U+002B	PLUS SIGN
−	U+002D	HYPHEN-MINUS (number difference)
×	U+002A	ASTERISK (number product)
/	U+2215	DIVISION SLASH
⟹	U+27F9	LONG RIGHTWARDS DOUBLE ARROW
▯	U+2AFE	WHITE VERTICAL BAR
@	U+0040	COMMERCIAL AT
∥	U+2225	PARALLEL TO (parallel composition)
^	U+2303	UP ARROWHEAD
≙	U+2259	ESTIMATES
⟵	U+27F5	LONG LEFTWARDS ARROW

Table 6.2: *(cont.)* Unicode code unit assignments for Ebba input characters.

The character mapping used (and partially planned to be used) by Ebba is summarized in Table 6.2. All characters in that table are considered punctuation except those whose general category is L&. Unicode identifier conventions are used — no normalization is currently done. Those characters in Table 6.2 that are valid identifiers, are reserved words. The usual numeric literal conventions are used.

In selecting Unicode code units for each notational atom in B, the mapping specified for ISO Z was used as a starting point. Whenever B and Z use the same notation for the same thing, the character used in ISO Z was considered. In many cases, it was decided to use the semantically correct character (for example, `U+2215` DIVISION SLASH) instead of a visually similar ASCII character (for example, `U+002F` SOLIDUS). Although this may present problems in current environments, the benefits in the long run were weighed more important. In some cases, it was deemed necessary to use two different code units for the same character in differing semantic contexts (for example, − is mapped to U+2212 MINUS SIGN when it denotes set difference and to U+002D HYPHEN-MINUS when it denotes numeric difference).

6.5 Backend

The backend is responsible for doing something useful with the absrtact syntax tree of the input. The backend contains a typechecker, a proof engine and a prettyprinter. Planned features for the backend include a code generator for implementation machines.

The typechecker takes an abstract syntax tree and uses a typechecking algorithm (a variant of the one described at the end of Subsection 5.3.4) to verify the type soundness of the input. For now, the typechecker only handles raw set notation; no generalized substitutions or machines can be checked at this time. The typechecker is described in detail in Appendix C.

The proof engine is preliminary. It is capable of proving simple statements in first-order logic and is based on the proof procedure given in 5.2.5).

6.6 Future plans

There are several issues with Ebba as it currently stands:

1. It does not understand the full language of the B method. Even not everything presented in Chapter 5 is supported.

2. The typechecker does not support even the full language supported by the frontend.

3. Proof obligations are not extracted from specifications.

4. It should be able to generate code from implementation machines.

5. The proof engine does not support set notation.

6. It is fragile and needs extensive testing.

The first four issues are mostly a matter of simple programming. The next issue will require further study and perhaps even reseach.

Notes

[1] We describe Release 1 of Ebba I.

[2] A recent full build took 20 minutes with G++ 3.0 on an AMD Athlon XP 1,5 GHz, although the SLOC count (excluding generated source) was only about 5000.

7 Conclusion

This thesis has presented a sketch for a formal reconstruction of the B method and for a freely available realization of its tool support. In addition, we have presented, as background, the role of formal methods in the general context of software engineering, a historically motivated review of formal logic and set theory, and a quick sketch of the main methods of automated reasoning.

This thesis has been hurt by its status. Its scope is so huge that a proper treatment would have required far more time, resources and pages than can be reasonably spent on a master's thesis. As such, this thesis feels like a torso: there is a skeleton for the whole thing, but only parts of the body has flesh. It is the hope of the author that the thesis still has a purpose to serve, other than as the vehicle to a degree.

Even as it stands, this thesis makes technical contributions: most importantly, it develops an input format for the B notation that is both efficiently implementable on a computer and strongly resembles the notation used in this thesis and other printed works on the B method. It also notes problems with the B method as publically documented and proposes remedies to some of them. Most notably, a problem with the typechecking algorithm was noted and resolved.

Many of the corrections and enhancements to the B method suggested in this thesis are not fully validated nor verified for correctness — examples are the improved typechecking algorithm and the formalization of substitution frames.

It would be interesting to develop a formal method derived from the B method that exploits the parallels between the concepts of AMN and Object-Orientation: is it possible to make a programming language that is at the same time a formal method to be reckoned with?

As a smaller matter, it would be good to find a rewrite rule for substitution opening that does not depend on rule application order nor on a machine context.

8 Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge: Cambridge University Press, 1996.

[2] Jean-Raymond Abrial and Louis Mussat. "On using conditional definitions in formal theories". In Bert et al. [12].

[3] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.

[4] Aristoteles. *Kategoriat, Tulkinnasta, Ensimmäinen analytiikka ja Toinen analytiikka*. No. 1 in Teokset. Helsinki: Gaudeamus, 1994.

[5] B-Core. *The B-Toolkit User's Documentation*. B-Core (UK) Ltd., Aug. 1999.
URL http://www.b-core.com/ONLINEDOC/Contents.html

[6] B-Core. "B-toolkit", 2001.
URL http://www.b-core.com/btoolkit.html

[7] Franz Baader and Wayne Snyder. "Unification theory". In Robinson and Voronkov [100].

[8] Leo Bachmair and Harald Ganzinger. "Resolution theorem proving". In Robinson and Voronkov [100].

[9] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. New York: Springer, 1998.

[10] John Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". *Communications of the ACM*, 21 (8), pp. 613–641, Aug. 1978.

[11] Paul Bernays and Abraham A. Fraenkel. *Axiomatic set theory with a historical introduction*. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland, 1958.

[12] Didier Bert, Jonathan P. Bowen, Martin C. Henson and Ken Robinson (eds.). *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of Z and B Users, Grenoble, France, January 2002*, no. 2272 in Lecture Notes in Computer Science. Berlin: Springer, 2002.

[13] Grady Booch. *Object-oriented analysis and design with applications*. Reading, MA: Addison-Wesley, 1994.

[14] George Boole. *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning*. London: Macmillan, Barclay & Macmillan, 1847. Reprinted by Basil Blackwell, Oxford, 1965.

[15] —. *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*. Dover, 1955. First American printing of the 1854 edition with all corrections made within the text.

[16] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995, anniversary ed.

[17] Cesare Buriali-Forti. "On well-ordered classes". In van Heijenoort [55].

[18] —. "A question on transfinite numbers". In van Heijenoort [55].

[19] Stanley Burris. "The laws of Boole's thought". Preprint, accessed on the 22th of May, 2002.
URL http://www.thoralf.uwaterloo.ca/htdocs/MYWORKS/PREPRINTS/aboole.pdf

[20] Michael Butler, Jim Grundy et al. "The refinement calculator: Proof support for program refinement". In Lindsay Groves and Steve Reeves (eds.), "Formal Methods Pacific'97: Proceedings of FMP'97", (pp. 40–61). Wellington, New Zealand: Springer-Verlag, 1997.
URL http://citeseer.nj.nec.com/butler97refinement.html

[21] Georg Cantor. *Gesammelte Abhandlungen mathematischen und philosophischen Inhalts mit erläuternden Anmerkungen sowie mit Ergänzungen aus dem Briefwechsel Cantor–Dedekind, herausgaben von Ernst Zermelo nebst einem Lebenslauf Cantors von Adolf Fraenkel*. Berlin: Springer, 1932. Reprinted in 1990.

[22] —. *Contributions to the Founding of the Theory of Transfinite Numbers*. New York: Dover, 1955. Translated by Philip E. B. Jourdain.

[23] —. "Letter to Dedekind". In van Heijenoort [55].

[24] George Cantor. "Über eine Eigenschaft des Ingbegriffes aller reellen algebraischen Zahlen". In "Gesammelte Abhandlungen mathematischen und philosophischen Inhalts mit erläuternden Anmerkungen sowie mit Ergänzungen aus dem Briefwechsel Cantor–Dedekind, herausgaben von Ernst Zermelo nebst einem Lebenslauf Cantors von Adolf Fraenkel", [21]. Reprinted in 1990.

[25] —. "Ein betrag zur mannigfaltigkeitslehre". In "Gesammelte Abhandlungen mathematischen und philosophischen Inhalts mit erläuternden Anmerkungen sowie mit Ergänzungen aus dem Briefwechsel Cantor–Dedekind, herausgaben von Ernst Zermelo nebst einem Lebenslauf Cantors von Adolf Fraenkel", [21]. Reprinted in 1990.

[26] D. Carrington, I. Hayes et al. *A tool for developing correct programs by refinement*. Technical report 95-49, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, Oct. 1996.
URL http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?95-49

[27] Alonzo Church. "An unsolvable problem of elementary number theory". *American Journal of Mathematics*, 58 (2), pp. 345–363, Apr. 1936.

[28] Stephen A. Cook and Robert A. Reckhow. "Time-bounded random access machines". In "Proceedings of the Fourth Annual ACM Symposium on Theory of Computing", ACM, 1972.

[29] Mark Davis. *Unicode Newline Guidelines*. Unicode Standard Annex 13, Unicode Inc., 2002.
URL http://www.unicode.org/unicode/reports/tr13/tr13-9.html

[30] Mark Davis, Michael Everson et al. *Unicode 3.1*. Unicode Standard Annex 27, Unicode Inc., May 2001.
URL http://www.unicode.org/unicode/reports/tr27/tr27-4.html

[31] Richard Dedekind. *Was sind und was sollen die Zahlen & Stetigkeit und Irrationale Zahlen*. Braunschweig: Vieweg, 1969. Originally published in 1872 (Was sind un was sollen die Zahlen) and in 1872 (Stetigkeit und Irrationale Zahlen).

[32] Thomas E. Dickey. "NCURSES — new Curses". Accessed on 18th of December, 2002.
URL http://dickey.his.com/ncurses/

[33] E. W. Dijkstra. "Structured programming". In J. N. Buxton and B. Randell (eds.), "Software Engineering Techniques: Report on a conference sponsored by the NATO Scuence Committee, Rome, Italy, 27–31 Oct. 1969", (pp. 84–88). Brussels: NATO Scientific Affairs Division, 1970.
URL http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/

[34] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Englewood Cliffs: Prentice-Hall, 1976.

[35] Steve Dunne. "A theory of generalised substitutions". In Bert et al. [12].

[36] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, vol. 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[37] Joseph Feller and Brian Fitzgerald. "A framework analysis of the open source software development paradigm". In Soon Ang, Helmut Krcmar et al. (eds.), "Proceedings of the 21st international conference on Information systems", (pp. 58–69). Association for Information Systems, 2000.

[38] Cormac Flanagan and Shaz Qadeer. "Predicate abstraction for software verification". *ACM SIGPLAN Notices*, 37 (1), pp. 191–202, Jan. 2002. Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02).

[39] Robert W. Floyd. "Assigning meanings to programs". In J. T. Schwartz (ed.), "Mathematical Aspects of Computer Science", vol. XIX of *Proceeding of Symposia in Applied Mathematics*. Providence: American Mathematical Society, 1967.

[40] Abraham A. Fraenkel. "The notion of "definite" and the independence of the axiom of choice". In van Heijenoort [55].

[41] G. Frege. *Die Grundlagen der Arithmetik, eine logisch matematische Untersuchung über den Begriff der Zahl. The Foundations of Arithmetic, a logico-mathematical enquiry into the concept of number*. Oxford: Basil Blackwell, 1959. A reprint of Wil-

helm Koebner's (Breslau) printing of 1884, with an English translation printed *en face*.

[42] Gottlob Frege. *Grundgesetze der Arithmetik begriffsschriftlich abgeleitet*. Darmstadt: Wissenschaftliche Buchgesellschaft, 1962. An unaltered Reprint of the 1893 edition.

[43] —. "*Begriffsschrift*, a formula language, modeled upon that of arithmetic, for pure thought". In van Heijenoort [55].

[44] —. "Letter to Russell". In van Heijenoort [55].

[45] —. "Frege on Russell's paradox, Grundgesetze der Arithmetik, vol ii, appendix, pp. 253–65". In Peter Geach and Max Black (eds.), "Translations from the Philosophical Writings of Gottlob Frege", Oxford: Basil Blackwell, 1970.

[46] Étienne Gagnon. *SableCC — an object-oriented compiler framework*. Master's thesis, School of Computer Science, McGill University, Mar. 1998.

[47] Etienne M. Gagnon and Laurie J. Hendren. "SableCC, an object-oriented compiler framework". In "Proceedings of the Technology of Object-Oriented Languages and Systems", 1998.

[48] Kurt Gödel. *On formally undecidable propositions of Principia Mathematica and related systems I*. New York: Dover, 1992. Reprint of the 1931 paper.

[49] W. Wayt Gibbs. "Trends in computing: Software's chronic crisis". *Scientific American*, 271 (3), Sep. 1994.

[50] John Gilmore. "Finally, a primary source on Mariner 1". *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*, 5 (73), Dec. 1987. URL http://catless.ncl.ac.uk/Risks/5.73.html#subj2

[51] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2000, 2nd ed.

[52] Jeremy Gray. "Did Poincaré say "set theory is a disease"?" *The Mathematical Intelligencer*, 13 (1), pp. 19–22, 1991.

[53] Wolfgang Grieskamp et al. "The ZETA system: Overview (and other documents)". Accessed on the 30th of April, 2002.
URL http://uebb.cs.tu-berlin.de/zeta/

[54] Les Hatton. "Software failures: follies and fallacies". *IEE Review*, 43 (2), pp. 49–52, Mar. 1997.

[55] Jean van Heijenoort (ed.). *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Cambridge, MA: Harvard University Press, 1967.

[56] D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Berlin: Springer, 1967, 5th ed. First edition was published in 1928.

[57] David Hilbert. "On the infinite". In van Heijenoort [55].

[58] C. A. R. Hoare. "An axiomatic basis for computer programming". *Communications of the ACM*, 12 (10), pp. 576–583, Oct. 1969.

[59] —. "Communicating sequential processes". *Communications of the ACM*, 21 (8), pp. 666–677, Aug. 1978.

[60] —. "How did software get so reliable without proof?" In "FME'96: industrial benefit and advances in formal methods: Third International Symposium of Formal Methods Europe, Oxford, UK, March 18–22", (pp. 1–17). Berlin: Springer, 1996.

[61] International Organization for Standardization. *Information technology — Programming languages — Ada*, 1995. ISO/IEC 8652:1995.

[62] —. *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language*, 1996. ISO/IEC 13817-1:1996.

[63] —. *Information technology — Syntactic metalanguage — Extended BNF*, 1996. ISO/IEC 14977:1996(E).

[64] —. *Programming languages — C++*, 1998. ISO/IEC 14882:1998(E).

[65] —. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, 2002. ISO/IEC 13568:2002.

[66] Thomas Jech. *Set Theory*. New York: Academic Press, 1978.

[67] Stefan Kahrs, Donald Sannella and Andrzej Tarlecki. "The definition of Extended ML: A gentle introduction". *Theoretical Computer Science*, 173 (2), pp.

445–484, 1997.
URL http://citeseer.nj.nec.com/kahrs95definition.html

[68] Antti-Juhani Kaijanaho. "PR — a Prolog-esque inference engine", Jun. 2001.
URL ftp://ftp.jyu.fi/private/antkaij/pr-0.1.tar.gz

[69] Antti-Juhani Kaijanaho, Björn Brill and J. H. M. Dassen. "CatDVI: a DVI to text/plain translator", Nov. 2002.
URL http://catdvi.sourceforge.net/

[70] Immanuel Kant. *The Critique of Pure Reason*. Project Gutenberg, 2003. Released ahead of schedule on 29th of December 2001. Translated by J. M. D. Meiklejohn. First publication in 1781.
URL ftp://ibiblio.org/pub/docs/books/gutenberg/etext03/cprrn10.txt

[71] Donald E. Knuth. *Literate Programming*. No. 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[72] Pasi Koikkalainen and Pekka Orponen. *Tietotekniikan perusteet*. Jyväskylän yliopisto, tietotekniikan laitos, 2002. Lecture notes on Fundamentals of Information Technology.

[73] Robert E. Kraut and Lynn A. Streeter. "Coordination in software development". *Communications of the ACM*, 38 (3), Mar. 1995.

[74] Lassi Kurittu. *Johdatus logiikkaan*. Luentomoniste 47, Jyväskylän yliopisto, Matematiikan laitos, Jyväskylä, 2000. Lecture notes on logic.

[75] Gottfried Wilhelm Leibniz. *Logical papers*. Oxford: Clarendon Press, 1966.

[76] Richard C. Linger. "Cleanroom process model". *IEEE Software*, 11 (2), pp. 50–58, Mar. 1994.

[77] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, vol. 6 of *Fundamental Studies in Computer Science*. Amsterdam: North-Holland, 1978.

[78] George F. Luger and William A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Redwood: Benjamin/Cummings, 1993, 2nd ed.

[79] Simon Marlow. "Happy: The parser generator for Haskell". Accessed on the 19th of December, 2002.
URL http://www.haskell.org/happy/

[80] Bertrand Meyer. *Eiffel: the language*. New York: Prentice Hall, 1992.

[81] Doug Mink. "Mariner 1 from NASA reports". *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*, 5 (73), Dec. 1987.
URL http://catless.ncl.ac.uk/Risks/5.73.html#subj2

[82] Marty Moore. "Mariner I". *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*, 5 (73), Dec. 1987.
URL http://catless.ncl.ac.uk/Risks/5.73.html#subj2

[83] Carroll Morgan. "The specification statement". *ACM Transactions on Programming Languages and Systems*, 10 (3), pp. 403–419, Jul. 1988.

[84] National Aeronautics and Space Administration, Washington, DC. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*, Dec. 1998.
URL http://eis.jpl.nasa.gov/quality/Formal_Methods/

[85] P. Naur and B. Randell (eds.). *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968*. Brussels: NATO Scientific Affairs Division, 1969.
URL http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/

[86] Greg Nelson. "A generalization of Dijkstra's calculus". *ACM Transactions on Programming Languages and Systems*, 11 (4), pp. 517–561, Oct. 1989.

[87] John von Neumann. "An axiomatization of set theory". In van Heijenoort [55].

[88] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. San Francisco, CA: Morgan Kaufmann, 1998.

[89] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual for X11 Release 4*, vol. 4 of *The X Window System*. Sebastopol: O'Reilly, 1990, 2nd ed.

[90] Pekka Orponen. *Laskennan teoria, syksy 1997*. Luentomoniste. Jyväskylä: Jyväskylän yliopisto, matematiikan laitos, 1997.

[91] David Lorge Parnas. "Teaching programming as engineering". In Jonathan P. Bowen and Michael G. Hinchey (eds.), "ZUM '95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 1995 Proceedings", vol. 967 of *Lecture Notes in Computer Science*, (pp. 471–481). Springer, 1995.

[92] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis and Charles V. Weber. *Capability Maturity Model™ for Software, Version 1.1*. Tech. rep., Software Engineering Institute, Carnegie Mellon University, 1996.

[93] Giuseppe Peano. "The principles of arithmetic, presented by a new method". In van Heijenoort [55].

[94] Alan J. Perlis. "Epigrams on programming". *SIGPLAN Notices*, 17 (9), Sep. 1982.

[95] Simon Peyton Jones (ed.). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. To appear.
URL http://research.microsoft.com/Users/simonpj/haskell98-revised/

[96] Shari Lawrence Pfleeger and Les Hatton. "Investigating the influence of formal methods". *Computer*, 30 (2), pp. 33–43, Feb. 1997.

[97] Roger S. Pressman and Darrel Ince. *Software Engineering: A Practitioner's Approach*. London: McGraw-Hill, 2000, 5th ed.

[98] Hilary Putnam. "Nonstandard models and Kripke's proof of the Gödel theorem". *Notre Dame Journal of Formal Logic*, 41 (1), 2000.
URL http://projecteuclid.org/Dienst/UI/1.0/Summarize/euclid.ndjfl/1027953483

[99] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an accidental revolutionary*. O'Reilly, 2001, revised ed.

[100] Alan Robinson and Andrei Voronkov (eds.). *Handbook of Automated Reasoning*. Amsterdam: Elsevier (North-Holland), 2001.

[101] J. A. Robinson. "A machine-oriented logic based on the resolution principle". *Journal of the Association for Computing Machinery*, 12 (1), 1965.

[102] John Rushby. *Formal Methods and the Certification of Critical Systems*. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993. Also issued under the title "Formal Methods and Digital Systems Validation for Airborne Systems" as NASA Contractor Report 4551, December 1993.
URL http://www.csl.sri.com/papers/csl-93-7/

[103] Bertrand Russell. "Letter to Frege". In van Heijenoort [55].

[104] —. "Mathematical logic as based on the theory of types". In van Heijenoort [55].

[105] Hannele Salminen and Jouko Väänänen. *Johdatus logiikkaan*. Jyväskylä: Gaudeamus, 1997. An introduction to logic.

[106] Dirk Schlimm. "A short history of primitive recursion", May 1998.
URL http://www.phil.cmu.edu/dschlimm/texts/prim_rec.dvi

[107] Steve Schneider. *The B-Method: An Introduction*. Cornerstones of computing. Hampshire: Palgrave, 2001.

[108] Henry Maurice Sheffer. "A set of five independent postulates for Boolean algebras, with application to logical constants". *Transactions of the American Mathematical Society*, 14 (4), Oct. 1913.

[109] J. C. Shepherdson and H. E. Sturgis. "Computability of recursive functions". *Journal of the ACM*, 10 (2), Apr. 1963.

[110] Thoralf Skolem. "The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains". In van Heijenoort [55].

[111] —. "Some remarks on axiomatized set theory". In van Heijenoort [55].

[112] Richard M. Stallman. "Free software definition". In Joshua Gay (ed.), "Free Software, Free Society: Selected Essays of Richard M. Stallman", Boston: GNU, 2002.
URL http://www.gnu.org/philosophy/free-sw.html

[113] Steria Méditerranée. "Atelier-B", 2001.
URL http://www.atelierb.societe.com/index_uk.html

[114] Alfred Tarski. "The concept of truth in formalized languages". In "Logic, Semantics, Metamathematics: Papers from 1923 to 1938", Oxford: Oxford University Press, 1969.

[115] Ken Thompson. "Reflections on trusting trust". *Communications of the ACM*, 27 (8), 1984.

[116] Ian Toyn. "Cadiℤ home page". Accessed on the 30th of April, 2002.
URL http://www-users.cs.york.ac.uk/~ian/cadiz/

[117] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". *Proceedings of the London Mathematics Society, Series 2*, 42, 1936.

[118] Unicode Consortium. *The Unicode Standard, Version 3.0.0*. Reading, MA, MA: Addison-Wesley, 2000.
URL http://www.unicode.org/unicode/uni2book/u2.html

[119] —. *Unicode 3.2.0*. Unicode Standard Annex 28, Unicode Inc., Mar. 2002.
URL http://www.unicode.org/unicode/reports/tr28/tr28-3.html

[120] —. "The Unicode standard, version 3.2.0". Defined in [118], as amended by [30] and [119], 2002.

[121] Hans van Vliet. *Software Engineering: Principles and Practice*. Chichester: Wiley, 2000, 2nd ed.

[122] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1978. The work comprises three volumes, originally published in 1910–1913.

[123] "Windows 2000 supports Unicode version 2.0, knowledge base -artikkeli — 227483", oct 2002.
URL http://support.microsoft.com/default.aspx?scid=KB;en-us;q227483

[124] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge Classics. Routledge, 2002. Originally published in 1921, translated by D. F. Pears and B. F. McGuinness.

[125] "Web WordNet 1.7.1 search — overview for "premises"". Accessed on 21th of April, 2002.

URL http://www.cogsci.princeton.edu/cgi-bin/webwn1.7.1?stage=1&word=premises

[126] Larry Wos, Ross Overbeek, Ewing Lusk and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[127] Georg Henrik von Wright. "Logiikka, filosofia ja kieli: Ajattelijoita ja ajatussuunita nykyajan filosofiassa". In "Logiikka ja humanismi", [128].

[128] —. *Logiikka ja humanismi*. Helsinki: Otava, 1998.

[129] —. "Looginen empirismi". In "Logiikka ja humanismi", [128].

[130] Ernst Zermelo. "Investigations in the foundations of set theory I". In van Heijenoort [55].

A Two formal theories

> *Later generations will regard set theory as a disease from which one has recovered.*
>
> — *Henri Poincaré (according to rumours)*[1]

The purpose of this appendix is to build a fairly standard version of first-order logic and a ZFCish set theory upon that logic. Proofs are omitted for reasons of space.

The mechanisms of this appendix are rather standard. This particular way of building first-order logic and set theory is a synthesis by the present author based on many sources, such as [1, 48, 66, 74, 105]. A partial record of sources for particular definitions is given in the endnotes.

The set theory presented in this appendix deviates from the usual in that we grant classes a proper place in the formalism. Both choices have necessitated modifications to the ZFC axiom system. The resulting axiom system is similar but not identical to von Neumann, Bernays and Gödel's axiomatization (cf. eg. [11]).

We assume that we work in a metalogical world where we do have the machinery of mathematics such as sets, using an informal version of the notation that we are developing. This should be just fine as we are interested more in the mechanization of logic and set theory than in their phiolosophical and metamathematical aspects.

A.1 Preliminaries

The set of natural numbers includes zero.

We say that a natural number n *divides* a natural number m if $\exists\, \alpha \in \mathbb{N}\colon\ \alpha \leq m \wedge \alpha n = m$, and we denote this by $n \mid m$.

We call a natural number $n > 1$ *prime* (denoted by $\mathrm{prime}(n)$), if 1 and n are the only ones that divide it.

We define the function $\mathrm{Pr}\colon \mathbb{N} \to \mathbb{N}$ recursively as follows[2]:

$$\begin{aligned} \mathrm{Pr}(0) &= 0 \\ \mathrm{Pr}(n+1) &= \min\{\, y \in \mathbb{N}\colon\ \mathrm{prime}(y) \wedge y > \mathrm{Pr}(n)\,\} \end{aligned}$$

A pair is an ordered set. The pair of $\mathbf{a} \in \mathbf{A}$ and $\mathbf{b} \in \mathbf{B}$, in that order, is denoted by $(\mathbf{a}, \mathbf{b})$, which is a member of the cartesian product $\mathbf{A} \times \mathbf{B}$. There are projection functions pr_1 and pr_2 for which the following hold:

$$\mathrm{pr}_1(\mathbf{a}, \mathbf{b}) = \mathbf{a}$$
$$\mathrm{pr}_2(\mathbf{a}, \mathbf{b}) = \mathbf{b}$$

A.2 Alphabets, strings and substitutions

An *alphabet* Σ is a pair: the first element of the pair is a countably infinite set (denoted by $\mathrm{symbols}(\Sigma) = \mathrm{pr}_1(\Sigma)$), whose elements are called *symbols*, and the second element of the pair is a bijection $\mathrm{numbering}(\Sigma)\colon \mathrm{symbols}(\Sigma) \to \mathbb{N} \setminus \{0\}$ (note that $\mathrm{numbering}(\Sigma) = \mathrm{pr}_2(\Sigma)$).

Let Σ be an alphabet. We define the following set:

$$\Sigma^* = \{\, (\alpha, \beta)\colon \alpha = \Sigma \wedge \beta \in \mathbb{N} \wedge \exists \gamma \in \mathbb{N}\colon\ \forall \delta \in \mathbb{N}\colon \\ \mathrm{prime}(\delta) \to ((\delta < \gamma \to \delta \mid \beta) \wedge (\delta \geq \gamma \to \neg(\delta \mid \beta)))\,\}.$$

We will call elements of the set Σ^* *strings* of Σ. We will also define functions $\mathrm{alphabet} = \mathrm{pr}_1$ and $\mathrm{gödel} = \mathrm{pr}_2$. For $S \in \Sigma^*$ we call $\mathrm{gödel}(S)$ the *Gödel number* of the string S.[3]

We define a length function as $l = \lambda\, \alpha\colon\ \max\{\, \gamma \in \mathbb{N}\colon\ \mathrm{Pr}(\gamma) \mid \mathrm{gödel}(\alpha)\,\}$.

The $n+1$th symbol of a string S is denoted by $S(n)$ and is defined, when n is a natural number less than $l(S)$, as $\mathrm{numbering}(\mathrm{alphabet}(S))^{-1}(k)$, where k is the largest natural number such that $\mathrm{Pr}(n+1)^k \mid \mathrm{gödel}(S)$.

Let Σ be an alphabet and let $\varphi \in \mathrm{symbols}(\Sigma)$. We denote the string

$$\left(\Sigma, 2^{\mathrm{numbering}(\Sigma)(\varphi)}\right)$$

by $\langle\varphi\rangle_\Sigma$. Furthermore, we denote by $\langle\rangle_\Sigma$ the string $(\Sigma, 0)$. If the alphabet is clear from the context, then the subscript Σ may be omitted.

Let S and T be strings where $\mathrm{alphabet}(S) = \mathrm{alphabet}(T)$. We define a sequence of strings $C_0, \ldots, C_{l(T)}$:

$$C_0 = S$$
$$C_{i+1} = \left(\Sigma, \mathrm{gödel}(C_i)\,\mathrm{Pr}(l(C_i)+1)^{\mathrm{numbering}(\Sigma)(T(i))}\right) \quad \text{where } i \in \{0, \ldots, l(T)-1\}.$$

Now, $C_{l(T)}$ is the *concatenation* of S and T, denoted by ST.

If $\varphi_0, \ldots, \varphi_n$ are symbols of an alphabet Σ, we denote the string $\langle\varphi_0\rangle_\Sigma \cdots \langle\varphi_n\rangle_\Sigma$ by $\langle\varphi_0 \cdots \varphi_n\rangle_\Sigma$. If the alphabet is clear from the context, then the subscript Σ may be omitted.

If S, T, U and V are strings where $\text{alphabet}(S) = \text{alphabet}(T) = \text{alphabet}(U) = \text{alphabet}(V)$, and $S = TUV$, then U is a *substring* of S, and T is an *initial substring* or *prefix* of S, and V is a *final substring* or *postfix* of S.

Let Σ be an alphabet. A *(string) substitution* σ is a partial function $\sigma\colon \text{symbols}(\Sigma) \nrightarrow \Sigma^*$. Let now S be a string where $\text{alphabet}(S) = \Sigma$. We will define a sequence of strings $C_0, \ldots, C_{l(S)+1}$:

$$C_0 = \langle\rangle_\Sigma$$
$$C_{i+1} = \begin{cases} C_i\ \langle S(i)\rangle_\Sigma, & \text{if } \sigma \text{ is not defined for } S(i), \text{ and} \\ C_i\ (\sigma(S(i))), & \text{otherwise.} \end{cases}$$

Now, $C_{l(S)+1}$ is denoted by $S\sigma$ and called the *application* of σ to S.

Let Σ be an alphabet. A partial function $\sigma\colon \Sigma^* \nrightarrow \Sigma^*$ is a *generalized (string) substitution*, if no member of its domain is an initial substring of another. Let $S \in \Sigma^*$. We will define the application $S\sigma$ of a generalized string substitution with recursive rewrite rules:

$$(ST)\sigma :\Rightarrow \sigma(S)(T\sigma) \qquad \text{if } \sigma \text{ is defined for } S$$
$$(\langle\varphi\rangle S)\sigma :\Rightarrow \langle\varphi\rangle(S\sigma) \qquad \text{otherwise}$$

A.3 First-order logic

Usually, by the term first-order logic we mean a family of similar logics with a common structure. In this section, we develop that common structure. We call the different members of the family of first-order logics an *application* of first-order logic. We will only consider first-order logic with equality.

A.3.1 Syntax The set of symbols for first-order logic is a union of the set of logical constants, individual constants, predicate symbols, variables, metavariables and additional symbols. The sets of logical constants, individual constants, predicate symbols, variables, metavariables and additional symbols are disjoint. The sets of individual constants, predicate symbols, variables and metavariables are specified by the application; there will be a countably infinite number of both variables

Character	Code position	Name
(	U+0028	LEFT PARENTHESIS
)	U+0029	RIGHT PARENTHESIS
$\neg$	U+00AC	NOT SIGN
$\rightarrow$	U+2192	RIGHTWARDS ARROW
$\forall$	U+2200	FOR ALL
,	U+002C	COMMA
$=$	U+003D	EQUALS SIGN

Table A.1: Logical constants

and metavariables. We will write variables as lowercase greek letters and typeset metavariables in boldface.

The set of logical constants consists of the Unicode characters specified in table A.1. The set of additional symbols is the complement of the union of the other sets.

Throughout this section we assume the following:

1. The alphabet Σ consisting of the symbols listed above and a particular numbering function are given.

2. No substitution is defined for any symbols other than the metavariables.

A *formula schema* of first-order logic is any string. Formula schemata are denoted by uppercase calligraphic letters, such as $\mathcal{F}$. A *formula* of first-order logic is a formula schema which contains no metavariables; a formula schema that is not a formula is a *proper formula schema*.

A formula schema $\mathcal{F}$ is said to *generate* another formula $\mathcal{F}'$, if there is a substitution σ such that $\mathcal{F}\sigma = \mathcal{F}'$. If we say that a metavariable *stands for*, *denotes* or *is* a certain kind of a formula, then only a formula of that kind may be mapped to that metavariable in the substitution. If we say that a metavariable *stands for*, *denotes* or *is* a certain kind of a symbol, then only a formula which consists solely of that kind of a symbol may be mapped to that metavariable in the substitution.

A formula is a *term* (also known as an *expression* , iff

- it consists solely of one individual constant or variable, or

- it is generated by the schema $\langle \mathbf{f}(\mathbf{a}) \rangle$, where **f** is an individual constant and **a** is a comma-separated list of terms.

A formula is a *comma-separated list of terms*, iff

- it is a term (in which case the length of the list is 1), or
- it is generated by the schema $\langle \mathbf{l}, \mathbf{t} \rangle$, where **l** is a comma-separated list of terms and **t** is a term (in which case the length of the list is $n+1$, where n is the length of **l**).

A formula schema is a *term schema* if all formulae it generates are terms.

A formula is *well-formed*, iff

- it is generated by the schema $\langle \mathbf{P}(\mathbf{a}) \rangle$, where **P** is a predicate symbol and **a** is a comma-separated list of terms,
- it is generated by the schema $\langle (\mathbf{aPb}) \rangle$, where **a** and **b** are terms and **P** is a predicate symbol,
- it is generated by the schema $\langle (\mathbf{a} = \mathbf{b}) \rangle$, where **a** and **b** are terms,
- it is generated by the schema $\langle (\neg \mathbf{P}) \rangle$, where **P** is a well-formed formula,
- it is generated by the schema $\langle (\mathbf{P} \rightarrow \mathbf{Q}) \rangle$, where **P** and **Q** are well-formed formulae, or
- it is generated by the schema $\langle (\forall \boldsymbol{\alpha} \colon \mathbf{P}) \rangle$, where $\boldsymbol{\alpha}$ denotes a variable symbol and **P** denotes a well-formed formula.

A formula schema is well-formed if all formulae it generates are well-formed.

Parentheses may be omitted from formulae if their recursive structure is clear even without the parentheses. Generally, $\neg$ binds stronger than $\rightarrow$. The subformula of a $\forall$-formula extends as far to the right as is possible.

A variable is *free* in a term, if

- the term consists solely of that variable, or
- the term is generated by the schema $\langle \mathbf{f}(\mathbf{a}) \rangle$, where **f** is an individual constant and **a** is a comma-separated list of terms, and the variable is free in **a**.

A variable is *free* in a comma-separated list of terms, if

- the comma-separated list of terms has a length of 1 and the variable is free in its only constituent term, or
- the comma-separated list is generated by the schema $\langle \mathbf{l}, \mathbf{t} \rangle$, where $\mathbf{l}$ is a comma-separated list of terms and $\mathbf{t}$ is a term, and the variable is free in $\mathbf{l}$ or $\mathbf{t}$ or both.

A variable is *free* in a well-formed formula, if

- the formula is generated by the schema $\langle \mathbf{P}(\mathbf{a}) \rangle$, where $\mathbf{P}$ is a predicate symbol and $\mathbf{a}$ is comma-separated list of terms, and the variable is free in $\mathbf{a}$,
- the formula is generated by the schema $\langle (\mathbf{aPb}) \rangle$, where $\mathbf{a}$ and $\mathbf{b}$ are terms and $\mathbf{P}$ is a predicate symbol, and the variable is free in $\mathbf{a}$ or in $\mathbf{b}$ or in both,
- the formula is generated by the schema $\langle (\mathbf{a} = \mathbf{b}) \rangle$, where $\mathbf{a}$ and $\mathbf{b}$ are terms, and the variable is free in $\mathbf{a}$ or in $\mathbf{b}$ or in both,
- the formula is generated by the schema $\langle (\neg \mathbf{P}) \rangle$, where $\mathbf{P}$ is a well-formed formula, and the variable is free in $\mathbf{P}$,
- the formula is generated by the schema $\langle (\mathbf{P} \to \mathbf{Q}) \rangle$, where $\mathbf{P}$ and $\mathbf{Q}$ are well-formed formulae, and the variable is free in $\mathbf{P}$ or in $\mathbf{Q}$ or in both, or
- the formula is generated by the schema $\langle (\forall \boldsymbol{\alpha} \colon \mathbf{P}) \rangle$, where $\boldsymbol{\alpha}$ denotes a variable other than the variable being considered and $\mathbf{P}$ denotes a well-formed formula where the variable being considered is free.

Otherwise, a variable is not free in a term or a well-formed formula.

Occasionally we will introduce *shorthand*. A shorthand introduction consists of a *shorthand* $\mathcal{S}$, which is a formula schema and of a *definition* $\mathcal{D}$, which is either a term schema or a well-formed formula schema. If the definition is a term schema, then the formulae generated by the shorthand are terms. If the definition is a well-formed formula schema, then the formulae generated by the shorthand are well-formed. Sometimes we will introduce several shorthands by the means of a two-column table; then each row in the table is a shorthand introduction, the left-hand column contains the shorthands and the right-hand column contains the definitions.

We will now introduce shorthand notations according to Table A.2.

Shorthand	Definition
$\langle(\exists\,\boldsymbol{\alpha}\colon \mathbf{P})\rangle$	$\langle(\neg\,\forall\,\boldsymbol{\alpha}\colon \neg\mathbf{P})\rangle$
$\langle(\mathbf{P} \vee \mathbf{Q})\rangle$	$\langle(\neg\mathbf{P} \to \mathbf{Q})\rangle$
$\langle(\mathbf{P} \wedge \mathbf{Q})\rangle$	$\langle(\neg(\neg\mathbf{P} \vee \neg\mathbf{Q}))\rangle$
$\langle(\mathbf{P} \leftrightarrow \mathbf{Q})\rangle$	$\langle((\mathbf{P} \to \mathbf{Q}) \wedge (\mathbf{Q} \to \mathbf{P}))\rangle$
$\langle(\boldsymbol{\alpha}[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\mathbf{t})\rangle$
$\langle(\boldsymbol{\beta}[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\boldsymbol{\beta})\rangle$
$\langle(\mathbf{c}[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\mathbf{c})\rangle$
$\langle((\mathbf{c}(\mathbf{l}))[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle((\mathbf{c})((\mathbf{l})[\boldsymbol{\alpha}/\mathbf{t}]))\rangle$
$\langle((\mathbf{l},\mathbf{u})[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle((\mathbf{l}[\boldsymbol{\alpha}/\mathbf{t}]),\mathbf{u}[\boldsymbol{\alpha}/\mathbf{t}])\rangle$
$\langle((\mathbf{p}(\mathbf{l}))[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\mathbf{p}(\mathbf{l}[\boldsymbol{\alpha}/\mathbf{t}]))\rangle$
$\langle((\mathbf{upv})[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle((\mathbf{u}[\boldsymbol{\alpha}/\mathbf{t}])\mathbf{p}(\mathbf{v}[\boldsymbol{\alpha}/\mathbf{t}]))\rangle$
$\langle((\neg\mathbf{P})[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\neg(\mathbf{P}[\boldsymbol{\alpha}/\mathbf{t}]))\rangle$
$\langle((\mathbf{P} \to \mathbf{Q})[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle((\mathbf{P}[\boldsymbol{\alpha}/\mathbf{t}]) \to (\mathbf{Q}[\boldsymbol{\alpha}/\mathbf{t}]))\rangle$
$\langle((\forall\,\boldsymbol{\alpha}\colon \mathbf{P})[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\forall\,\boldsymbol{\alpha}\colon \mathbf{P})\rangle$
$\langle((\forall\,\boldsymbol{\gamma}\colon \mathbf{P})[\boldsymbol{\alpha}/\mathbf{t}])\rangle$	$\langle(\forall\,\boldsymbol{\gamma}\colon (\mathbf{P}[\boldsymbol{\alpha}/\mathbf{t}]))\rangle$

Here $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are distinct variables, $\mathbf{P}$ and $\mathbf{Q}$ are well-defined formulae, $\mathbf{t}$, $\mathbf{u}$ and $\mathbf{v}$ are terms, $\mathbf{p}$ is a predicate symbol, $\mathbf{l}$ is a comma-separated list of terms and $\mathbf{c}$ is an individual constant, with the additional restriction that $\boldsymbol{\gamma}$ must not be free in $\mathbf{t}$.

Table A.2: Shorthand definitions for first-order logic.

A.3.2 Truth A *model* of first-order logic is a tuple (C, P, c, p), where C is a sequence of sets $C_0, C_1, \ldots$, and C_0 is a nonempty set, and each C_n for $n > 0$ is a set of functions $(C_0)^n \to C_0$, P is a sequence of sets $P_1, P_2, \ldots$, and P_n is a set of functions $(C_0)^n \to \{0, 1\}$, and c is a sequence of functions $c_0, c_1, \ldots$, where

c_i: the set of individual constants $\to C_i$

for all $i \in \mathbb{N}$, and p is a sequence of functions $p_1, p_2, \ldots$, where

p_i: the set of predicate symbols $\to P_i$

for all $i \in \mathbb{N} \setminus \{0\}$.

We define the interpretation of terms under a model (C, P, c, p) as follows:

- The interpretation of a term consisting solely of one individual constant a is $c_0(a)$.
- The interpretation of a term generated by the schema $\langle \mathbf{f}(\mathbf{a}) \rangle$, where $\mathbf{f}$ is a term and $\mathbf{a}$ is a formula consisting of one or more terms separated by commas, is $c_n(\mathbf{f})(\mathbf{a}')$, where $\mathbf{a}'$ consists of the interpretations of the terms in $\mathbf{a}$ separated by commas, and n is the number of terms in $\mathbf{a}$.
- The interpretation of a term generated by a shorthand is the interpretation of the term generated by the definition of the shorthand under the same string substitution.

We then define the *truth value* of well-defined formulae under the same model as follows[4]:

- The truth value of a formula generated by the schema $\langle \mathbf{P}(\mathbf{a}) \rangle$, where $\mathbf{P}$ is a predicate symbol and $\mathbf{a}$ is a formula consisting of one or more terms separated by commas, is true, if $p_n(\mathbf{P})(\mathbf{a}') = 1$ and false otherwise, where $\mathbf{a}'$ consists of the interpretations of the terms in $\mathbf{a}$ separated by commas, and n is the number of terms in $\mathbf{a}$.
- The truth value of a formula generated by the schema $\langle (\mathbf{aPb}) \rangle$, where $\mathbf{a}$ and $\mathbf{b}$ are terms and $\mathbf{P}$ is a predicate symbol, is true, if $p_2(\mathbf{P})(c(\mathbf{a}), c(\mathbf{b})) = 1$ and false otherwise.
- The truth value of a formula generated by the schema $\langle (\mathbf{a} = \mathbf{b}) \rangle$, where $\mathbf{a}$ and $\mathbf{b}$ are terms, is true if $c(\mathbf{a}) = c(\mathbf{b})$ and false otherwise.

- The truth value of a formula generated by the schema $\langle(\neg \mathbf{P})\rangle$, where $\mathbf{P}$ is a well-formed formula, is true if the truth value of $\mathbf{P}$ is false, and false otherwise.
- The truth value of a formula generated by the schema $\langle(\mathbf{P} \rightarrow \mathbf{Q})\rangle$, where $\mathbf{P}$ and $\mathbf{Q}$ are well-formed formulae, is true if the truth value of $\mathbf{P}$ is false and when the truth value of both $\mathbf{P}$ and $\mathbf{Q}$ is true, and false otherwise.
- The truth value of a formula generated by the schema $\langle(\forall \boldsymbol{\alpha}\colon \mathbf{P})\rangle$ under the substitution σ, where $\boldsymbol{\alpha}$ denotes a variable and $\mathbf{P}$ denotes a well-formed formula, is true if for every individual symbol s the truth value of the formula generated by $\langle \mathbf{P}[\boldsymbol{\alpha}/s]\rangle$ is true, and false otherwise.
- The truth value of a well-formed formula generated by a shorthand is the truth value of the well-formed formula generated by the definition of the shorthand under the same string substitution.

If the truth value of a well-formed formula is true or false, then we say that the well-formed formula is true or false, respectively.

A well-defined formula $\mathcal{F}$ is a *tautology*, denoted by $\vDash \mathcal{F}$, if it is true under all models of first-order logic. A well-defined formula is a *contradiction* if it is false under all models of first-order logic. A well-defined formula is *contingent* if it is neither a tautology nor a contradiction.

A well-defined formula generated by $\langle \mathbf{F}\rangle$ is a *(semantic) consequence* of the well-defined formulae generated by $\langle \mathbf{F}\rangle_i$ (where i is a member of a given nonempty set I), denoted by $\mathcal{F} \vDash \langle \mathbf{F}\rangle$, where $\mathcal{F} = \{\, \alpha\colon\ \exists\, \beta \in I\colon \alpha = \langle \mathbf{F}\rangle_i \,\}$, if in every model where every formula generated by $\langle \mathbf{F}\rangle_i$ is true, also the formula generated by $\langle \mathbf{F}\rangle$ is true.

A.3.3 Inference The *axiom schemata* of first-order logic are the following well-formed formula schemata, where $\mathbf{P}$, $\mathbf{Q}$ and $\mathbf{R}$ are well-formed formulae, $\mathbf{a}$ is an individual constant or a variable, $\boldsymbol{\alpha}$ is a variable, $\boldsymbol{\beta}$ is a variable which is not free in $\mathbf{P}$, and $\mathbf{t}$ and $\mathbf{u}$ are terms.[5]:

$$\langle \mathbf{P} \rightarrow (\mathbf{Q} \rightarrow \mathbf{P})\rangle \tag{Ax1}$$

$$\langle (\mathbf{P} \rightarrow (\mathbf{Q} \rightarrow \mathbf{R})) \rightarrow ((\mathbf{P} \rightarrow \mathbf{Q}) \rightarrow (\mathbf{P} \rightarrow \mathbf{R}))\rangle \tag{Ax2}$$

$$\langle \neg\neg \mathbf{P} \rightarrow \mathbf{P}\rangle \tag{Ax3}$$

$$\langle (\forall \boldsymbol{\alpha}\colon \mathbf{P}) \rightarrow (\mathbf{P}[\boldsymbol{\alpha}/\mathbf{a}])\rangle \tag{Ax4}$$

$$\langle (\forall \boldsymbol{\beta}\colon \mathbf{P} \rightarrow \mathbf{Q}) \rightarrow (\mathbf{P} \rightarrow \forall \boldsymbol{\beta}\colon \mathbf{Q})\rangle \tag{Ax5}$$

$$\langle (\mathbf{t} = \mathbf{u} \wedge \mathbf{P}[\boldsymbol{\alpha}/\mathbf{t}]) \rightarrow \mathbf{P}[\boldsymbol{\alpha}/\mathbf{u}] \rangle \tag{Ax6}$$

$$\langle \mathbf{e} = \mathbf{e} \rangle \tag{Ax7}$$

Additionally, for every shorthand $\langle \mathbf{P} \rangle$ introduced into the logic with the well-formed formula schema $\langle \mathbf{Q} \rangle$ as the definition, the formula schema $\langle \mathbf{P} \leftrightarrow \mathbf{Q} \rangle$ is an axiom schema of first-order logic, and for every shorthand $\langle \mathbf{t} \rangle$ introduced into the logic with the term schema $\langle \mathbf{u} \rangle$ as the definition, the formula schema $\langle \mathbf{t} = \mathbf{u} \rangle$ is an axiom schema of first-oder logic. Every well-formed formula generated by the axiom schemata are *axioms* of first-order logic.

A well-formed formula $\mathcal{F}$ of first-order logic is a *(syntactic) consequence* of the well-formed formulae $\mathcal{A}_1, \ldots, \mathcal{A}_n$, denoted by $\mathcal{A}_1, \ldots, \mathcal{A}_n \vdash \mathcal{F}$, iff

- it is an axiom of first-order logic,
- it is $\mathcal{A}_i$ for some $i \in \{1, \ldots, n\}$,
- it is generated by the schema $\langle \mathbf{Q} \rangle$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n \vdash \langle \mathbf{P} \rangle$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n \vdash \langle \mathbf{P} \rightarrow \mathbf{Q} \rangle$, where $\mathbf{P}$ and $\mathbf{Q}$ are well-formed formulae, or
- it is generated by the schema $\langle \forall\, \boldsymbol{\alpha}\colon \mathbf{P} \rangle$, where $\boldsymbol{\alpha}$ is a variable and $\mathcal{A}_1, \ldots, \mathcal{A}_n \vdash \mathbf{P}$.

If $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are all axioms (or $n = 0$), then $\mathcal{F}$ is a *theorem*, denoted by $\vdash \mathcal{F}$.

It can be shown that every theorem is a tautology. Gödel proved the converse in 1930: every tautology is a theorem.

A.4 Set theory

Let $\in$ stand for a predicate symbol and let Ω stand for an individual constant in first-order logic. Then, set theory is an application of first-order logic where there are no individual constants apart from Ω, no function symbols, and no predicate symbols apart from $\in$, modified suitably as described in this section.

A formula of set theory is a *term*, iff

- it is a term of first-order logic, or
- it is generated by the schema $\langle \{\, \boldsymbol{\alpha}\colon \mathbf{P} \,\} \rangle$, where $\boldsymbol{\alpha}$ is a variable and $\mathbf{P}$ is a well-formed formula of set theory.

A variable is *free* in a term of set theory, iff

- it is free in the term by the rules of first-order logic, or
- the term is generated by the schema $\langle\{\, \boldsymbol{\alpha}\colon \mathbf{P}\,\}\rangle$, where $\boldsymbol{\alpha}$ is a variable distinct from the variable being considered and $\mathbf{P}$ is a well-formed formula of set theory, and the variable being considered is free in $\mathbf{P}$.

We introduce more shorthands according to Table A.3[6], and the axiom schemata described below[7].

A.4.1 Axiom of the universal class The first axiom of set theory asserts that there is a $\subset$-maximal class, the *universal class*, which we denote $\langle\Omega\rangle$.

$$\langle\forall\,\tau\colon \tau \subset \Omega\rangle \tag{Ax8}$$

We call the elements of the universal class *sets*, and we call classes that are not sets *proper classes*. By definition and by this axiom, there can be no class of proper classes.

A.4.2 Axiom of extensionality The second axiom states that classes are extensional, in other words that classes are determined by their elements.

$$\langle\forall\,\tau\colon\ \forall\,\upsilon\colon (\forall\,\alpha\colon \alpha \in \tau \leftrightarrow \alpha \in \upsilon) \rightarrow \tau = \upsilon\rangle \tag{Ax9}$$

The converse, $\langle\forall\,\tau\colon\ \forall\,\upsilon\colon \tau = \upsilon \rightarrow (\forall\,\alpha\colon \alpha \in \tau \leftrightarrow \alpha \in \upsilon)\rangle$, is a theorem.

A.4.3 Axiom schema of class comprehension The third axiom schema gives meaning to the class comprehension notation. It states that for each well-formed formula there is a class that consists of exactly those sets for which the well-formed formula is true.

$$\langle\forall\,\tau\colon (\tau \in \Omega \mathbin{\wedge} \mathbf{P}[\alpha/\tau]) \leftrightarrow t \in \{\, \alpha\colon \mathbf{P}\,\}\rangle \tag{Ax10}$$

Here $\mathbf{P}$ is a well-formed formula.

Note that this axiom does not lead to the Russell paradox, since it does not allow a proper class to be a member of a class.

Note that by the three axiom schemata described so far, $\langle\Omega = \{\, \alpha\colon \alpha = \alpha\,\}\rangle$ holds.

Shorthand	Definition
$\langle \mathbf{t} \subset \mathbf{u} \rangle$	$\langle \forall\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \in \mathbf{t} \to \boldsymbol{\alpha} \in \mathbf{u} \rangle$
$\langle \varnothing \rangle$	$\langle \{\, \alpha\colon\, \alpha \neq \alpha \,\} \rangle$
$\langle \bigcup \mathbf{t} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \exists\, \boldsymbol{\beta}\colon\, \boldsymbol{\beta} \in \mathbf{t} \wedge \boldsymbol{\alpha} \in \boldsymbol{\beta} \,\} \rangle$
$\langle \bigcap \mathbf{t} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \forall\, \boldsymbol{\beta}\colon\, \boldsymbol{\beta} \in \mathbf{t} \to \boldsymbol{\alpha} \in \boldsymbol{\beta} \,\} \rangle$
$\langle \{\mathbf{t}\} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} = \mathbf{t} \,\} \rangle$
$\langle \{\mathbf{t}, \mathbf{u}\} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} = \mathbf{t} \vee \boldsymbol{\alpha} = \mathbf{u} \,\} \rangle$
$\langle \mathbf{t} \cup \mathbf{u} \rangle$	$\langle \bigcup \{\mathbf{t}, \mathbf{u}\} \rangle$
$\langle \{\mathbf{l}, \mathbf{u}\} \rangle$	$\langle \{\mathbf{l}\} \cup \{\mathbf{u}\} \rangle$
$\langle \mathbf{t} \cap \mathbf{u} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \in \mathbf{t} \wedge \boldsymbol{\alpha} \in \mathbf{u} \,\} \rangle$
$\langle \mathbf{t} \setminus \mathbf{u} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \in \mathbf{t} \wedge \neg \boldsymbol{\alpha} \in \mathbf{u} \,\} \rangle$
$\langle \{\, \boldsymbol{\alpha} \in \mathbf{t}\colon\, \mathbf{P} \,\} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \in \mathbf{t} \wedge \mathbf{P} \,\} \rangle$
$\langle \forall\, \boldsymbol{\alpha} \in \mathbf{t}\colon\, \mathbf{P} \rangle$	$\langle \forall\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \in \mathbf{t} \to \mathbf{P} \rangle$
$\langle \exists\, \boldsymbol{\alpha} \in \mathbf{t}\colon\, \mathbf{P} \rangle$	$\langle \exists\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \in \mathbf{t} \wedge \mathbf{P} \rangle$
$\langle \{\, (\boldsymbol{\alpha}, \boldsymbol{\beta})\colon\, \mathbf{P} \,\} \rangle$	$\langle \{\, \boldsymbol{\delta}\colon\, \exists\, \boldsymbol{\alpha}\colon\, \exists\, \boldsymbol{\beta}\colon\, \boldsymbol{\delta} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \wedge \mathbf{P} \,\} \rangle$
$\langle \{\, ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma})\colon\, \mathbf{P} \,\} \rangle$	$\langle \{\, \boldsymbol{\delta}\colon\, \exists\, \boldsymbol{\alpha}\colon\, \exists\, \boldsymbol{\beta}\colon\, \exists\, \boldsymbol{\gamma}\colon\, \boldsymbol{\delta} = ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma}) \wedge \mathbf{P} \,\} \rangle$
$\langle \exists (\boldsymbol{\alpha}, \boldsymbol{\beta})\colon\, \mathbf{P} \rangle$	$\langle \exists\, \boldsymbol{\delta}\colon\, \boldsymbol{\delta} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \wedge \mathbf{P} \rangle$
$\langle \forall (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathbf{t}\colon\, \mathbf{P} \rangle$	$\langle \exists\, \boldsymbol{\delta} \in \mathbf{t}\colon\, \boldsymbol{\delta} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \wedge \mathbf{P} \rangle$
$\langle 2^{\mathbf{t}} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \boldsymbol{\alpha} \subset \mathbf{t} \,\} \rangle$
$\langle (\mathbf{t}, \mathbf{u}) \rangle$	$\langle \{\{\mathbf{t}\}, \{\mathbf{t}, \mathbf{u}\}\} \rangle$
$\langle (\mathbf{l}, \mathbf{u}) \rangle$	$\langle \{(\mathbf{l}), \{(\mathbf{l}), \mathbf{t}\}\} \rangle$
$\langle \mathbf{t} \times \mathbf{u} \rangle$	$\langle \{\, \boldsymbol{\alpha}\colon\, \exists\, \boldsymbol{\beta} \in \mathbf{t}\colon\, \exists\, \boldsymbol{\gamma} \in \mathbf{u}\colon\, \boldsymbol{\alpha} = (\boldsymbol{\beta}, \boldsymbol{\gamma}) \,\} \rangle$

Here $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ are distinct variables, $\mathbf{P}$ is a well-defined formula where $\boldsymbol{\delta}$ is not free, $\mathbf{t}$, $\mathbf{u}$ and $\mathbf{v}$ are terms where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ are not free, and $\mathbf{l}$ is a comma-separated list of terms.

Table A.3: Shorthand definitions for set theory*(cont.)*

Shorthand	Definition
$\langle \mathbf{tuv} \rangle$	$\langle (\mathbf{t}, \mathbf{v}) \in \mathbf{u} \rangle$
$\langle \mathbf{t}[\mathbf{u}] \rangle$	$\langle \boldsymbol{\alpha} \colon \exists\, \boldsymbol{\beta} \in \mathbf{t} \colon \exists\, \boldsymbol{\gamma} \in \mathbf{u} \colon \boldsymbol{\beta} = (\boldsymbol{\gamma}, \boldsymbol{\alpha}) \rangle$
$\langle \mathbf{t}(\mathbf{u}) \rangle$	$\langle \bigcup \mathbf{t}[\mathbf{u}] \rangle$
$\langle \Upsilon \rangle$	$\langle \{\, \alpha \colon \varnothing \in \alpha \wedge \forall\, \beta \in \alpha \colon \beta \cup \{\beta\} \in \alpha \,\} \rangle$
$\langle \lambda\, \boldsymbol{\alpha} \colon \mathbf{t} \rangle$	$\langle \{\, (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon \boldsymbol{\beta} = \mathbf{t} \,\} \rangle$
$\langle \lambda (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon \mathbf{t} \rangle$	$\langle \{\, ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma}) \colon \boldsymbol{\gamma} = \mathbf{t} \,\} \rangle$
$\langle \lambda\, \boldsymbol{\alpha} \in \mathbf{t} \colon \mathbf{u} \rangle$	$\langle \{\, (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon \boldsymbol{\alpha} \in \mathbf{t} \wedge \boldsymbol{\beta} = \mathbf{u} \,\} \rangle$
$\langle \lambda (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathbf{t} \colon \mathbf{u} \rangle$	$\langle \{\, ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma}) \colon (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathbf{t} \wedge \boldsymbol{\gamma} = \mathbf{u} \,\} \rangle$
$\langle \operatorname{dom} \mathbf{t} \rangle$	$\langle (\lambda (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon \boldsymbol{\alpha})[\mathbf{t}] \rangle$
$\langle \operatorname{ran} \mathbf{t} \rangle$	$\langle (\lambda (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon \boldsymbol{\beta})[\mathbf{t}] \rangle$
$\langle \iota \boldsymbol{\alpha} \colon \mathbf{P} \rangle$	$\langle \{\, (\boldsymbol{\delta}, \boldsymbol{\alpha}) \colon \boldsymbol{\delta} = \varnothing \wedge \mathbf{P} \,\}(\varnothing) \rangle$
$\langle \iota \boldsymbol{\alpha} \in \mathbf{t} \colon \mathbf{P} \rangle$	$\langle \iota \boldsymbol{\alpha} \colon \boldsymbol{\alpha} \in \mathbf{t} \wedge \mathbf{P} \rangle$
$\langle \iota (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon \mathbf{P} \rangle$	$\langle \iota \boldsymbol{\delta} \colon \boldsymbol{\delta} = (\boldsymbol{\alpha}, \boldsymbol{\beta}) \wedge \mathbf{P} \rangle$
$\langle \iota ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma}) \colon \mathbf{P} \rangle$	$\langle \iota \boldsymbol{\delta} \colon \boldsymbol{\delta} = ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma}) \wedge \mathbf{P} \rangle$
$\langle \operatorname{inj} \mathbf{t} \rangle$	$\langle \forall (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathbf{t} \colon \forall (\boldsymbol{\gamma}, \boldsymbol{\delta}) \in \mathbf{t} \colon \boldsymbol{\alpha} = \boldsymbol{\gamma} \leftrightarrow \boldsymbol{\beta} = \boldsymbol{\delta} \rangle$
$\langle \mathbf{t} \colon \mathbf{u} \to \mathbf{v} \rangle$	$\langle \mathbf{t} \in \mathbf{u} \times \mathbf{v} \wedge \operatorname{fun} \mathbf{t} \rangle$
$\langle \mathbf{t}^{-1} \rangle$	$\langle \{\, (\boldsymbol{\alpha}, \boldsymbol{\beta}) \colon (\boldsymbol{\beta}, \boldsymbol{\alpha}) \in \mathbf{t} \,\} \rangle$

Here $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ are distinct variables, $\mathbf{P}$ is a well-defined formula where $\boldsymbol{\delta}$ is not free, $\mathbf{t}$, $\mathbf{u}$ and $\mathbf{v}$ are terms where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ are not free, and $\mathbf{l}$ is a comma-separated list of terms.

Table A.3: *(cont.)* Shorthand definitions for set theory.

A.4.4 Axiom of separation The fourth axiom states that every subclass of a set is a set.

$$\langle \forall\, \tau\colon\ \forall\, \upsilon \in \Omega\colon \tau \cap \upsilon \in \Omega \rangle \tag{Ax11}$$

Note that this is not a proper axiom schema as in ZFC; this is because we have the concept of a class that we can use to our advantage.

A.4.5 Construction axioms The fifth axiom starts a series of construction axioms: they tell us how to "grow" sets. The first of these states that a pair class of sets is a set, the second states that the union of sets is a set, and the third states that the powerclass of a set is a set.

$$\langle \forall\, \tau\colon\ \forall\, \upsilon\colon (\tau \in \Omega \mathbin{\wedge} \upsilon \in \Omega) \to \{\tau, \upsilon\} \in \Omega \rangle \tag{Ax12}$$

$$\left\langle \forall\, \tau \in \Omega\colon \bigcup \tau \in \Omega \right\rangle \tag{Ax13}$$

$$\langle \forall\, \tau \in \Omega\colon 2^{\tau} \in \Omega \rangle \tag{Ax14}$$

A.4.6 Axiom of infinity An inductive set is one that contains the empty set, and for every element $\mathbf{t}$ it contains $\langle \{\, \alpha\colon\ \alpha = \{\varnothing, \mathbf{t}\}\,\} \rangle$ as an element. We denote the class of inductive sets with Υ. The axiom of infinity states that there is at least one inductive set.

$$\langle \exists\, \alpha\colon \alpha \in \Upsilon \rangle \tag{Ax15}$$

A.4.7 Axiom of replacement The axiom of replacement declares that the image of a set under a function (which may be a proper class) is a set.

$$\langle \forall\, \tau\colon (\forall\, \alpha\colon\ \exists\, \beta\colon \tau[\{\alpha\}] = \{\beta\}) \to (\forall\, \alpha \in \Omega\colon \tau[\alpha] \in \Omega) \rangle \tag{Ax16}$$

Note that again, this axiom is not a proper schema thanks to the fact that classes are proper objects in this axiomatization.

A.4.8 Axiom of regularity The axiom of regularity states that all nonempty sets have a $\in$-minimal member.

$$\langle \forall \tau \colon (\tau \in \Omega \wedge \exists \alpha \colon \alpha \in \tau) \to \exists \alpha \in \tau \colon \tau \cap \alpha = \varnothing \rangle \tag{Ax17}$$

A.4.9 Axiom of choice The axiom of choice allows us to postulate a function that selects one element from an arbitrary set.

$$\langle \forall \tau \colon (\forall \alpha \in \tau \colon \alpha \neq \varnothing) \to \exists \alpha \in \Omega \colon \forall \beta \in \tau \colon \exists \gamma \in \beta \colon \alpha[\{\beta\}] = \{\gamma\} \rangle \tag{Ax18}$$

A.4.10 Models of set theory A model of first-order logic with equality is a model of set theory if the axioms (Ax8)–(Ax18) are true in it.

A.4.11 Arithmetic We will define more shorthands according to Table A.4[8].

The following formulae are theorems:

$$\langle 0 \in \mathbb{N} \rangle \tag{A.1}$$
$$\langle \forall \alpha \in \mathbb{N} \colon \alpha + 1 \neq 0 \rangle \tag{A.2}$$
$$\langle \forall \alpha \in \mathbb{N} \colon \alpha + 1 \in \mathbb{N} \rangle \tag{A.3}$$
$$\langle \forall \alpha \in \mathbb{N} \colon \forall \beta \in \mathbb{N} \colon \alpha + 1 = \beta + 1 \to \alpha = \beta \rangle \tag{A.4}$$
$$\langle \forall \alpha \colon (0 \in \alpha \wedge \forall \beta \in \mathbb{N} \colon \beta \in \alpha \to \beta + 1 \in \alpha) \to \mathbb{N} \subset \alpha \rangle \tag{A.5}$$

These formulae are, incidentally, the Peano axioms for arithmetic, rephrased for this set theory. Thus, this set theory includes arithmetic.

Notes

[1]This quote is usually attributed to Henri Poincaré, but in fact he never seems to have said or written it. Gray [52] traced the history of this quote, and his reconstruction of the events has Pierport treating a badly translated summary of Poincaré's talk as a quotation.

[2]This definition is from Gödel [48].

Shorthand	Definition
$\langle \mathbb{N} \rangle$	$\langle \bigcap \Upsilon \rangle$
$\langle \#\mathbf{t} \rangle$	$\langle \iota\boldsymbol{\alpha} \in \mathbb{N}\colon\ \exists\, \boldsymbol{\beta}\colon\ \operatorname{dom} \boldsymbol{\beta} = \mathbf{t} \wedge \operatorname{ran} \boldsymbol{\beta} = \boldsymbol{\alpha} \wedge \operatorname{inj} \boldsymbol{\beta} \rangle$
$\langle 0 \rangle$	$\langle \varnothing \rangle$
$\langle 1 \rangle$	$\langle \{\varnothing\} \rangle$
$\langle \mathbf{t} \leq \mathbf{u} \rangle$	$\langle \mathbf{t} \subset \mathbf{u} \rangle$
$\langle \mathbf{t} \dot{-} \mathbf{u} \rangle$	$\langle \#(\mathbf{t} \setminus \mathbf{u}) \rangle$
$\langle \mathbf{t} + \mathbf{u} \rangle$	$\langle \iota\boldsymbol{\alpha}\colon \mathbf{t} = \boldsymbol{\alpha} \dot{-} \mathbf{u} \wedge \mathbf{u} \leq \boldsymbol{\alpha} \rangle$
$\langle \mathbf{t} \cdot \mathbf{u} \rangle$	$\langle \#(\mathbf{t} \times \mathbf{u}) \rangle$
$\langle \mathbf{t}^{\mathbf{n}} \rangle$	$\langle (\iota\boldsymbol{\alpha}\colon (\forall\, \boldsymbol{\beta} \in \boldsymbol{\alpha}\colon\ \exists\, \boldsymbol{\gamma}\colon\ \exists\, \boldsymbol{\delta} \in \mathbb{N} \setminus \{0\}\colon\ \exists\, \boldsymbol{\epsilon}\colon\ \boldsymbol{\beta} = ((\boldsymbol{\gamma}, \boldsymbol{\delta}), \boldsymbol{\varepsilon})) \wedge$ $\forall\, \boldsymbol{\beta}\colon \boldsymbol{\alpha}(\boldsymbol{\beta}, 1) = \boldsymbol{\alpha} \wedge \forall\, \boldsymbol{\gamma} \in \mathbb{N} \setminus \{0, 1\}\colon \boldsymbol{\alpha}(\boldsymbol{\beta}, \boldsymbol{\gamma}) = \boldsymbol{\beta} \times \boldsymbol{\alpha}(\boldsymbol{\beta}, \boldsymbol{\gamma} \dot{-} 1))(\mathbf{t}, \mathbf{n}) \rangle$
$\langle \mathbf{t} \uparrow \mathbf{n} \rangle$	$\langle (\iota\boldsymbol{\alpha}\colon (\forall\, \boldsymbol{\beta} \in \boldsymbol{\alpha}\colon\ \exists\, \boldsymbol{\gamma} \in \mathbb{N}\colon\ \exists\, \boldsymbol{\delta} \in \mathbb{N}\colon\ \exists\, \boldsymbol{\epsilon} \in \mathbb{N}\colon\ \boldsymbol{\beta} = ((\boldsymbol{\gamma}, \boldsymbol{\delta}), \boldsymbol{\varepsilon})) \wedge$ $\forall\, \boldsymbol{\beta}\colon \boldsymbol{\alpha}(\boldsymbol{\beta}, 0) = 1 \wedge \forall\, \boldsymbol{\gamma} \in \mathbb{N} \setminus \{0\}\colon \boldsymbol{\alpha}(\boldsymbol{\beta}, \boldsymbol{\gamma}) = \boldsymbol{\alpha}(\boldsymbol{\beta}, \boldsymbol{\gamma} \dot{-} 1) \cdot \boldsymbol{\beta})(\mathbf{t}, \mathbf{n}) \rangle$
$\langle \mathbf{n} \mid \mathbf{m} \rangle$	$\langle \exists\, \boldsymbol{\alpha} \in \mathbb{N}\colon\ \boldsymbol{\alpha} \leq \mathbf{m} \wedge \boldsymbol{\alpha}\mathbf{n} = \mathbf{m} \rangle$
$\langle \operatorname{pr}_{\mathbf{n}}\, \mathbf{t} \rangle$	$\langle (\iota\varphi\colon \varphi = \{\, ((\boldsymbol{\alpha}, \boldsymbol{\beta}), \boldsymbol{\gamma})\colon (\exists\, \boldsymbol{\delta}\colon\ \exists\, \boldsymbol{\varepsilon}\colon\ \boldsymbol{\beta} = (\boldsymbol{\delta}, \boldsymbol{\varepsilon}) \rightarrow$ $((\boldsymbol{\alpha} = 1 \rightarrow \boldsymbol{\gamma} = \boldsymbol{\delta}) \wedge (\boldsymbol{\alpha} \in \mathbb{N} \setminus \{0, 1\} \rightarrow \boldsymbol{\gamma} = \varphi(\boldsymbol{\alpha} - 1, \boldsymbol{\varepsilon})))) \wedge$ $(\neg(\exists\, \boldsymbol{\delta}\colon\ \exists\, \boldsymbol{\varepsilon}\colon\ \boldsymbol{\beta} = (\boldsymbol{\delta}, \boldsymbol{\varepsilon})) \rightarrow (\boldsymbol{\alpha} = 1 \wedge \boldsymbol{\gamma} = \boldsymbol{\beta}))\,\})(\mathbf{n}, \mathbf{t}) \rangle$
$\langle \operatorname{pr} \rangle$	$\langle \lambda\, \alpha\colon\ \lambda\, \beta\colon\ \operatorname{pr}_{\alpha}\, \beta \rangle$
$\min \mathbf{t}$	$\langle \iota\boldsymbol{\alpha} \in \mathbf{t}\colon\ \forall\, \boldsymbol{\beta} \in \mathbf{t}\colon \boldsymbol{\alpha} \leq \boldsymbol{\beta} \rangle$
$\langle \operatorname{st}_{\mathbf{t}}\, \mathbf{u} \rangle$	$\left\langle (\iota\boldsymbol{\alpha}\colon \boldsymbol{\alpha} = \{\, ((\boldsymbol{\beta}, \boldsymbol{\gamma}), \boldsymbol{\delta})\colon \boldsymbol{\beta} \in 2^{\mathbb{N}} \wedge \boldsymbol{\delta} = (\operatorname{pr}_{\min \boldsymbol{\beta}}\, \boldsymbol{\gamma}, \boldsymbol{\alpha}(\boldsymbol{\beta} \setminus \{\min \boldsymbol{\beta}\}, \boldsymbol{\gamma}))\,\})(\mathbf{t}, \mathbf{u}) \right\rangle$
$\langle \{\mathbf{m}, \ldots, \mathbf{n}\} \rangle$	$\langle \{\, \boldsymbol{\alpha} \in \mathbb{N}\colon\ \mathbf{m} \leq \boldsymbol{\alpha} \wedge \boldsymbol{\alpha} \leq \mathbf{n}\,\} \rangle$

Here $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, $\boldsymbol{\delta}$ and $\boldsymbol{\varepsilon}$ are distinct variables, $\mathbf{t}$, $\mathbf{u}$, $\mathbf{n}$ and $\mathbf{m}$ are terms where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, $\boldsymbol{\delta}$ and $\boldsymbol{\varepsilon}$ are not free, and $\mathbf{n}$ and $\mathbf{m}$ is such that $\mathbf{n} \in \mathbb{N}$ and $\mathbf{m} \in \mathbb{N}$ are theorems.

Table A.4: Shorthand definitions for arithmetic.

[3]I have not seen this definition of strings anywhere before, but the idea of encoding strings as prime factorizations of natural numbers is due to Gödel [48] in a slightly different context. The definitions of length and nth symbol of a string are also due to Gödel.

[4]This kind of a truth definition is due to Alfred Tarski [114].

[5]This axiom system except for the last two axioms and the following rules of inference are taken from Kurittu [74], but I don't know their origin. The last two axiom schemata are taken from [1]; I don't know their origin.

[6]The *definite descriptor* ι can be traced back to *Principia mathematica* [122]. The intended reading of "$\iota\boldsymbol{\alpha}$: **P**" is *the unique set* $\boldsymbol{\alpha}$ *for which* **P** *is true*. If $\boldsymbol{\alpha}$ is not uniquely determined by **P**, the descriptor does denote some class (not necessarily a set), but the class it denotes is usually inappropriate in the context. In Bernays [11] it would denote the empty set in that case, but there is no such guarantee by this definition.

[7]These axiom schemata are adapted from Jech [66], where they are credited as the Zermelo–Fraenkel axioms.

[8]The definitions of addition and multiplication are from Jech [66]; I don't know their origin.

B A summary of AbstractSyntax.hs

This appendix summarizes the AbstractSyntax module of Ebba H.

```
module AbstractSyntax
    where
```

We define five mutually recursive data types, four of which correspond to the nonterminals in the abstract syntax given in Chapter 5.

The declarations of the data types list alternatives, much like the corresponding EBNF specification. Each alternative is given a *constructor* (for example, LogicalNegation), which is usually the first identifier in the alternative (prefix notation). Occasionally an infix constructor can be used: such a constructor needs to be a symbol string starting with a colon or an identifier enclosed in backquotes. Each constructor is given signature, the parameters it takes, which are specified by giving their type. Thus, for example, LogicalAnd is a two-parameter constructor, taking two Predicates as parameters. A type name enclosed in brackets denotes a list whose elements are of that type.

```
data Predicate = Predicate `LogicalAnd` Predicate
               | Predicate `LogicalOr` Predicate
               | Predicate `LogicalImplication` Predicate
               | Predicate `LogicalEquivalence` Predicate
               | LogicalNegation Predicate
               | UniversalQuantification Variable Predicate
               | ExistentialQuantification Variable Predicate
               | Expression `SetMembership` Expression
               | Expression :== Expression
               | Subset Expression Expression
               | SubstitutedPredicate Substitution Predicate
               | MetavariablePredicate Integer [SideCondition]
               | PredicateApplication Name [Expression]
               | NoPredicate
               | IllFormedPredicate String
```

```
data Variable = SingletonVariable Name
              | EmptyVariable
              | Frame Substitution
              | PairVariable Variable Variable
              | MetavariableVariable Integer [SideCondition]

data Expression = IdentifierExpression Name
                | OperatorExpression Operator [Expression]
                | PairExpression Expression Expression
                | SubstitutedExpression Substitution Expression
                | Expression `CartesianProduct` Expression
                | SetComprehensionExpression Variable Predicate
                | MetavariableExpression Integer [SideCondition]
                | FunctionApplication Expression [Expression]
                | IllFormedExpression String

data Substitution = Variable := Expression
                  | [Integer] `PSubst` [Predicate]
                  | Skip
                  | PrecondSubst Predicate Substitution
                  | BoundedChoice Substitution Substitution
                  | GuardedSubst Predicate Substitution
                  | UnboundedChoice Variable Substitution
```

The last data type records side conditions on metavariables.

```
data SideCondition = Variable `NonfreeInE` Expression
                   | Variable `NonfreeInP` Predicate
                   | Substitute Variable Expression
                   | IsVariable Integer
```

C A description of TypeChecker.hs

This module implements typechecking of B set theory. Much more than the set theory is not supported at this time.

```
module TypeChecker (typecheck)
    where
```

Here we have the abstract syntax for types and type predicates.

```
data Type = TypeOf Expression
          | Supertype Expression
          | Type ‘Producttype‘ Type
          | Powertype Type
          | TypeId Name
data TypePredicate = Name ‘InType‘ Expression
                   | GivenSet Name
```

The typechecking environment is a list of TypePredicates, although it may not be the most efficient structure with large inputs. We have two functions for checking whether some type predicate can be found in the environment.

```
type Environment = [TypePredicate]
```

lookupEnv :: Environment → Name → Maybe Expression

isGiven :: Environment → Name → Bool

The process of typechecking is written in monadic style. The typechecking monad is the type TypecheckM, which stores notices (warnings and error messages) gathered during the typechecking.

A monad is an abstract data type that models *actions*. A value of a monad type is a *potential* action; in other words, it *denotes* an action. These potential actions can be composed using

- sequencing (>>), which has the same role in monadic programming as the semicolon has in many imperative programming languages; and

- data-connected sequencing (>>=), which is essentially a way to sequence two actions while also letting the earlier action communicate a value to the latter action.

In every monad there are also two ways to construct simple actions:

- The (overloaded) function *return* takes a value and generates an action that, if it is performed, passes the parameter value to the next action and does nothing else.
- The (overloaded) function *fail* takes a string parameter and generates an action that, if it is performed, fails. The parameter may be propagated to whoever catches the exception caused by the failure.

Additionally, most monads have some other ways to construct primitive actions, and a way to cause an action to be performed.

The typechecking monad is mainly used to thread the notices through the typechecking process and to make it convenient (via the failure mechanism) to communicate typing errors.

```
newtype TypecheckM α = TypecheckM ([String] → (Maybe α, [String]))
instance Monad TypecheckM
    where
```

We have auxiliary functions to produce notices. The function *rule* takes a formula, a description of the rule to be applied, and produces an action that, if it is performed, adds a notice containing the formula and the rule description to the (implicit) notices list. The function *erule* does a similar thing but it is designed to be used in rules that test equality of types.

```
rule :: Show f ⇒ f → String → TypecheckM ()
```

```
erule :: Bool → Type → Type → String → TypecheckM ()
```

The *eitherM* combinator function takes two potential action and returns a potential action that, if it is ever performed, performs the first parameter action, and if that fails, it performs the second parameter action, too. Thus it runs one or the other if one of them succeeds, and fails only if both fail.

```
eitherM :: TypecheckM α → TypecheckM α → TypecheckM α
eitherM (TypecheckM m) (TypecheckM m') = TypecheckM $ λ s →
                                          let ms@(mr, _) = m s
                                          in if isJust mr
                                                then ms
                                                else m' s
```

The function *typecheck* starts the typechecking process by creating the potential action of typechecking and then forcing its performance.

```
typecheck :: Predicate → (Bool, [String])
typecheck p = (ok, reverse l)
    where (TypecheckM m) = check [GivenSet $ inject "BIG"] p'
          (ma, l) = m [ ]
          ok = isJust ma
          p' = rewrite p
```

The following function is one of the two main typechecking functions. This one operates on a check(**p**) type judgment, and operates by cases based on the form of the predicate. We will show here only some cases, ones that we deem interesting.

```
check :: Environment → Predicate → TypecheckM ()
```

The following case is straighforward: we just check both component predicates.

The **do** notation is syntactic sugar for the monadic sequencing operators, designed to look familiar to imperative programmers.

```
check env f@(p `LogicalAnd` q) = do rule f "T 1"
                                    check env p
                                    check env q
```

The next case implements the enhanced typing suggested at the end Subsection 5.3.4. The function *typeQ*, defined below, finds types for the variables x and returns a type environment containing these types. We then check the component predicate with this new environment.

```
check env f@(UniversalQuantification x p) = do rule f "T 4 / T 5"
                                               (_, env') ← typeQ env f x p
                                               check env' p
```

The following case demonstrates the handling of relations between expressions. We merely translate them into appropriate relations between the appropriate types.

```
check env f@(e :== e') = do rule f "T 7"
```

The *eq* function is the entry point to testing equality of types. It tries the same rules in both orders of the parameters, since equality is a symmetric relation.

```
eq :: Environment → Type → Type → TypecheckM ()
eq env e f = eitherM (eqv env False e f) (eqv env True f e)
```

The *eqv* function does the real work in checking equality of types. As before, we show only some interesting cases.

```
eqv :: Environment → Bool → Type → Type → TypecheckM ()
```

The first case handles the case of type rule T 9. We lookup the type of the identifier in the environment. If we succeed, we pass it on. If we fail, we fail.

```
eqv env retro t@(TypeOf (IdentifierExpression x)) u = erule retro t u "T 9" >>
                                                      case lookupEnv env x of
                                                          Just s → eq env (Supertype s) u
                                                          Nothing → fail "Type error"
```

The final case checks the equality of two type names. If they are lexically equivalent, we check if they denote a given set. If not, we fail, otherwise we succeed immediately. If they are not lexically equivalent, we fail.

```
eqv env retro t@(TypeId i) u@(TypeId i') = do erule retro t u "T 21"
                                              if i == i'
                                                then if isGiven env i
                                                       then return ()
                                                       else fail "Unknown set"
                                                else fail $ "Types do not match"
```

The penultimate functions implement the type finding procedure discussed at the end of Subsection 5.3.4.

```
findType :: Environment → Name → Predicate → Maybe Expression
findType env x (LogicalAnd p q)                    = case findType env x p of
                                                         Just e → Just e
                                                         _ → findType env x q
findType env x p@(LogicalOr _ _)                   = findType env x $ rewrite p
findType env x (LogicalImplication (LogicalAnd p q) r) = findType env x (LogicalAnd p (LogicalImplication q r))
findType env x p@(LogicalEquivalence _ _)          = findType env x $ rewrite p
findType _ _ (LogicalNegation _)                   = Nothing
```

```
findType _ x (SetMembership (Identifi erExpression x') e)
    | x == x'   = Just e
    | otherwise = Nothing
findType env x ((:==) (Identifi erExpression x') (Identifi erExpression x''))
    | x == x'   = lookupEnv env x''
    | x == x''  = lookupEnv env x'
    | otherwise = Nothing
findType _ _ _                                                    = Nothing

typeQ :: Show α ⇒ Environment → α → Variable → Predicate → TypecheckM (Expression, Environment)
typeQ env context v@(SingletonVariable x) p = case findType env x p of
                                                Nothing → fail $ "No type found"
                                                (Just e) → do m2m ("Forbidden self−reference") (nonfreeM v e)
                                                              m2m ("Cannot shadow") $ mapM_ (nonfreeM v) env
                                                              return (e, (x 'InType' e):env)
typeQ env context (PairVariable x y) p = do (e, env') ← typeQ env context x p
                                            (f, env'') ← typeQ env' context y p
                                            return ((PairExpression e f), env'')
```

The final function translates a certain simple monad (Maybe) into the typechecking monad. The idea is that if the Maybe action fails, we fail with the given message but if it succeeds, we succeed with the same return value.

```
m2m :: String → Maybe α → TypecheckM α
m2m _ (Just a)   = return a
m2m msg Nothing = fail msg
```

D Summary of the ebba-unicode library

D.1 Unicode.hs

This module exports everything the module UnicodeDataDef exports, as well as the functions described below.

module Unicode (**module** UnicodeDataDef, *unidata*, *coerce*, *generalCategory*, *splitAtNLF*)
where

The *coerce* function should not really be in this module, but it is handy. It converts any integral value to any numeric type.

coerce :: (Integral α, Num β) $\Rightarrow$ $\alpha \rightarrow \beta$
coerce = *fromInteger* $\circ$ *toInteger*

The *unidata* function looks up the Unicode data for the given character.

unidata :: Char $\rightarrow$ UD

The *generalCategory* function looks up the general category of the given character.

generalCategory :: Char $\rightarrow$ GeneralCategory
generalCategory = *gc* $\circ$ *unidata*

The final function splits the given string into its first line and the rest of the string. It uses the Unicode newline conventions [29] to decide on the location of the first line break.

splitAtNLF :: String $\rightarrow$ (String, String)

D.2 UnicodeDataDef.hs

The purpose of this module is to declare the data structure types that the rest of the library (and its clients) will be needing.

module UnicodeDataDef
where

For efficiency reasons, we add a great number of strictness annotations. These take the form of an exclamation mark before a parameter type for a constructor. The effect is that when such a value is constructed, the evaluation of that parameter is immediately forced instead of delayed.

Also for efficiency reasons, we include a custom (parametrized) type for strict rational numbers. The type parameter is the integer type that will be used to represent the quotient and the remainder.

```
data Ratio α = !α :% !α
```

The type Bidi is an enumeration of bidirectional categories.

```
data Bidi = L
          | LRE
          | LRO
          | R
          | AL
          | RLE
          | RLO
          | PDF
          | EN
          | ES
          | ET
          | AN
          | CS
          | NSM
          | BN
          | B
          | S
          | WS
          | ON
```

The next type encodes the possible values for the character decomposition mapping.

```
data Decomp = NoDecomp
            | Decomp ![Int]
            | Font ![Int]
            | NoBreak ![Int]
            | Initial ![Int]
            | Medial ![Int]
            | Final ![Int]
            | Isolated ![Int]
            | Circle ![Int]
            | Super ![Int]
            | Sub ![Int]
            | Vertical ![Int]
            | Wide ![Int]
            | Narrow ![Int]
            | Small ![Int]
            | Square ![Int]
            | Fraction ![Int]
            | Compat ![Int]
  deriving (Show, Read)
```

The general category is the most important type. It describes whether the character is an uppercase letter, opening punctuation or something else.

```
data GeneralCategory = Lu
                     | Ll
                     | Lt
                     | Lm
                     | Lo
                     | Mn
                     | Mc
                     | Me
                     | Nd
                     | Nl
                     | No
                     | Pc
                     | Pd
                     | Ps
                     | Pe
                     | Pi
                     | Pf
                     | Po
                     | Sm
                     | Sc
                     | Sk
                     | So
                     | Zs
                     | Zl
                     | Zp
                     | Cc
                     | Cf
                     | Cs
                     | Co
  deriving (Show, Read, Eq)
```

The final type is a collection of all data the Unicode character database has to offer about any given character.

```
data UD = UD {cp :: !Int, cn :: !String, gc :: !GeneralCategory, cc :: !Int, bd :: !Bidi, dc ::
                !Decomp, ddv :: !(Maybe Int), dv :: !(Maybe Int), nv :: !(Maybe
                (Ratio Int)), m :: !Bool, n1 :: !String, c :: !String, um :: ![Int], lm ::
                ![Int], tm :: ![Int]}
          | NotACharacter Int
    deriving (Show, Read)
```

D.3 Octet.hs

The Octet module contains a simple mechanism for reading octet streams from the file system.

```
module Octet
    where
```

An octet is an eight-bit unsigned integer. An octet stream is a list of octets. Note that Word8 is not standard Haskell!

```
type Octet = Word8
type OctetStream = [Octet]
```

The *assume8bit* function takes a string and converts it into an octet stream, checking that no character in the string lies beyond the 8-bit range.

```
assume8bit :: String → OctetStream
```

The standard input stream is available as *stdinOF*.

```
stdinOF :: IO OctetFile
```

The standard output stream is available as *stdoutOF*.

```
stdoutOF :: IO OctetFile
```

The standard error stream is available as *stderrOF*.

```
stderrOF :: IO OctetFile
```

A file can be opened given its file name and open mode (in the style of C's *fopen*).

```
type OpenMode = String

openOF :: String → OpenMode → IO OctetFile
```

A file can be closed using *closeOF*.

closeOF :: OctetFile → IO ()

The octet currently at the point is returned by *peekOctet*.

peekOctet :: OctetFile → IO Octet

The point is advanced by one octet by *eatOctet*.

eatOctet :: OctetFile → IO ()

The octet currently at the point is returned by *getOctet*. It also advances the point by one octet after peeking the octet at the point.

getOctet :: OctetFile → IO Octet

If the point has advanced beyond the end of the file, *isEOF* returns True.

isEOF :: OctetFile → IO Bool

The function *getOctets* reads the content of the file lazily. After calling this function, the other functions can no longer be invoked on this file. The use of this function is discouraged (it uses black magic internally and cannot fully conceal it), but it is sometimes useful.

getOctets :: OctetFile → IO OctetStream

D.4 UTF.hs

module UTF
  where

The following declarations introduce type aliases for two function types.

type UTFDecoder = OctetStream → String
type UTFEncoder = String → OctetStream

Type classes are a mechanism for controlled overloading of functions in Haskell. The following declaration means that potentially any type α can be made the first argument of *encode* and *decode* — all one needs is an instance declaration. In this case, the type α is merely a mechanism for indicating which version of those functions is to be used.

class UTF α
  where *encode* :: α → String → OctetStream
        decode :: α → OctetStream → String

Here we declare a placeholder type for UTF-8. We also declare it an instance of the above type class, so that it can be used as the first argument to the functions. At the same time, we specify which actual functions are to be used as *encode* and *decode* for the case of UTF-8.

```
data UTF8 = UTF8
instance UTF UTF8
    where encode UTF8 = encodeUTF8
          decode UTF8 = decodeUTF8

decodeUTF8 :: UTFDecoder

encodeUTF8 :: UTFEncoder
```