Technical Report

UCAM-CL-TR-201
ISSN 1476-2986

Number 201

Formalizing abstraction mechanisms for hardware verification in higher order logic

Thomas Frederick Melham

August 1990

This technical report is based on a dissertation submitted August 1989 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville & Caius College.

ISSN 1476-2986

Acknowledgements

I should like to thank my research supervisor, Dr Mike Gordon, who allowed me free rein in my early days as a research student, and who spurred me on towards the end. His guidance, advice, and the influence of his ideas have been of great value throughout the development of this work. I should also like to thank Dr Graham Birtwistle for his continued support over many years.

Thanks are also due to the members of the Cambridge hardware verification group, both past and present, for many valuable discussions on various aspects of this work. I am especially grateful to Albert Camilleri, Avra Cohn, Inder Dhingra, Mike Gordon, and Tim Leonard for their helpful comments on earlier drafts of this dissertation. Graham Birtwistle also very kindly read an early draft, and made several valuable suggestions. Thanks are also due to Glynn Winskel for useful discussions about relating transistor models.

Financial support for this work came from a number of sources. Early work was supported by a scholarship from the Royal Commission for the Exhibition of 1851 and by funding provided by the ORS Awards scheme. The Commissioners for the Exhibition of 1851, Pembroke College, and Graham Birtwistle helped with funding to attend conferences. I am grateful to Gonville and Caius College for support in the form of an unofficial fellowship, during the tenure of which much of this dissertation was written.

Contents

Chapter 1

Introduction

Recent advances in microelectronics have given designers of digital hardware the potential to build electronic devices of remarkable size and complexity. With increasing size and complexity, however, it becomes increasingly difficult to ensure that such devices are free of design errors which may cause them to malfunction. Exhaustive simulation of even moderately-sized circuits is impossible, and partial simulation offers only partial assurance of functional correctness.

This is an especially serious problem in safety-critical applications, where failure due to hardware design errors may cause loss of life or extensive damage. In these applications, functional errors in circuit designs cannot be tolerated. But even where safety is not the primary consideration, there may be important economic reasons for doing everything possible to eliminate design errors—and to eliminate them early in the design process. A flawed design may mean costly and time-consuming refabrication, and mass-produced devices with design errors may have to be recalled and replaced.

1.1 Hardware Verification by Formal Proof

A solution to these design correctness problems is one of the goals of recent research in *hardware verification*. With this approach, the functional behaviour of hardware is described mathematically, and formal proof is used to verify that hardware designs meet rigorous specifications of intended behaviour. Such proofs can be very large and complex, so mechanized theorem-proving tools are often used to construct them.

Considerable progress has been made in this area in the past few years. Notable large-scale applications of hardware verification include: Hunt's verification of the FM8501 microprocessor using the Boyer-Moore theorem prover [53], Herbert's verification of a network interface chip using the HOL proof assistant [48], Joyce's verification in HOL [56,55,58] of a simple microprocessor originally verified by Gordon using the LCF_LSM theorem prover [33], and the verification by Narendran and Stillman of an image processing chip using a method based in rewriting [73].

Hardware verification techniques are now considered mature enough to be applied to simple circuits intended for safety-critical applications. Cohn's work on verifying the Viper microprocessor is a preliminary experiment in this area [16,17]. Viper [20] is a simple microprocessor designed at the Royal Signals and Radar Establishment with formal verification in mind. It is intended for use in safety-critical applications and is now commercially available. Cohn's formal proof of the correctness of Viper is not complete, but it covers some important aspects of the machine's design.

1.1.1 The Hardware Verification Method

Two things are needed for any method of hardware verification based on rigorous specification and formal proof. The first is a formal or mathematical *language* for describing the behaviour of hardware and expressing propositions about it. The ideal language for this purpose is expressive enough to describe hardware in a natural and concise notation yet still has a well-understood and reasonably simple semantics. The second requirement is a *deductive calculus* for proving propositions about hardware expressed in this language. This must, of course, be logically sound; and it should be powerful enough to make it possible to prove all the true propositions about hardware behaviour that arise in practice.

Various formal languages and associated proof techniques have been proposed as a basis for hardware verification. These range from special-purpose hardware description languages—with *ad hoc* proof rules—to systems of formal logic, and subsets of ordinary mathematics. Formal methods for reasoning about hardware have been based, for example, on algebraic techniques [4,12,66], various kinds of temporal logic [8,22,25,59,72], functional programming techniques [76], predicate calculus [23,51,53,83], and higher order logic [11,34,43].

Details of the methods for proving hardware correctness based on these different formalisms vary. But many of them share a common general approach, in which a formal proof of the correctness of a hardware device typically involves the following four steps.

1. Write a formal specification S to describe the behaviour which the device must exhibit for it to be considered correct.

2. Write a specification for each kind of primitive hardware component used in the device. These specifications are intended to describe the actual behaviour of real hardware components.

3. Define an expression D which describes the behaviour of the device to be proved correct. The definition of D has the general form

   $$D \overset{\text{def}}{=} P_1 + \cdots + P_n$$

   where P_1, ..., P_n specify the behaviours of the constituent parts of the device, and $+$ is a *composition* operator which models the effect of wiring components together. The expressions $P_1, \ldots, P_n$ used here are instances of the specifications for primitive devices defined in step 2.

4. Prove that the device described by D is correct with respect to the formal specification S. This is done by proving a theorem of the form

   $$\vdash D \ \mathsf{satisfies}\ S$$

   where '$\mathsf{satisfies}$' is some *satisfaction* relation on formal specifications of hardware behaviour. This correctness theorem asserts that the behaviour described by D satisfies the specification of intended behaviour S.

When the device to be proved correct is large, this method is usually applied hierarchically. The device is structured into a hierarchy of components, and formal specifications which describe 'primitive components' at one level of the hierarchy become specifications of intended behaviour at the next level down. The structure of the proof mirrors this hierarchy: the top-level specification is shown to be satisfied by an appropriate connection of components; at the next level down, each of these components is shown to be correctly implemented by a connection of sub-components, and so on—down to the lowest level, where the components used correspond to devices available as hardware primitives.

1.1.2 Limitations of Hardware Verification[1]

A correctness proof of the kind described above cannot *guarantee* that a hardware device will not malfunction: the *design* of a device may be proved correct, but the hardware actually built can still behave in ways not intended by the designer. There are some obvious reasons for this. There may, for example, be fabrication defects in the manufactured device. Or, since many mechanized theorem-proving tools for hardware verification are not yet integrated with the CAD systems used to generate circuit layouts, the design which is verified may not correspond exactly to the device which is built. But in addition to these pragmatic problems, there are also more fundamental reasons why a physical device, although built to a verified design, may still fail to behave as intended by the designer. Two of these are particularly relevant here. Both are results of completely obvious but important limits to what can be established by formal proof.

First, a formal proof cannot demonstrate that a design will behave 'as intended' by the designer; a proof can show only that the design behaves as prescribed by a written, and possibly inaccurate, formal specification. A hardware device may therefore exhibit unexpected behaviour because its design was proved correct with respect to a specification that fails to reflect the designer's intent. This is an

[1]This section is based, in part, on Cohn's discussion in [18] of the limitations of hardware verification in the context of the Viper verification project.

obvious point. But it is an especially important one when the specification is large or the behaviour being specified is complex, for then it may be far from clear to the designer that the specification is itself correct.

Second, a formal proof cannot, strictly speaking, demonstrate anything about a physical hardware device. All that can be proved is that a mathematical *model* of the device has the properties prescribed by a specification of required behaviour. But a model may fail to capture important aspects of the real hardware it is intended to describe. Design errors may therefore escape discovery because the undesirable behaviour resulting from them is not reflected in the formal model on which a proof of correctness is based.

Because of these fundamental limits to the scope of formal proof, the hardware verification method described in Section 1.1.1 can never establish with complete certainty that a physical device will function exactly as intended. A correctness result obtained by this method can be only as good as the formal model and the specification it relates. There are, however, two important practical measures which can be taken to help justify confidence in the significance of correctness results obtained by formal proof. First, specifications of required behaviour can be made as clear and concise as possible, so that their fidelity to the designer's intent can be evaluated easily. This reduces the likelihood of a proof failing to yield a meaningful result because the specification of required behaviour is itself incorrect.[2] Second, formal models can be used which describe the empirical behaviour of real devices as accurately as possible. The more accurate the model used, the less likely it is that a design error will escape discovery by formal proof.

Adopting these two measures—making models as accurate as possible, and making specifications clear and concise—is essential if design verification by formal proof is to be an effective way of dealing with the problem of hardware correctness. An important consequence of this is that the notion of *abstraction* plays a central role in effective formal methods for hardware verification.

1.2 Abstraction

Abstraction is the process by which the important properties of a complex object are isolated for further consideration or use and the remaining ones ignored as being irrelevant to the task at hand. An example is the process of procedural abstraction in programming. Once a procedure has been defined, it is treated as an atomic operation with only one important attribute: *what* the procedure does. The exact sequence of computation steps which achieve this operation—*how* the procedure does it—is ignored [36]. Programming language constructs that support

[2]Another approach is to make specifications executable, so that the designer can run them to gain confidence in their accuracy (see [10,65,80]). This is a special case of the more general strategy of evaluating specifications by deriving consequences of them.

abstraction in this way are fundamental tools for dealing with the complexity of the programming task. By allowing the attributes of a complex object which are important to the task at hand to be isolated from those which are not, software abstraction mechanisms provide a way of limiting the amount of detail that must be considered at any one time [37].

Abstraction plays a similar role in hardware verification. Here, an abstraction mechanism establishes a formal *relationship* of abstraction between a complex description of hardware behaviour and a conceptually simpler one. Abstraction mechanisms of this kind provide a means for controlling the complexity of both formal specifications of hardware behaviour and proofs of design correctness. By suppressing the irrelevant information in detailed formal descriptions of hardware behaviour, and thereby isolating the properties of these descriptions which are most important, effective abstraction mechanisms help to control the size and complexity of the specifications at each level in a hierarchically-structured proof of correctness. In this way, abstraction mechanisms and hierarchical structuring can help to control the complexity of correctness proofs for large and complex hardware designs—in the same way that software abstraction mechanisms are used to manage the complexity of program development.

Considered in the context of the fundamental limitations to the scope of formal proof discussed in the previous section, it is clear that the concept of abstraction must play a central role in two areas of hardware verification: (1) in formulating *meaningful* assertions about the correctness of large or complex hardware designs, and (2) in assessing the *accuracy* of the formal models on which correctness proofs are based. These two aspects of the role of abstraction in hardware verification are discussed briefly in Sections 1.2.1 and 1.2.2 below.

1.2.1 Abstraction and Correctness

In proving the correctness of a hardware device, it is desirable to use a formal model of hardware behaviour whose mathematical properties reflect as accurately as possible the empirical behaviour of the physical device itself. This does not mean that it is always necessary to model the physics of electronic circuits in as much detail as possible: a simplified model can often be justified by the fact that the device which is to be verified is implemented in a particular technology or design style. But a model should nonetheless reflect as accurately as possible the actual behaviour of a device built using the technology or design style in question. The reason for this was discussed above in connection with the limitations of hardware verification by formal proof—the more accurate the formal model of a device, the less likely it is that errors in its design will escape discovery.

Specifications of required behaviour, on the other hand, must be clear and concise, so that they are intelligible enough to be seen to reflect the designer's intent. Most of the details about the actual behaviour of device which is to be proved correct must therefore be left out the formal specification of its intended behaviour. Only the essential aspects of its required behaviour can be included. Furthermore, the larger and more complex the device being verified, the more detailed information about its actual behaviour must be ignored in order to keep the specification of its required behaviour small.

This means that specifications must, in general, present more abstract views of the behaviour of hardware devices than models do. The concept of a relationship of abstraction between two formal descriptions of hardware behaviour is therefore fundamental to expressing formally what it *means* for a device to be ‘correct’. Formally, the correctness of a device is stated by a theorem which asserts that a mathematical model of its actual behaviour in some sense ‘satisfies’ a specification of its intended behaviour. For all but the simplest devices, the satisfaction relation used to formulate this correctness relationship must relate a *detailed* design model to an *abstract* specification of required behaviour. This notion of correctness as a relationship of abstraction is discussed in detail in Chapter 4 of this dissertation.

1.2.2 Abstraction and the Accuracy of Models

Although the value of a correctness result depends on how accurately a model reflects the actual behaviour of the physical device it represents, a very accurate formal model of hardware behaviour may be unnecessarily complex, and it may be possible to adopt a circuit design style for which a simpler model will do. The *functional* correctness of a fully complementary CMOS circuit, for example, does not critically depend on transistor size ratios [84]. A very accurate formal model of CMOS transistor behaviour, which takes into account transistor size ratios, would therefore be inappropriate for this conservative design style. In this case, a less accurate—but also simpler and more tractable—formal model of CMOS transistor behaviour can be used. The validity of using this simpler model can be justified on empirical grounds, in the light of what is known about the actual behaviour of fully complementary CMOS circuit designs.

In general, a simplified model can often be justified empirically by the fact that the device which is to be proved correct is implemented in a restricted style of circuit design. Such a model may be less ‘accurate’ than a more complex formal model of hardware behaviour, but the restrictions on device behaviour imposed by the design style itself will ensure that the extra accuracy of a more complex model is not needed. In this case, the simple formal model of hardware behaviour will in fact be an *abstraction* of a more accurate—but also more complex—formal model of hardware behaviour. Both models will describe the same physical hardware device, but the simpler model will describe only *some* of the aspects of device behaviour that are captured by the more accurate model.

Although it may be possible to justify this use of a simplified model of device behaviour *empirically*, it is also desirable to assess the accuracy of a simplified model by more rigorous means. This can be done by means of a formal proof which demonstrates that a simplified model is sufficiently accurate (with respect to a more complex model) for the particular circuit design style in question. Such a proof would show that a simple formal model of hardware behaviour is, in some sense, a *valid* abstraction of a more complex formal model of hardware behaviour for a restricted class of circuit designs. This notion of an abstraction relationship between two formal models of hardware behaviour, and the connection between this idea and the concept of the relative accuracy of two models, is discussed in detail in Chapters 4 and 7 of this dissertation.

1.3 Motivation

The research reported in this dissertation was originally motivated by the author's experience with using the LCF_LSM theorem prover [32] to prove the correctness of an associative memory device intended for use in a local area network [6,7]. This device comprised 37 SSI and MSI TTL chips, the most complex of which was an AM2910 microprogram controller. Although the design of this device was relatively straightforward, its formal verification in the LCF_LSM system proved to be remarkably difficult. The main problem was simply the almost intractably large size of the intermediate theorems generated during the proof. Single theorems were often generated by the system which were hundreds of lines long, and several minutes, or even several hours, of CPU time were needed to manipulate them. The proof was completed only with considerable difficulty—some time after the LCF_LSM system had become obsolete.

The difficulties encountered during this exercise were for the most part due, not to problems with the LCF_LSM theorem prover itself, but to deficiencies in the underlying formalism for hardware verification supported by the system. Two main problems were encountered in the course of the proof. First, the LCF_LSM formalism (which is described in detail in [32]) limited the extent to which both the associative memory itself and the hardware components used in its design could be described by concise *abstract* formal specifications. Second, this formalism provided only limited and inflexible *abstraction mechanisms* for relating formal specifications of hardware behaviour at different levels of abstraction.

These two limitations imposed considerable restrictions on the extend to which abstraction could be used to simplify specifications at intermediate levels in the hierarchically-structured proof of correctness for the associative memory device. For example, fully detailed information about the effect of executing any of 16 possible opcodes using the AM2910 microprogram controller had to be retained in formal specifications throughout much of the proof, even though the actual microcode for the associative memory made use of only 5 different opcodes. It became clear from this exercise that more effective abstraction mechanisms were needed for controlling the complexity of specifications and proofs than were made available by the LCF_LSM formalism and theorem prover.

The research reported in this dissertation was originally undertaken in order to develop some practical and widely applicable abstraction techniques for dealing with the kinds of problems encountered in this application.

1.4 The Contribution of this Work

In this dissertation, it is shown how reasoning about the correctness of hardware by formal proof can be done using certain fundamental abstraction mechanisms to relate specifications of hardware behaviour at different levels of abstraction. The formalism used here is a variety of higher order logic [29]. The general approach taken in this work is pragmatic and example-driven. Correctness proofs were done in higher order logic for a variety of simple hardware devices, each of which was chosen to isolate the issues involved in some particular aspect of abstraction, formal specification, or proof. Chapter 4 provides a general account of some fundamental principles derived from these examples. Chapters 5–7 give a selection of detailed examples which illustrate these basic principles. The aim of these examples is to provide clear illustrations of some specific aspects of the use of abstraction mechanisms in hardware verification. The examples given here are therefore generally very simple (an exception is the case study in Chapter 6).

One of the main aims of this dissertation is to provide a clear and motivated account of the role of abstraction in hardware verification, and to describe some basic techniques by which abstraction mechanisms for hardware verification can be formalized in higher order logic—by purely *definitional* means (see below). In addition to this general account of some basic principles behind the formalization of abstraction mechanisms for hardware verification in higher order logic, the main contributions of the work reported in this dissertation are the following:

- A systematic method is developed for defining any instance of a wide class of concrete data types in higher order logic (Appendix A). This method has been fully automated in the HOL theorem prover for higher order logic. The types definable by this method provide a firm logical basis for representing *data* in formal specifications of hardware behaviour at various different levels of data abstraction (Chapter 5).

- A new technique is described for modelling the behaviour of entire *classes* of hardware designs in higher order logic. The technique is based on a formal representation in logic for the structure of circuit designs which makes use of recursive types definable by the systematic method mentioned above.

This approach can be used both to prove the correctness of certain classes of circuit designs (Chapter 5) and to support reasoning about the relative accuracy of formal models of hardware behaviour in general (Chapter 7).

- A technique is described by which the correctness of a model that contains detailed information about the time-dependent behaviour of a device can be formulated with respect to a specification of required behaviour written in terms a more abstract representation of *time*. The method is illustrated by a substantial case study—the formal verification in higher order logic of a simple ring communication network (Chapter 6).

Two fundamental principles underlie all this work. The first is that all reasoning about hardware behaviour should be done by strictly formal proof, using only the deductive calculus provided by the primitive basis of higher order logic. This provides the greatest possible assurance that only logically sound reasoning is used. The second principle is that any special-purpose notation needed to specify hardware behaviour and to formulate propositions about hardware correctness should be introduced by definitional means only. This ensures that inconsistency is not introduced by postulating *ad hoc* axioms intended to characterize non-primitive mathematical objects.

A consequence of adopting these two principles was that mechanized theorem proving support was essential for the work reported in this dissertation, since for any but the simplest theorems of higher order logic it is not feasible to carry out fully detailed and completely formal proofs manually. Except where noted, formal proofs for all the major theorems about hardware in this dissertation were generated using the HOL interactive proof assistant [30]. Some special-purpose theorem proving tools were also developed in the HOL system for the work on abstraction reported in this dissertation. But detailed discussion of these theorem proving tools and the fully formal proofs done using them is avoided in the body of the dissertation and relegated to the appendixes.

1.5 Outline of the Dissertation

A brief outline of the dissertation is given below. There is no chapter devoted exclusively to an account of related work. This is instead discussed *passim*.

- Chapter 2 gives an overview of higher order logic and its mechanization in the HOL system. The purpose of this chapter is to make the dissertation self-contained, and the emphasis is therefore on the features of the logic which are important to an understanding of later chapters. Of particular importance are Sections 2.1.4–2.1.7, in which the definitional mechanisms of the logic are explained. These play a central role in formalizing the abstraction mechanisms discussed in later chapters. Except in the details of presentation, none of the material in Chapter 2 is new. Both the formulation of higher order logic described in this chapter and its mechanization in the HOL system are due to Gordon [29,30].

- Chapter 3 introduces the basic techniques for hardware verification using higher order logic. A method for specifying the behaviour of hardware is described, together with a method for constructing behavioural models of composite devices from the specifications of their components. An example proof is given which illustrates the general approach to verification in higher order logic. The techniques described in this chapter are well-established and widely-used (see, for example, [11,34,43,51]), and no claim to novelty is made for the basic ideas behind them. The particular approach described here is due to Gordon [34]. The account given in this chapter represents the present author's view of this approach. The chapter concludes with an overview of related work on hardware verification based on formal logic.

- Chapter 4 shows how the two basic types of abstraction introduced above in Sections 1.2.1 and 1.2.2 can be formalized in higher order logic. These two types of abstraction are referred to as abstraction *within* a model of hardware behaviour, and abstraction *between* models of hardware behaviour. This chapter is concerned only with the general ideas behind the formalization of these two kinds of abstraction in higher order logic. The technical details are covered in the three chapters that follow.

  Abstraction within a model is discussed in Section 4.1. This section shows how the idea of correctness as a relationship of abstraction can be expressed formally in logic and incorporated into the method of hardware verification introduced in Chapter 3. Three basic abstraction mechanisms are discussed in this section: behavioural abstraction, data abstraction, and temporal abstraction. Each of these abstraction mechanisms can be used to relate detailed formal models of hardware designs to more abstract and concise specifications of intended behaviour. The role of abstraction in hierarchical verification is also discussed in this section.

  Section 4.2 introduces the idea of an abstraction relationship between two formal models of hardware behaviour. An abstraction relationship of this kind describes the conditions under which correctness results obtained in an abstract or simplified model of hardware behaviour agree with correctness results obtained using a more accurate but less tractable model. Only a very brief introduction to the idea of an abstraction relationship between two models of hardware behaviour is given in this section, but a detailed example is provided in Chapter 7.

- The subject of Chapter 5 is data abstraction. In particular, the emphasis of this chapter on the use of defined logical *types* to represent 'data' in higher order logic. In Section 5.1, it is shown how an arbitrary instance of a wide class of concrete data types can be characterized formally in higher order logic. In Sections 5.2 and 5.3, two detailed examples are given to show how these defined logical types be used to support formal reasoning about hardware behaviour where data abstraction is involved. A common theme of these two examples is the importance of an appropriate choice of data types for use in formal specifications of hardware behaviour.

- Chapter 6 describes some techniques developed for temporal abstraction in higher order logic and illustrates the use of these techniques by a case study. Temporal abstraction involves relating formal specifications that describe hardware behaviour using different notions of time. In Sections 6.1–6.2, a technique is described for constructing *time mappings* in higher order logic, and it is shown how mappings of this kind can be used to formulate the correctness of hardware devices with respect to temporally abstract specifications of required behaviour. In Section 6.3, a substantial case study involving temporal abstraction is presented: a proof of correctness in higher order logic for the design of a simple ring communication network.

- In Chapter 7, a worked example is given to illustrate the concept of an abstraction relationship between two models of hardware behaviour which was introduced in Chapter 4.

- In Chapter 8, a brief summary is given of the main contributions of this work, and some directions for future research are proposed.

- Appendix A gives a detailed account of a systematic method for defining an arbitrary, possibly recursive, concrete data type in higher order logic. A fully automatic implementation of this method in the HOL system is also discussed. This work, although presented in an appendix, is one of the main contributions of this dissertation. The method for defining logical types explained here is fundamental to the work on abstraction discussed in Chapters 5 and 7. The text of Appendix A is adapted with very few changes from the self-contained account of this work published as [64], and there is therefore a small amount of repetition in the appendix of material already covered in the less detailed summary provided in Chapter 5.

- Appendix B contains an example interactive HOL session which illustrates the use of the automatic theorem proving tools based on the method for defining recursive types discussed in Appendix A.

Chapter 2

Higher Order Logic and the HOL System

This chapter provides an overview of the formulation of higher order logic used in later chapters for reasoning about hardware. A brief description is also given of the mechanization of this logic in the HOL theorem proving system.

Higher order logic is described in Section 2.1. The description given in this section is not complete; only those aspects of the logic which are important to an understanding of later chapters are covered. Of particular importance are Sections 2.1.4–2.1.7, which describe formal mechanisms for making various kinds of definitions in higher order logic. These play a central role in formalizing the abstraction mechanisms for hardware verification discussed in Chapters 4–7.

An overview of the HOL theorem prover for higher order logic is given in Section 2.2. A full description of the HOL system is beyond the scope of this dissertation, and only a sketch of the approach to theorem proving used in the HOL system is provided here. Except for some of the theorems in the T-ring correctness proof given in Chapter 6, all the theorems of higher order logic in this dissertation were generated in the HOL system. Details of how these proofs were carried out in the system will not be given, but an example interactive session with the HOL system can be found in Appendix B.

2.1 An Overview of Higher Order Logic

The version of higher order logic described in this section was developed by Dr M. Gordon at the University of Cambridge [29]. Gordon's version of higher order logic is based on Church's formulation of the simple theory of types [14], which combines some of the features of the λ-calculus with a simplification of the early type theory of Russell and Whitehead [82]. Gordon's machine-oriented formulation extends Church's theory in two significant ways: the syntax of types includes the polymorphic type discipline developed by Milner for the LCF logic PPλ [35], and the primitive basis of the logic includes formal rules of definition for consistently extending the logic with new constants and new types.

The following sections give a brief introduction to this formulation of higher order logic. For a full account of the logic see [29,30]. In what follows, and in the remaining chapters, the phrase 'higher order logic' should generally be understood to mean the particular formulation described here. Detailed descriptions of related versions of higher order logic can be found in [1,47].

2.1.1 Types

Higher order logic is a typed logic. Every term of the logic has an associated logical *type* which represents the kind of value it denotes. A term is considered to be syntactically well-formed only if its type is consistent with the types of its subterms. As a syntactic device, types are necessary to eliminate certain 'paradoxical' statements which, if they could be expressed in the logic, would make it inconsistent (e.g. formulations of Russell's paradox). Ensuring that every term has a type which is consistent with those of its subterms simply makes such paradoxical expressions syntactically ill-formed.

2.1.1.1 The Syntax of Types

The syntax of types in higher order logic is given by

$$ty \quad ::= \quad c \;\mid\; v \;\mid\; (ty_1, \ldots, ty_n)op$$

where c ranges over type *constants*, v ranges over type *variables*, op ranges over n-ary type *operators* (for $n \geq 1$), and ty, ty_1, …, ty_n range over types. Type constants and type variables are called *atomic* types. Types constructed using type operators are called *compound* types.

Type constants are identifiers that name fixed sets of values. An example is the primitive type constant $bool$. This type constant denotes the two-element set of boolean truth-values.

Type variables are used to stand for 'any type'. They are written α, β, γ, etc. Type expressions that contain type variables are called *polymorphic* types. A *substitution instance* of a polymorphic type ty is a type obtained by substituting types for all occurrences of one or more of the type variables in ty. Type variables occur in Church's formulation of higher order logic [14] as *metavariables* ranging over types; in the version of higher order logic used here they are part of the object language. This allows a limited form of implicit universal quantification over types within the logic, since theorems that contain polymorphic types are also true for any substitution instance of them.

A compound type of the form $(ty_1, \ldots, ty_n)op$ denotes a set constructed from the sets denoted by the types ty_1 through ty_n. The n-ary type operator op is the name of the operation that constructs this set. The compound type $(bool, bool)fun$, for example, denotes the set of all total functions from values of type $bool$ to values of type $bool$. This compound type is constructed using the binary type operator fun, which denotes the function space operation on sets.

2.1.1.2 Primitive and Defined Types

There are two primitive type constants in higher order logic: *bool* and *ind*. The type constant *bool* denotes the two-element set of boolean truth-values. The type constant *ind* denotes the set of 'individuals', which in this formulation of higher order logic is simply a set with infinitely many distinct elements. There is only one primitive type operator in higher order logic: the binary type operator *fun*. If ty_1 and ty_2 are any two types, then the compound type $(ty_1, ty_2)fun$ is the type of all total functions from values of type ty_1 to values of type ty_2.

In principle, every type needed for doing proofs in higher order logic can be written using only type variables, the primitive type constants *bool* and *ind*, and the primitive type operator *fun*. But in practice it is desirable to add more type constants and operators to the logic than are strictly necessary to prevent inconsistency. In the version of higher order logic used here, this is done by defining new types and type operators in terms of primitive types (or other already defined types). A type definition extends the language of types in higher order logic by introducing a new type constant or type operator not already present in it. Formally, a type definition is an axiom which is added to the logic to define the meaning of a new type expression. The primitive basis of higher order logic includes an explicitly-stated *rule of definition* for introducing axioms of this kind. This rule is explained in detail in Section 2.1.7.

Two logical types which can be defined formally using this rule are *num*, the type of natural numbers, and $(ty_1, ty_2)prod$, the cartesian product of ty_1 and ty_2. A summary of some notation associated with these two basic types is given in Section 2.1.2.6, and a full account of how they are defined is given in Appendix A. Other defined types are also introduced in Appendix A, and in Chapters 5 and 7.

2.1.1.3 Notational Abbreviations for Types

Some notational abbreviations are used to make type expressions more readable. Compound types constructed with the type operators *fun* and *prod* can be written using the infix notation shown below.

Infix Abbreviations for Types	
Type	*Abbreviation*
$(ty_1, ty_2)fun$	$(ty_1 \rightarrow ty_2)$
$(ty_1, ty_2)prod$	$(ty_1 \times ty_2)$

Expressions written using this infix notation are *metalinguistic* abbreviations for the corresponding object-language types; unlike defined types, they are not part of an extended syntax of object-language type expressions. The expression '$bool{\rightarrow}(bool{\rightarrow}bool)$', for example, is simply shorthand for the less readable type expression $(bool, (bool, bool)fun)fun$.

By convention, it is assumed that the infix symbols $\rightarrow$ and $\times$ associate to the right. The expression $\mathit{bool}{\rightarrow}(\mathit{bool}{\rightarrow}\mathit{bool})$, for example, can be written without parentheses as $\mathit{bool}{\rightarrow}\mathit{bool}{\rightarrow}\mathit{bool}$. In addition, the infix symbol $\times$ is assumed to be more tightly binding than $\rightarrow$. So, for example, the expression $\mathit{bool} \times \mathit{bool} \rightarrow \mathit{bool}$ means $(\mathit{bool} \times \mathit{bool}) \rightarrow \mathit{bool}$.

2.1.2 Terms

The higher order logic notation for *terms* can be viewed informally as an extension of the conventional syntax of predicate calculus in which variables can range over functions (higher order variables) and functions can take functions as arguments or yield functions as results (higher order functions). These two extensions are illustrated by the proposition of higher order logic shown below.

$$\forall x f.\, \exists \mathit{fn}.\, (\mathit{fn}\; 0 = x) \wedge \forall n.\, \mathit{fn}\; (n{+}1) = (f\; (\mathit{fn}\; n))\; n \tag{2.1}$$

This proposition states that functions can be defined on the natural numbers such that they satisfy primitive recursive equations. It asserts that for any value x and any function f, there is a function fn that yields x when applied to 0 and satisfies the recursive equation $\mathit{fn}\; (n{+}1) = (f\; (\mathit{fn}\; n))\; n$ for all values of n. The quantified variables f and fn in (2.1) are examples of higher order variables; they both range over functions. The function f is also an example of a higher order function; when applied to the value $(\mathit{fn}\; n)$ on the right hand side of the recursive equation in (2.1) it yields a function as a result.

The proposition shown above is written in an abbreviated notation. It stands for a much less readable expression written in the ‘pure’ syntax of higher order logic terms. This pure syntax of terms is described below in Section 2.1.2.1. Some notational abbreviations for terms are introduced in Sections 2.1.2.4 and 2.1.2.5. These define the notation used in proposition (2.1) and allow terms to be written in a form which resembles the conventional syntax of predicate calculus.

2.1.2.1 The Syntax of Terms

The syntax of (untyped) terms in higher order logic is given by

$$\mathit{tm} \quad ::= \quad c \;\mid\; v \;\mid\; (\mathit{tm}_1\; \mathit{tm}_2) \;\mid\; \lambda v.\, \mathit{tm}$$

where c ranges over constants, v ranges over variables, and tm, tm_1, and tm_2 range over terms. Terms of the form $(\mathit{tm}_1\; \mathit{tm}_2)$ are called *applications*, and terms of the form $\lambda v.\, \mathit{tm}$ are called *abstractions*. In this dissertation, sans serif identifiers (e.g. a, b, c, Const) and non-alphabetical symbols (e.g. $\supset$, $=$, $\forall$) are generally used for constants, and italic identifiers (e.g. v, x, x_1, fn, F, G) are used for variables.

2.1.2.2 Free and Bound Variables and Substitution

An occurrence of variable v in a term tm is *bound* if it occurs after the dot in a subterm of the form '$\lambda v.\, t$'. An occurrence of a variable which is not bound is called *free*. If tm_1, ..., tm_n are terms and v_1, ..., v_n are distinct variables, then the metalinguistic notation $tm[tm_1, \ldots, tm_n/v_1, \ldots, v_n]$ stands for the result of simultaneously substituting tm_i for v_i for $1 \leq i \leq n$ at every free occurrence of v_i in the term tm, with the condition that no free variable in any tm_i becomes bound in the result of the substitution. When the notation $tm[tm_1, \ldots, tm_n]$ is used, it should be understood to represent a term obtainable as the result of such a substitution. In particular, '$tm[v_1, \ldots, v_n]$' means a term obtainable by a substitution of the form $tm[v_1, \ldots, v_n/v'_1, \ldots, v'_n]$ such that none of the variables v_1, ..., v_n becomes bound in the result. Thus $tm[v_1, \ldots, v_n]$ represents a term in which there may be free occurrences of the variables v_1, ..., v_n. As used in later chapters, this notation should be understood simply to mean a term with exactly n distinct free variables v_1, ..., v_n. And when a term is written $tm[v_1, \ldots, v_n]$, the notation $tm[tm_1, \ldots, tm_n]$ should, throughout the context in which this term is discussed, be understood to mean $tm[tm_1, \ldots, tm_n/v_1, \ldots, v_n]$.

2.1.2.3 Well-typed terms

Every term in higher order logic must be *well-typed*. Writing $tm{:}ty$ indicates explicitly that the term tm is well-typed with type ty. The well-typed terms of higher order logic are defined inductively as follows:

- *Constants:* Each constant c has a fixed type ty, called its *generic* type. If the generic type ty is polymorphic, then $\mathsf{c}{:}ty'$ is a well-typed term for any substitution instance ty' of ty.

- *Variables:* If an identifier id is not already the name of a constant then $id{:}ty$ is a well-typed variable for any type ty.

- *Applications:* If $tm_1{:}ty_1{\to}ty_2$ and $tm_2{:}ty_1$ are well-typed terms then the application $(tm_1\ tm_2){:}ty_2$ is a well-typed term. It represents the result of applying the function denoted by tm_1 to the value denoted by tm_2.

- *Abstractions:* If $x{:}ty_1$ is a variable and $tm{:}ty_2$ is a well-typed term then the abstraction $(\lambda x.\, tm){:}ty_1{\to}ty_2$ is a well-typed term. It represents the function whose value for an argument a is given by $tm[a/x]$.

Only well-typed terms are considered syntactically well-formed in higher order logic. There is an algorithm, due to Milner [68], which can be used to *infer* the type of a term from the generic types of the constants it contains. The HOL mechanization of higher order logic uses this algorithm to assign consistent types to logical terms entered by the user. In general, types will not be mentioned explicitly when it is clear from the form or context of a term what its type must be. When necessary, the notation *tm*:*ty* will be used to indicate the types of variables or constants.

2.1.2.4 Primitive and Defined Constants

There are three primitive constants in higher order logic:

$$= :\alpha{\to}\alpha{\to}\mathit{bool}, \qquad \supset :\mathit{bool}{\to}\mathit{bool}{\to}\mathit{bool} \qquad \text{and} \qquad \varepsilon :(\alpha{\to}\mathit{bool}){\to}\alpha.$$

The two constants $=$ and $\supset$ denote the binary relations of equality and material implication respectively. These relations are represented by higher order functions; when applied to a value, both $=$ and $\supset$ yield boolean-valued functions. For example, the application $(\supset P)$ denotes a function of type $\mathit{bool}{\to}\mathit{bool}$. When applied to a boolean term Q, the resulting term $(\supset P)\ Q$ expresses the proposition that P implies Q. Likewise, the application $(= x)\ y$ means x equals y. The third primitive constant—the constant ε—is a *selection* operator [47,60] for higher order logic. A description of this operator is deferred until Section 2.1.5.

The three symbols $=$, $\supset$, and ε are the only primitive constants in higher order logic. All other constants are introduced by means of constant definitions. These extend the logic by defining new constants as atomic abbreviations for particular logical terms. Formally, constant definitions—like type definitions—are axioms that extend the object-language syntax of higher order logic. The formal rule of definition which allows these axioms to be added to the logic is explained in detail in Section 2.1.4.

Some basic constants which can be defined formally using this rule of definition are listed in the table shown below.

Basic Defined Constants

Defined Constant	*Generic Type*	*Description*
$\neg$	$\mathit{bool}{\to}\mathit{bool}$	negation
$\wedge$	$\mathit{bool}{\to}\mathit{bool}{\to}\mathit{bool}$	conjunction
$\vee$	$\mathit{bool}{\to}\mathit{bool}{\to}\mathit{bool}$	disjunction
$\forall$	$(\alpha{\to}\mathit{bool}){\to}\mathit{bool}$	universal quantification
$\exists$	$(\alpha{\to}\mathit{bool}){\to}\mathit{bool}$	existential quantification
$\exists!$	$(\alpha{\to}\mathit{bool}){\to}\mathit{bool}$	unique existence
T, F	*bool*	truth-values: *true* and *false*

These constants are part of the conventional notation of mathematical logic and can be defined formally in higher order logic so that they have their usual logical properties. The formal definitions of these basic constants will not be given here, but full details can be found in [29].

2.1.2.5 Notational Abbreviations for Terms

The pure syntax of higher order logic terms described above can be made to resemble the conventional notation of predicate calculus by means of the following metalinguistic abbreviations.

Infixes. Certain applications of the form $(\mathsf{c}\ tm_1)\ tm_2$ are abbreviated by writing the constant c in infix position. These include (among others) applications of the constants $=$, $\wedge$, $\vee$, and $\supset$. The expression $(a \wedge b) \supset a$, for example, should be read as a metalinguistic abbreviation for the term $(\supset\ ((\wedge\ a)\ b))\ a$. Other infix notation will be introduced as needed.

Omission of Parentheses. The following conventions allow parentheses to be omitted when writing terms. Application associates to the left, so terms of the form $(\ldots((tm_1\ tm_2)\ tm_3)\ldots tm_n)$ can also be written $tm_1\ tm_2\ tm_3\ \ldots\ tm_n$. When the constants $\supset$, $\wedge$, and $\vee$ occur in infix position, they associate to the right. For example, the expression $a \supset b \supset c$ means $a \supset (b \supset c)$. Application of the basic constants $\neg$, $\wedge$, $\vee$, $\supset$, and $=$ binds less tightly than application of other functions. For example, the expression $f\ x \wedge g\ x$ means $(f\ x) \wedge (g\ x)$. These basic constants are themselves ranked in decreasing order of tightness of binding as follows: $\neg$, $\wedge$, $\vee$, $\supset$, $=$. That is, application of $\neg$ binds more tightly than application of $\wedge$, which in turn binds more tightly than application of $\vee$, and so on. For example, the expression $\neg a \wedge b \supset c$ means $((\neg a) \wedge b) \supset c$. Finally, the scope of '$\lambda x.$' is assumed to extend as far to the right as possible. For example, the expression $\lambda f.\ f\ x$ means $\lambda f.\ (f\ x)$ rather than $(\lambda f.\ f)\ x$.

Quantifiers. In higher order logic, the quantifiers $\forall$, $\exists$, and $\exists!$ are functions that map predicates (i.e. boolean-valued functions) to truth-values. For example, the application $\exists\ (\lambda x.\ x{=}x)$ expresses the proposition that there is at least one value for which the predicate denoted by $\lambda x.\ x{=}x$ is true. In the usual notation of predicate calculus, this proposition is written $\exists x.\ x{=}x$. In higher order logic, this conventional notation for quantifiers is defined by means of the metalinguistic abbreviations shown below.

Abbreviations for Quantifiers

Term	*Abbreviation*	*Term*	*Abbreviation*
$\forall\ (\lambda x.\ tm)$	$\forall x.\ tm$	$\forall v_1.\ \forall v_2.\ \cdots \forall v_n.\ tm$	$\forall v_1\ v_2\ \ldots\ v_n.\ tm$
$\exists\ (\lambda x.\ tm)$	$\exists x.\ tm$	$\exists v_1.\ \exists v_2.\ \cdots \exists v_n.\ tm$	$\exists v_1\ v_2\ \ldots\ v_n.\ tm$
$\exists!\ (\lambda x.\ tm)$	$\exists! x.\ tm$	$\exists! v_1.\ \exists! v_2.\ \cdots \exists! v_n.\ tm$	$\exists! v_1\ v_2\ \ldots\ v_n.\ tm$

A similar abbreviated notation is used for nested λ-abstractions. For example, the expression $\lambda x\, y.\ tm$ is an abbreviation for the term $\lambda x.\ \lambda y.\ tm$.

Other Notation. The expression $f \circ g$ denotes the composition of the two functions f and g and satisfies the usual defining equation: $\forall x.\,(f \circ g)\; x = f(g(x))$. The symbol $\circ$ is a defined constant written in infix position. The expression $(b \Rightarrow tm_1 \,|\, tm_2)$ means 'if b then tm_1 else tm_2' and is an abbreviation for the term $\mathsf{Cond}\; b\; tm_1\; tm_2$, where Cond is an appropriately-defined constant. (See [29] for its definition.) Other metalinguistic abbreviations and notational conventions are introduced in later chapters.

2.1.2.6 Constants for the Defined Types *num* and $\alpha \times \beta$

Some constants associated with the basic defined types *num* and $\alpha \times \beta$ are shown in the table below.

Constants for the Defined Types *num* and $\alpha \times \beta$

Type	*Constants*	*Generic Type*	*Description*
num	$0, 1, 2, \ldots$	*num*	numerals of type *num*
	Suc	$num{\to}num$	successor
	$+, \times, \mathsf{Exp}$	$num{\to}num{\to}num$	arithmetic functions
	$<, \leq, >, \geq$	$num{\to}num{\to}bool$	ordering relations
$\alpha \times \beta$	,	$\alpha{\to}\beta{\to}(\alpha \times \beta)$	pairing (infix)
	Fst Snd	$(\alpha \times \beta){\to}\alpha$ $(\alpha \times \beta){\to}\beta$	projection functions for the components of pairs

These constants are standard mathematical notation and can be defined formally using the rule for constant definitions described in Section 2.1.4. (See [29] for the definitions.) The usual elementary theorems about the natural numbers (e.g. Peano's Postulates), theorems of arithmetic, and theorems about pairs follow from the definitions of these constants. These theorems are used in proving the propositions about hardware discussed in later chapters, but proofs of these simple theorems about arithmetic and pairs will not be given.

2.1.3 Sequents, Theorems and Inference Rules

The style of proof supported by Gordon's formulation of higher order logic is a form of natural deduction [47] based on Milner's formulation of PPλ [35]. In this style of proof, *sequents* are used to keep track of assumptions. A sequent is written $\Gamma \vdash tm$, where Γ is a set of boolean terms called the assumptions and tm is a boolean term called the conclusion. The assumptions and conclusion of a sequent correspond to formulas in predicate calculus. In higher order logic, however, there is no special syntactic class of formulas—these are simply terms of type *bool*.

The sequent notation $\Gamma \vdash tm$ can be read as the metalinguistic assertion that there exists a natural deduction proof of the conclusion tm from the assumptions in Γ. When the set of assumptions Γ is empty, the notation $\vdash tm$ is used. In this case, *tm* is a formal *theorem* of the logic. The same notation is used for the *axioms* of the logic—these are theorems which are simply postulated to be true.

In [29], the inference rules of the logic are stated using the notation illustrated by the rule shown below.

$$\textsc{Modus Ponens:} \qquad \frac{\Gamma_1 \vdash t_1 \supset t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

This rule states that from the two formulas $t_1 \supset t_2$ and t_1 one can immediately infer the formula t_2 by *modus ponens*. In a natural deduction proof, formulas occurring as lines in the proof can depend on assumptions, in the sense of having been deduced from them. The sequent notation used in the rule shown above makes this dependence on assumptions explicit. If $t_1 \supset t_2$ depends on assumptions Γ_1 and t_1 depends on assumptions Γ_2, then the inferred formula t_2 depends on the union of Γ_1 and Γ_2. This notation for inference rules resembles a sequent calculus [26], in which sequents are assertions in the object language. The proof system of [29], however, is essentially natural deduction. Sequents are used merely to keep track of assumptions, and a sequent $\Gamma \vdash t$ should be read as a meta-theorem about provability by natural deduction.

In Gordon's formulation of higher order logic [29] there are five axioms, eight inference rules, and two rules of definition. All the theorems about hardware in this dissertation follow by formal proof using only this primitive logical basis. Fully detailed formal proofs will not be given for these theorems, so a complete list of axioms and inference rules need not be given here. Of particular relevance to later chapters, however, are the mechanisms provided by the primitive basis of higher order logic for making object-language definitions. Definitions of this kind allow the logic to be consistently extended with new notation (in particular, with new constants and types) without postulating *ad hoc* axioms in order to give meaning to this notation. This provides a means for introducing special-purpose notation for hardware verification into the logic in a rigorous and purely formal way. This forms the basis for formalizing the abstraction mechanisms for hardware verification discussed in later chapters. Sections 2.1.4–2.1.7 below describe how these definitional mechanisms are supported by the primitive basis of the logic.

2.1.4 Constant Definitions

The only primitive constants of higher order logic are $=$, $\supset$, and ε. All other constants are introduced by means of the following rule of definition.

CONSTANT DEFINITIONS: If *tm*:*ty* has no free variables, *tm* does not contain the identifier c, there are no type variables in *tm* not also present in ty, and c is not already the name of a constant, then a new constant $\mathsf{c}{:}ty$ can be defined by extending the syntax of the logic to include c as a constant and adding the axiom $\vdash \mathsf{c} = tm$ to the primitive basis of the logic.

The axiom $\vdash \mathsf{c} = tm$ introduced by this rule simply makes the new constant c an object-language abbreviation for the term tm. Adding a new constant by postulating a definitional axiom of this kind is a *conservative extension* of the logic. That is, for any formula t not containing the new constant c being defined, $\vdash t$ is a theorem of the extended logic if and only if it is a theorem of the original logic. In particular, $\vdash \mathsf{F}$ is a theorem of the extended logic if and only if it is a theorem of the original logic. Thus adding axioms that define new constants to the logic will not introduce inconsistency that was not already there; adding definitional axioms is 'safe'.

2.1.4.1 Derived Constant Definitions

The rule for constant definitions described above can be used to justify a derived rule for making definitions of the form

$$\vdash \mathsf{f}\ x_1\ x_2\ \ldots\ x_n = tm \qquad \text{or} \qquad \vdash \mathsf{f}(x_1, x_2, \ldots, x_n) = tm,$$

where all the free variables in tm are included among x_1, x_2, $\ldots$, x_n. For every such definition there is a provably equivalent equation of the more basic kind allowed by the primitive rule for constant definitions.

For example, every theorem of the form $\vdash \mathsf{f}\ x = tm[x]$, where x is the only free variable in $tm[x]$, is equivalent to a corresponding definitional axiom of the form $\vdash \mathsf{f} = \lambda x.\, tm[x]$. An equation of the form $\vdash \mathsf{f}\ x = tm[x]$ can therefore be regarded as a definition of the function constant f, justified by means of a *derived* principle of definition based on the primitive rule for constant definitions. Any defining equation of the two forms shown above can be justified in a similar way. Proofs of these defining equations are straightforward and will therefore be omitted when definitions of this kind are made.

2.1.5 The Primitive Constant ε

The primitive constant $\varepsilon{:}(\alpha{\rightarrow}bool){\rightarrow}\alpha$ is a function that maps predicates on values of type α to values of type α. The semantics of ε can be described informally as follows. If $P{:}ty{\rightarrow}bool$ denotes a predicate on values of type ty, then the application $\varepsilon\ P$ denotes some value of type ty for which P is true. If there is no such value, then $\varepsilon\ P$ denotes some fixed but unknown value of type ty.

This informal semantics is formalized in higher order logic by the single axiom for ε shown below.

$$\vdash \forall P\, x.\, P\, x \supset P(\varepsilon\ P) \tag{2.2}$$

It follows from this axiom that ε can be used to obtain a logical term which provably denotes a value having some property P from a theorem merely stating that such a value exists. Formally, if P denotes a predicate, and $\vdash \exists x.\, P\, x$ is a theorem of the logic, then so is $\vdash P(\varepsilon\ P)$. The only axiom for ε is (2.2), so when $\vdash \exists x.\, P\, x$ holds the only (non-trivial) facts that can be proved about $\varepsilon\ P$ are the logical consequences of $\vdash P(\varepsilon\ P)$. In particular, if more than one value satisfies P, then it is not possible to prove which of these values $\varepsilon\ P$ denotes. And if no value satisfies P, then nothing significant can be proved about $\varepsilon\ P$.

In practice, applications of the form '$\varepsilon\ P$' are usually treated as atomic names for values having the property P, and it is often convenient to abbreviate such ε-terms by defining new constants that denote them. If $P[x]{:}\mathit{bool}$ is a term with no free variables other than x, and $\vdash \exists x.\, P[x]$ is a theorem of the logic, then the equation $\vdash \mathsf{c} = (\varepsilon\ \lambda x.\, P[x])$ defines a constant c such that $\vdash P[\mathsf{c}/x]$. In presenting proofs, this fact will be used as a derived rule of inference to justify omitting the intermediate inference steps needed to introduce a constant c and prove $\vdash P[\mathsf{c}/x]$, given a theorem of the form $\vdash \exists x.\, P[x]$.

2.1.6 Recursive Definitions

In a constant definition, $\vdash \mathsf{c} = tm$, the constant c being defined must not occur in tm on the right hand side of the equation. This restriction rules out the possibility of making inconsistent recursive definitions like $\vdash \mathsf{c} = \neg\, \mathsf{c}$. Constants that satisfy recursive equations are therefore not directly definable by the rule for constant definitions. To define such a constant in higher order logic it is first necessary to prove that the desired recursive equation is in fact satisfiable. The constant can then be defined non-recursively using ε, and the desired recursive equation can be derived from this definition using the method outlined in Section 2.1.5.

Suppose, for example, the aim is to define a constant c such that $\vdash \mathsf{c} = tm[\mathsf{c}]$, where $tm[\mathsf{c}]$ is a term that contains c. To ensure that this equation is consistent, one must first show that it can be satisfied by *some* value. This is done by proving the theorem $\vdash \exists c.\, c = tm[c]$. Using ε, the constant c can then be defined non-recursively by the equation:

$$\vdash \mathsf{c} = (\varepsilon\ \ \lambda c.\, c = tm[c]).$$

Using the axiom for ε, discussed in Section 2.1.5, the desired recursive equation $\vdash \mathsf{c} = tm[\mathsf{c}]$ then follows from this non-recursive definition of c and the previously-proved consistency theorem $\vdash \exists c.\, c = tm[c]$.

2.1.6.1 Primitive Recursive Definitions

An important application of the method described above is in defining constants that denote primitive recursive functions on the natural numbers. Many functions that arise in proofs about hardware are primitive recursive, and constants that denote such functions can be defined formally in higher order logic by means of the *primitive recursion theorem* shown below.

$$\vdash \forall x f.\, \exists fn{:}num{\to}\alpha.\, (fn\ 0 = x) \wedge (\forall n.\, fn\ (\mathsf{Suc}\ n) = f\ (fn\ n)\ n) \tag{2.3}$$

An outline of the proof of this theorem is given by Gordon in [29]. The theorem states the validity of primitive recursive definitions on the natural numbers: for any x and f there exists a corresponding total function $fn{:}num{\to}\alpha$ which satisfies the primitive recursive definition whose form is determined by x and f.

Theorem (2.3) can be used to justify formally the introduction of a constant to denote any particular primitive recursive function. Choosing appropriate values for x and f in (2.3) yields a theorem which asserts the existence of the desired function, and a new constant can then be introduced to denote this function. For example, taking x and f in a suitably type-instantiated version of (2.3) to be $\lambda m.\, m$ and $\lambda f\, x\, m.\, \mathsf{Suc}(f\ m)$ yields, after some simplification, the following theorem.

$$\vdash \exists fn.\, (\forall m.\, fn\ 0\ m = m) \wedge (\forall n\, m.\, fn\ (\mathsf{Suc}\ n)\ m = \mathsf{Suc}(fn\ n\ m))$$

This theorem asserts the existence of a recursively-defined *addition* function on the natural numbers. As discussed in Section 2.1.5, a new constant + can be introduced to denote this function using ε. This yields the theorem

$$\vdash (\forall m.\, +\ 0\ m = m) \wedge (\forall n\, m.\, +\ (\mathsf{Suc}\ n)\ m = \mathsf{Suc}(+\ n\ m)),$$

which states that + satisfies the usual primitive recursive definition of addition. If the constant + is written in infix position, this is equivalent to the two equations shown below.

$$\begin{aligned} &\vdash 0 + m &&= m \\ &\vdash (\mathsf{Suc}\ n) + m &&= \mathsf{Suc}(n + m) \end{aligned} \tag{2.4}$$

In [30], Gordon outlines how recursion equations like these can in general be derived by formal proof from the primitive recursion theorem (2.3). The procedure is straightforward, and the details of justifying primitive recursive definitions will therefore be omitted from proofs presented here. Only the resulting recursion equations will be shown, in the form illustrated by (2.4).

The primitive recursion theorem justifies recursive definitions on the natural numbers only. Recursive functions defined on other logical types (e.g. recursive data types such as lists and trees) are discussed in Chapter 5 and Appendix A.

2.1.7 Type Definitions

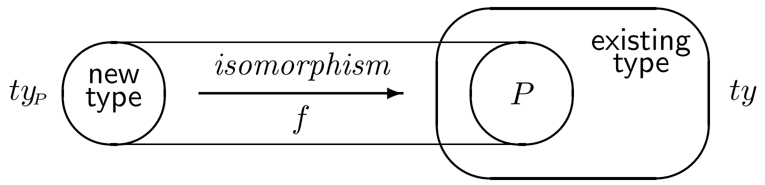
The primary function of types in higher order logic is to eliminate inconsistency. For this purpose, all that is needed are the primitive types of the logic. But there is a pragmatic reason for having a richer syntax of types than is strictly necessary for consistency. Extending the syntax of types in higher order logic allows more natural and concise formulations of propositions about hardware than are possible with only primitive type expressions. For example, introducing new types to name sets of values that arise naturally in specifications of hardware behaviour helps make these specifications clear and concise. This pragmatic motivation for a rich syntax of types is similar to the motivation for the use of abstract data types in high-level programming languages: using higher-level data types reduces complexity by abstracting away from the details of how values are represented.

Sections 2.1.7.1 and 2.1.7.2 below describe a method for extending the logic with new type constants and type operators. This method is based on a formal rule of definition which allows axioms of a restricted form to be added to the primitive basis of the logic. These axioms are analogous to definitional axioms for new constants; they define new types in terms of other type expressions already present in the logic. Like the rule for constant definitions, the rule for type definitions ensures that adding a new type is a conservative extension of the logic.

2.1.7.1 The Rule for Type Definitions

The mechanism for defining logical types described in this section was suggested by Dr M. Fourman and formalized by Gordon in [29]. It is analogous to the mechanism for defining abstract data types in the programming language ML [19]. The basic idea is that a type definition is made by adding an axiom to the logic which asserts that the set of values denoted by a new type is isomorphic to an appropriate subset of the values denoted by a type expression already present in the logic.

Suppose that *ty* is a type of the logic, and $P{:}ty{\to}bool$ is a predicate on values of type *ty* which defines some useful subset of the set denoted by *ty*. A type definition introduces a new type expression ty_P which denotes a set of values having exactly the same properties as the subset defined by P. Formally, this is done by adding an axiom to the logic which states that there is an isomorphism f from the new type ty_P to the set of values that satisfy P:



The function f can be thought of as a representation function that maps each value of the new type ty_P to the value of type ty that represents it. Because f is an isomorphism, it can be shown that the set denoted by ty_P has the same properties as the subset of ty defined by P. The new type ty_P is therefore defined in terms of ty, and its properties are determined by the choice made for the predicate P.

This method is used to define both type constants and type operators. When ty contains no type variables, the new type ty_P being defined is a type constant. When ty contains type variables, the new type ty_P is an expression of the form $(\alpha_1, \ldots, \alpha_n)op$, where α_1, …, α_n are the type variables in ty. In this case, the type definition has the effect of introducing a new n-ary type operator op.

The formal rule of definition for adding new types is shown below. For clarity, the rule is stated for the case when the type ty_P being defined is a type constant. The general rule, which also allows definitions of type operators, is similar. It will not be shown here, but the details can be found in [29,30].

> Type Definitions: If $P{:}ty{\to}bool$ has no free variables, both ty and P contain no type variables, $\vdash \exists x.\, P\, x$ is a theorem of the logic, and ty_P is not already the name of a type constant, then a new type constant ty_P can be defined by extending the syntax of types to include ty_P and adding the axiom
>
> $$\vdash \exists f{:}ty_P{\to}ty.\, (\forall a_1\, a_2.\, f\ a_1{=}f\ a_2 \supset a_1{=}a_2) \land (\forall r.\, P\ r{=}(\exists a.\, r{=}f\ a))$$
>
> to the primitive basis of the logic.

The axiom introduced by this rule simply states that there is an isomorphism from ty_P to the values of type ty that satisfy P. The restriction that P satisfies the theorem $\vdash \exists x.\, P x$ ensures that the defined type constant ty_P denotes a non-empty set. This restriction is necessary because all type expressions in the logic must denote non-empty sets.

2.1.7.2 Deriving Abstract Axioms for New Types

A type definition of the form described above merely states that a new type is isomorphic to a particular subset of an existing type. From such type definitions, it is possible to derive theorems that characterize new types more abstractly. The idea is to prove a collection of theorems that state the essential properties of a new type without reference to how it is defined. These theorems constitute a derived 'axiomatization' of the new type, and once they have been proved they become the basis for all further reasoning about it.

With this approach, introducing a new type (or type operator) in higher order logic involves two steps:

1. Find an appropriate representation for the new type, and make a type definition based on this representation.

2. Use the definition of the new type and the properties of its representation to prove a set of theorems that abstractly characterizes it.

The motivation for first defining a new type and then deriving abstract 'axioms' for it is that this process guarantees consistency. Simply postulating plausible-looking axioms to express the properties of a new type can inadvertently make the logic inconsistent.[1] But deriving abstract 'axioms' from a type definition amounts to giving a formal proof of their consistency—by showing that there is a model for them—and this process avoids the potential for inconsistency associated with postulating *ad hoc* axioms to describe new types.

The usual axioms for the cartesian product type $\alpha\times\beta$ provide a simple example of the result of this two-step process. The essential properties of this type are captured formally by the three theorems shown below.

$$\vdash \forall a{:}\alpha.\, \forall b{:}\beta.\ \mathsf{Fst}(a,b) = a$$
$$\vdash \forall a{:}\alpha.\, \forall b{:}\beta.\ \mathsf{Snd}(a,b) = b$$
$$\vdash \forall p{:}(\alpha\times\beta).\, p = (\mathsf{Fst}\ p, \mathsf{Snd}\ p)$$

These three theorems can be derived by formal proof from an appropriate type definition for the type operator $\times$ and suitable definitions of the three constants ',' (the infix pairing operator), Fst, and Snd. All the usual properties of pairs follow from these theorems, and once they have been proved it becomes unnecessary to know how the type $\alpha\times\beta$ was represented and defined.

All the non-primitive types and type operators used in this dissertation have been defined formally (by the author) in the HOL system using the primitive rule of definition described in Section 2.1.7.1 and the two-step 'methodology' discussed above. Full details of the formal definitions for these types (which include the basic types *num* and $\alpha\times\beta$), and outline proofs of the abstract 'axiomatizations' for them, are given in Appendix A.

2.2 The HOL System

The HOL system [30] is a mechanized proof-assistant developed by Gordon at the University of Cambridge for conducting proofs in the formulation of higher order logic described in the previous section. It has been used primarily to reason about the correctness of hardware, but much of what has been developed in HOL for hardware verification—the theory of arithmetic, for example—is also fundamental to many other applications. The underlying logic and basic facilities of the system are completely general and can in principle be used to support reasoning in any area that can be formalized in higher order logic.

[1]The axioms for the type $(\alpha)list$ in Gordon's definition of higher order logic [29] illustrate this danger. These list axioms are inconsistent—and this seems to have survived notice since [29] first appeared in 1985.

HOL is based on the LCF approach to interactive theorem proving[2] and has many features in common with the LCF systems developed at Cambridge [74] and Edinburgh [35]. Like LCF, the HOL system supports secure theorem proving by representing its logic in the strongly-typed functional programming language ML [19]. Propositions and theorems of the logic are represented by ML abstract data types, and interaction with the theorem prover takes place by executing ML procedures that operate on values of these data types. Because HOL is built on top of a general-purpose programming language, the user can write arbitrarily complex programs to implement proof strategies. Furthermore, because of the way the logic is represented in ML, such user-defined proof strategies are guaranteed to perform only valid logical inferences.

2.2.1 The Representation of Higher Order Logic in ML

ML is a strongly-typed language. All expressions in the language have types, and only consistently-typed expressions are syntactically well-formed. The syntax of types in ML resembles that of types in higher order logic. For example, the ML type `int->bool` is the type of functions (i.e. functional programs) that take an integer as a parameter and return a boolean as a result. The rules for type-checking ML expressions are similar to the rules for well-typed logical terms given on page 16. An ML function call '`f(x)`', for example, will be accepted by the type-checker only if there are ML types ty_1 and ty_2 such that the function `f` has ML type ty_1`->` ty_2 and the value `x` has ML type ty_1.

This type discipline is the basis for the soundness of proofs in the HOL system. HOL is built on top of ML by extending the set of built-in ML data types with a special-purpose abstract data type `thm`, values of which are *theorems* of higher order logic. There are no literals of type `thm`. That is, it is not possible to obtain a value of type `thm` simply by typing one in. There are, however, certain ML identifiers which are given values of type `thm` when the system is built. These values correspond to the axioms of higher order logic. In addition, HOL makes available several built-in ML functions that take theorems as arguments and return theorems as results. Each of these corresponds to one of the primitive inference rules of the logic and returns only theorems that logically follow from its input theorems using the corresponding inference rule. The ML type checker ensures that values of type `thm` can be generated only by applying these functions either to previously-generated values of type `thm`, or to the values of type `thm` that represent axioms. Every value of type `thm` must therefore either be an axiom or have been obtained by computation using the functions that represent the primitive

[2]The current implementation of HOL is a modified version of Cambridge LCF [74], which is itself a development of the Edinburgh LCF system [35]. The basic approach to mechanized theorem proving used in all these systems is due to Milner.

inference rules of the logic—i.e. every theorem in HOL must be generated from the axioms using the inference rules. In this way, the ML type checker guarantees the soundness of the HOL theorem prover: a theorem can be generated in the system only by valid formal proof.

In addition to the primitive inference rules, there are many *derived inference rules* available in HOL. These are ML procedures which perform commonly-used sequences of primitive inferences by calling the appropriate ML functions which represent the primitive inference rules. Derived inference rules relieve the HOL user of the need to give explicitly all the primitive inference steps of a proof. The ML code for a derived rule can be arbitrarily complex. But it will never return a theorem that does not follow by valid logical inference, since a theorem can be obtained only by a series of calls to the primitive inference rules.

2.2.2 Interactive Proof in HOL

HOL supports two styles of interactive proof: *forward* proof and *backward* proof. In the forward style, inference rules are simply applied in sequence to previously proved theorems until the desired theorem is obtained. The user specifies which rule is applied at each step of the proof, either interactively or by writing an ML program that calls the appropriate sequence of procedures. This is usually not the easiest way of doing a proof in the system, since the exact details of a proof are rarely known in advance.

It is often simpler to find the proof by working backwards from the statement to be proved (called a *goal*) to previously proved theorems which imply it. This is the backward, or goal-directed, proof style. The HOL system, following LCF, supports this style of proof by means of ML functions called *tactics*. These break goals down into increasingly simple subgoals—until the subgoals obtained can be proved directly from theorems already derived. Again, the user specifies which tactic to use at each step. In addition to breaking a goal down into subgoals, a tactic also constructs a sequence of forward inference steps which can be used to derive the goal, once the subgoals have themselves been proved. This is necessary because all theorems in the system must ultimately be obtained by forward proof. This approach to proof using tactics is due to Milner. It is described in detail in [30,35,74].

2.2.3 Efficiency in HOL

The LCF approach to theorem proving used in HOL ensures the soundness of any proof done in the system. This approach, however, is computationally very expensive. Completely formal proofs of even simple theorems in higher order logic can take thousands of primitive inferences, and when these proofs are done in the

HOL system, all the inferences involved must actually be carried out by executing the corresponding ML procedures.

There are, however, two important features of the HOL system which, together, allow *efficient* proof strategies to be programmed. The first of these is a feature inherited from LCF: theorems proved in HOL (or LCF) can be saved on disk and therefore do not have to be generated each time they are needed in future proofs. The second feature is the expressive power of higher order logic itself, which allows useful and very general 'lemmas' to be stated in the logic. The amount of inference that a programmed proof rule must do can therefore be reduced by pre-proving general theorems from which the desired results follow by a relatively small amount of deduction. These theorems can then be saved and used by the derived inference rule in future proofs. This strategy of replacing 'run time' inference by pre-proved theorems is possible in HOL because type polymorphism and higher-order variables make the logic expressive enough to yield theorems of sufficient generality.

This simple strategy for making derived rules efficient is illustrated by the method for automating recursive type definitions described in Appendix A. In Section A.5.4, a theorem is described from which an abstract 'axiomatization' for any concrete recursive type can be deduced with relatively little inference.

Chapter 3

Hardware Verification using Higher Order Logic

This chapter describes the basic techniques for using higher order logic to specify hardware behaviour and to prove the correctness of hardware designs.

The advantages of higher order logic as a formalism for hardware verification are discussed by Gordon in [34] and by Hanna and Daeche in [43,44]. Higher order logic makes available the results of general mathematics, and this allows the construction of any mathematical tools which are needed for the verification task in hand. Its expressive power permits hardware behaviour to be described directly in logic; a specialized hardware description language is not needed. In the formulation used here, new constants and types can be introduced by means of the definitional mechanisms described in Chapter 2. This allows special-purpose notation for hardware verification to be introduced as a conservative extension of the logic, without the need to postulate *ad hoc* axioms. In addition, the inference rules of the logic provide a secure basis for proofs of correctness. A specialized deductive calculus for reasoning about hardware behaviour is not required.

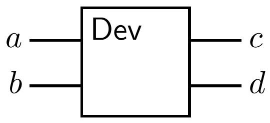
An overview of formal proof as a method for demonstrating the correctness of hardware was given in Section 1.1.1 of the introductory chapter. This chapter outlines how higher order logic supports the approach described there. Section 3.1 describes how the behaviour of hardware is specified in the notation of higher order logic. Section 3.2 describes how a specification of the behaviour of a composite device is constructed from the specifications of its parts. Section 3.3 describes a simple and direct approach to formulating the correctness of hardware designs in logic. In Section 3.4, an example is given to illustrate this approach. The chapter concludes with an account of related work based on formal logic.

3.1 Specifying Hardware Behaviour

The approach to specifying hardware behaviour described in this section is well-known (see, for example, [11,34,43,51]). The basic idea is to specify the behaviour of a hardware device by stating which combinations of values can be observed on its external wires. Such a specification is expressed formally in logic by a boolean-valued term with free variables that correspond to these external wires.

This term imposes a constraint on the values of these free variables. To reflect the behaviour of the device it specifies, the term is chosen such that the combinations of values that satisfy this constraint are precisely those which can be observed simultaneously on the corresponding external wires of the device itself.

For example, consider the device Dev shown below.



This device has four external wires: a, b, c, and d. A formal specification of its behaviour in logic is a boolean-valued term of the form $tm[a, b, c, d]$. This term is constructed such that for all values of the free variables a, b, c, and d:

$$tm[a,b,c,d] = \begin{cases} \mathsf{T} & \text{if the values denoted by } a,\ b,\ c, \text{ and } d \text{ could occur simultaneously on the corresponding external wires of } \mathsf{Dev} \\ \mathsf{F} & \text{otherwise} \end{cases}$$

There is no restriction on the form that the specification $tm[a, b, c, d]$ must take. Any mathematical concepts and notation needed to describe the behaviour of Dev may be used, provided they can be defined formally. Since higher order logic is a formalism intended as a foundation for mathematics, it has the advantage that it is possible (in principle) to define in the logic the mathematical tools required to describe the behaviour of any particular device.

This approach to specifying hardware in logic describes its behaviour only in terms of the values that can be observed externally. No information about internal state is used in a specification. Furthermore, there is no distinction between the inputs and the outputs of a device—the constraint imposed by a specification on its free variables need not be a functional one. Both specifications of the hardware primitives used in designs and specifications of the intended behaviour of designs can be expressed formally in logic by this method. Such specifications can describe either purely combinational behaviour or time-dependent (sequential) behaviour. These two possibilities are discussed below in Sections 3.1.2 and 3.1.3. This approach can also be used to write formal specifications that only partially describe the behaviour of a device. This is discussed in Section 3.1.4. A simple method for abbreviating specifications is first introduced in Section 3.1.1.

3.1.1 Abbreviating Specifications

The rule for constant definitions discussed in Section 2.1.4 of Chapter 2 provides a formal mechanism for abbreviating hardware specifications of the kind described above. Such a specification is just a boolean-valued term with free variables, and an object-language abbreviation for it can be introduced simply by defining a predicate constant to name the constraint it imposes on these variables.

For example, the specification of the device shown in the previous section can be abbreviated by means of the predicate Dev which is defined formally by:

$$\vdash \mathsf{Dev}(a, b, c, d) = tm[a, b, c, d]$$

This introduces a new predicate constant 'Dev' into the logic and makes the logical term '$\mathsf{Dev}(a, b, c, d)$' an object-language abbreviation for the specification $tm[a, b, c, d]$. The constant Dev itself is an atomic name for a class of devices, each of which exhibits the same sort of behaviour as the others but has differently labelled external wires. Any particular device in this class is specified by an application of Dev to an appropriate 4-tuple of variables; for example, the term $\mathsf{Dev}(w, x, y, z)$ specifies a device with external wires named w, x, y, and z. This can be viewed as providing an object-language notation for generic, 'parameterized' specifications of hardware behaviour.

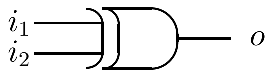
An alternative way of abbreviating the term $tm[a, b, c, d]$ is by introducing the defining equation:

$$\vdash \mathsf{Dev}\ a\ b\ c\ d = tm[a, b, c, d]$$

This makes the constant Dev a higher order function, rather than predicate on 4-tuples. This higher order form of defining equation has the advantage that the function defined by it need not be applied to all four variables a, b, c, and d at once. For example, the term '$\mathsf{Dev}\ a$' is a well-formed 'partial application' of Dev to the variable a only.

3.1.2 Specifying Combinational Behaviour

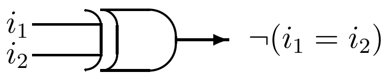
The most direct application of the hardware specification method discussed above is in describing the purely combinational behaviour of hardware devices. Here, a highly simplified view is taken of hardware behaviour: only the static behaviour of a device is specified, and the possibility that its behaviour may change over time is ignored. An example is the combinational specification of an exclusive-or gate shown below.



$$\vdash \mathsf{Xor}(i_1, i_2, o) = (o = \neg(i_1 = i_2))$$

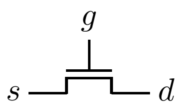
In this specification, the values T and F are used to represent the two logic levels 'true' and 'false', and the variables i_1, i_2, and o range over these two boolean truth-values. The term $\mathsf{Xor}(i_1, i_2, o)$ describes a relationship between these variables which corresponds to the way an exclusive-or gate works in practice: the output o is true exactly when either i_1 or i_2 is true but not both.

The output of the exclusive-or gate shown above is a simple function of its two inputs. The full generality of the 'relational' method of specifying behaviour is therefore not needed; the combinational behaviour of this gate could be specified equally well by the function 'Xor' defined below.



$\vdash \mathsf{Xor}(i_1, i_2) = \neg(i_1 = i_2)$

In general, however, a hardware device may have bidirectional external wires, used for both input and output. And one advantage that relational specifications have over functional ones is that they directly support formal specifications of such devices. This is illustrated by the N-type transistor and its formal specification shown below.



$\vdash \mathsf{Ntran}(g, s, d) = (g \supset (d = s))$

In this specification, the source s and the drain d of the transistor are bidirectional. It follows from the specification that if the gate g has the value T then s and d must have the same boolean value. But the direction of signal flow (if any) between s and d is not expressed by the term '$\mathsf{T} \supset (d = s)$', which merely states that the values on s and d must be equal.

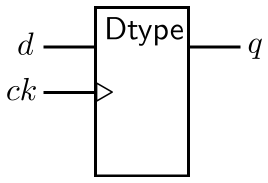
Because relational specifications do not distinguish between inputs and outputs, they can be interpreted in ambiguous ways. For example, Hoare [51] has pointed out that the transistor specification shown above might lead one to conclude that if opposite values are supplied to the source and drain then this will cause the gate to have the value F. The exclusive-or specification defined above also admits of ambiguous interpretation. It can be viewed as the specification of a gate with inputs i_1 and i_2, and output o. But it can also be viewed as the specification of an exclusive-or gate with inputs i_1 and o—and output i_2. The purely logical properties of the term '$\mathsf{Xor}(i_1, i_2, o)$' are appropriate to both interpretations.

This problem of ambiguous interpretation is partly due to the inadequacy of combinational specifications in general. Real hardware has delay: a change of input values takes time to produce a change of output. But a combinational specification presents only a static view of hardware behaviour, and it cannot suggest the temporal relationship between a change of input and a change of output. For this, another way of describing behaviour is needed.

3.1.3 Specifying Sequential Behaviour

The sequential behaviour of hardware devices can be specified in logic by using functions to describe the sequences of values that appear on their external wires at successive moments of time. With this approach, time is represented by the natural numbers, which in higher order logic are denoted by the defined logical type *num*. The sequence of values that appears on a wire is represented by a function $f{:}num{\to}bool$, so that the value present on the wire at any particular time t is given by the application $f(t)$. A specification of the time-dependent behaviour of a device is then a constraint on variables that range over such functions.

For example, the behaviour of a rising-edge triggered D-type flip flop can be specified in logic by the term $\mathsf{Dtype}(ck, d, q)$ defined, together with an auxiliary function Rise, as shown below.



$$\vdash \mathsf{Rise}\ ck\ t = \neg ck(t) \wedge ck(t{+}1)$$

$$\vdash \mathsf{Dtype}(ck, d, q) = \forall t.\, q(t{+}1) = (\mathsf{Rise}\ ck\ t \Rightarrow d(t) \mid q(t))$$

In this specification, instants of discrete time are represented by natural numbers, and the variables *ck*, *d*, and *q* range over functions of logical type $num{\to}bool$. The term $\mathsf{Dtype}(ck, d, q)$ specifies the sequential behaviour of a D-type flip-flop by imposing constraints on the functions *ck*, *d*, and *q*. Whenever the clock *ck* rises, the value on the input *d* is sampled and appears on the output *q* one time unit later. When the clock does not rise, the output *q* remains stable over time.[1]

This example illustrates why it is advantageous to use *higher order* logic, rather than first order logic, to specify hardware behaviour. In general, a specification of sequential behaviour is a term that impose constraints on higher order variables. In the D-type specification, for example, the variables *ck*, *d*, and *q* are higher order variables. They range over functions of logical type $num{\to}bool$. The constant Rise is also higher order. It denotes a function which both takes a function as an argument and yields a function as a result. Higher order logic directly supports these higher order entities, and this expressive power allows natural and direct specifications of sequential behaviour in logic.

3.1.4 Partial Specifications

A partial specification is one that does not completely describe the behaviour of a hardware device, but only selected aspects of it. The ability to write such specifications is essential if formal verification is to be applied to very large or complex devices. To be intelligible, the formal specification of intended behaviour for such a device must concentrate on only the essential features of it—a complete description may be too complex.

[1]This is, of course, a highly simplified view of the behaviour of a D-type flip flip. More realistic formal specifications of this device can be found in [34,44,48,79].

Partial specifications of behaviour can be expressed in logic in a natural and direct way. Specifications are just terms that describe constraints on the values that can appear on the external wires of a device. A partial specification simply constrains these values only in the situations that are significant or relevant. In all other situations, it leaves them unconstrained. The D-type flip flop specification shown above is a simple example. The equation for the output wire q in this specification constrains the value of $q(t{+}1)$ for all t, but the value of $q(0)$ is left unconstrained. This specification is therefore only a partial specification of behaviour: the value of the output q at 'time zero' is not specified.

This D-type example is a somewhat special case: the output q is unspecified only at one particular point in time. A more general application of partial specifications is illustrated by the specification shown below.

$$\vdash \mathsf{Dev}(i, o) = (\mathsf{P}\ i \supset (o = \mathsf{f}\ i))$$

Here, a partial specification is used to leave undefined the value on the output wire o for certain values on the input wire i. The term '$\mathsf{P}\ i$' is a condition on this input value, and the output o is specified only for inputs that satisfy this condition. This is reflected formally in the specification by the fact that if $\mathsf{P}\ i = \mathsf{F}$ then the term '$\mathsf{P}\ i \supset (o = \mathsf{f}\ i)$' is satisfied for any value o.

This is a commonly-used technique for writing partial specifications that define the behaviour which a device is required to exhibit only for selected input values (examples can be found in [34,44,48,55,63]). Such specifications are appropriate when it is known that the device will be used in an environment where only these input values arise, and it is therefore unnecessary to specify its required behaviour for all input values.

3.2 Deriving Behaviour from Structure

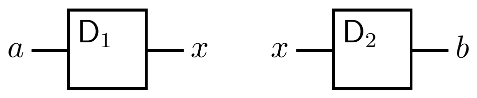
To prove the correctness of a composite hardware device—a device built by wiring together components—a method is needed for constructing a formal description of its behaviour in logic. This is done by first writing a formal specification for each kind of primitive hardware component used in its design. Instances of these specifications are then combined syntactically to obtain a logical term that describes the net behaviour of the entire device. This term is called a *model* of the composite device.

Two syntactic operations on terms, called *composition* and *hiding*, are used in constructing models. These operations represent two ways in which a physical device can be constructed from its parts: composition represents the operation of wiring parts together, and hiding represents the operation of 'insulating' wires from the environment. A model is constructed syntactically by applying these two operations to the logical terms that describe the constituent parts of a device.[2] The next two sections describe how these operations can be represented in higher order logic. The techniques described in these sections are well known and widely used (see, for example, [11,34,43,51]).

3.2.1 Composition

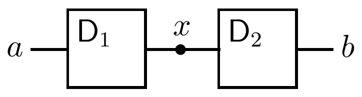
Composition models the effect of joining two devices together by connecting them at all identically-labelled external wires. Syntactically, composition is done simply by forming the *conjunction* of the logical terms which specify the devices that are connected together.

For example, suppose that the two devices D_1 and D_2 are specified by the boolean terms $tm_1[a,x]$ and $tm_2[x,b]$ respectively, as shown below.



$tm_1[a,x]$ $\qquad$ $tm_2[x,b]$

The two terms $tm_1[a,x]$ and $tm_2[x,b]$ describe the values that can be observed independently on the external wires of the devices D_1 and D_2. If these two devices are connected together by the wire x, the values that can be observed on the external wires of resulting composite device are just those that can be observed simultaneously on the wires of *both* its components. A model of the resulting behaviour is given by the logical conjunction of the terms which specify these components:



$tm_1[a,x] \wedge tm_2[x,b]$

The result is a term with three free variables: a, x, and b. This term constrains the values on the wires of the composite device to be exactly those allowed by the constraints imposed by both $tm_1[a,x]$ and $tm_2[x,b]$.

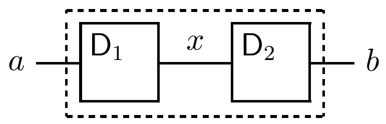
3.2.2 Hiding

In general, a model constructed by composition may have free variables which correspond to wires that are used only for internal communication between the components of a device. Hiding models the effect of insulating these wires from

[2]This approach is based on the algebraic method of modelling concurrent systems, originally proposed by Milne and Milner in [67].

the environment, making them internal to the device. Syntactically, hiding is done by *existentially quantifying* the free variables in a term which correspond to internal wires. This results in a term in which these variables are bound—and therefore no longer represent external wires of a device.

Consider, for example, the term $tm_1[a, x] \wedge tm_2[x, b]$ which describes the composite device shown in the previous section. Suppose that the wire which corresponds to the free variable x is used for internal communication only. A model in which this wire is internal to the device, hidden from the environment, is given by existentially quantifying over the variable x:



$$\exists x.\, tm_1[a, x] \wedge tm_2[x, b]$$

The result is a term in which only the variables a and b, corresponding to the external wires of the device, are free. This term expresses the constraint that two values a and b can be observed on the external wires of the device exactly when there is *some* internal value x such that the constraints for the components of the device are satisfied.

3.2.3 A Note on Terminology

A specification constructed by the methods described in the preceding sections is called a *model* of the device it describes. In general, what is meant by a 'model' is a logical expression which describes the behaviour of a particular hardware device. Such an expression 'models' the behaviour of a device, in the sense that its purely formal properties are intended to reflect at least some aspects of how the device really behaves. For example, term '$\mathsf{Dtype}(ck, d, q)$' defined in Section 3.1.3 is a formal model of the behaviour of a D-type flip-flop. A term constructed by composition and hiding from the specifications of the parts used in a design is also referred to as a 'model'. Such a term models the behaviour of a particular composite hardware device.

In what follows—and especially in Chapters 4 and 7—the term 'model' is also sometimes used to mean a *collection* of formal specifications which describe the primitive components used in hardware designs. For example, a collection of specifications for the primitive components used in building CMOS circuits (i.e. transistors) will be called a CMOS 'transistor model'. Such a model consists of a particular choice of specifications for the primitive hardware components used in all CMOS designs. A model, in this sense of the word, provides the *basis* for constructing descriptions of particular composite devices. The behaviour of any particular CMOS circuit, for example, can be described by a term constructed from the primitive specifications that constitute a CMOS transistor model.

Both senses of the word 'model' are used in later chapters (and the remaining sections of this chapter), but the sense in which the term 'model' is used is always indicated explicitly when it may not be clear from the context in which it occurs.

3.3 Formulating Correctness

Once a model of a device has been constructed by the method discussed above, the correctness of the device can be expressed by a proposition which asserts that this model in some sense 'satisfies' an appropriate specification of required behaviour. The most direct way of formulating this satisfaction relationship is by logical *equivalence.* With this formulation, the correctness of a hardware design is asserted by a theorem of the form:

$$\vdash M[v_1, \ldots, v_n] = S[v_1, \ldots, v_n]$$

where the term $M[v_1, \ldots, v_n]$ is the model of the device which is asserted to be correct, and the term $S[v_1, \ldots, v_n]$ is the specification of required behaviour. This theorem states that the truth-values denoted by these two terms are the same for any assignment of values to the free variables v_1, …, v_n. This means that the design is clearly 'correct' with respect to this specification, since the behaviour described by the model is *identical* to that expressed by the specification.

Formulating correctness by an equivalence of this kind is usually appropriate only for small or relatively simple hardware designs. For more complex devices, it is usually impractical to formulate correctness in this way. Indeed, for all but the simplest devices, it is clear that satisfaction *must not* be logical equivalence. If correctness is formulated by an equivalence of the form shown above, then the specification $S[v_1, \ldots, v_n]$ must denote the same constraint on the free variables v_1, …, v_n as the model $M[v_1, \ldots, v_n]$ does. But if the device to be proved correct is large or complex, and if the model of the device is at all realistic, then both the model and *any logically equivalent specification* are likely to be large and complex as well.[3] This means that the specification of intended behaviour may be too complex to be seen to reflect the designer's intent—the correctness of the *specification* may be no more obvious than the correctness of the design itself.

For the specification of a complex device to be intelligible to the designer, it must generally be limited to a more abstract view of its behaviour than is given by a detailed model of its design. The relationship of satisfaction which is used to express correctness must therefore, in general, be one of *abstraction*, rather than

[3]The specification may, of course, be expressed in a more concise notation than the model, but the perspicuity obtainable in this way is limited.

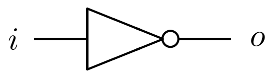
strict equivalence. The formalization of this notion of correctness is discussed in detail in Chapter 4, and in the chapters that follow.

3.4 An Example Correctness Proof

In this section, a very simple example is given to illustrate the basic approach to verification introduced above. The example is the formal verification of the standard CMOS circuit design for a inverter. The purpose of this example (which has been taken from [11,34]) is to provide a very simple preliminary illustration of hardware verification using higher order logic, and it is not suggested that the particular correctness result demonstrated here has any significant practical value. More realistic examples are given in later chapters.

3.4.1 The Specification of Required Behaviour

The first step in the verification of an inverter is to write a formal specification of required behaviour for the design. A specification of the combinational behaviour which a correctly implemented inverter is required to exhibit is given by the term '$\mathsf{Not}(i, o)$' defined below.

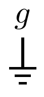


$$\vdash \mathsf{Not}(i, o) = (o = \neg i)$$

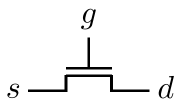
This specification simply asserts that the boolean value on the output o must be the negation of the value on the input i.

3.4.2 Specifications of the Primitive Components

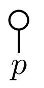
The formal specifications shown in Figure 3.1 describe the four different kinds of primitive components used in CMOS circuit designs. The terms $\mathsf{Pwr}\ p$ and $\mathsf{Gnd}\ g$ specify the behaviour of power (VDD) and ground (VSS) nodes respectively. The specifications $\mathsf{Ntran}(g, s, d)$ and $\mathsf{Ptran}(g, s, d)$ specify the behaviour of N-type and P-type transistors. These are modelled as ideal switches which are controlled by the boolean values present on their gates. For example, $\mathsf{Ntran}(g, s, d)$ acts as an



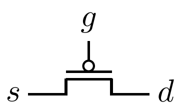
$\vdash \mathsf{Gnd}\ g = (g = \mathsf{F})$



$\vdash \mathsf{Ntran}(g, s, d) = (g \supset (d = s))$



$\vdash \mathsf{Pwr}\ p = (p = \mathsf{T})$



$\vdash \mathsf{Ptran}(g, s, d) = (\neg g \supset (d = s))$

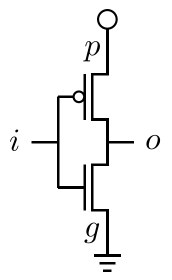
Figure 3.1: CMOS Primitives.

ideal switch which is closed when g=T and open when g=F. This is, of course, a highly simplified model of CMOS transistor behaviour.

These four specifications constitute a model of CMOS designs in general, in the sense that a description of the behaviour of any particular CMOS circuit can be constructed from instances of these primitives using composition and hiding. This is an example of a 'model' in the *second* sense discussed above in Section 3.2.3. That is, the specifications shown in Figure 3.1, together with the operations of composition and hiding, form the basis for constructing a formal description of any particular circuit design.

3.4.3 The Design Model

Given the CMOS primitives defined in the previous section, a model '$\mathsf{Inv}(i, o)$' of the behaviour of a CMOS inverter can be constructed using the operations of composition and hiding discussed in Section 3.2.



$$\vdash \mathsf{Inv}(i, o) = \exists g\, p.\, \mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g \wedge \mathsf{Ntran}(i, g, o) \wedge \mathsf{Ptran}(i, p, o)$$

Figure 3.2: A Formal Model of a CMOS Inverter.

The definition of $\mathsf{Inv}(i, o)$ is shown in Figure 3.2. The model is constructed using logical conjunction '$\wedge$' to compose the specifications which describe the four constituent parts of a standard CMOS inverter. The variables p and g in the definition represent wires which are internal to the inverter's design. They are therefore 'hidden' from the environment using the existential quantifier '$\exists$'. The net effect is that the term $\mathsf{Inv}(i, o)$ is satisfied precisely when the values i and o satisfy the constraint imposed by the specifications of the parts used in the design, for some internal values p and g.

3.4.4 The Proof of Correctness

The correctness of the inverter design is expressed formally by the theorem of higher order logic shown below:

$$\vdash \mathsf{Inv}(i, o) = \mathsf{Not}(i, o)$$

Here, the satisfaction relation between the design model and the specification is simply logical equivalence—the design of an inverter is easily simple enough for correctness to be formulated by equivalence. The correctness theorem shown above states that the behaviour described by the model '$\mathsf{Inv}(i,o)$' exactly matches the required behaviour which expressed by the specification '$\mathsf{Not}(i,o)$'. The CMOS circuit shown in Figure 3.2 therefore correctly implements this required behaviour.

An outline of the formal proof of this correctness theorem is given below. Each step in this proof can be justified formally using only the inference rules of higher order logic, but only an informal sketch of the proof is given here. A completely formal proof of this particular correctness result is trivial to generate in the HOL theorem proving system.

Proof Outline:

1. The definition of the constant Inv is:

   $$\vdash \mathsf{Inv}(i,o) = \exists g\, p.\ \mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g \wedge \mathsf{Ntran}(i,g,o) \wedge \mathsf{Ptran}(i,p,o)$$

2. Expanding with the definitions of Pwr, Gnd, Ntran, and Ptran yields:

   $$\vdash \mathsf{Inv}(i,o) = \exists g\, p.\ (p = \mathsf{T}) \wedge (g = \mathsf{F}) \wedge (i \supset (o = g)) \wedge (\neg i \supset (o = p))$$

3. By the meta-theorem $\vdash (\exists v.\ v{=}tm \wedge t[v]) = t[tm/v]$ this is equivalent to:

   $$\vdash \mathsf{Inv}(i,o) = (i \supset (o = \mathsf{F})) \wedge (\neg i \supset (o = \mathsf{T}))$$

4. Simplifying using the laws $\vdash (o = \mathsf{F}) = \neg o$ and $\vdash (o = \mathsf{T}) = o$ gives:

   $$\vdash \mathsf{Inv}(i,o) = (i \supset \neg o) \wedge (\neg i \supset o)$$

5. Replacing $i \supset \neg o$ by its contrapositive $o \supset \neg i$ yields:

   $$\vdash \mathsf{Inv}(i,o) = (o \supset \neg i) \wedge (\neg i \supset o)$$

6. By the definition of boolean equality, this is equivalent to:

   $$\vdash \mathsf{Inv}(i,o) = (o = \neg i)$$

7. Abbreviating the right hand side using the definition of Not gives:

   $$\vdash \mathsf{Inv}(i,o) = \mathsf{Not}(i,o)$$

Steps 1–3 of this proof illustrate a procedure which is commonly used in proofs of hardware correctness. This consists of first expanding with the definitions of the parts used in the model, and then eliminating the equations for internal wires of the device using the meta-theorem shown in step 3. For other examples of the use of this technique, see [11,34,63]. Step 4 of the proof makes use of the fact that formulas in higher order logic are simply boolean terms. Steps 5–7 amount to an elementary proof in standard propositional calculus.

This example, although trivial in itself, illustrates the general approach to proving a hardware design correct in higher order logic. First, a specification of intended behaviour is written. A specification is then written for each different kind of primitive device used in the design, and instances of these specifications are composed to obtain a formal model of the design. Finally, a theorem is proved which asserts that this model in some sense 'satisfies' the specification of required behaviour. In the example given above, this satisfaction relation between the model and the specification is just logical equivalence. Formal mechanisms for expressing satisfaction by a relationship of abstraction, rather than simply by equivalence, are discussed in detail in the next chapter.

3.5 Related Work

Various approaches to hardware design verification have been proposed based on specification and proof in formal logic. These include methods based on first order logic, higher order logic, and temporal logic. A brief outline of some of this work is given below.

Early Work using First Order Logic

Early work in applying first order logic to hardware verification was done by T.J. Wagner at Stanford. In [83] Wagner uses a mechanized proof checker for first order logic to prove the correctness of clocked sequential circuits. The approach is based on an *ad hoc* axiomatization in logic of an algebra of signal transitions. Device specifications are conjunctions of first order terms that describe conditional register transfer operations. A similar approach, based on resolution and a clausal form for conditional assignments to registers, is discussed by Wojcik in [87].

The Boyer-Moore Logic and Theorem Prover

A notable application of first order logic is W. Hunt's verification of a 16-bit microprocessor using the Boyer-Moore theorem prover [9,53]. The Boyer-Moore logic is quantifier-free first order logic with equality and axiom-schemes of induction. Hunt describes hardware behaviour in this logic using a functional approach: the behaviour of a hardware device is modelled by a function from inputs to outputs. Sequential behaviour is described by recursive functions defined on lists. These lists represent sequences of discrete moments in time. Specifications of required behaviour are also functions, and the correctness of designs is formulated by function equality. Structural induction on lists is used to prove the equality of functions that describe sequential behaviour. All the proofs in the microprocessor verification were performed using the Boyer-Moore theorem prover.

Conlan CHDL's and First Order Theories

Eveking [23,24] uses the standard notion of a *theory* [47] to formalize hardware descriptions in first order logic. This approach was developed in connection with the CONLAN project [75], which provides a family of computer hardware description languages (CHDL's) for describing digital systems. Eveking's approach is to associate a first order theory with each hardware description written in one of these languages.

Such a theory consists of the predicate calculus augmented with a particular collection of extra constants and axioms. These axioms are of two kinds: CHDL-specific axioms, and description-specific axioms. CHDL-specific axioms simply describe the various constructs of the CHDL in which the hardware description is written. These axioms are analogous to the theorems of higher order logic which characterize the defined entities used in specifications of hardware (e.g. the defined type *num* and the defined constants $+$, $\times$, etc.).

The description-specific axioms of a theory formalize the particular hardware description associated with it. Taken as a group, these axioms are analogous to a model (in the sense of a formal description of an individual hardware device) in the higher-order approach described in this chapter. Each axiom states which values can be observed at certain points in a circuit. A typical example (taken from [23]) is shown below.

$$\vdash \forall t.\ ((0<t) \wedge (\mathsf{a}(t-1) = \mathsf{T})) \supset (\mathsf{x}(t) = \mathsf{y}(t-1))$$

The functions a, x, and y in this axiom represent the sequences of values that can be observed over time at three particular points in a circuit. With the higher-order approach described in Section 3.1.3, these sequences would be represented by higher order variables ranging over functions. With Eveking's first order approach, however, these sequences are represented by function *constants*. The description-specific axiom shown above characterizes the relationship which holds between the values observed at the three points of the circuit represented by these three function constants. In Eveking's work, both specifications of required behaviour and models of hardware designs are theories which contain description-specific axioms of this kind.

Eveking formulates design correctness using the notion of one theory being an *extension* of another.[4] A theory T_2 is an extension of a theory T_1 if every axiom of T_2 is a theorem of T_1. The correctness of a device is demonstrated formally

[4]The notion of *interpreting* one theory in another is also used to formulate correctness. A discussion of this way of formulating the correctness of designs is deferred until Section 4.3.

by showing that a theory describing the behaviour which the device is required to exhibit is an extension of a theory which models the actual behaviour of the device itself. This is similar to formulating correctness in higher order logic by an implication of the form $\vdash M \supset S$, where M is the model of a design and S is the specification of required behaviour.

As presented in [23], the approach described above is based on *first order* logic only. Certain concepts which often arise in reasoning about hardware behaviour, however, are most naturally represented by higher order entities. The function Rise defined on page 34 is a typical example. In later papers (e.g. [24]) Eveking extends the logic he uses by including certain higher-order constructs, so that such naturally higher-order entities can be represented. These extensions include second-order functions (i.e. 'functionals') and λ-abstractions.

First Order Theories for Asynchronous Devices

Barros and Johnson [2] use first order logic to reason about the ideal behaviour of four commonly-used asynchronous devices: the arbiter, the synchronizer, the latch, and the inertial delay. The approach taken is axiomatic—each device is described by a collection of first order axioms (i.e. a first order theory). These axioms are of three kinds: *output* axioms, *forward* axioms, and *reverse* axioms. Output axioms express idealizing assumptions about the values on the output wires of a device. The formal description of each device includes, for example, an output axiom which asserts that every transition between boolean values on its output wires takes a bounded amount of time. Forward axioms specify the effect which a particular sequence of values at the inputs of a device has on the values at the outputs. These axioms stipulate conditions on the inputs of a device which are *sufficient* for a certain kind of output behaviour to occur. Reverse axioms specify the conditions which the inputs of a device *must* satisfy for a particular output behaviour to occur—i.e. reverse axioms express *necessary* conditions on the inputs for a certain kind of output behaviour to be observed. One of the notable features of this approach is this systematic approach to describing asynchronous hardware.

Barros and Johnson use equivalence of theories[5] to formulate the functional equivalence of the four asynchronous devices mentioned above. Two of these devices are considered to be equivalent if they can be used to implement, or 'simulate', each other. More precisely, two devices D_1 and D_2 are considered to be equivalent if a circuit which behaves like D_1 can be built using instances of D_2 as primitive components, and vice versa. This notion of equivalence of devices is expressed formally by equivalence of theories. Two devices D_1 and D_2 are equivalent if: (1) a theory describing D_1 is an extension of a theory which describes an appropriate implementation of D_1 constructed using instances of D_2

[5]Two first-order theories are *equivalent* if each one is an extension of the other.

as primitive components, and (2) a similar proposition holds in which the roles of D_1 and D_2 reversed. This approach was used to demonstrate the equivalence of the four asynchronous devices mentioned above. The complete proof for all four devices is not given in [2], but an outline of the formal proof of equivalence for the inertial delay and the latch is given to illustrate the general approach.

Higher Order Logic: Veritas

Higher order logic was first proposed as a formalism for hardware verification in the early account of the VERITAS project given by F.K. Hanna in [41]. The VERITAS approach to hardware verification is described in detail by Hanna and Daeche in [43], and a case study which illustrates this approach (the verification of a D-type flip flop) can be found in [44]. The term 'VERITAS' is used to refer both to the general approach to hardware verification described in these papers and to the particular species of higher order logic which is used in this approach.

An implementation of the VERITAS logic in the purely functional programming language Miranda[6] [81] is described by Hanna and Daeche in [42]. This logic is a version of polymorphic higher-order logic based on the typed λ-calculus. The syntax of terms in this formulation of higher order logic is similar to the syntax of terms described in Chapter 2, but the type system of VERITAS includes elements of Martin-Löf's Intuitionistic Type Theory [61] and is therefore more flexible and expressive than the type system used here. For example, the type system of the VERITAS logic includes *subtypes*; these are not available in the formulation of higher order logic described in Chapter 2.

The VERITAS approach to hardware verification is similar in many ways to the approach described in the preceding sections of this chapter. A full account of the VERITAS methodology will therefore not be given here (for this, see [43]). There is, however, one aspect of VERITAS which relates to the work described in Chapter 7 of this dissertation. This is discussed briefly below.

In the VERITAS approach, the *structure* of hardware is modelled independently of its *behaviour*. This is done by using a hierarchy of axiomatically characterized types and subtypes to represent various kinds of purely structural entities (e.g. input ports, output ports, and n-input logic gates). This differs from the approach outlined in this chapter, where the primary emphasis is on behaviour. Here, the structure of hardware is modelled only to the extent that it is reflected by the syntactic structure of the logical terms which model its behaviour. In Chapter 7 of this dissertation, however, a technique is developed by which the structure of hardware can be modelled more explicitly. This technique resembles the VERITAS approach to structure mentioned above—at least in so far as logical types are used to model the structure of circuits independently of their behaviour. The details are given in Chapter 7.

[6]Miranda is a trade-mark of Research Software Ltd.

Higher Order Logic: HOL

The approach to hardware verification using higher order logic described in this chapter is due to Dr. M. Gordon and was first presented by him in [31]. More widely-known early accounts of Gordon's approach are the papers [34] and [11]. A more recent general account, in which the emphasis is on reasoning about CMOS circuits, is given by Hoare and Gordon in [51].

Several researchers have based their work on Gordon's formulation of higher order logic, its mechanization in the HOL system, and the general approach to hardware verification proposed in the papers cited above. Herbert [48] uses higher order logic and the HOL system to prove the correctness of an ECL chip used as a component in the Cambridge Fast Ring Network [52]. Dhingra [21] uses higher order logic to formalize the design rules of a CMOS design style called CLIC. Joyce [56,55,58] uses higher order logic and HOL to prove the correctness of a microprocessor. The formal verification of the Viper microprocessor in HOL is documented by Cohn in [16,17]. Aspects of this research that relate specifically to the subject of the present work are discussed in later chapters.

Temporal Logic

In addition to the work mentioned above, which is all based on either first-order or higher-order predicate calculus, formal methods for specifying hardware behaviour and reasoning about hardware correctness have also been based on *temporal logic.*

An early application of temporal logic to hardware verification is the correctness proof for the design of an arbiter given by Bochmann in [8]. Bochmann uses temporal operators such as $\Box$ ('henceforth') and $\bigtriangledown$ ('eventually') to capture the time-dependent properties hardware components. The arbiter correctness proof in [8] is done by proving that a collection of 'invariant assertions' hold for all states that can be reached from the initial state of the device. The correctness of the arbiter design then follows from these invariant assertions.

Moszkowski [72] defines a logical formalism for specifying hardware behaviour called ITL (Interval Temporal Logic). The truth of a formula in ITL is defined relative to a finite *interval* of discrete time. Modal operators are used to express temporal concepts in terms of these time intervals. Moszkowski shows how this can be used to describe formally a wide variety of time-dependent aspects of hardware behaviour. In [71], Moszkowski describes a logic programming language called 'Tempura' which is based on a subset of ITL. Moszkowski shows how this language can be used to simulate formal specifications of hardware behaviour written in ITL. Some extensions to ITL and further applications of ITL to hardware verification are discussed by Leeser in [59]. Hale [40] gives a formal semantics for ITL in higher order logic, and uses the HOL system to prove properties of Tempura programs.

In [22], Dill and Clarke present a method for automating the verification of asynchronous circuits using temporal logic. Specifications of required behaviour are written in a version of propositional temporal logic called CTL. A circuit is shown to satisfy a CTL specification by first translating a gate-level description of the circuit into a state transition graph. The CTL specification for the circuit is then checked automatically against this state graph representation by a program called a *model checker*. This program checks that the paths, or sequences of states, in the state transition graph satisfy the given CTL specification. The automatic verification of an arbiter is given as an example to illustrate this approach.

Chapter 4

Abstraction

Abstraction plays a central role in making formal proof an effective method for dealing with the problem of hardware correctness. The reasons for this were given in Chapter 1, in the context of a discussion of some fundamental limits to the scope of hardware verification by formal proof. In Section 1.2, two ways in which the notion of abstraction plays an important role in hardware verification were introduced. This chapter describes how these two types of abstraction—which will be referred to here as abstraction *within* a model[1] of hardware behaviour and abstraction *between* models of hardware behaviour—can be formalized in higher order logic.

Abstraction within a model is discussed in Section 4.1. This type of abstraction has to do with the way in which correctness is formulated in logic. With the approach to hardware verification introduced in Chapter 3, the correctness of a device is stated formally by a proposition of logic which asserts that some relationship of 'satisfaction' holds between a formal model of its design and an appropriate specification of intended behaviour. For the reasons discussed in Chapter 1 and Section 3.3, this satisfaction relationship must, in general, be one of abstraction. That is, a formulation of correctness must relate a *detailed* model of an actual hardware device to a more *abstract* specification of required behaviour. Section 4.1 shows how this notion of correctness as a relationship of abstraction can be formalized in logic and incorporated into the method of hardware verification introduced in Chapter 3.

The second type of abstraction—abstraction between models—is discussed in Section 4.2. Here, the concern is not with the correctness of individual hardware designs, but with the idea of an abstraction relationship between two different collections of formal specifications for the primitive components used in designs. One such collection can be an abstraction of another, in the sense that it presents a more abstract view of the same primitive components. In this case, either set of primitive specifications can be used to construct a design model for any particular hardware device. Design models constructed using the more abstract primitives, however, will generally be simpler—but also less accurate—than design

[1]Here, a 'model' means a collection of formal specifications for the different kinds of primitive hardware components from which devices are built (cf. Section 3.2.3).

models constructed using the more detailed primitives. But for certain *kinds* of devices, the design models constructed from these two sets of primitives may be effectively equivalent (in a sense which is explained in Section 4.2). For this class of devices, the more abstract primitives should be used, since this will result in models which are just as accurate as the ones constructed from the detailed primitives, but which are more tractable. Section 4.2 shows how the concept of an abstraction relationship between models can be expressed formally in logic, and explains how this relates to the idea that a class of design models constructed from a set of abstract primitives may be effectively equivalent to a class of design models constructed from a set of more detailed primitives.

This chapter is concerned only with the general ideas behind the formalization of these two kinds of abstraction in logic. Only a very simplified account is given here, and some of the complexities which arise in practice are either ignored or mentioned only briefly. In subsequent chapters, however, concrete examples are given to show how the general principles introduced here are applied in practice.

4.1 Abstraction within a Model

There are three important ways in which the specification of required behaviour for a hardware device can present a more abstract view of its behaviour than is given by a realistic formal model of its design.

First, the specification may be only a *partial* specification of required behaviour, which leaves unspecified some aspects of the behaviour given by the model of the device itself. Such a specification typically stipulates how a device is expected to behave only when it is used in certain environments. In all other environments, the behaviour of the device is not specified. The formal model of a hardware device, however, generally gives more detail than this. It describes how the device behaves when it is placed in an arbitrary environment, not just how it behaves in only selected environments. In this case, a partial specification of required behaviour contains less information about the behaviour of the actual device than the model does—the formal relationship between the model and the specification of required behaviour is one of *behavioural abstraction*.

Second, the specification may be expressed in terms of a higher-level notion of *data* than is used in the model. The design model for a hardware multiplier, for example, might describe its behaviour in terms of the individual binary values present on each of its input and output wires. An abstract specification for this device, however, is more likely to describe its functional behaviour in terms of the numbers being multiplied than in terms of individual bits. In this case, the formal relationship between the design model and the more abstract specification is one of *data abstraction*: the specification is written using a more abstract notion of data than the concrete binary representation of numbers used in model.

Finally, the specification may be formulated in terms of a less detailed notion of *time* than is used in the model. The formal specification of required behaviour for a large device—a microprocessor, for example—is unlikely to include as much information about how the device behaves over time as will be given by a detailed model of its design. A realistic model of a microprocessor might, for example, describe its behaviour at a level of temporal detail which includes information about system timing and propagation delay. But an abstract specification for this device is more likely to describe it as a finite state machine, in which the emphasis is on the sequence of operations carried out by the device, rather than the exact times at which these operations occur. This specification would then represent a *temporal abstraction* of the more detailed behaviour given by the model.

The next three sections show how correctness statements can be formulated in logic which express these three basic kinds of abstraction relationship between a design model and an abstract specification of required behaviour. In the most general case, a correctness statement may involve all three types of abstraction. The aim of the following three sections, however, is to provide a clear account of the motivation for each type of abstraction and to discuss how each one can help to simplify specifications of required behaviour. Each type of abstraction is therefore considered separately in these three sections. Example correctness statements in which all three types of abstraction are combined are considered in later sections of this chapter.

4.1.1 Behavioural Abstraction

A specification of required behaviour is simply a logical term which expresses a constraint on the values that can appear on the external wires of a device. As was discussed in Section 3.1.4, a *partial* specification is one which imposes only a partial constraint on these values. This constraint will be satisfied by only very restricted combinations of values in the situations or contexts for which a specific behaviour is required of the device. But in the situations where the behaviour of the device is intended to be left unspecified, the constraint imposed by the specification will be satisfied by a relatively wide range of combinations of values.

The formal model of a device, however, will generally describe the combinations of values which actually appear on its external wires—even in the situations where a *particular* combination of values is not called for by a more abstract partial specification of required behaviour. This means that there will be combinations of values which are allowed by a partial specification of required behaviour, but which do not satisfy the constraint imposed by a more detailed model. A satisfaction relation based on behavioural abstraction must therefore express a relationship between a strong constraint (the model) and less restrictive one (the specification).

It is straightforward to formulate a correctness statement which expresses this abstraction relationship in logic. Suppose that the two terms $M[v_1, \ldots, v_n]$ and $S[v_1, \ldots, v_n]$ are the formal model of a hardware device and a partial specification of required behaviour respectively. The idea that the specification imposes a less restrictive constraint on the free variables v_1, ..., v_n than the model is expressed formally by the correctness theorem shown below.

$$\vdash M[v_1, \ldots, v_n] \supset S[v_1, \ldots, v_n]$$

This theorem asserts that any combination of values which satisfies the constraint imposed by the model also satisfies the constraint imposed by the more abstract partial specification of required behaviour.

This is a weaker correctness statement than the one used in the inverter example given in the previous chapter, where correctness was stated as a logical equivalence between the model and the specification. Here, the model is required only to *imply* the specification. Every combination of values that satisfies the model must also satisfy the specification. But there may also be combinations of values which are allowed by the specification, but which (according to the model) never actually appear on the external wires of the device itself. This reflects the fact that the partial specification is a behavioural abstraction of the model: the specification stipulates only selected aspects of the device's behaviour and therefore defines a range of *allowable* behaviour for the device. The criterion of correctness expressed by logical implication is that the behaviour actually exhibited by the device must lie somewhere within this range.

This use of logical implication to state correctness is natural and well known, and examples can be found in many papers on hardware verification using formal logic [11,43,51]. What is emphasized in the present discussion is merely that this formulation of correctness expresses a relationship of *abstraction* between the formal model of a hardware device and a more abstract specification. Relaxing the criterion of correctness from logical equivalence to logical implication allows the specification of required behaviour for a device to stipulate only some of the aspects of its behaviour which are captured formally by a more detailed formal model of its design. Only those properties of the device which are relevant to its functional correctness need be 'mentioned' explicitly in the specification. The specification can therefore be more succinct than is possible when the model and the specification are required to be logically equivalent.

4.1.2 Data Abstraction

Besides being only a partial specification, an *abstract* specification of intended behaviour for a device may also be written in terms of a correspondingly abstract notion of the kinds of values it operates on. The free variables which occur in such a specification will not stand for the values actually present on the external wires of the device itself, but for more abstract externally observable quantities. And the specification will be written using operations which can be meaningfully applied to these abstract quantities, rather than the operations which are carried out by the actual hardware on a more concrete representation of these values. The logical types of the free variables which represent these abstract quantities in the specification will therefore generally differ from those of the free variables in a formal model of the design itself. A satisfaction relation based on data abstraction must therefore relate 'concrete' values of one logical type in a model to more 'abstract' values of another logical type in a specification.

In the simplest case of data abstraction, both the model and the specification express a constraint on free variables that directly correspond to the physical wires of the device, but use different logical types to represent the range of values that can appear on these wires. In this case, the model and the specification will be logical terms of the form:

$$M[\underbrace{c_1, \ldots, c_n}_{\text{type } ty_1}] \qquad \text{and} \qquad S[\underbrace{a_1, \ldots, a_n}_{\text{type } ty_2}]$$

In this very simple case, each variable a_i in the specification represents the same externally observable value as the corresponding variable c_i in the model. The specification, however, is expressed as a constraint on abstract values of type ty_2, instead of the concrete values of type ty_1 in the model that represent the values actually present on the external wires of the device.

To formulate a correctness statement that relates these two specifications, it is necessary to 'translate' the constraint on values of type ty_1 expressed by the model into a constraint on more abstract values of type ty_2. This can be done by using an appropriately-defined *data abstraction function* to map concrete values of type ty_1 in the model to abstract values of type ty_2 in the specification. Given such a function $\mathsf{f}{:}ty_1{\rightarrow}ty_2$, a correctness statement which expresses a relationship of data abstraction between the model and the specification can be formulated as shown below.

$$\vdash M[c_1, \ldots, c_n] \supset S[\mathsf{f}\ c_1, \ldots, \mathsf{f}\ c_n]$$

This theorem states that every combination of values c_1, ..., c_n which, according to the model, actually appears on the external wires of the device is a concrete *representation* at a lower level of data abstraction for a combination of more abstract values which is allowed by the specification.

As with behavioural abstraction, correctness is expressed in this theorem using a satisfaction relation based on logical implication. But here, it is a translation of the values present in the model that must satisfy the specification of required behaviour, rather than the values themselves. This translation is obtained by point-wise application of the data abstraction function f to the free variables of the model: the abstraction function f is used to map each concrete value c_i in the model to a corresponding abstract value '$\mathsf{f}\ c_i$'. The combination of abstract values thus obtained must satisfy the constraint expressed by the *abstract* specification of required behaviour. That is, the values $\mathsf{f}\ c_1, \ldots, \mathsf{f}\ c_n$ must, when substituted for the free variables $a_1, \ldots, a_n$, satisfy the constraint given by the abstract specification $S[a_1, \ldots, a_n]$. The resulting correctness statement asserts that the operations on concrete values carried out by the device correctly implement the required operations on abstract values expressed by the specification.

In this very simple example, there is a one-to-one correspondence between the free variables in the model and the free variables in the specification. But in general this may not always be the case: an observable value represented by a single variable in the abstract specification for a device may in fact correspond to a *collection* of values in the model of its design. For example, the model of an 8-bit binary counter might contain eight boolean variables, one for each output wire; but an abstract specification for this device may simply represent its output by a single variable ranging over numbers.

A correctness statement based on data abstraction may therefore involve more complex functions of the variables in the model than was suggested by the simple example given above. An example is the correctness statement shown below.

$$\vdash M[c_1, \ldots, c_n] \supset S[\mathsf{f}_1(c_1, \ldots, c_i),\ \mathsf{f}_2(c_{i+1}, \ldots, c_n)]$$

Here, the specification of required behaviour is a constraint of the form '$S[a_1,\ a_2]$' on two abstract quantities a_1 and a_2. In the model these two abstract values are represented by a collection of n concrete values. Two data abstraction functions f_1 and f_2 are used in this correctness statement to relate these concrete values in the model to the abstract values which they represent: the function f_1 maps $c_1, \ldots, c_i$ to one abstract value in the specification, and the function f_2 maps $c_{i+1}, \ldots, c_n$ to the other. Many other patterns of correspondence between the free variables in a model and the abstract values in a specification are, of course, also possible.

The advantage of data abstraction is that it allows the specification of required behaviour for a device to be written in terms of abstract 'high-level' operations on data, without having to specify precisely how this data is represented in the device itself. In a correctness theorem of the kind discussed above, data abstraction functions can be used to separate the details of data representation from the formal specification of required behaviour. This allows the specification to be expressed in terms of the mathematical entities and notation most appropriate to an intuitively clear and textually succinct abstract statement of the computation a device is intended to carry out.

4.1.3 Temporal Abstraction

In addition to being expressed at a higher level of data abstraction, an abstract specification of required behaviour may also give less detail about how a device behaves over time than a realistic model of the device itself. It may, for example, state the behaviour a device is expected to exhibit at only certain significant or 'interesting' points of time, and leave unspecified the details of any intermediate states through which the device must pass to realize this behaviour. In this case, the specification will employ a more abstract notion of time than would be used in a more detailed design model—i.e. a model that *does* describe the device's behaviour at these intermediate states. A satisfaction relation based on temporal abstraction must therefore establish a relationship between two different formal representations of time: an 'abstract' representation of time in the specification, and a 'concrete' representation of time in the model.

In the general case, each *unit* of discrete time in the specification corresponds to an *interval* of discrete time in the more detailed design model. In this case, the specification describes the values that appear on the external wires of the device at fewer points of 'real' time than the model does. Each point of 'abstract time' in the specification corresponds to a particular point of 'concrete time' in the model. And, at these corresponding points in time, the model and the specification impose the same constraint on the values that can appear on the external wires of the device. But the model also constrains these values at points of concrete time which lie *between* what are considered to be adjacent points of time at the more abstract level of the specification. A correctness statement that relates this model to the more abstract specification must therefore establish a correspondence between two different time-scales: a 'fine-grained' time-scale in the model and a 'coarse-grained' time-scale in the specification.

This correspondence can be described formally by a function that maps each point of abstract time in the specification to a corresponding point of concrete time in the model. Such a function is a *time mapping* that describes the precise relationship between the abstract time-scale used in the specification and the concrete time-scale used in the more detailed model. A simple example is shown

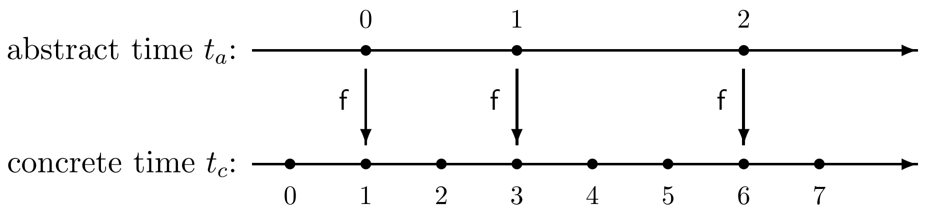


Figure 4.1: A Mapping between Time-scales.

in Figure 4.1. The solid lines in this diagram represent continuous or real time. The dots represent the points of real time which constitute the two discrete time-scales involved: the concrete time-scale t_c used in the model, and the abstract time-scale t_a used in the specification. The mapping f describes the relationship between these two time-scales. To every point of time t on the abstract time-scale, the function f assigns a corresponding point of concrete time '$\mathsf{f}\ t$' such that the order of time is preserved:

$$\vdash \forall t_1\, t_2.\ (t_1 < t_2) \supset (\mathsf{f}\ t_1 < \mathsf{f}\ t_2)$$

This establishes a correspondence between units of time on the abstract time-scale and intervals of time on the concrete time-scale by mapping successive points of abstract time to *selected* points of concrete time.

Any correspondence between successive units of abstract time and contiguous intervals of concrete time can be described formally in logic by a time mapping of this kind. The particular point of concrete time assigned by such a function to each point of abstract time will, of course, depend on the exact relationship between the model and the specification involved. For example, each unit of abstract time in the specification for a clocked synchronous device might correspond to an interval of concrete time between two rising edge of a clock signal in the model. In this case, the function f would map points of time on the abstract time-scale to the points of concrete time at which these rising edges of the clock occur. A detailed account of how such a function can be defined will not be given here, but will be deferred until Chapter 6.

Given an appropriately-defined mapping f from abstract time to concrete time, a correctness statement that relates a model to a specification at a higher level of temporal abstraction can be formulated in logic as follows. Suppose that the two logical terms

$$M[c_1, \ldots, c_n] \qquad \text{and} \qquad S[a_1, \ldots, a_n]$$

are the model of a device and an abstract specification of required behaviour, respectively. To simplify matters, assume that the free variables in both the model and the specification are functions of type $num{\rightarrow}bool$, and that c_i corresponds to a_i for $1 \leq i \leq n$. If the specification is a temporal abstraction of the model (in the sense discussed above) and if the device is correct, then each sequence of values a_i in the specification will correspond to a *subsequence* of the values given by the variable c_i in the model. Each function a_i in the specification will represent a sequence of values which could be obtained by 'sampling' the function c_i at only those points of concrete time which are significant at the abstract level of the specification, and therefore correspond to points of discrete time on the abstract time-scale used in the specification.

Given a function f that describes this correspondence, a correctness statement that relates the model to the specification can be formulated as shown below.

$$\vdash M[c_1, \ldots, c_n] \supset S[c_1 \circ \mathsf{f}, \ldots, c_n \circ \mathsf{f}]$$

This theorem states that if the functions c_1, ..., c_n satisfy the temporally detailed constraint imposed by the model, then the functions $c_1 \circ \mathsf{f}$, ..., $c_n \circ \mathsf{f}$ will satisfy the temporally abstract specification of required behaviour. Here, the model describes the values that appear on each external wire c_i at points of fine-grained or concrete time. The function f specifies which of these points of concrete time correspond to points of time on the abstract time-scale. Composition on the right with f constructs an abstract sequence of values '$c_i \circ \mathsf{f}$' from each detailed sequence of values c_i by sampling the function c_i at these selected points of concrete time. The combination of abstract sequences obtained in this way must satisfy the abstract specification of required behaviour.

The resulting correctness statement asserts that the combinations of values present on the external wires of the device satisfy the specification of required behaviour at each point in time that is regarded as significant or important at the abstract level of description. That is, the behaviour of the device when observed at only those selected points of concrete time specified by the function f satisfies the temporally abstract specification of its required behaviour.

The advantage of temporal abstraction is that it hides irrelevant details about intermediate state transitions from the abstract specification of required behaviour for a device. Points of time on the abstract time-scale in a correctness theorem based on temporal abstraction correspond to *selected* points of time on the more detailed concrete time-scale, and it is only at these selected points of time that the device's behaviour is stipulated by the abstract specification. Furthermore, the use of a time mapping to relate abstract and concrete time-scales not only allows the behaviour of the device at *other* points of time to be left unconstrained by the specification, but also makes intermediate states transitions completely invisible to the specification. Intermediate states represented by points of time on the concrete time-scale used in the model simply do not exist on the abstract time-scale used in the specification. This allows the specification to describe the required behaviour of a device at only significant points of time, without also having to indicate precisely *which* points of time are in fact of interest.

4.1.4 Two Problems

The underlying satisfaction relation in all three forms of correctness discussed above is *logical implication*. In each case, correctness is stated by an implication of the form $M \supset S$, in which the model is the antecedent and a substitution instance of the specification is the consequent. There are two problems that can arise when correctness is stated by an implication of this form. These are discussed briefly in the two sections that follow.

4.1.4.1 Underspecification

Whenever the behaviour which a device is required to exhibit is stated formally by a partial specification, there is the possibility that this partial specification in fact *under*specifies the intended behaviour of the device. That is, a partial specification may inadvertently fail to stipulate some important aspect of the device's intended behaviour, and therefore be satisfied by a wider range of values than is actually intended by the designer. In this case, the constraint expressed by the specification will be satisfied by some combinations of values which in fact ought *not* to appear on the external wires of the device. But when correctness is formulated as logical implication, a model which is satisfied by these undesirable combinations of values (and therefore represents an *incorrect* design) will, according to this formal notion of correctness, be considered correct with respect to this specification.

This is much less likely to happen when correctness is stated formally by logical equivalence. If the specification and the model are required to express the *same* constraint on the free variables which they contain, then any weakness in the specification must either: (1) be matched exactly by a corresponding degree of 'nondeterminism' in the model, or (2) make it impossible to complete the proof of correctness. But if the criterion of correctness is relaxed to logical implication, then the specification is allowed to express a strictly weaker constraint than the model. An inadequate specification is therefore less likely to be detected during the course of a proof, since the behaviour given by the model is required only to lie somewhere within the range of behaviour stipulated by the specification.

There is no complete solution to this problem, since it is a problem of inaccuracy in the specification of intended behaviour for a device. For the reasons already discussed in Chapter 1, it is not possible to *prove* that a partial specification in fact covers all the essential aspects of a device's intended behaviour. Whenever it is possible to leave *something* unspecified, it is also possible to leave something *essential* unspecified.

4.1.4.2 Inconsistent Models

A second problem with using logical implication to express correctness is that an *inconsistent* model then trivially satisfies any specification.[2] An inconsistent model is one which cannot be satisfied by any assignment of values to its free variables. A simple example is the term '$\mathsf{Pwr}\ x \wedge \mathsf{Gnd}\ x$', where $\mathsf{Pwr}\ x$ and $\mathsf{Gnd}\ x$ are instances of the specifications for power and ground defined in Chapter 3.

[2]In [11], this is called the 'false implies anything problem'. The pragmatic solution to this problem mentioned in this section was suggested by M. Fourman.

This term is logically equivalent to *falsity*, since no boolean value x can satisfy both $\mathsf{Pwr}\ x$ and $\mathsf{Gnd}\ x$. If satisfaction is formulated as logical implication, then this inconsistent model satisfies (i.e. implies) *any* specification. In general, if the model on the left hand side of the implication:

$$M[v_1, \ldots, v_n] \supset S[v_1, \ldots, v_n]$$

is false for all values of the variables v_1, ..., v_n, then this implication is a *theorem*, no matter what constraint is imposed on these variables by the term on the right hand side of the implication. This is clearly unsatisfactory, since a formal theorem of this kind provides no meaningful assurance of functional correctness.

The ideal solution to this problem would be to have a collection of specifications for the primitive components used in designs that always yields a consistent model, no matter how this model is constructed from these primitives using the syntactic operations of composition ('$\wedge$') and hiding ('$\exists$'). This, however, may require the specifications for primitive components to be of considerable complexity. A more pragmatic solution is to check the consistency of the particular design model on which proof of correctness is based. This can be done by proving a consistency theorem of the form:

$$\vdash \exists v_1 \ldots v_n.\, M[v_1, \ldots, v_n]$$

in addition to proving a correctness statement of the general form illustrated by the implication shown above. Proving this extra consistency theorem ensures that the model shown above can be satisfied by at least one combination of values for the variables v_1, ..., v_n. It therefore shows that this model does not satisfy a specification merely because it is inconsistent. If none of the external wires of a device are bidirectional (i.e. every wire of the device is either an input or an output), then a stronger consistency theorem can be formulated:

$$\vdash \forall i_1 \ldots i_n.\, \exists o_1 \ldots o_m.\, M[i_1, \ldots, i_n, o_1, \ldots, o_m]$$

This theorem states that for any collection of input values i_1, ..., i_n, there are output values o_1, ..., o_m which, according to the model, are consistent with them. Again, this shows that the model does not satisfy a specification of required behaviour merely because it is inconsistent. An example of a consistency theorem of this second kind can be found in Section 5.3.3.4 of Chapter 5.

In general, it is necessary to prove a consistency theorem of one of these two kinds, in addition to a correctness theorem, whenever a satisfaction relation based on implication is used. Consistency theorems are usually not proved in most of the examples presented in the literature, since the models which are used are generally simple enough to be easily seen to be consistent. But when formal verification is applied to much larger examples, it may be necessary to consider more explicitly the possibility that the models involved might be inconsistent.

4.1.5 Discussion

For clarity, only a highly simplified account was given in the preceding sections of the three most basic ways in which correctness can be expressed in logic by a satisfaction relation based on abstraction. Some complexities which arise in practice, but which were left out of consideration in the simplified account given above, are discussed below. Some of the aspects of data and temporal abstraction which will be discussed in Chapters 5 and 6 are also mentioned briefly below.

- One aspect of the formalization of data abstraction in higher order logic which was not discussed above is the task of defining logical *types* to provide formal representations of 'data' in logic. To make effective use of data abstraction, a variety of types are needed, both for writing abstract specifications of intended behaviour and for defining realistic design models. The free variables in both models and specifications stand for the values by which a device communicates with its environment. To provide a direct and natural representation for these values, it is generally necessary to introduce new logical types whose formal properties are appropriate to the kinds of values involved. As was discussed in Chapter 2, this must be done by first defining these types and then proving that they have the desired properties. This aspect of the formalization of data abstraction in higher order logic is discussed in detail in Chapter 5.

- In the overview of temporal abstraction given above, what was not discussed in any detail is the task of defining the time mappings needed to relate abstract and concrete time-scales. The points of time at which values are of interest to the abstract specification for a device usually depend in some way on the operation of the device itself. For example, an abstract specification may (as was mentioned above) stipulate the values that must appear on the inputs and outputs of a device at points of abstract time which correspond to the rising edges of a clock. In this case, the correspondence between abstract time and concrete time will depend on the behaviour of the clock, and a mapping between time-scales that describes this correspondence must therefore be constructed from the clock signal itself. A general technique by which functions that map from one time-scale to another can be constructed in this way is discussed in detail in Chapter 6.

- In the correctness theorem given above as an example of temporal abstraction:

  $$\vdash M[c_1, \ldots, c_n] \supset S[c_1 \circ \mathsf{f}, \ldots, c_n \circ \mathsf{f}]$$

  a single time mapping f is used to construct a subsequence '$c_i \circ \mathsf{f}$' from each sequence of values c_i in the model. In general, however, it is not necessarily the case that the same mapping of time-scales can be used for every variable in a model. For example, the abstract specification for a device driven by a

2-phase clock might represent a temporal abstraction obtained by sampling the values present at certain points in the circuit on the one phase of the clock and sampling the values present at other points in the circuit on the other phase of the clock. A statement of correctness in this case must employ two different time mappings to relate this specification to a more detailed model of the device—one for each clock phase. Examples of temporal abstraction in which different mappings between an abstract time-scale and a concrete time-scale occur in the same correctness statement can be found in Chapter 6.

- The distinction between data and temporal abstraction made in the preceding sections is that data abstraction involves a translation of the *values* by which a device communicates with its environment, whereas temporal abstraction involves a translation of the *times* at which these communications occur. In many correctness statements in which both types of abstraction are involved, the satisfaction relation used to formulate correctness can be expressed as a combination of two distinct components: a data abstraction and a temporal abstraction. A formulation of correctness which combines data abstraction and temporal abstraction, and which illustrates this factorization of the abstraction relationship into separate data and temporal components, is discussed below in Section 4.1.7.

- Although it is a generally useful principle of organization to distinguish between data and temporal abstraction, some abstraction relationships do not fit neatly into one category or the other, but are best regarded as hybrid combinations of both data and temporal abstraction. For example, the translation from a bit-serial representation of numeric data in a design model to a sequence of numbers in an abstract specification of required behaviour is both a data abstraction (translating bits to numbers) and a temporal abstraction (relating a group of values spread out over points of concrete time to a value at a single point of abstract time). Although correctness statements based on translations of this kind are not easily expressed as simple combinations of two distinct types of abstraction (i.e. data and temporal abstraction) the formalization in higher order logic of correctness statements of this kind does not present any special problems.

- The distinction made in the previous section between temporal abstraction or data abstraction and behavioural abstraction might seem—from a purely formal point of view—to be a somewhat artificial one. All the correctness statements given above as examples of data and temporal abstraction can also be seen as examples of behavioural abstraction, in which the specification of required behaviour is simply a weaker constraint than the model. Consider, for instance, the correctness theorem which was given in Section 4.1.2 to illustrate the idea of a relationship of data abstraction:

$$\vdash M[c_1, \ldots, c_n] \supset S[\mathsf{f}\ c_1, \ldots, \mathsf{f}\ c_n]$$

As was discussed in Section 4.1.2, a theorem of this kind can be viewed as expressing a relationship of data abstraction between the model $M[c_1, \ldots, c_n]$ and an abstract specification of required behaviour $S[a_1, \ldots, a_n]$. But it can also be seen as expressing a relationship of behavioural abstraction, in which the specification of required behaviour is the term '$S[\mathsf{f}\ c_1, \ldots, \mathsf{f}\ c_n]$'.

- The distinction between these two views of the same correctness statement is a pragmatic distinction, rather than a strictly formal or syntactic one. It is only when the specification for the correctness theorem shown above is *taken to be* the constraint on abstract values '$S[a_1, \ldots, a_n]$', where externally observable values are represented by abstract *variables*, rather than data representations constructed from concrete variables, that this theorem express a relationship of *data* abstraction between the model and the specification. The distinction between the two possible views of this formal theorem is therefore purely a matter of what is considered to be 'the specification' of intended behaviour for the device in question.

- To provide a clear motivation for the three types of abstraction discussed above, correctness was treated as a relationship between a fully concrete design model for an actual hardware device and an abstract specification intended behaviour for the device in question. In general, however, the correctness proof for a large device must be structured hierarchically, so that a logical term which is considered to be the *model* of a component at one level in the hierarchy becomes the *specification* of required behaviour for a more concrete component at the next level down.[3] A correctness proof therefore usually involves a hierarchy of nested abstractions, rather than a single abstraction from one fully detailed concrete model to a top-level specification of required behaviour. The notion of hierarchical proof is discussed in further detail in Section 4.1.8 below.

4.1.6 Validity Conditions

Another complexity not mentioned in the preceding account of abstraction is that an abstraction is normally valid or appropriate only under certain conditions. A given specification may in fact present a valid abstract view of the actual behaviour of a device only under certain conditions, which will be referred to here as *validity conditions* on a correctness statement expressed as a relationship of abstraction.

[3]Of course, the free variables in such a term do not generally stand for the values on the physical 'wires' of an actual device, but usually represent more abstract externally observable quantities.

Suppose, for example, that the term $M[c_1, \ldots, c_n]$ is the formal model of a hardware device, and that $S[a_1, \ldots, a_n]$ is a specification of required behaviour expressed at a higher level of data abstraction than the model. For a given design model and a given abstract specification, it may not always be possible to prove a correctness theorem of the simple form discussed above in Section 4.1.2:

$$\vdash M[c_1, \ldots, c_n] \supset S[\mathsf{f}\ c_1, \ldots, \mathsf{f}\ c_n]$$

where f is an appropriate data abstraction function that maps concrete values in the model to abstract values in the specification. The device modelled by the term $M[c_1, \ldots, c_n]$ may in fact behave as stipulated by the abstract specification of required behaviour only for a restricted range of values on its input wires, or in only certain well-behaved contexts or environments.

In this case, it will not be possible to prove a correctness statement of the simple form shown above. Instead, the correctness statement must involve an extra constraint that states the conditions under which the device in fact *does* behave as required by the abstract specification:

$$\vdash C[c_1, \ldots, c_n] \supset (M[c_1, \ldots, c_n] \supset S[\mathsf{f}\ c_1, \ldots, \mathsf{f}\ c_n])$$

The term '$C[c_1, \ldots, c_n]$' in this correctness statement is a *validity condition* on the abstraction relationship between the model and the specification. By imposing a constraint on the free variables $c_1, \ldots, c_n$ this condition describes the external environments in which the device modelled by $M[c_1, \ldots, c_n]$ in fact does behave as stated by the more abstract specification of intended behaviour. The specification present a valid abstract view of the behaviour described by the more detailed formal model of its design only when the device is placed in an environment that satisfies this constraint.

A validity conditions of the kind illustrated by this example can arise whenever correctness is formulated by a satisfaction relationship based on any of the three types of abstraction discussed in the preceding sections. In each case, a validity condition expresses a constraint that must be satisfied by the environment in which a device is operating in order ensure that it will behave as required by the abstract specification. Particular examples of the sorts of validity conditions that arise when correctness is formulated as a relationship of data abstraction or temporal abstraction can be found in Chapters 5 and 6.

4.1.7 A Notation for Correctness

In the most general case, a correctness statement can involve a combination of all three types of abstraction—data, temporal, and behavioural. Furthermore, as was pointed out above, a correctness statement based on temporal abstraction may involve several different mappings from abstract to concrete time, and a correctness statement based on data abstraction may involve functions that map a collection of concrete values represented by several free variables in the model to a single abstract value in the specification of required behaviour.

For notational clarity, however, the discussion of abstraction in the remaining sections of this chapter will be restricted to correctness statements expressible by theorems of the general form shown below:

$$\vdash C[c_1, \ldots, c_n] \supset (M[c_1, \ldots, c_n] \supset S[\mathsf{f} \circ c_1 \circ \mathsf{g}, \ldots, \mathsf{f} \circ c_n \circ \mathsf{g}]) \tag{4.1}$$

A theorem of this form expresses correctness as a combination of behavioural, temporal, and data abstraction. The term $M[c_1, \ldots, c_n]$ is the design model for the device in question. The specification of required behaviour is term of the form $S[a_1, \ldots, a_n]$. Each abstract variable a_i in the specification corresponds to the concrete variable c_i in the model for $1 \leq i \leq n$. The condition $C[c_1, \ldots, c_n]$ is a validity condition on the abstraction relationship between these two formal descriptions of the device's behaviour. Each variable c_i in a correctness theorem of this form is assumed to have logical type $num{\to}ty_1$, and represents a sequence of externally observable values of type ty_1 at points of time on a concrete time-scale. The function f is a data abstraction function of type $ty_1{\to}ty_2$ that maps concrete values of type ty_1 in the model to abstract values of type ty_2 in the specification. The function g is a time mapping from the abstract time-scale used in the specification to the concrete time-scale used in the model.

A correctness statement of the general form shown above relates a detailed design model to a more abstract specification of required behaviour by means of a point-wise mapping from the free variables in the model to corresponding values in the specification. Each temporally detailed sequence c_i of concrete values in the model is mapped to a temporally abstract sequence '$\mathsf{f} \circ c_i \circ \mathsf{g}$' of abstract values by composition on the left with the data abstraction function f and composition on the right with the time mapping g. A correctness statement of this general kind therefore involves both temporal abstraction and data abstraction. It is furthermore assumed that the abstract constraint '$S[a_1, \ldots, a_n]$' may be only a partial specification of required behaviour, in which case the specification is also a behavioural abstraction of the model.

When the functions f and g in a correctness statement of the form shown above are both *identity* functions, a theorem of this kind simply expresses a relationship of behavioural abstraction between the model and the specification (in this case, the abstract and concrete types ty_1 and ty_2 are in fact the same). Similarly, when only the data abstraction function f is the identity function, a theorem of this form expresses a relationship of temporal and behavioural abstraction; and when only the time mapping g is the identity function, a theorem of the form shown above states correctness as a relationship of data and behavioural abstraction.

In what follows, a correctness statements written in the form shown above can be assumed to be qualified by a validity condition $C[c_1, \ldots, c_n]$ which is non-trivial, in the sense that it is not simply satisfied for *every* possibly combination of values for the free variables c_1, $\ldots$, c_n. An abstraction relationship which is not qualified by a validity condition will be written:

$$\vdash M[c_1, \ldots, c_n] \supset S[\mathsf{f} \circ c_1 \circ \mathsf{g}, \ldots, \mathsf{f} \circ c_n \circ \mathsf{g}] \tag{4.2}$$

I.e. correctness will be stated in this case without a validity condition on the satisfaction relationship between the model and the abstract specification.

4.1.7.1 An Abbreviated Notation for Satisfaction

To simplify the discussion of abstraction in the remaining sections of this chapter, the following notation is introduced to abbreviate correctness statements of the form explained above. Any correctness statement of the general form given by theorem-scheme (4.1) can be written:

$$\vdash C[c_1, \ldots, c_n] \supset (M[c_1, \ldots, c_n] \supset S[F\ c_1, \ldots, F\ c_n]) \tag{4.3}$$

where F is the higher order *abstraction function* denoted by $\lambda c.\, \mathsf{f} \circ c \circ \mathsf{g}$. A correctness theorem of this form will therefore be abbreviated by writing:

$$\vdash C[c_1, \ldots, c_n] \supset M[c_1, \ldots, c_n] \underset{F}{\mathsf{sat}} S[a_1, \ldots, a_n] \tag{4.4}$$

Whenever a correctness statement is written using this notation,[4] it is assumed that F stands for an abstraction function of the form $\lambda c.\, \mathsf{f} \circ c \circ \mathsf{g}$, where the function f is a data abstraction function and the function g is a time mapping. The notation used in a correctness statement of this kind should be regarded as a metalinguistic abbreviation for a correctness theorem of the form given by theorem-scheme (4.3). Correctness theorems which are not qualified by a validity condition (i.e. theorems of the form given by theorem-scheme (4.2) shown above) can also be abbreviated using this notation.

4.1.7.2 Notation for Behavioural Abstraction

In an abbreviated correctness theorem written using the metalinguistic notation introduced above, it is always assumed that the function F has the form $\lambda c.\, \mathsf{f} \circ c \circ \mathsf{g}$, for some data abstraction function f and time mapping g. In the sections that follow, however, it will not be necessary to mention particular data abstraction functions and time mappings—except in the following special case.

[4]Although this notation expresses a concept of satisfaction which is similar to that formalized by the sat relation in CSP [49], the '$\underset{F}{\mathsf{sat}}$' notation used here is not the same as 'sat' in CSP.

When both f and g are identity functions, a correctness theorem of the kind discussed above simply expresses a relationship of behavioural abstraction between the model and the specification. In this case, a correctness statement will be abbreviated by:

$$\vdash C[c_1, \ldots, c_n] \supset M[c_1, \ldots, c_n] \underset{I}{\mathsf{sat}} S[c_1, \ldots, c_n]$$

Here, the function I is simply the identity function that maps each variable c_i in the model to the same value c_i in the specification. A theorem written in this form expresses a relationship of behavioural abstraction between the model and the specification (i.e. the relation $\underset{I}{\mathsf{sat}}$ can just be read as logical implication). A behavioural abstraction which is not qualified by a validity condition will be written using the same notation, but without the condition '$C[c_1, \ldots, c_n]$'.

4.1.7.3 Omission of Free Variables

When it is not necessary to draw attention to the particular free variables in the logical terms involved in correctness theorems of the kind discussed above, these variables will simply be omitted. In this case, correctness statement involving abstraction will be abbreviated by writing:

$$\vdash C \supset M \underset{F}{\mathsf{sat}} S \qquad \text{and} \qquad \vdash M \underset{F}{\mathsf{sat}} S$$

When this abbreviated notation is used, it assumed that C and M represent a validity condition and a design model respectively, and stand for logical terms that contain free variables c_1, …, c_n. It is also assumed that S represents an abstract specification of required behaviour, and stands for a logical term containing free variables a_1, …, a_n ranging over *abstract* values. Again, the function F is assumed to be a function of the form $\lambda c. \mathsf{f} \circ c \circ \mathsf{g}$. Finally, when only behavioural abstraction is involved, correctness statements will be written:

$$\vdash C \supset M \underset{I}{\mathsf{sat}} S \qquad \text{and} \qquad \vdash M \underset{I}{\mathsf{sat}} S$$

where I stands for the identity mapping (as discussed in the previous section).

4.1.8 Abstraction and Hierarchical Verification

In the simple inverter example given in Chapter 3, the correctness proof was based on a single model of the entire circuit, constructed by composing the formal specifications of all the primitive components used in its design. For designs of any considerable size, however, this direct approach is usually impractical. Instead, the design of the device which is to be proved correct must be structured into a hierarchy of models, and its correctness demonstrated by *hierarchical verification*.

4.1.8.1 Hierarchical Verification

Figure 4.2 shows a simple example of hierarchical verification. The design to be verified is structured into a two-level hierarchy of components. At the top level of this hierarchy (level 0), the design consists of two components S_1 and S_2, connected together by the internal wire z. At this level, these two components are considered to be primitive devices, and are modelled formally by the terms S_1 and S_2. The term M models the behaviour of the entire device at this level. It is constructed by composing S_1 and S_2 and hiding the internal wire z. The correctness statement at this level of the proof asserts that the model M satisfies the specification S, which describes the intended behaviour of the entire design.

At the next level down (level 1), the terms S_1 and S_2 become specifications of required behaviour for the two devices modelled by M_1 and M_2. These two models are constructed from the specifications of the primitive components P_1, P_2, P_3, and P_4. At this level, there are two separate correctness theorems to be proved. These two theorems assert that the devices modelled by M_1 and M_2 correctly implement the abstract behaviours required by the specifications S_1 and S_2 respectively. It follows from this, and from the correctness result for M proved at level 0, that wiring together the two devices modelled by M_1 and M_2 gives a concrete implementation of the entire design which is correct with respect to the top-level abstract specification S.

Level 0

Model:

$$\vdash M \;=\; \exists z.\, S_1 \wedge S_2$$

Correctness:

$$\vdash M \underset{F}{\mathsf{sat}} S$$

Level 1

Models:

$$\vdash M_1 \;=\; \exists x.\, P_1 \wedge P_2$$
$$\vdash M_2 \;=\; \exists y.\, P_3 \wedge P_4$$

Correctness:

$$\vdash M_1 \underset{G}{\mathsf{sat}} S_1$$
$$\vdash M_2 \underset{G}{\mathsf{sat}} S_2$$

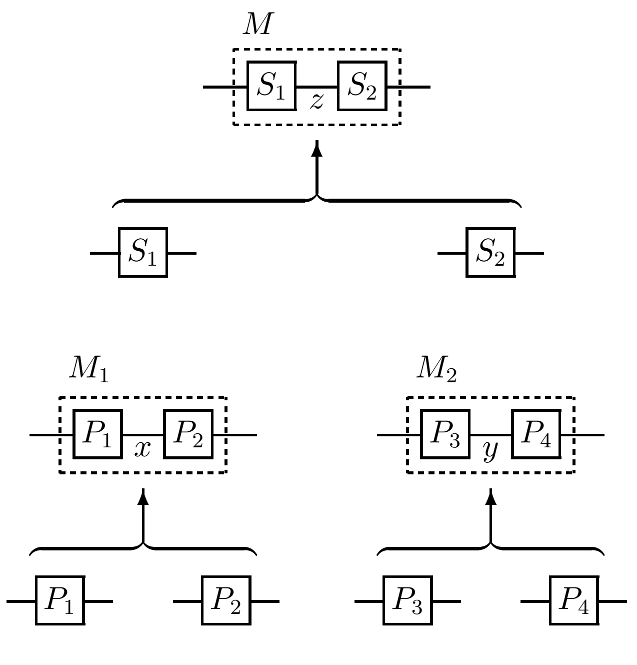


Figure 4.2: Hierarchical Verification

This hierarchical approach to hardware verification is possible in logic because both design models and specifications of required behaviour have exactly the same syntactic status. Both are simply boolean terms; and the model-building operations of composition ('$\wedge$') and hiding ('$\exists$') can be applied to both. Logical terms which are used as abstract *specifications* at one level in a hierarchical proof can therefore be treated as *models* at the next higher level. In a formalism in which specifications of required behaviour and models are syntactic entities of two distinct kinds, this direct approach to hierarchical verification is not possible.

An important advantage of hierarchical verification, of course, is that if multiple instances of the same kind of component are used at one level of the hierarchy, then a design for that kind of component has to be verified only once at the next level down. Furthermore, if a component is proved correct with respect to a concise, *abstract* specification of required behaviour, then this can be used in place of a more detailed design model at the next higher level. In the hierarchical proof shown in Figure 4.2, for example, the design model M at the top level of the proof is constructed from the abstract specifications S_1 and S_2, rather than the more detailed concrete models M_1 and M_2. This gives a more tractable model at the top level than would otherwise be possible, and helps to control the complexity of the proof at that level. In this way, abstraction mechanisms, together with hierarchical structuring and regularity, can help control the complexity of large correctness proofs.

4.1.8.2 Putting Hierarchical Proofs Together

A hierarchical proof of correctness for the design of large device will typically involve many intermediate levels of specification between the fully concrete design model and the top-level abstract specification of required behaviour. At each level in the hierarchy, correctness theorems will relate each subcomponent to an abstract specification at the next higher level. In the general case, there will be several independent correctness theorems at each level in the hierarchy, one for each different kind of 'primitive' component used at that level. To complete such a proof, it is necessary to combine the intermediate correctness results that relate models and specifications at each level of the hierarchy into a single correctness theorem that relates a complete and fully concrete model of the entire design to the top-level abstract specification of intended behaviour.

For example, in the simple two-level hierarchical correctness proof discussed above, there are three separate correctness theorems:

$$\text{Level 1:}\quad \vdash M_1 \underset{G}{\mathsf{sat}} S_1 \quad \vdash M_2 \underset{G}{\mathsf{sat}} S_2 \qquad\quad \text{Level 0:}\quad \vdash M \underset{F}{\mathsf{sat}} S$$

Two correctness theorems are derived at the lowest level in the hierarchy. These relate the concrete models M_1 and M_2 to the intermediate abstract specifications

S_1 and S_2 respectively. At the next level, a third correctness theorem relates the model M to the top-level abstract specification S. The intermediate model M at this level is constructed from the specifications S_1 and S_2 for the concrete components at the next level down.

Proving these three theorems shows only that each component in the hierarchy is correct with respect to its abstract specification. To complete the proof of this device, one must also derive a correctness theorem for the entire design with respect to the top-level abstract specification. That is, in addition to proving the three separate correctness theorems shown above, one must also derive from these theorems a correctness theorem for a model of the entire design:

$$\vdash (\exists z.\, (\exists x.\, P_1 \wedge P_2) \wedge (\exists y.\, P_3 \wedge P_4)) \underset{H}{\mathsf{sat}} S \qquad \text{where } H = F \circ G$$

This theorem states that a complete and fully detailed design model constructed from the concrete primitive specifications P_1, P_2, P_3, and P_4 satisfies the top-level abstract specification of required behaviour S. The abstraction function H used to formulate satisfaction in this correctness theorem is simply the composition of the functions F and G used to formulate satisfaction at levels 0 and 1 respectively.

To obtain a correctness theorem for a hierarchically-structured design from a collection of separate correctness results for each level of the hierarchy, the satisfaction relation used to formulate correctness must satisfy the three rules shown in Figure 4.3 below. The sat-TRANS rule requires the satisfaction relation which is used to formulate correctness to be transitive. If this rule holds, then any two correctness theorems $\vdash M_1 \underset{F}{\mathsf{sat}} M_2$ and $\vdash M_2 \underset{G}{\mathsf{sat}} S$ derived for adjacent levels in a hierarchical proof can be composed to obtain a correctness theorem

- sat-TRANS: $$\frac{\vdash M_1 \underset{F}{\mathsf{sat}} M_2 \qquad \vdash M_2 \underset{G}{\mathsf{sat}} S}{\vdash M_1 \underset{G\circ F}{\mathsf{sat}} S}$$

- $\wedge$-MONO: $$\frac{\vdash M_1 \underset{F}{\mathsf{sat}} S_1 \qquad \vdash M_2 \underset{F}{\mathsf{sat}} S_2}{\vdash (M_1 \wedge M_2) \underset{F}{\mathsf{sat}} (S_1 \wedge S_2)}$$

- $\exists$-EXT: $$\frac{\vdash M \underset{F}{\mathsf{sat}} S}{\vdash (\exists c.\, M) \underset{F}{\mathsf{sat}} (\exists a.\, S)}$$

Figure 4.3: Three Meta-theorems about Satisfaction.

$\vdash M_1 \underset{G \circ F}{\mathsf{sat}} S$ which relates the detailed model M_1 to the abstract specification S. The abstraction function F in this rule maps the free variables in the concrete model M_1 to abstract values in the intermediate model M_2, and the abstraction function G maps values in the intermediate model M_2 to values in the top-level specification S. When the two correctness theorems are composed, the abstraction function used in the resulting satisfaction relation is simply the composition of the functions G and F.

The $\wedge$-MONO rule states that the operation of composition is monotonic with respect to satisfaction. If this rule holds of the satisfaction relation used to state correctness, then a correctness theorem for a composite hardware device can be derived from separate correctness theorems for each of its subcomponents. The rule states that if the models M_1 and M_2 satisfy the abstract specifications S_1 and S_2 respectively, then the composition of the models will satisfy the conjunction of the abstract specifications. It is assumed that the same abstraction function F is used to formulate correctness for each of the two models M_1 and M_2. In the $\wedge$-MONO rule, the function F is also the abstraction function for the derived correctness result.

The $\exists$-EXT rule states that satisfaction must be preserved when internal wires are hidden using the hiding operator $\exists$. The variables c and a in this rule stand for corresponding concrete and abstract values in the model M and the specification S respectively. The rule states that if the model M satisfies the specification S, then the model '$\exists c.\, M$' obtained by hiding the internal value c will satisfy the abstract specification '$\forall a.\, S$' obtained by existentially quantifying the abstract variable a. The abstraction function F in the derived correctness result is the same as the abstraction function in the original theorem.

In general, meta-theorems similar to the three rules shown in Figure 4.3 must hold for any formulation of satisfaction for the correctness of components in a hierarchically-structured correctness proof. For the general class of correctness theorems introduced in Section 4.1.7 (here abbreviated using the $\underset{F}{\mathsf{sat}}$ notation) it is straightforward to prove that these rules hold. Similar rules for combining correctness results can also be derived when several different abstraction functions are used in a single correctness statement, or when several concrete variables in a design model are mapped to a single abstract value in an abstract specification.

Given these rules about satisfaction, the separate correctness theorems derived for each level of a hierarchically structured design can always be combined to obtain a correctness statement that relates a fully detailed model for the entire design to the top-level abstract specification of required behaviour. There is a straightforward systematic procedure for deriving a correctness theorem for a hierarchically structured design from the correctness theorems obtained for its subcomponents at each level in the hierarchy. For each composite component in the hierarchy, a correctness theorem can be obtained by combining the correctness results for its constituent parts using the two rules $\wedge$-MONO and $\exists$-EXT. The rule sat-TRANS can be used to compose the correctness results that link adjacent levels in the hierarchy to obtain a correctness theorem that relates the most concrete design model to the top-level abstract specification. For the simple example of hierarchical verification shown above in Figure 4.2, the process of assembling the separate correctness theorems obtained at each level into a correctness theorem for the entire design consists of the derivation shown below:

1. $\vdash M_1 \underset{G}{\mathsf{sat}} S_1$ [correctness of M_1]

2. $\vdash M_2 \underset{G}{\mathsf{sat}} S_2$ [correctness of M_2]

3. $\vdash (M_1 \wedge M_2) \underset{G}{\mathsf{sat}} (S_1 \wedge S_2)$ [$\wedge$-MONO: **1**, **2**]

4. $\vdash (\exists z.\, M_1 \wedge M_2) \underset{G}{\mathsf{sat}} (\exists z.\, S_1 \wedge S_2)$ [$\exists$-EXT: **3**]

5. $\vdash (\exists z.\, S_1 \wedge S_2) \underset{F}{\mathsf{sat}} S$ [correctness of M]

6. $\vdash (\exists z.\, M_1 \wedge M_2) \underset{F \circ G}{\mathsf{sat}} S$ [sat-TRANS: **4**, **5**]

7. $\vdash (\exists z.\, (\exists x.\, P_1 \wedge P_2) \wedge (\exists y.\, P_3 \wedge P_4)) \underset{F \circ G}{\mathsf{sat}} S$ [**6**, and definitions of M_1, M_2]

In the first four steps of this proof, the two correctness theorems for the each of the two components at level 1 are combined using the rules $\wedge$-MONO and $\exists$-EXT to obtain the correctness theorem for the entire design which is shown in step **4**. The abstract specification '$\exists z.\, S_1 \wedge S_2$' in this theorem is just the design model at used at level 0, and the desired correctness theorem therefore follows immediately by the transitivity property of satisfaction expressed by the rule sat-TRANS. For a hierarchical design of more than two levels, the sequence of deductions illustrated by this example is applied recursively.

4.1.8.3 Hierarchical Verification and Validity Conditions

The rules given in the previous section are somewhat over-simplified, in that they do not take validity conditions into account. In a hierarchical proof where validity conditions are involved, a more complex process of reasoning is required to combine the correctness theorems obtained at each level in the hierarchy. In particular, one must show that any validity conditions which arise at *intermediate* levels in the hierarchy are satisfied, since the final result must be a correctness statement that relates a fully concrete design model to the top-level abstract specification. In general, this may involve strengthening the intermediate abstract specifications in the hierarchy. This in turn may entail strengthening the validity conditions at lower levels of the hierarchy.

Rules for combining correctness results which are similar to those discussed in the previous section but which also take validity conditions into account can be

- sat-TRANS: $$\frac{\vdash C_1 \supset M_1 \underset{F}{\mathsf{sat}} (C_2 \wedge M_2) \quad \vdash C_2 \supset M_2 \underset{G}{\mathsf{sat}} S}{\vdash C \supset M_2 \underset{G \circ F}{\mathsf{sat}} S}$$

- $\wedge$-MONO: $$\frac{\vdash C_1 \supset M_1 \underset{F}{\mathsf{sat}} S_1 \quad \vdash C_2 \supset M_2 \underset{F}{\mathsf{sat}} S_2}{\vdash (C_1 \wedge C_2) \supset (M_1 \wedge M_2) \underset{F}{\mathsf{sat}} (S_1 \wedge S_2)}$$

- $\exists$-EXT: $$\frac{\vdash C \supset M \underset{F}{\mathsf{sat}} S}{\vdash C \supset (\exists c.\, M) \underset{F}{\mathsf{sat}} (\exists a.\, S)} \quad [c \text{ not free in } C]$$

Figure 4.4: Extended Meta-theorems about Satisfaction.

formulated in a number of different ways. One possible formulation consists of the three extended meta-theorems about satisfaction shown in Figure 4.4 above.

The most straightforward of these rules is the extended $\wedge$-MONO rule. This rule states that the validity condition for a composite design is just the logical conjunction of the validity conditions for its components. The extended sat-TRANS rule reflects the fact that intermediate validity conditions must be satisfied in order to compose the correctness results for two adjacent levels of a hierarchical proof. Here, the abstract specification '$C_2 \wedge M_2$' at the lower level consists of both the model and the validity condition for the higher level. In the $\exists$-EXT rule, the validity condition C on the derived correctness theorem is the same as the validity condition for the given correctness theorem. But the rule can be applied only when the existentially quantified variable c does not occur in the condition C. That is, the validity condition may not *directly* constrain the value of the wire which is to be hidden.

In addition to the three extended meta-theorems shown in Figure 4.4, the following VCOND rule is needed:

- VCOND: $$\frac{\vdash (C_1 \wedge M) \supset C_2 \quad \vdash C_2 \supset M \underset{F}{\mathsf{sat}} S}{\vdash C_1 \supset M \underset{F}{\mathsf{sat}} S}$$

This rule allows a strong validity condition C_2 to be replaced by a weaker validity condition C_1 when it is known that the constraint imposed by C_1 and the model M is sufficient to ensure that C_2 is satisfied. This rule is needed in order to simplify validity conditions that arise in connection with the three rules shown in Figure 4.4. For example, the VCOND rule may be needed to eliminate a free variable c in the validity condition of a correctness theorem obtained using the $\wedge$-MONO rule, so that the $\exists$-EXT theorem can subsequently be applied.

An example proof which illustrates the process of reasoning represented by these extended meta-theorems about satisfaction is given in Section 6.3.7 of the case study discussed in Chapter 6.

4.2 Abstraction between Models

The type of abstraction discussed in preceding sections of this chapter is called abstraction 'within' a model because it takes place within the context of a fixed choice of formal specifications for the constituent parts of a device. A formal specification is written for each different kind of primitive component used in the design of the device to be verified. A design model is then constructed by applying the operations of composition and hiding to instances of these primitives, and a correctness statement is proved which relates this design model to a more abstract specification of required behaviour. This correctness statement expresses relationship of abstraction between the design model for the device in question and an appropriate abstract specification of required behaviour.

In this section, a brief overview is given of the formalization in logic of a different type of abstraction relationship—abstraction *between* models. Here, the concern is not with expressing the correctness of an individual *design*, but with the relationship between two formal models for the primitive hardware components from which designs are built. In Chapter 7, a detailed example is given to show how this second type of abstraction relationship can be expressed formally in higher order logic. Only a brief overview of the basic idea of abstraction between models is therefore given here.

In logic, a *model* of hardware behaviour—in the sense of a formal basis for describing the behaviour of any particular hardware device—is just a collection of logical terms that describe the primitive components from which hardware designs are constructed. An example is the simple model of CMOS transistor behaviour defined in Chapter 3, consisting of the primitive specifications for power, ground, and N-type and P-type transistors which were defined in Section 3.4.2. The idea of a relationship of abstraction between *two* such models, both of which describe the same collection of primitive hardware components, can be expressed formally in logic as an assertion about the correctness results that can be proved about individual design descriptions constructed using the two models.

Suppose, for example, that $\mathrm{P}_c=\{\mathrm{P}^c_1, \ldots, \mathrm{P}^c_n\}$ and $\mathrm{P}_a=\{\mathrm{P}^a_1, \ldots, \mathrm{P}^a_n\}$ are two sets of formal specifications for the primitive components used in hardware designs, i.e. two models of hardware behaviour in general. Informally, the model P_a is an abstraction of the model P_c if the primitive specifications that constitute P_a are, in some sense, abstractions or simplifications of the primitive specifications that constitute P_c. In this case, the abstract model of hardware behaviour P_a will capture only some of the aspects of device behaviour which are modelled by the more detailed primitives P_a. This means that there will be correctness statements—i.e. assertions about the behaviour of particular hardware designs—which can be formulated using the detailed primitives P_c, but which cannot be expressed the more abstract primitives P_a. A formalization of the abstraction relationship between these two models of hardware behaviour must therefore be based on correctness statements which can be expressed in both models.

Furthermore, because the abstract primitives P_a are simpler than the detailed primitives P_c, they are also likely to be less 'accurate' formal descriptions of the way actual primitive devices behave. In general, inaccuracy in a formal model of hardware behaviour is manifested by the ability to prove the correctness of hardware devices with respect to specifications that do not reflect the actual behaviour of the physical device itself. This means that there will be correctness theorems which can be proved in the abstract model of hardware behaviour given by the abstract primitives P_a, but which cannot be proved using the more detailed model given by the detailed primitives P_c. That is, P_c will be a more accurate formal model of hardware than P_a. For some devices, it may be possible to prove correctness results in *both* models. But there will also be circuit designs which can be proved correct in the abstract model P_a, but which, according to the more detailed model P_c, are in fact incorrect. A formal characterization of the abstraction relationship between these two models reflects this difference between the relative accuracy of the two models of hardware behaviour.

These ideas can be expressed formally in higher order logic as follows. Let the term '$M_a[a_1, \ldots, a_n]$' be an arbitrary design model constructed from the abstract primitive specifications in the set P_a. The most general form of a correctness theorem for such a model is a behavioural abstraction of the form:

$$\textbf{1:}\quad M_a[a_1, \ldots, a_n] \underset{I}{\mathsf{sat}} S[a_1, \ldots, a_n]$$

since any correctness statement for the model $M_a[a_1, \ldots, a_n]$ which is based on data or temporal abstraction can be expressed by logical implication. Suppose that the term $M_c[c_1, \ldots, c_n]$ is a formal model of the same design, constructed using the detailed primitive specifications in the set P_c. A proposition which asserts the correctness of this more detailed design model with respect to the *same* specification as used in the behavioural abstraction shown above will have the general form:

$$\textbf{2:}\quad \vdash C[c_1, \ldots, c_n] \supset M_c[c_1, \ldots, c_n] \underset{F}{\mathsf{sat}} S[a_1, \ldots, a_n]$$

where $C[c_1, \ldots, c_n]$ and F are an appropriate validity condition and an appropriate abstraction function, respectively. An abstraction function is needed here because the model in this correctness statement is more detailed than the model in the behavioural abstraction shown above. A validity condition may be necessary for the same reason.

These two propositions express the same (or, at least, equivalent) correctness assertion about the design of a particular device modelled using the two sets of primitive specifications P_c and P_a. Proposition **1** asserts the correctness of the design modelled using the abstract primitives P_a with respect to the specification $S[a_1, \ldots, a_n]$. Proposition **2** is a translation of this correctness assertion into the 'language' of the more detailed formal model of hardware behaviour given by the primitives P_c. For clarity, the two correctness statements can be abbreviated by:

$$\textbf{1:}\;\; M_a \underset{I}{\mathsf{sat}} S \qquad \text{and} \qquad \textbf{2:}\;\; C \supset M_c \underset{F}{\mathsf{sat}} S$$

Given this translation from an arbitrary correctness assertion formulated in the abstract model of hardware behaviour P_c into an equivalent correctness assertion expressed in the more detailed model P_c, the abstraction relationship *between* these two models can be characterized formally as follows. The idea that the P_a is an abstraction of P_c can be expressed formally by the assertion that any correctness statement which is (1) expressible in both models, and (2) provable in the detailed model of hardware behaviour given by P_c, is also provable in the more abstract model of hardware behaviour given by P_a. That is, for an arbitrary design modelled by M_c and M_a and an arbitrary specification S:

$$\text{if} \quad \vdash C \supset M_c \underset{F}{\mathsf{sat}} S \quad \text{then} \quad \vdash M_a \underset{I}{\mathsf{sat}} S$$

The idea that the abstract model P_a may be less accurate than the detailed model P_c can be expressed formally by the assertion that the converse implication holds for only *some* designs. That is, for only some pairs of design models M_c and M_a will the converse implication:

$$\text{if} \quad \vdash M_a \underset{I}{\mathsf{sat}} S \quad \text{then} \quad \vdash C \supset M_c \underset{F}{\mathsf{sat}} S$$

hold for all specifications S. For *these* designs, every correctness result obtained using the abstract (i.e. simple) formal model of behaviour also provable using the more detailed (but also more complex) model. Proving this converse implication shows that if only certain correctness assertions are of interest, then the abstract model of hardware behaviour given by the abstract primitives P_a is a sound basis for reasoning about the class of designs for which this implication holds.

This section has given only a general overview of how the idea of an abstraction relationship between two models of hardware can be expressed formally in logic. A detailed example is given in Chapter 7.

4.3 Related Work

The idea of proving correctness with respect to *abstract* specifications is well known in software verification, and many of the ideas presented in this chapter have analogues in formal methods for reasoning about program correctness. In particular, the idea of using abstraction functions to relate formal specifications at two different levels of abstraction was inspired by the approach to reasoning about the correctness of software data representations originally proposed by Hoare in [50].

The idea of expressing the correctness of hardware designs by theorems similar in form to those discussed in Section 4.1 is reasonably well-known. Correctness theorems for particular hardware devices based on all three types of abstraction relationship discussed in Section 4.1 are presented in a number of papers in the hardware verification literature (expressed, perhaps, in a slightly different form). The aim of this chapter, however, was not to give particular examples of abstraction, but to provide a motivated and general account of some basic techniques for expressing certain abstraction relationships in higher order logic. Again, the basic idea of hierarchical verification is, of course, not new. But the general rules for putting hierarchical proofs together presented in Sections 4.1.8.2 and 4.1.8.3 have not been stated explicitly in previous work.

Interpretation of Theories in First Order Logic

One other researcher who has concentrated on the role of abstraction in hardware verification is Hans Eveking, whose basic approach to verification was described in Chapter 3. In an early work [23], Eveking discusses three abstraction mechanisms for suppressing detail in formal descriptions of hardware behaviour. The first of these is a form of 'structural' abstraction, in which the values on internal wires are hidden from abstract specifications. The second abstraction mechanism uses CONLAN assertions—logical assumptions on which proofs are based—to express behavioural abstraction. A form of temporal abstraction is provided by the third of Eveking's abstraction mechanisms. He calls this mechanism 'partially defined behaviour in the time-dimension' and gives an example in which he abstracts from a timing level flip flop description to a register transfer level description of a conditional transfer device.

In [24], Eveking shows how abstraction can be expressed by interpreting one first order theory in another. An interpretation of a theory T_2 in a theory T_1 is a syntactic translation of the axioms of T_2 such that each translated axiom is a provable theorem of T_1. In [24], Eveking shows how this concept can be used to express the correctness of a simple clocked synchronous device with respect to a temporally abstract specification.

Chapter 5

Data Abstraction

To make effective use of data abstraction in higher order logic, it is generally necessary to define special-purpose logical types for use in both models of hardware designs and formal specifications of required behaviour, since the formal properties which these types are required to have will depend on the kind of behaviour being specified, on the level of abstraction at which the devices are described, and on how accurate the specifications are intended to be.

This means that no fixed collection of logical types is likely to be an adequate basis for specifying all devices. The basic logical types *bool* and *num→bool*, for example, are sufficient for specifying hardware behaviour at the level of abstraction where the devices used are flip flops and combinational logic gates. But at the level of abstraction where the primitive components are transistors, an accurate model of behaviour has to account for more kinds of values on the wires of a device than can be represented by the boolean truth-values T and F. It may be necessary to represent electrical signals of several different strengths, or to represent 'undefined' or 'floating' values. The types *bool* and *num→bool* are also insufficient for specifications at the register-transfer level of abstraction, where it is often necessary to specify behaviour not in terms of the values on single wires but in terms of vectors of bits and arithmetical operations on fixed-width binary words. And at the architecture level, concise specifications may require comparatively complex abstract data types, such as stacks and queues.

In the formulation of higher order logic used here, a new type can be introduced into the logic only by *defining* it using the type definition mechanism described in Chapter 2. An appropriate representation must be found for the values of the new type, and an abstract characterization for the new type must be *derived* from its definition. This can involve a significant amount of formal proof. To facilitate the introduction of special-purpose logical types for describing hardware at various different levels of data abstraction, a method was developed to *automate* the formal definitions of certain types in the HOL system. This chapter provides an overview of the class of types definable by this method, and gives two examples to show how these types can be used to support formal reasoning about hardware behaviour where data abstraction is involved.

5.1 Defining Concrete Types in Logic

Many of the sorts of values which arise naturally in specifications of hardware behaviour—especially at lower levels of abstraction—can be represented by the values of what are commonly called 'concrete data types'. These are types whose values are generated by a set of *constructors* (i.e. functions) which yield concrete representations for these values. Examples include types which denote finite sets of atomic values (enumerated types), types which denote sets of structured values (record types) or finite disjoint unions of structured values (variant records), and types which denote sets of recursive data structures (recursive types).

To make types of this kind readily available in logic, a method was developed for defining an arbitrary, possibly recursive, concrete type automatically in the HOL system. This method is based on the two-step process for introducing a new logical type explained in Section 2.1.7.2. First, the required type is defined formally using the primitive rule for type definitions described in Section 2.1.7.1. An abstract 'axiomatization' for the newly-defined type is then derived by formal proof from its definition. This consists of a single theorem of higher order logic which provides a complete and abstract characterization of the newly-defined type, and forms the basis for all further proofs about it.

Full details of the logical basis for this mechanization of type definitions in HOL are given in Appendix A. The sections that follow provide an overview of the class of types definable by this method, the form of the abstract axioms[1] used to characterize these types in logic, and some fundamental properties which follow from these axioms. In Section 5.1.1, a preliminary example is provided to illustrate the general approach. Concrete types in general are then discussed in Section 5.1.2. A brief account is then given in Section 5.1.3 of the mechanization in the HOL system of the method by which these types are defined.

5.1.1 An Example: A Type of Lists

A simple example of a recursive concrete type is the type of finite, homogeneous lists. This type can be described informally[2] by the equation shown below.

$$\mathit{list} \quad ::= \quad \mathsf{Nil} \quad | \quad \mathsf{Cons}\ \alpha\ \mathit{list} \tag{5.1}$$

This equation is an informal recursive 'definition' of the set of all finite-length lists of values of type α. The symbols Nil and Cons are the usual *constructors* by which list values are formed. The symbol Nil stands for the empty list, and the symbol

[1]In referring to these theorems as 'axioms', quotation marks will henceforth be omitted—it being understood that they are not *axioms*, in the sense of having been postulated without proof, but are derived *abstract characterizations* of the corresponding defined logical types.

[2]Here, and in what follows, *informally* means not in the language of higher order logic.

Cons stands for the operation which constructs a list of length $n+1$ by adding a value of type α onto the front of a list of length n. The set of values defined informally by this recursive equation consists of the smallest set that contains the list denoted by Nil and is closed under the list-forming operation denoted by Cons.

5.1.1.1 Characterizing Lists in Logic

To make the set of values described informally by the recursive equation shown above into a logical type, a definition for a non-primitive type expression '$(\alpha)list$' must be introduced formally by means of the type definition mechanism explained in Chapter 3. The details of this definition will not be discussed here (for this, see Appendix A) but once the type $(\alpha)list$ has been defined, its definition can be used to prove the abstract characterization of lists consisting of the single theorem of higher order logic shown below.

$$\vdash \forall e\, f.\, \exists! fn.\, (fn\ \mathsf{Nil} = e) \wedge (\forall h\, t.\, fn(\mathsf{Cons}\ h\ t) = f\ (fn\ t)\ h\ t) \tag{5.2}$$

This theorem, which captures the essential properties of the defined type $(\alpha)list$ and therefore amounts to an axiomatization of finite-length lists, is analogous to the primitive recursion theorem for natural numbers discussed in Section 2.1.6.1. It asserts that a function $fn{:}(\alpha)list{\to}\beta$ can be defined uniquely by 'primitive recursion' on lists—i.e. by giving a value e to define $fn\ \mathsf{Nil}$, and a function f to define $fn(\mathsf{Cons}\ h\ t)$ in terms of $(fn\ t)$, h, and t.

5.1.1.2 Theorems about Lists

All the usual properties of lists follow from theorem (5.2) above. In particular, it is possible to derive from this abstract characterization of lists the three fundamental properties of lists shown below.

$$\begin{array}{l}
\vdash \forall h\, t.\, \neg(\mathsf{Nil} = \mathsf{Cons}\ h\ t) \\
\vdash \forall h_1\, h_2\, t_1\, t_2.\, (\mathsf{Cons}\ h_1\ t_1 = \mathsf{Cons}\ h_2\ t_2) \supset ((h_1 = h_2) \wedge (t_1 = t_2)) \\
\vdash \forall P.\, (P\ \mathsf{Nil} \wedge \forall t.\, P\ t \supset \forall h.\, P(\mathsf{Cons}\ h\ t)) \supset \forall l.\, P\ l
\end{array}$$

These three theorems about lists are analogous to Peano's postulates for the natural numbers. The first two theorems state that Nil and Cons yield distinct values of type $(\alpha)list$ and that Cons is one-to-one. The third theorem states the validity of structural induction on lists, and provides the formal means for proving properties of lists by structural induction. It also follows from this induction theorem that every value of type $(\alpha)list$ is either equal to Nil, or is constructed from Nil by finitely many applications of the function Cons.

5.1.1.3 Recursive Function Definitions on Lists

In addition to the three basic properties of lists discussed above, defining equations for *primitive recursive functions* on lists can also be derived from theorem (5.2). As was discussed in Chapter 2, function constants that satisfy recursive equations are not directly definable in higher order logic by the primitive rule for constant definitions. To define a constant which denotes a recursive function—indeed to 'obtain' such a function, whether denoted by a constant or not—one must prove that the desired recursive equation is in fact satisfiable. It follows immediately from theorem (5.2), however, that any primitive recursive function definition on lists can be satisfied by a (unique) total function. This provides a direct means for constructing such functions in logic, and for justifying the introduction of function constants which denote them.

For example, given theorem (5.2) it is straightforward to define a constant which denotes a primitive recursive *length* function on lists. Specializing the variables e and f in a suitably type-instantiated version of theorem (5.2) so that $e = 0$ and $f = \lambda x\, y\, x.\, x{+}1$ yields (after some simplification) the theorem shown below.

$$\vdash \exists! fn.\, (fn\ \mathsf{Nil} = 0) \wedge (\forall h\, t.\, fn(\mathsf{Cons}\ h\ t) = (fn\ t){+}1)$$

This asserts the (unique) existence of a function which satisfies the usual primitive recursive definition of the length of a list. Given this theorem, a constant 'Length' can then be introduced to denote the function whose existence it asserts. This can be done formally by an appropriate non-recursive definition involving the primitive constant ε, as was explained in Section 2.1.6. The result is a pair of theorems, which constitute the usual primitive recursive 'definition' of the length of a list:

$$\begin{array}{lcl} \vdash \mathsf{Length}\ \mathsf{Nil} & = & 0 \\ \vdash \mathsf{Length}\ (\mathsf{Cons}\ h\ t) & = & (\mathsf{Length}\ t) + 1 \end{array}$$

Any function constant which satisfies a primitive recursive definition on lists can be defined formally in a similar way—i.e. by first specializing the variables e and f in theorem (5.2) to obtain a theorem which asserts the existence of the desired function, and then introducing a constant to name this function by a non-recursive definition involving ε. This method for defining primitive recursive functions on lists is completely analogous to the method for justifying primitive recursive definitions on the natural numbers which was discussed in Section 2.3.

5.1.2 Concrete Types in General

The type of lists discussed in the preceding sections is a simple instance of the general class of concrete types which can be defined automatically using the HOL implementation of the method explained in Appendix A. Every type which can be defined by this method can be described 'informally' by an equation which is similar in form to the equation (5.1) for lists shown on page 77. The general form of such an equation is shown below.

$$rty \quad ::= \quad \mathsf{C}_1\, ty \,\ldots\, ty \quad | \quad \cdots \quad | \quad \mathsf{C}_n\, ty \,\ldots\, ty \tag{5.3}$$

An equation of this kind is an informal 'definition' of a logical type rty with n different constructors: C_1, C_2, …, C_n. The symbol rty on the left-hand side of the equation is the name of the type (or type operator) being defined. Each type ty which occurs on the right-hand side of the equation must be either a type expression already present in the logic or the name rty itself. In any equation of this kind, at least one of the expressions $\mathsf{C}_i\ ty\ \ldots\ ty$ on the right-hand side of the equation must contain ty's which are all existing types of the logic, rather than the type rty being defined.

An equation of this kind is similar to a 'datatype' declaration in Standard ML [46]. It simply states the names of the constructors for the type it describes and the types of their arguments. When rty occurs somewhere among the ty's on the right-hand side of the equation shown above, the type rty denotes a set of recursively-defined structures (essentially a set of *trees*). When the symbol rty occurs *nowhere* among the ty's on the right-hand side of the equation, the type described by the equation denotes a disjoint union of n different kinds of 'records', each one of which is represented by one of the constructors C_1, C_2, …, C_n.

5.1.2.1 Characterizing Concrete Types in Logic

Any type described informally by an equation of the form shown above can be characterized formally in logic by a single theorem of the form:

$$\begin{array}{l} \vdash \forall f_1\ f_2\ \ldots\ f_n.\ \exists!fn{:}rty{\to}\alpha. \\ \qquad \begin{array}{rcl} \forall x_1 \ldots x_i.\ fn(\mathsf{C}_1\ x_1\ \ldots\ x_i) & = & f_1\ (fn\ x_1)\ \ldots\ (fn\ x_i)\ x_1\ \ldots\ x_i\ \wedge \\ \forall x_1 \ldots x_j.\ fn(\mathsf{C}_2\ x_1\ \ldots\ x_j) & = & f_2\ (fn\ x_1)\ \ldots\ (fn\ x_j)\ x_1\ \ldots\ x_j\ \wedge \\ \vdots & & \\ \forall x_1 \ldots x_k.\ fn(\mathsf{C}_n\ x_1\ \ldots\ x_k) & = & f_n\ (fn\ x_1)\ \ldots\ (fn\ x_k)\ x_1\ \ldots\ x_k \end{array} \end{array} \tag{5.4}$$

where the right-hand sides of the equations include recursive applications '$fn\ x$' only for variables x of type rty. (See, for example, theorem (5.2) above.) A theorem of this form asserts the unique existence of primitive recursive functions defined by cases on the constructors C_1, C_2, …, C_n. This is a slight extension of the *initiality* property by which structures of this kind are characterized in the 'initial algebra' approach to specifying abstract data types [28]. This property provides an abstract characterization of the type rty which is both succinct and complete, in the sense that it completely determines the structure of the values of rty up to isomorphism.

A major practical advantage of this characterization is its uniform treatment of all concrete types. Every type which can be described informally by an equation of the kind given by (5.3) is characterized by a single theorem of the same general form. This uniformity is the basis for the *efficient* automation in HOL of the process of defining these types and proving abstract axioms for them. Because the same form of derived axiomatization is used for every type, the axiom for any particular type can be inferred relatively easily from a pre-proved theorem stating that the property given by (5.4) holds for *all* concrete types. This is explained in more detail in Appendix A.

5.1.2.2 Theorems about Concrete Types

A further advantage of characterizing concrete types by theorems of the kind shown above is that many useful standard properties follow from these theorems in a uniform way, with relatively short formal proofs.

For example, from a theorem of the form given by (5.4) it is straightforward to prove an alternative characterization of the type *rty* which is analogous to the conventional axiomatization of the natural numbers given by Peano's postulates. For the type *rty* described by theorem-scheme (5.4), this characterization consists of the following fundamental properties: (1) the constructors C_1, C_2, ..., C_n yield distinct values of type *rty*; (2) each constructor which is not a constant is one-to-one; and (3) the principle of structural induction holds for *rty*. The three theorems about lists discussed in Section 5.1.1.2 show how these properties are stated formally for the type $(\alpha)list$. Similar theorems hold for any concrete type which can be described by an equation of the general form given by equation (5.3).

5.1.2.3 Recursive Function Definitions on Concrete Types

Another important property of the characterization of *rty* given by theorem-scheme (5.4) is that a theorem of this form provides a formal means for defining a wide class of useful functions on *rty*. When *rty* is a recursive type, a theorem of the form given by (5.4) can be used to prove the existence of any *primitive recursive* function on *rty* and to define constants which denote such functions. This is illustrated by the method for defining primitive recursive functions on lists explained in Section 5.1.1.3. Primitive recursive definitions of functions on any concrete recursive type can be justified by a process of reasoning which is similar to that illustrated by the example given in this section. When *rty* is a non-recursive type, the characterization given by the theorem-scheme (5.4) also provides a means for defining functions. In this case, the characterizing theorem for *rty* states the unique existence of functions defined by cases on its constructors, and this provides a simple and direct way of constructing particular instances of these functions. This is illustrated by an example given in Section 5.2.4.3.

5.1.3 Mechanization in HOL

To provide mechanized support for reasoning about concrete types of the kind described in the preceding sections, a collection of automatic theorem-proving tools was implemented in the HOL system. These tools were used in the HOL proofs of the theorems about hardware discussed in later sections of this chapter.

The main component of these tools is an ML procedure which carries out all the logical inferences which are necessary to define an arbitrary concrete type in higher order logic and to prove an abstract axiomatization for it. The user input to this programmed proof rule is an 'informal' specification of the logical type which is to be defined, in the form of an equation of the kind discussed above in Section 5.1.2. The output is an abstract characterization of the required logical type, in the form of an instance of theorem-scheme (5.4). This theorem is proved automatically, by purely logical inference, from an automatically-constructed formal definition of the particular concrete type requested by the user.

The theorem-proving tools implemented in HOL also include procedures for proving the 'standard' properties of concrete types discussed in Section 5.1.2.2. A procedure is provided, for example, for proving a structural induction theorem for any concrete type. The justification of arbitrary primitive recursive definitions on concrete recursive types was also automated. An example HOL session which shows how these tools are used in practice is given in Appendix B.

5.2 Example: A Transistor Model

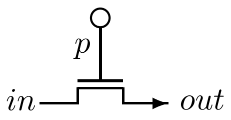
This section shows how a simple instance of the class of concrete types discussed above can be used to formulate a CMOS transistor model which is more realistic than the one used in Chapter 3. An example is also given to show how a proof of correctness involving data abstraction can be done based on a circuit model defined in terms of this concrete type. The section begins with a discussion of the inadequacies of the simple transistor model defined in Chapter 3.

5.2.1 Inadequacies of the Switch Model

In the simple transistor model defined in Chapter 3, transistors are modelled as ideal switches which are controlled by the boolean logic level present on their gates. For example, the formal specification for an N-type transistor in this model describes this device as an ideal switch which is closed when its gate has the value T and open when its gate has the value F. Although this very simple *switch model* of transistor behaviour can be useful for some purposes, it clearly fails to capture many important aspects of the way real CMOS devices behave.

One of these aspects is the fact that the switching behaviour of a real CMOS transistor does not depend simply on the 'logic level' present on its gate, but on the magnitude of the gate-to-source voltage Vgs, compared to some non-zero threshold voltage Vt. This means that a transistor does not behave like an ideal switch which can transmit both logic levels equally well. An N-type transistor, for example, transmits logic *low* well, but transmits logic *high* poorly. In the switch model, however, the specifications for N-type and P-type transistors do not reflect this important aspect of transistor behaviour—transistors are modelled as if they can transmit both logic levels equally well.

This simplification makes it possible to prove, using the switch model, the 'correctness' of certain CMOS circuits which do not work in practice. An example is the simple device shown below, where the value on the input *in* is transmitted through an N-type transistor to drive a capacitative load at the output *out*:



$$\vdash (\exists p.\, \mathsf{Pwr}\; p \wedge \mathsf{Ntran}(p, in, out)) = (out = in)$$

This circuit is simply an N-type transistor with its gate connected directly to power. In the switch model, this circuit is equivalent to a *wire* which connects the output directly to the input. This is stated formally by the correctness theorem shown on the right, which asserts that a formal model of this circuit, constructed using the switch model primitives defined in Chapter 3, is logically equivalent to the specification '$out = in$'. In reality, however, the circuit shown above does not behave like a direct connection between *out* and *in*. If the output drives a capacitative load, and the input is at logic level *high*, then the voltage at *out* will only reach a level which is the threshold voltage Vt less than VDD. This voltage may be too low to drive the gate of another transistor, so it must be treated as distinct from the logic level *high*. The switch model correctness statement shown above is therefore misleading, for it asserts that an N-type transistor with its gate wired to VDD provides a completely transparent connection between *out* and *in*.

5.2.2 A Three-valued Logical Type

The fundamental problem with the switch model is that it specifies the behaviour of transistors using a logical type with only two values. In this very simple model, each wire in a circuit must have either the value *high* (modelled by T) or the value *low* (modelled by F). The physical phenomenon of a 'degraded' logic level—a logic level which is distinct from both these values, and which cannot be used to drive the gate of a transistor—is not even a *possibility* in this model.

To overcome this problem, a type with more than two values is needed. The simplest solution is to use a defined logical type with exactly three distinct values.

Using the informal notation introduced in Section 5.1.2, an appropriate logical type '*tri*' is defined by the equation shown below.

$$tri \ ::= \ \mathsf{Hi} \ | \ \mathsf{Lo} \ | \ \mathsf{X}$$

This informal definition of the type *tri* states that it denotes a set which contains exactly three distinct values—namely Hi, Lo, and X. The corresponding abstract axiomatization for this three-valued type consists of the following single theorem of higher order logic.

$$\vdash \forall a\,b\,c.\ \exists! fn{:}tri{\to}\alpha.\ (fn\ \mathsf{Hi} = a) \wedge (fn\ \mathsf{Lo} = b) \wedge (fn\ \mathsf{X} = c) \tag{5.5}$$

This theorem—which can be proved automatically by the HOL mechanization of type definitions discussed in Section 5.1.3—provides a complete and abstract characterization of the defined logical type *tri*. This characterization takes the form of a degenerate 'primitive recursion' theorem for the concrete type *tri*. Since *tri* is an enumerated type with no recursive constructors, the theorem simply states that any function defined by cases on the three constants Hi, Lo, and X exists and is uniquely defined.

It follows immediately from this theorem that the type constant *tri* denotes a set containing exactly three values: the fact that *fn* always exists implies that the constants Hi, Lo, and X denote distinct values of type *tri*, and the fact that *fn* is uniquely determined by its values for Hi, Lo, and X implies that these constants denote the only values of type *tri*. These two properties of the type *tri* are stated formally by the theorems shown below.

$$\vdash\ \neg(\mathsf{Hi} = \mathsf{X}) \wedge \neg(\mathsf{Lo} = \mathsf{X}) \wedge \neg(\mathsf{Hi} = \mathsf{Lo}) \tag{5.6}$$

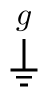
$$\vdash \forall P.\ (P\ \mathsf{Hi} \wedge P\ \mathsf{Lo} \wedge P\ \mathsf{X}) \supset \forall t{:}tri.\ P\ t \tag{5.7}$$

These theorems correspond to two of the 'standard' properties of concrete types discussed in Section 5.1.2.2. The first theorem is the statement for the particular type *tri* of the general property that the constructors of a concrete type yield distinct values. The second theorem is a 'degenerate' example of the structural induction theorem which holds of every concrete type. Both these theorems can be proved automatically by the HOL tools mentioned in Section 5.1.3.

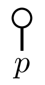
5.2.3 A Threshold Switching Model[3]

Once the type *tri* has been defined—and the properties discussed above have been proved—this three-valued logical type can be used as the basis for a transistor model which at least partly captures the threshold switching behaviour of real

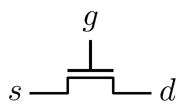
[3]The transistor model defined in this section is based on a suggestion made by M. Fourman at the workshop on *Theoretical Aspects of VLSI Architectures* at the University of Leeds in 1986.



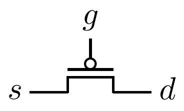
$$\vdash \mathsf{Gnd}\ g = (g = \mathsf{Lo})$$



$$\vdash \mathsf{Pwr}\ p = (p = \mathsf{Hi})$$



$$\vdash \mathsf{Ntran}(g, s, d) = ((g = \mathsf{Hi}) \supset ((d = \mathsf{Lo}) = (s = \mathsf{Lo})))$$



$$\vdash \mathsf{Ptran}(g, s, d) = ((g = \mathsf{Lo}) \supset ((d = \mathsf{Hi}) = (s = \mathsf{Hi})))$$

Figure 5.1: A Threshold Switching Model.

CMOS devices. The basic idea of this model is to represent the strongly-driven logic levels *high* and *low* by the values Hi and Lo, and to represent all degraded logic levels, which cannot reliably drive the gates of transistors, by the value X.

The formal specifications shown in Figure 5.1 constitute a *threshold switching* model of CMOS transistor behaviour based on this representation of logic levels. The specifications Pwr p and Gnd g model VDD and VSS as constant sources of Hi and Lo respectively. The specifications for N-type and P-type transistors are intended to reflect the fact that these devices do not transmit both logic levels equally well. For example, it follows from the specification for an N-type transistor $\mathsf{Ntran}(g, s, d)$ that when the gate g has the value Hi and the source s has the value Lo (i.e. when the gate-to-source voltage is large) then the drain d must also have value Lo. This reflects the fact that the logic level modelled by Lo is transmitted unchanged through an N-type transistor. But when both g and s have the value Hi, then the value of d may be either Hi or X. The specification '$\mathsf{Ntran}(g, s, d)$' is satisfied in both cases. This reflects the fact that the value Hi can be degraded to X when it is transmitted through an N-type transistor. The specification for a P-type transistor is similar. In this case, when g and s are both Lo the value of d can be either Lo or X, reflecting the fact that the logic level modelled by Lo is only imperfectly transmitted through a P-type transistor.

5.2.4 An Example of Data Abstraction

This section shows how the CMOS inverter proof which was done in Chapter 3 using the switch model of transistors can be redone using the more accurate threshold switching model. The aim of this section is to provide a simple example which illustrates the approach to data abstraction introduced in Chapter 4. Here, data abstraction is used to relate a model defined in terms of the three-valued type *tri* to a specification written in terms of the primitive type *bool*.

5.2.4.1 The Specification

One of the simplest possible formal specifications for a CMOS inverter is the one used in the correctness proof given in Chapter 3:

$$\vdash \mathsf{Not}(i,o) = (o = \neg i)$$

Here, the values on the external wires of the device are modelled by booleans, and the variables i and o have logical type *bool*. This simple specification will also be used in the threshold model proof given here.

5.2.4.2 The Model

Given the threshold switching primitives defined above in Section 5.2.3, a formal model of an inverter can be defined as shown below.

$$\vdash \mathsf{Inv}(i{:}tri,\ o{:}tri) = \exists g\,p.\,\mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g \wedge \mathsf{Ntran}(i,g,o) \wedge \mathsf{Ptran}(i,p,o)$$

The structure of this definition is identical to the one given in Chapter 2, where the switch model primitives are used. In the model defined here, however, the terms $\mathsf{Pwr}\ p$, $\mathsf{Gnd}\ g$, $\mathsf{Ntran}(i,g,o)$ and $\mathsf{Ptran}(i,p,o)$ are instances of the threshold switching primitives shown in Figure 5.1, rather than the similarly-named switch model primitives defined in Chapter 3.

5.2.4.3 Defining the Data Abstraction Function

To formulate a correctness statement which relates the model $\mathsf{Inv}(i,o)$ to the more abstract specification $\mathsf{Not}(i,o)$, a data abstraction function is needed to relate values of type *tri* in the model to values of type *bool* in the specification. In the model, the two strongly-driven logic levels *high* and *low* are represented by the two values Hi and Lo. In the specification, these two logic levels correspond to the two boolean values T and F. The required data abstraction function must therefore map the value Hi to T and the value Lo to F.

Given theorem (5.5), it is trivial to define a constant 'abs' which denotes the required function. Taking an instance of this theorem in which the type *bool* is substituted for the type variable α, and specializing the universally-quantified variables a and b to T and F respectively, yields the following theorem.

$$\vdash \forall c.\,\exists!\mathit{fn}.\,(\mathit{fn}\ \mathsf{Hi} = \mathsf{T}) \wedge (\mathit{fn}\ \mathsf{Lo} = \mathsf{F}) \wedge (\mathit{fn}\ \mathsf{X} = c)$$

This theorem asserts the unique existence of a function which has *at least* the properties which are needed for an appropriate data abstraction function from *tri* to *bool*. From this, one can immediately infer the existence of a function which has *precisely* the desired properties:

$$\vdash \exists fn.\,(fn\ \mathsf{Hi} = \mathsf{T}) \wedge (fn\ \mathsf{Lo} = \mathsf{F})$$

Using the technique for defining constants explained in Section 2.1.5, a constant 'abs' can be introduced to denote the function whose existence is guaranteed by the theorem shown above. The result is the following theorem about abs:

$$\vdash (\mathsf{abs}\ \mathsf{Hi} = \mathsf{T}) \wedge (\mathsf{abs}\ \mathsf{Lo} = \mathsf{F})$$

This theorem states that the function abs maps the value Hi to T and the value Lo to F, as required. A fully *formal* proof of this theorem (the details of which are not relevant here) can be done completely automatically in the HOL system using the mechanized theorem-proving tools mentioned in Section 5.1.3.

5.2.4.4 A Note about the Definition of abs

There is one important formal consequence of the definition of abs given above which may not be immediately obvious. In higher order logic, all functions of type *tri→bool* are *total* functions, and the data abstraction function abs:*tri→bool* therefore yields *some* boolean value when applied to X. But with the definition given above, the constant abs is defined so that it is impossible to prove *which* boolean value is denoted by the application 'abs X'. That is, neither 'abs X = T' nor 'abs X = F' is a formal theorem of the logic. One of these two equations must be 'true', but neither of them can be *proved* to be true.

This is a consequence of the way in which the constant abs is defined using the primitive constant ε. The underlying definition for abs—which was not shown the derivation given above—is in fact the following equation.

$$\vdash \mathsf{abs} = \varepsilon\ (\lambda fn.\,(fn\ \mathsf{Hi} = \mathsf{T}) \wedge (fn\ \mathsf{Lo} = \mathsf{F}))$$

All that can be proved from this definition is that the constant function abs satisfies the predicate '$\lambda fn.\,(fn\ \mathsf{Hi} = \mathsf{T}) \wedge (fn\ \mathsf{Lo} = \mathsf{F})$'. That there *is* a function which satisfies this predicate was proved in the previous section. It follows from this, and from the axiom for ε discussed in Section 2.1.5, that abs in fact has the property expressed by this predicate. That is, it follows that:

$$\vdash (\mathsf{abs}\ \mathsf{Hi} = \mathsf{T}) \wedge (\mathsf{abs}\ \mathsf{Lo} = \mathsf{F}) \tag{5.8}$$

But this is the only significant fact that can be proved about abs, since the single axiom for ε shown in Section 2.1.5 is the only property of ε made available by the primitive basis of the logic. (See Section 2.1.5 for further discussion of this point.)

The reason that the function abs is defined in this way is to make it impossible to prove certain misleading correctness statements, which if abs were otherwise defined would be provable, but which in fact give a false assurance of correctness. Suppose, for example, that instead of being defined as shown above, the constant abs is defined such that the formal theorem $\vdash \mathsf{abs}\,\mathsf{X} = \mathsf{T}$ can be derived from its definition, in addition to theorem (5.8). If abs is defined in this way, then it is also possible to prove the theorem shown below.

$$\vdash (\exists p.\, \mathsf{Pwr}\, p \wedge \mathsf{Ntran}(p, in, out)) \supset (\mathsf{abs}\, out = \mathsf{abs}\, in)$$

But this theorem is a 'correctness' statement for the simple CMOS device shown above on page 83. It asserts that an N-type transistor with its gate connected directly to power can be viewed, at a higher level of data abstraction, as a direct connection between *in* and *out*. This formal theorem—which is exactly the sort of correctness statement that the threshold model is designed to eliminate—can be proved *only* if $\vdash \mathsf{abs}\,\mathsf{X} = \mathsf{T}$ can be derived from the definition of abs. So when abs is defined formally as shown in Section 5.2.4.3, this fallacious correctness statement is *not* in fact a theorem of higher order logic and cannot be proved.

5.2.4.5 The Proof of Correctness

Once the function abs is defined, it is straightforward to formulate a correctness statement for the inverter which uses data abstraction to relate the model $\mathsf{Inv}(i, o)$ to the more abstract specification $\mathsf{Not}(i, o)$.

This correctness statement must be qualified by a *validity condition* which restricts the range of values on the input of the inverter to the strongly-driven logic levels represented by Hi and Lo. This condition is necessary because the specification $\mathsf{Not}(i, o)$ represents a valid abstract view of an inverter only when the device is used in an environment in which the input i is always strongly-driven.

A correctness statement which includes the required validity condition can be formulated in logic as shown below.

$$\vdash \neg(i = \mathsf{X}) \supset \mathsf{Inv}(i, o) \supset \mathsf{Not}(\mathsf{abs}\, i,\, \mathsf{abs}\, o)$$

This correctness theorem states that if the circuit modelled by $\mathsf{Inv}(i, o)$ is used in an environment in which its input is always strongly driven, then it will behave as required by the specification $\mathsf{Not}(i, o)$. The function abs is used in this theorem to translate values of type *tri* in the model to the corresponding values of type *bool* in the more abstract specification. The term '$\neg(i = \mathsf{X})$' is a validity condition on the abstraction relationship between the model and the specification. This condition limits the correctness statement to the assertion that the inverter circuit will behave as required *only if* the value on its input is not 'X'.

The proof of the correctness theorem shown above is simple. The 'induction' theorem for tri shown on page 84 allows the proof to be done by case analysis on the value of input i. When $i = \mathsf{X}$, the validity condition is false, and the implication is therefore vacuously true. When $i = \mathsf{Hi}$ or $i = \mathsf{Lo}$, it follows from theorem (5.6) that the validity condition '$\neg(i = \mathsf{X})$' is true. The proof therefore reduces to showing that the implication

$$\mathsf{Inv}(i, o) \supset \mathsf{Not}(\mathsf{abs}\ i,\ \mathsf{abs}\ o) \tag{5.9}$$

holds for $i = \mathsf{Hi}$ and $i = \mathsf{Lo}$. By a simple derivation which is similar to the first few steps in the switch model proof shown on page 41, it follows that:

$$\vdash \mathsf{Inv}(i, o) \ = \ ((i = \mathsf{Hi}) \supset (o = \mathsf{Lo})) \wedge ((i = \mathsf{Lo}) \supset (o = \mathsf{Hi}))$$

So proving that (5.9) holds for $i = \mathsf{Hi}$ and $i = \mathsf{Lo}$ reduces to proving:

$$\vdash \mathsf{Not}(\mathsf{abs\ Hi},\ \mathsf{abs\ Lo}) \qquad \text{and} \qquad \vdash \mathsf{Not}(\mathsf{abs\ Lo},\ \mathsf{abs\ Hi}).$$

These two theorems follow immediately from the definition of Not and the defining equations for abs given by the theorem $\vdash (\mathsf{abs\ Hi} = \mathsf{T}) \wedge (\mathsf{abs\ Lo} = \mathsf{F})$.

5.2.5 Summary and Discussion

This section has shown how a simple instance of the general class of concrete types introduced earlier in this chapter can be used to formulate a transistor model which at least partly captures the threshold switching behaviour of real CMOS devices. By modelling the range of values which can appear on the wires of a circuit using the special-purpose concrete type tri, instead of the primitive type *bool*, this threshold switching model reflects the actual behaviour of CMOS transistors more accurately than the simpler switch model of transistors defined in Chapter 3. Design errors are therefore less likely to escape discovery if correctness proofs are based on the threshold switching model instead of the switch model, since a circuit which can be proved correct using the switch model may in fact be incorrect according to the more accurate threshold switching model. The relationship between these two transistor models is examined in more detail in Chapter 7.

In addition to providing a straightforward example of data abstraction, the inverter correctness proof given in this section also illustrates two general aspects of the approach taken here to formalizing data types and data abstraction in higher order logic. These are discussed in the two sections that follow.

5.2.5.1 Data Abstraction without Partial Functions

The first general point illustrated by the inverter example is that in using data abstraction to formulate correctness it is possible to avoid dealing explicitly with data abstraction functions which are also, properly speaking, *partial* functions.

Why such functions might seem to be appropriate, or even necessary, can be explained by considering how 'data' is represented in the example given above. Two logical types are used in this example to represent data: the three-valued type *tri* and the two-valued type *bool*. In addition to the values Hi and Lo, which correspond to the two boolean values T and F, the type *tri* also has a third value: 'X'. In circuit models based on the threshold switching primitives, this value represents the physical phenomenon of a 'degenerate' logic level. This kind of logic level, however, is *not represented* in a specification of required behaviour based on the two-valued type *bool*. It is therefore undesirable (as was noted above in Section 5.2.4.4) for a data abstraction function from *tri* to *bool* to assign a *particular* boolean value to X, since the physical phenomenon modelled by X is not even represented in specifications based on the primitive type *bool*. It might seem natural, then, to make the data abstraction function from *tri* to *bool* a partial function—a function which is in fact *undefined* when applied to X.

The method used in this example to define the function abs, however, avoids the need to develop a full theory of partial functions in higher order logic. Instead, abs is a total function, but is defined (using ε) so that nothing definite can be proved about the boolean value denoted by 'abs X'. This avoids the possibility of proving misleading correctness theorems (like the one shown on page 88) which might otherwise be provable simply because the boolean value given by 'abs X' just happens to satisfy the specification of required behaviour which is involved.

Although the data abstraction function abs is a total function, the validity condition used in the correctness statement for the inverter effectively restricts its domain to only those values of type *tri* for which a partial function would in fact be defined. The net effect is to 'simulate' a partial data abstraction function by the combination of a total function (abs) and a predicate which restricts the range of this total function (the validity condition). This is a natural (and obvious) general technique for representing partial functions by total functions. The same method can be used whenever the representation of data in a model is richer than the representation of data in a specification, and it is therefore necessary to restrict the range of a data abstraction function to only a subset of the values that can arise in the model.

5.2.5.2 Defining Data Abstraction Functions

The method used in this example to define the data abstraction function abs illustrates one of the pragmatic advantages of the approach taken here to the characterization of concrete types in logic. Every concrete type is characterized by a theorem which asserts the existence of a wide class of functions *from* the type itself *to* any other type. This provides a direct means for defining data abstraction functions from the concrete types used in models to the higher-level types used in specifications. In the example given above, the model was based on the three-valued concrete type *tri*. This type is characterized by a theorem which asserts the existence of an arbitrary function defined by cases on Hi, Lo, and X. It was therefore trivial to define a data abstraction function by cases on Hi and Lo.

This particular example is, of course, very simple. But when more complex *recursive* types are used in models of hardware behaviour, the way in which these types are characterized in logic provides a direct and powerful means for defining data abstraction functions on the values of these types by *primitive recursion*. One application in which such recursive data abstraction functions arise naturally is in reasoning about hardware devices that operate on vectors of bits—i.e. finite sequences of binary digits. This application is considered in the next section.

5.3 Example: Bit Vectors

In reasoning about the correctness of devices that operate on vectors of bits, a commonly used technique is to prove a single correctness theorem for the entire *class* of all n-bit wide implementations of such a device [11,34,53,78]. An example to which this technique has often been applied is the class of all n-bit ripple-carry binary adders. The recursive structure of this class of devices (an $n{+}1$ bit adder is just a 1-bit adder connected to an n-bit adder) makes it straightforward to formulate and prove a single correctness theorem which states the correctness of every n-bit adder. The correctness of a 16-bit adder, or an adder of any other particular size, can then be inferred directly from this more general correctness result for the class of all n-bit adders.

An important advantage of this approach is that it makes it unnecessary to prove a correctness theorem from first principles for any particular instance of an entire class of related devices. Instead, a single theorem is proved which states the correctness of *every* device in the class, and the correctness of any particular device can then be inferred from this more general result. A further advantage is that this approach often makes proofs simpler than would otherwise be possible. For example, a *direct* proof of correctness for a 16-bit adder is likely to involve the manipulation of large algebraic expressions which define the relationships that hold among the individual binary digits in the design.[4] But a correctness theorem for the class of all n-bit adders (which directly implies a correctness theorem for the 16-bit adder) is much simpler to prove, since it can be proved by *induction*.

[4]In fact, this is how the correctness of a 16-bit adder would be proved automatically by Barrow's VERIFY system [3], which does not support proofs of correctness for a class of n-bit devices.

To support this approach to reasoning about a class of n-bit devices in higher order logic, a logical *type* is needed to model the set of all bit vectors, or n-bit binary words. Gordon [34] describes a representation for n-bit words based on the logical type *num→bool*, the type of total functions from the natural numbers to the booleans. Adopting this representation of n-bit words has the advantage of making it unnecessary to define a special-purpose type to model bit vectors in logic,[5] but there is also a problem with this representation that can lead to unnecessary complexity in specifying and reasoning about the behaviour of a class of n-bit devices. This problem, and an alternative representation of bit vectors based on the concrete type of *lists* introduced earlier in this chapter, are discussed in the sections that follow.

In addition to a representation for n-bit words, two other things are needed in order to support formal reasoning about a class of n-bit devices in higher order logic. The first is a method for constructing a model for an entire class of n-bit circuit designs. The second is a method for defining data abstraction functions on n-bit words. These two requirements are also discussed, in the context of the representation for bit vectors proposed here, in the sections that follow.

5.3.1 Representing Bit Vectors by *num→bool*

The type *num→bool* was proposed as a representation for bit vectors (n-bit words) in higher order logic by Gordon in [34], and has been used by Camilleri in [10,11] and adapted for vectors of other values by Joyce in [55] . The idea of this representation is that each bit vector is modelled by a function of type *num→bool* which maps bit positions (represented by natural numbers) to bits (represented by booleans).

The n bit-positions in a bit vector of length n are numbered from 0 to $n-1$. When a bit vector is interpreted as n-bit binary number, the least significant bit occurs at position 0, and the most significant bit occurs at position $n-1$. Following this convention, a bit vector of length n is represented by a function f:*num→bool* that maps each bit position (i.e. each number from 0 to $n-1$) to a corresponding boolean value. For example, the sequence of booleans 'TFTT', which represents to the 4-bit binary number '1101', would be modelled by a function f, where $f(0)=\mathsf{T}$, $f(1)=\mathsf{F}$, $f(2)=\mathsf{T}$, and $f(3)=\mathsf{T}$.

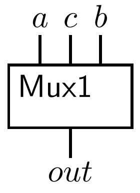
5.3.1.1 Defining Models based on *num→bool*

Associated with the representation of bit vectors described above is a method[6] for defining a 'parameterized' model to represent a class of n-bit designs. This method

[5]Of course, the type *num* is not a *primitive* type of the logic, but it is so fundamental that it can hardly be considered 'special-purpose'.

[6]again, originally due to Gordon [34], and subsequently adopted by others [10,11,55].

is based on primitive recursion on the natural numbers, and is best explained by considering an example. A one-bit multiplexer, which selects one of two inputs, a or b, depending on the value of a control input c, can be modelled in logic by the term '$\mathsf{Mux1}(c, a, b, out)$', defined as shown below.



$$\vdash \mathsf{Mux1}(c, a, b, out) = (out = (c \Rightarrow a \mid b))$$

Each of the free variables a, b, c, and out in this definition has type *bool*, and corresponds to a single external wire of the device. The device modelled by the term defined above is a 'one-bit' multiplexer, a device whose inputs and output are each one bit wide.

An (n+1)-bit multiplexer can be built by placing n+1 instances of this one-bit multiplexer in parallel, as shown by the diagram in Figure 5.2. To describe the behaviour of the entire class of n+1 bit wide devices represented by this diagram, it is necessary to define a model in which composition ('$\wedge$') is applied to n+1 instances of the specification for a one-bit multiplexer shown above. This is done by using primitive recursion on the natural numbers to define a model which is paramaterized by the size n of the multiplexer design it represents. The primitive recursive defining equations for this model are as follows.

$$\begin{array}{lcl} \vdash \mathsf{Mux}\ 0\ (c, a, b, out) & = & \mathsf{Mux1}(c,\ (a\,0),\ (b\,0), (out\,0)) \\ \vdash \mathsf{Mux}\ (n{+}1)\ (c, a, b, out) & = & \mathsf{Mux1}(c,\ a(n{+}1),\ b(n{+}1),\ out(n{+}1)) \wedge \\ & & \mathsf{Mux}\ n\ (c, a, b, out) \end{array}$$

These equations define a term '$\mathsf{Mux}\ n\ (c, a, b, out)$' to model the behaviour of the (n+1)-bit multiplexer structure shown in Figure 5.2. The variable n in this term stands for the size of the multiplexer whose behaviour it models. When n is equal to 0, the device modelled by this term consists of a single one-bit multiplexer. When n is greater than 0, the device consists of a one-bit multiplexer connected in parallel to an (n−1)-bit multiplexer.

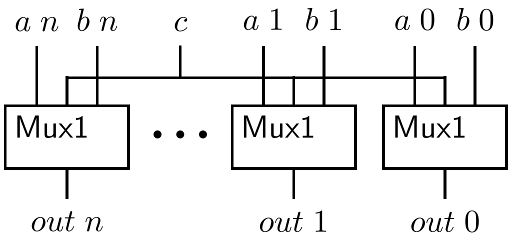


Figure 5.2: An n+1 Bit Multiplexer.

The variables a, b, and out in this definition range over functions of type $num{\rightarrow}bool$, and represent the $(n{+}1)$-bit words which appear on the inputs and the output of an $(n{+}1)$-bit multiplexer. The recursive definition shown above simply applies the composition operation '$\wedge$' to $n{+}1$ instances of the specification for a one-bit multiplexer. Each of these iterated specifications constrains one of the $n{+}1$ bit-positions numbered from 0 to n in the three bit vectors represented by the functions a, b and out. When $n{=}3$, for example, the model defined recursively by the equations shown above imposes the following constraint on the values of the functions c, a, b, and out.

$$\vdash \mathsf{Mux}\ 3\ (c, a, b, out) = \forall n.\,(n \leq 3) \supset \mathsf{Mux1}(c, (a\ n), (b\ n), (out\ n))$$

This models the effect of placing four instances of a one-bit multiplexer in parallel, to obtain a multiplexer that operates on 4-bit words. These 4-bit words are represented by functions of type $num{\rightarrow}bool$, as discussed above.

5.3.1.2 The Problem

The problem with using $num{\rightarrow}bool$ to represent bit vectors can be illustrated by considering the parameterized model $\mathsf{Mux}\ n\ (c, a, b, out)$ defined above. This model does not determine a *unique* representation for the combinations of bit vectors which can appear on the external wires of the class of devices which it is intended to describe. This means that certain assertions which one might want to prove about this class of devices cannot be stated in the most direct and natural way—as a simple constraint on the variables a, b, c, and out.

Suppose, for example, that $n{=}3$, and the control wire c has the value T. One might want to prove a theorem which states that the 4-bit word on the input a is, in this case, selected by the multiplexer and appears on the output out:

$$\mathsf{Mux}\ 3\ (\mathsf{T}, a, b, out) \supset (out = a) \tag{5.10}$$

This implication, however, is not true. The term '$\mathsf{Mux}\ 3\ (\mathsf{T}, a, b, out)$' implies only that the values $out\ n$ and $a\ n$ are equal for $0 \leq n \leq 3$. It does not imply that these two values are equal for all n, and therefore that the two functions out and a are themselves equal. Similarly, the parameterized model $\mathsf{Mux}\ n\ (c, a, b, out)$ does not satisfy the following natural formulation of correctness for the class of all multiplexer designs:

$$\forall n.\,\mathsf{Mux}\ n\ (c, a, b, out)\ \supset\ (out = (c \Rightarrow a \mid b))$$

This correctness statement cannot be proved, since there are functions a, b, and out which satisfy the term on the left hand side of the implication, but do not satisfy the term on the right hand side.

The source of the problem is that each bit vector is not *uniquely* represented by a single function of type $num \to bool$. In the implication (5.10) shown above, for example, the functions a and out are intended to model 4-bit words, and the term $\mathsf{Mux}\ 3\ (\mathsf{T}, a, b, out)$ is expected to imply that these 4-bit words are equal. But there are an infinite number of different representations in $num \to bool$ for any particular 4-bit word, and the constraint imposed by this term on the functions a and out does not ensure that these functions are equal *representations* for the same 4-bit word.

There are several *ad hoc* solutions to this problem. One of these is to include the size parameter n in any assertions which are to be proved about the behaviour of the class of devices modelled by $\mathsf{Mux}\ n\ (c, a, b, out)$. For example, one could prove (by induction on n) the following correctness theorem for the class of all multiplexer designs.

$$\vdash \forall n.\, \mathsf{Mux}\, n\ (c, a, b, out) \ \ = \ \ \forall i.\, i \leq n \supset (out\ i = (c \Rightarrow a\ i \,|\, b\ i))$$

But this means having to retain the size parameter n in future reasoning based on this correctness statement—for example, in a hierarchical correctness proof in which the specification in this correctness theorem is used as a model at a higher level in the hierarchy. Another solution is to strengthen the constraint imposed by the model on the functions a, b, and out in such a way as to ensure that each bit vector of fixed length n is represented by a unique function. But a better solution is to model bit vectors by a more appropriate logical type, in which each n-bit word already has a unique representation.

5.3.2 A Better Representation

The concrete recursive type of *lists* discussed earlier in this chapter provides a representation for bit vectors in which each finite sequence of bits is modelled by exactly one value. This type was defined informally by the equation:

$$list \ \ ::= \ \ \mathsf{Nil} \ \ | \ \ \mathsf{Cons}\ \alpha\ list$$

A substitution instance of the polymorphic type defined by this equation is the type $(bool)list$, the type of finite-length lists of booleans. Each bit vector can be represented by a value of this type constructed using the two constants Nil and Cons. The 4-bit word corresponding to the sequence of boolean values 'TTFT', for example, can be represented by the term shown below.

$$\mathsf{Cons\ T\ (Cons\ T\ (Cons\ F\ (Cons\ T\ Nil)))}$$

Any finite sequence of bits can be represented by a value of type $(bool)list$ in a similar way. The empty bit vector can be represented by the empty list Nil, and a non-empty vector of n boolean values $b_1, \ldots, b_n$ can be represented by the list '$\mathsf{Cons}\ b_1\ (\mathsf{Cons}\ b_2\ \ldots\ (\mathsf{Cons}\ b_n\ \mathsf{Nil})\ldots))$'.

The advantage of using lists to model bit vectors is that it provides a *unique* representation in logic for each possible finite sequence of bits. Each bit vector corresponds to precisely one list, and every list corresponds to a vector of bits. This follows from the properties of the defined type $(bool)list$ stated formally by the three theorems about lists discussed Section 5.1.1.2:

$$\vdash \forall h\,t.\,\neg(\mathsf{Nil} = \mathsf{Cons}\ h\ t)$$
$$\vdash \forall h_1\,h_2\,t_1\,t_2.\ (\mathsf{Cons}\ h_1\ t_1 = \mathsf{Cons}\ h_2\ t_2) \supset ((h_1 = h_2) \wedge (t_1 = t_2))$$
$$\vdash \forall P.\,(P\ \mathsf{Nil} \wedge \forall t.\,P\ t \supset \forall h.\,P(\mathsf{Cons}\ h\ t)) \supset \forall l.\,P\ l$$

The first two theorems shown above imply that two lists constructed using Nil and Cons are equal if and only if they represent exactly the same sequence of values. In other words, two lists are equal if and only if they model the same bit vector. The third theorem (induction) implies that every value of type $(bool)list$ is either equal to Nil or is constructed from Nil by finitely many applications of the function Cons. This means that the set denoted by $(bool)list$ contains only values that represent vectors of bits: there are no 'extra' values. Thus, the concrete recursive type $(bool)list$ has precisely the logical properties needed to provide an unambiguous formal representation of bit vectors in higher order logic.

5.3.2.1 Notational Abbreviations for Lists

In the next section, it is shown how a model for a class of n-bit circuit designs can be based on the representation of bit-vectors introduced above. The metalinguistic abbreviations shown in the following table will be used in this section to make list expressions more readable.

Abbreviations for Lists

Term	*Abbreviation*
Nil	$[\,]$
$\mathsf{Cons}\ b\ t$	$[h \mid t]$
$\mathsf{Cons}\ b_1\ (\mathsf{Cons}\ b_2\ \ldots\ (\mathsf{Cons}\ b_n\ \mathsf{Nil})\ldots))$	$[b_1;\ b_2;\ \ldots\ ; b_n]$

These abbreviations introduce a notation for lists in higher order logic which is similar to the syntax for lists in Prolog [15]. The notation introduced by the third abbreviation in the table shown above, however, does not include 'nested' list expressions like '$[x; [b; c]]$', which are allowed in Prolog but cannot be represented in higher order logic by well-typed terms constructed using Nil and Cons.

5.3.2.2 Defining Models based on $(bool)list$

When bit vectors are represented by lists, an explicitly-stated size parameter 'n' is not needed to define a model of a class of n-bit circuit designs. Models can instead be defined by primitive recursion on the *lists* which represent bit vectors in a circuit design. The size of the design represented by such a model is then implicitly determined by the values of the variables which represent the n-bit inputs and outputs of the design itself.

Consider, for example, the multiplexer design shown in the diagram on page 93. The class of n-bit circuit designs represented by this diagram can be modelled in logic by the term '$\mathsf{Mux}(c, a, b, out)$' defined by primitive recursion as follows:

$$\begin{aligned}
&\vdash \mathsf{Mux}(c, a, b, [\,]) &=&\ (a = [\,]) \wedge (b = [\,])\\
&\vdash \mathsf{Mux}(c, a, b, [h_o \,|\, t_o]) &=&\ \exists h_a\, h_b\, t_a\, t_b.\ (a = [h_a \,|\, t_a]) \wedge (b = [h_b \,|\, t_b]) \wedge\\
& & &\qquad\qquad\quad \mathsf{Mux1}(c, h_a, h_b, h_o) \wedge \mathsf{Mux}(c, t_a, t_b, t_o)
\end{aligned}$$

These two equations define a model $\mathsf{Mux}(c, a, b, out)$ in which the variables a, b, and out represent bit vectors on the inputs and output of a multiplexer by values of type $(bool)list$. The definition is done by primitive recursion on the output list out, and simply applies the constraint imposed by the specification of a one-bit multiplexer to the appropriate triples of bits taken from the three lists a, b, and out. As was discussed in Section 5.1.1.3, the validity of this primitive recursive definition follows directly from the abstract characterization of $(\alpha)list$ given by theorem (5.2).

The defining equations shown above are slightly complicated by the fact that it is necessary to ensure that the resulting model $\mathsf{Mux}(c, a, b, out)$ is satisfied only if the three lists a, b, and out represent bit vectors of the same length. A less cluttered picture of the essence of this recursive definition is given by the two theorems shown below.

$$\begin{aligned}
&\vdash \mathsf{Mux}(c, [\,], [\,], [\,]) = \mathsf{T}\\
&\vdash \mathsf{Mux}(c, [h_a \,|\, t_a], [h_b \,|\, t_b], [h_o \,|\, t_o]) = \mathsf{Mux1}(c, h_a, h_b, h_o)\ \wedge\ \mathsf{Mux}(c, t_a, t_b, t_o)
\end{aligned}$$

These two equations, which follow immediately from the more complex primitive recursive definition given above, define the model $\mathsf{Mux}(c, a, b, out)$ for lists a, b, and out of equal length. The first equation defines a '0-bit' multiplexer. This does not correspond, of course, to any real physical device; it merely provides a 'zero' (namely T) for the binary operation on models of composition ('$\wedge$'). The second equation is recursive: it defines an $(n{+}1)$-bit multiplexer to be a one-bit multiplexer connected in parallel an with n-bit multiplexer.

The correctness of the class of multiplexers modelled by $\mathsf{Mux}(c, a, b, out)$ can be stated with respect to a very simple and clear specification of required behaviour. An explicit size parameter n is not needed in the specification, and correctness can be stated simply by:

$$\vdash \mathsf{Mux}(c, a, b, out) = (out = (c \Rightarrow a \mid b))$$

This theorem is straightforward to prove by structural induction on the list out, using the induction theorem for the defined concrete type $(\alpha)list$.

5.3.2.3 Discussion

The main advantage of using lists to represent bit vectors is that it provides a direct and unambiguous representation for each finite sequence of bits. When bit vectors are represented by functions, there are an infinite number of possible representations for each fixed-length bit vector. And this means that either the lengths of bit vectors must be mentioned explicitly, by including a size parameter 'n' in both models and specifications, or a standard representation must somehow be chosen for each bit vector, perhaps by strengthening the constraint imposed by a model. But when bit vectors are modelled by lists, it is not necessary to impose extra constraints to obtain a unique representation.

The concrete type of lists provides not only a unique representation for each bit vector, but also a *structured* one. This allows a model for a regular class of n-bit circuit designs to be defined recursively on the structure of the lists which represent data in the design itself, rather than on an explicitly stated size parameter n. Such a model is *implicitly* parameterized by size, in the sense that a model for a device of any fixed size n can be obtained from it by taking its free variables to be particular lists of length n.

This implicit form of parameterization helps to keep specifications of required behaviour clear and simple, as was illustrated by the example given above. It also, however, tends to make the definitions of models more complex than the definitions of explicitly-parameterized models based on $num{\rightarrow}bool$ (this is also illustrated by the example given above). Furthermore, it is only possible when there is a straightforward correspondence between structure of a list (or lists) in the model and the structure (usually linear) of the class of circuit designs which the model represents.

This is *not* the case in the example considered in the next section, where a different technique is developed in order to model (and prove correct) a class of *tree-shaped* circuit designs with n-bit inputs. In this example, a design cannot be 'indexed' by the size of the bit vector on its input, since the class of devices to be modelled includes many alternative designs for each possible size of input vector.

5.3.3 An Example

Many devices can be implemented in hardware by tree-shaped structures. The example given in this section shows how concrete recursive types can be used to support formal reasoning about devices of this kind in higher order logic. The basic idea is to construct a model for a class of tree-structured circuit designs using a recursive type whose values have the same sort of structure as the devices themselves. In the example given here, a recursive type of *binary trees* is used to define a model for a simple class of n-bit test-for-zero devices constructed using a tree of 2-input OR-gates.

This example also shows how data abstraction can be used to formulate the correctness of a class of n-bit circuit designs. The inputs to the devices considered in this section are n-bit binary words, and these are represented in the model using the recursive type $(bool)list$ discussed above. A recursively-defined data abstraction function is used to relate this model to a more abstract specification of required behaviour in which the value on the input of the device is represented by a natural number of type *num*.

5.3.3.1 The Class of Devices

An n-bit test-for-zero device can be implemented in hardware by a tree of 2-input OR-gates connected to an inverter. Figure 5.3 shows two correct implementations of this kind for a 6-bit test-for-zero device. Each of these devices takes a 6-bit binary word on its input wires. The output of each device is a boolean value which is *true* if the binary number represented by its 6-bit input is zero, and is *false* otherwise.

Both of the circuits shown in Figure 5.3 are functionally correct. The circuit on the right (circuit b), however, is an *optimal* tree-shaped implementation for a

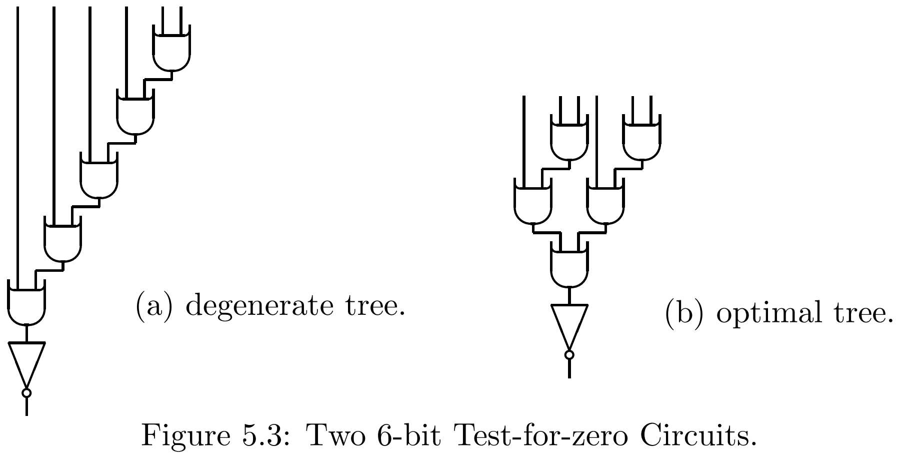


Figure 5.3: Two 6-bit Test-for-zero Circuits.

6-bit device, in the sense that the length of the path from inputs to output is the shortest possible, and the gate delay through the device is therefore minimal. On the other hand, the structure of circuit (a) is that of a 'degenerate' binary tree: it's structure is essentially that of a linear *list*. If the criterion by which circuits are judged is that of minimising delay, this structure is the *worst* implementation of a 6-bit test-for-zero device.

In general, the best implementation of an n-bit test-for-zero device using a binary tree of 2-input OR-gates will be a tree of height $\lceil \log_2 n \rceil$, and the worst implementation will be a degenerate tree of height $n-1$. The class of all n-bit devices of the *latter* kind (i.e. the set containing only the degenerate tree-shaped circuits) is straightforward to model in logic using the techniques explained in the previous sections; a model for this class of devices can be constructed by primitive recursion on the list which represents the n-bit input of the device. But a model for the class of *all* n-bit test-for-zero devices cannot be constructed in this way, since there are many possible OR-gate trees for each input vector size. To define a model for the class of all n-bit tree-shaped devices, another technique is needed.

5.3.3.2 A Type of Binary Trees

The technique proposed here is to use a concrete recursive type of *binary trees* to represent the structure of tree shaped circuits. Using the notation introduced in Section 5.1.2, this type can be described informally by the following equation.

$$btree \quad ::= \quad \mathsf{Leaf} \quad | \quad \mathsf{Node}\ btree\ btree$$

The type *btree* defined by this equation has two constructors: $\mathsf{Leaf}{:}btree$, and $\mathsf{Node}{:}btree{\to}btree{\to}btree$. The constant Leaf denotes the trivial tree consisting of a single leaf node. The function Node is used to build binary trees from smaller binary trees; if t_1 and t_2 are trees, then the term '($\mathsf{Node}\ t_1\ t_2$)' denotes the binary tree with left subtree t_1 and right subtree t_2.

The abstract axiomatization for the type informally defined by the equation shown above is the following theorem.

$$\vdash \forall e\, f.\, \exists! fn.\, \forall n.\, (fn\ \mathsf{Leaf} = e) \wedge (\forall t_1\, t_2.\, fn\ (\mathsf{Node}\ t_1\ t_2) = f\ (fn\ t_1)\ (fn\ t_2)\ t_1\ t_2)$$

This theorem states the validity of primitive recursive definitions on binary trees, and can be proved from an automatically-constructed formal type definition for the type constant *btree*, using the method explained in Appendix A. The usual 'standard properties' of a concrete type (see Section 5.1.2.2) follow from this abstract characterization of binary trees (e.g. a structural induction theorem). As was discussed in Section 5.1.2.3, a characterization of binary trees stated in this form can also be used to justify the introduction of function constants defined by primitive recursion on trees. These facts are used in the sections that follow.

5.3.3.3 Defining the Model

Given the recursive type *btree* defined above, it is straightforward to define a term '$\mathsf{Tfztree}\ t\ (in, out)$' which models the class of all tree-shaped implementations of an n-bit test-for-zero device. The variables *in* and *out* in this model represent the n-bit input and boolean output of the device. The variable t ranges over values of type *btree*, and represents the shape of the tree of OR-gates in the device. The possible values of this variable have the same kind of structure as the trees of 2-input OR-gates in the class of circuit designs which are represented by this model, and for each value t of type *btree* the model $\mathsf{Tfztree}\ t\ (in, out)$ describes the tree-shaped test-for-zero device whose internal structure is the same as the binary tree denoted by t. Since the variable t ranges over the set of all binary trees, this parameterized model describes the class of all such circuits.

The first step in the formal definition of this model, is to define an infix *append* function '$+\!\!+$' on lists. This function can be defined in logic by primitive recursion on lists as follows:

$$\begin{array}{lcl}\vdash [\,] +\!\!+ l & = & l \\ \vdash [h \mid t] +\!\!+ l & = & [h \mid (t +\!\!+ l)]\end{array}$$

These two equations are just the usual recursive definition of the concatenation of two lists. They can be proved formally from the abstract characterization of $(\alpha)list$ given by theorem (5.2).

Given this infix function on lists, a model '$\mathsf{Ortree}\ t\ (in, o)$' can be defined to represent the tree of OR-gates in an n-bit test-for-zero device. The variable t in this model has type *btree*, and its value determines the shape of the tree of OR-gates which the model represents. Each internal node in the binary tree t corresponds to a 2-input OR-gate in the circuit being modelled. Each leaf node in the tree t corresponds to one of the input wires which make up the bit-vector input *in*. The model is defined by primitive recursion on the tree t, as follows:

$$\begin{array}{lcl}\vdash \mathsf{Ortree}\ \mathsf{Leaf}\ (in, o) & = & (in = [o]) \\ \vdash \mathsf{Ortree}\ (\mathsf{Node}\ t_1\ t_2)\ (in, o) & = & \exists i_1\, i_2\, o_1\, o_2.\ (in{=}i_1 +\!\!+ i_2) \wedge \mathsf{Or}(o_1, o_2, o) \wedge \\ & & \quad \mathsf{Ortree}\ t_1\ (i_1, o_1) \wedge \mathsf{Ortree}\ t_2\ (i_2, o_2)\end{array}$$

When the tree t is a leaf node, model $\mathsf{Ortree}\ t\ (in, o)$ simply represents a wire which connects the input *in* (which must be a bit-vector of length one) directly to the output o. In the recursive case, when t is an internal node with two subtrees t_1 and t_2, the bit vector modelled by the list *in* is split into two sublists i_1 and i_2 by the expression '$in = i_1 +\!\!+ i_2$'. These become the inputs to the two OR-gate trees constructed recursively from the subtrees t_1 and t_2. The outputs o_1 and o_2 of these two recursively constructed OR-gate trees are also the inputs of a single OR-gate at the root of the entire tree. The output o of this single OR-gate is the output of the entire tree of gates. The OR-gate specification given by the predicate Or is the obvious combinational one, and its definition is omitted.

The model '$\mathsf{Tfztree}\ t\ (in, out)$' itself is defined simply by composing the model defined above and the model of an inverter:

$$\mathsf{Tfztree}\ t\ (in, out) = \exists o.\, \mathsf{Ortree}\ t\ (in, o) \wedge \mathsf{Not}(o, out)$$

The internal wire represented by o simply connects the tree of OR-gates modelled by $\mathsf{Ortree}\ t\ (in, o)$ to the inverter modelled by $\mathsf{Not}(o, out)$.

5.3.3.4 Consistency of the Model

The term '$\mathsf{Tfztree}\ t\ (in, out)$' defined above models the class of all tree-shaped test-for-zero devices, *both* for all possible shapes of the internal tree of OR-gates and for all possible lengths (except 0) of the input bit-vector *in*. Before basing a correctness proof on this model, however, it is worth checking that the model $\mathsf{Tfztree}\ t\ (in, out)$ is not inconsistent, and can in fact be satisfied for reasonable combinations of values for the variables t and *in*.

To formulate what is meant by 'reasonable', a recursive function is needed which computes the number of leaf nodes in a binary tree. An appropriate function $\mathsf{Leaves}{:}tree{\rightarrow}num$ is straightforward to define by primitive recursion on trees:

$$\begin{array}{lcl} \vdash \mathsf{Leaves}\ \mathsf{Leaf} & = & 1 \\ \vdash \mathsf{Leaves}\ (\mathsf{Node}\ t_1\ t_2) & = & (\mathsf{Leaves}\ t_1) + (\mathsf{Leaves}\ t_2) \end{array}$$

The validity of this definition follows directly from the abstract characterization of the concrete recursive type *btree*.

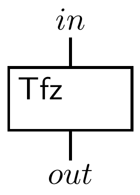
Given this function, and the Length function on lists defined in Section 5.1.1.3, a theorem can be proved which asserts that the model defined in the previous section is consistent for all appropriate combinations of *in* and t:

$$\vdash \forall t\, in.\, \neg(in = [\,]) \supset ((\exists out.\, \mathsf{Tfztree}\ t\ (in, out)) = (\mathsf{Leaves}\ t = \mathsf{Length}\ in))$$

This theorem states that the model $\mathsf{Tfztree}\ t\ (in, out)$ can be satisfied by some value for the variable *out* exactly when the number of leaves in the tree t matches the length of the input word *in*. This means that the model at least represents *some* 'circuit' for every value of the parameter t. The proof of this theorem can be done by structural induction on the variable t ranging over trees. As was mentioned above, the principle of structural induction on trees follows formally from the abstract characterization of the recursive type *btree*.

5.3.3.5 The Specification

An abstract specification of required behaviour for the class of devices modelled by $\mathsf{Tfz}\ t\ (in, out)$ is given by the definition shown below:



$$\vdash \mathsf{Tfz}(in, out) = (out = (in = 0))$$

This specification simply states that the boolean output *out* is true if and only if the numerical value present on the input *in* is equal to zero. The behaviour stipulated by this specification is a *data abstraction* of the behaviour given by the model defined above, and a data abstraction function is therefore needed in the formulation of correctness for the class of all n-bit test-for-zero devices with respect to this abstract specification.

5.3.3.6 Defining the Data Abstraction Function

In the model $\mathsf{Tfztree}\ t\ (in, out)$, the input *in* is an n-bit binary word, and is represented by a variable ranging over values of type $(bool)list$. In the abstract specification, however, this input is simply represented by a number: the free variable *in* in the term '$\mathsf{Tfz}(in, out)$' defined above has type *num*.

In order to translate the bit vector representation of data in the model to the numerical representation in the specification, a data abstraction function $\mathsf{Val}{:}(bool)list{\rightarrow}num$ is defined. This function, which maps n-bit binary numbers to natural numbers, is defined by primitive recursion on lists as follows:

$$\begin{aligned} &\vdash \mathsf{Val}\ [\,] &&= 0 \\ &\vdash \mathsf{Val}\ [b \,|\, l] &&= (2 \times (\mathsf{Val}\ l)) + (b \Rightarrow 1 \mid 0) \end{aligned}$$

It is assumed in this definition that an n-bit binary word is represented by a list of booleans which is ordered from least significant bit (at the start of the list) to most significant bit (at the end of the list).[7] The list '$[\mathsf{T};\mathsf{T};\mathsf{F};\mathsf{T}]$', for example, is assumed to represent the binary number '1011'. The equations for Val shown above compute the natural number corresponding to the unsigned binary representation given by the sequence of bits in any list of boolean values. For example, it follows from these two equations that $\vdash \mathsf{Val}\ [\mathsf{T};\mathsf{T};\mathsf{F};\mathsf{T}] = 11$.

Again, these primitive recursive defining equations can be proved directly (and automatically) from the characterization of the concrete recursive type $(\alpha)list$ given by theorem (5.2). As was mentioned in Section 5.2.5.2, this is one of the major pragmatic advantages of the approach taken here to the characterization

[7]In the test-for-zero example, of course, it does not matter.

of concrete recursive data types in higher order logic: recursively-defined data abstraction functions on concrete types are immediately justified by the theorems which characterize these types in logic.

5.3.3.7 The Correctness Proof

It remains to show that every device in the class of circuit designs modelled by $\mathsf{Tfztree}\ t\ (in, out)$ is functionally correct with respect to the abstract specification of required behaviour. The correctness statement is the theorem shown below:

$$\vdash \forall t.\, \mathsf{Tfztree}\ t\ (in, out) \supset \mathsf{Tfz}((\mathsf{Val}\ in),\ out)$$

This theorem follows easily by structural induction on the binary tree t. It states that every implementation for an n-bit test-for-zero device constructed from an appropriate binary tree t of OR-gates exhibits the functional behaviour stipulated by the relation Tfz. The model is parameterized by the variable t of type *btree*, and this effectively 'quantifies' over all possible shapes that the circuit can have. The correctness statement therefore asserts that *every* circuit is functionally correct with respect to the abstract specification of required behaviour. This theorem also asserts the correctness of these designs for every possible size of the n-bit input in. Finally, the recursively-defined data abstraction function Val is used in this formulation of correctness to relate values of type $(bool)list$ in the model to values of type num in the more abstract specification.

5.4 Summary and Discussion

A fundamental requirement for the effective use of data abstraction in reasoning about hardware behaviour is, of course, a formal representation of *data*. This chapter has shown how a wide class of concrete data types can be characterized formally in higher order logic, and has given two examples to show how these types can be used to support reasoning about hardware behaviour.

A common theme of these examples is the importance of an appropriate choice of types in defining models. In Section 5.2, the simple enumerated type *tri* was used to formulate a CMOS transistor model which is slightly more 'accurate' than the switch model defined in Chapter 3. In Section 5.3, the recursive type $(\alpha)list$ was used to provide a direct and unambiguous formal representation in logic of bit vectors. A new technique was also developed in this section for modelling a class of tree-shaped circuit designs using a concrete type of binary trees.

Two correctness proofs were also given in this chapter to illustrate the approach to data abstraction introduced in Chapter 4. The data abstraction functions in both examples were very simple, and the basic idea behind the data abstraction function Val, in particular, is not new—functions similar to Val have been defined by (for example) Gordon [34], Camilleri [10], Hunt [53], and Joyce [57]. The main aim of this chapter, however, was not to give examples of complex data abstraction functions, but to describe a general approach to the formal characterization in logic of data *types*, these being an essential prerequisite for the specification of hardware behaviour at various different levels of data abstraction.

The logical basis for all the work discussed in this chapter is the systematic method for defining concrete types explained in Appendix A. This method has been automated in the HOL system, and can be used to construct a completely rigorous and formal *definition* for any instance of the general class of concrete types discussed in this chapter. Other tools which were implemented in HOL include programs for proving a structural induction theorem for any concrete type and for defining arbitrary primitive recursive functions on these types. An interactive HOL session which illustrates the use of these tools in proving some of the theorems in this chapter can be found in Appendix B.

The examples given in this chapter do not, of course, exhaust the possible applications for concrete data types in hardware verification. These examples have emphasized the use of special-purpose concrete types in defining *models*, but many applications for concrete data types can also be found in abstract specifications of required behaviour. Some kinds of data which arise naturally at higher levels of data abstraction are, however, more appropriately represented formally in logic by *abstract* data types, and the method described in Appendix A applies to only concrete types. The automation in HOL of formal definitions for abstract types is an important area for future work.

Certain abstract data types, however, can be given straightforward concrete *representations* in logic using types of the kind discussed in this chapter. The integers are a simple example. A formal representation for the integers in higher order logic is given by the concrete type '*int*' defined by:

$$int \;::=\; \mathsf{Neg}\ num \;\mid\; \mathsf{Zero} \;\mid\; \mathsf{Pos}\ num$$

The concrete type *int* defined by this equation provides a representation for the integers in which the negative integers $\{-1, -2, \ldots\}$ correspond to the values $\{\mathsf{Neg}\ 0, \mathsf{Neg}\ 1, \ldots\}$, the integer 0 corresponds to the value Zero, and the positive integers $\{+1, +2, \ldots\}$ correspond to the values $\{\mathsf{Pos}\ 0, \mathsf{Pos}\ 1, \ldots\}$. Once this type has been defined, it is straightforward (but tedious) to develop integer arithmetic in logic by defining appropriate operations (e.g. addition, subtraction, etc.) on values of type *int*. The concrete type *int*, together with the operations which have been defined on it, can then be used as an abstract type of integers in specifications of hardware behaviour.

A final point must be mentioned here concerning the type $(\alpha)list$. This type is used as a logical 'building block' in the general method for defining concrete types explained in Appendix A, and is therefore given an *ad hoc* (but, of course, completely formal) definition in Section A.3.2 of the appendix. The reason for this is that the type $(\alpha)list$ is used to formulate and prove a key general theorem about the class of all concrete type. (See the appendix for details). For the purposes of the present chapter, however, this can be regarded as an accidental feature of the particular method by which concrete types are defined, and the type $(\alpha)list$ can be considered to be simply an *example* of the general class of types definable by this method.

5.5 Related Work

In most of the published work based on the HOL formulation of higher order logic, only the type constants *num* and *bool* and the type operators $\rightarrow$ and $\times$ are used to construct formal representations of data for reasoning about hardware behaviour. A notable exception is Dhingra's use of a four-valued type in [21] to define a CMOS transistor model. (A minor modification of this model, based on the same four-valued type, is also used by Joyce in [55].) But prior to the implementation in the HOL system of the method for defining concrete types reported in this dissertation, substantial and rigorous built-in theories were provided in HOL for only the basic types mentioned above, and many of the proofs about hardware done in the system have been based on only these types.

Early versions of the HOL system, however, had a collection of built-in types for modelling fixed-width binary words and memories (e.g. a type '$word4$' of the set of all 4-bit words). These types were used extensively in the Viper correctness proof [16,17], and were also used by Joyce in [56] and by Camilleri in [10]. But these special-purpose '$wordn$' types (which were inherited from LCF_LSM [32]) were never formally *defined* in HOL. Instead, they were characterized by a built-in collection of constants, axioms, and *ad hoc* inference rules—none of which were given a firm logical foundation in terms of the primitive basis of higher order logic.

The first release of HOL also had a built-in theory of lists. Recursive definitions and induction on lists, however, were not supported by the system, and the type of lists itself was axiomatized[8] rather than defined. In a comparative case study of theorem provers for hardware verification [78], the lack of tools for induction and recursive definitions on lists ('bit vectors') is cited as a serious deficiency of the HOL system. The HOL tools for reasoning about concrete types mentioned in Section 5.1.3 address precisely this problem, and allow induction and recursive definitions to be done easily in HOL on lists or *any* other concrete type. These tools were installed in the 1988/89 release of the public-domain HOL software.

Reasoning about recursive types has been mechanized in a number of other theorem-proving systems. Milner developed a package in LCF for automating the

[8]incompletely, but not, in fact, by the inconsistent axioms in [29,30] (cf. note 1 on page 26).

construction of lazy recursive types and the derivation of structural induction [69]. This package was later extended by Monahan [70]. The structural induction tools in Cambridge LCF [74] are another development of Milner's package. The HOL system is based on the LCF approach to mechanized theorem proving, and the present author's mechanization of type definitions and (in particular) structural induction in HOL in some ways resembles that of the LCF tools mentioned above. But the purely logical details differ considerably, since the two systems are based on different formalisms. The most important difference is that the HOL tools are based on the highly restrictive primitive rule for type *definitions* in higher order logic. The LCF type constructions are more axiomatic in flavour (consistency of the axioms is justified by domain theory).

The Boyer-Moore theorem prover includes an axiom-scheme, called the *shell principle*, which allows the user to axiomatize recursive structures in the Boyer-Moore logic [9]. This was used by Hunt to introduce several recursive data types in the correctness proof of the FM8501 microprocessor [53], including a recursive type of *lists* used to represent bit-vectors. Other types axiomatized for the FM8501 proof include the natural numbers and a concrete representation of the integers. The difference between Hunt's work and the approach presented in this chapter is that the shell principle introduces a type simply by postulating a Peano-like axiomatization for it. In higher order logic, however, a type must be *defined* and an abstract characterization of the type *derived* by formal proof. The main contribution of the present work is the automation of this process.

Veritas is also a typed higher order logic, and types play an important role in the Veritas approach to hardware specification and verification [43]. The Veritas logic (as used in [43]) does not have an explicitly-stated rule of definition for the introduction of new types. New types are instead introduced axiomatically—i.e. by postulating axioms that ascribe properties to them. But there seems to be no reason (in principle) why a restriction to axioms of a definitional kind could not also be imposed in Veritas.[9]

Finally, it must be mentioned that the basic idea behind the representation for bit vectors proposed in this chapter is, of course, not new. Lists are a natural representation for bit vectors, and have been used to represent them both in the Boyer-Moore logic [53] (as mentioned above) and in Interval Temporal Logic [72]. Chin *et al.* [13] have also used lists to represent bit vectors in HOL.[10]

[9]Indeed, this has been done in a recent development of Veritas called 'Veritas$^+$' [45].

[10]This work was done using an early prototype of the present author's implementation in HOL of the theory of lists and related tools for induction and recursive definitions on lists. These are now superseded by the HOL tools for the general class of concrete types.

Chapter 6

Temporal Abstraction

Temporal abstraction, as was discussed in Chapter 4, involves relating formal specifications that describe hardware behaviour using different notions of discrete time. This type of abstraction is used when the formal model of a device gives more detail about how it behaves over time than the abstract specification of its required behaviour. With the mechanism of temporal abstraction, information about a device's behaviour at moments of time that are not of interest can be hidden from the abstract specification, allowing the specification to concentrate on how the device behaves at only significant or interesting points of time.

6.1 Temporal Abstraction by Sampling

The idea behind temporal abstraction by sampling was introduced in Chapter 4. With this type of temporal abstraction, the abstract specification for a device simply describes its externally observable behaviour at fewer points of time than the formal model of its design. The grain of discrete time used in the specification is 'coarser' than the grain of discrete time used in the model, and each *unit* of discrete time at the abstract level of description corresponds to an *interval* of time at the more detailed level of description.

To express this abstraction relationship formally in logic, the idea of a mapping between time-scales was introduced in Chapter 4. A mapping of this kind specifies a correspondence between successive points of time on an abstract time-scale and selected points of time on a concrete time-scale. Given an appropriate time mapping f, a correctness statement based on temporal abstraction by sampling is formulated in logic as shown below:

$$\vdash M[c_1, \ldots, c_n] \supset S[c_1 \circ \mathsf{f}, \ldots, c_n \circ \mathsf{f}]$$

The model $M[c_1, \ldots, c_n]$ in this correctness theorem describes the values that appear on each external wire c_i at points of fine-grained or concrete time. The abstract specification is a constraint of the form $S[a_1, \ldots, a_n]$, and specifies the desired behaviour in terms of the values allowed on its external wires at points of coarse-grained or abstract time. The time mapping f defines the intervals of concrete time that correspond to each unit of abstract time. The correctness theorem states that whenever a sequences of values denoted by c_i satisfies the temporally detailed model, the subsequence $c_i \circ \mathsf{f}$, obtained by sampling c_i at the points of concrete time specified by f, will satisfy the temporally abstract specification of required behaviour. Proving a correctness statement of this form involves showing that if the sequences $c_1, \ldots, c_n$ take on the intermediate values defined by model, then the values of these sequences the selected points of time specified by the time mapping f will satisfy the abstract specification.

This correctness relationship is formulated as an implication, rather than an equivalence, because there may be several non-equivalent ways of implementing behaviour specified by the abstract constraint $S[a_1, \ldots, a_n]$. The implementation in which the sequences $c_1, \ldots, c_n$ take on the intermediate values defined by the model $M[c_1, \ldots, c_n]$ is only *one* such method. In the example given above, every sequence c_i in the model (every '*signal*') is sampled using the same time mapping f. In general, however, it is not necessary that the same time mapping be used for every signal. For example, some signals may be sampled at points of time corresponding to the rising edges of a clock, while others are sampled at points of time corresponding to the falling edges of a clock. Examples of this kind occur in the T-ring case study discussed below in Section 6.3.

Any correspondence between successive units of abstract time and contiguous intervals of concrete time can be specified formally in logic by a time mapping of the kind used in the correctness statement shown above. Such a mapping is just a function f of type $num{\rightarrow}num$ that assigns a particular point of concrete time to each point of abstract time, as shown in Figure 6.1. Not every function of logical type $num{\rightarrow}num$, however, specifies a valid correspondence between time-scales. A mapping from abstract to concrete time must be a strictly *increasing* function on the natural numbers. This requirement on a time mapping f can be expressed formally by the predicate Incr defined as follows.

$$\vdash \mathsf{Incr}\ f = \forall t_1\, t_2.\ (t_1 < t_2) \supset (f\ t_1 < f\ t_2)$$

This ensures that if time t_1 comes before time t_2 on the abstract time-scale, then this relationship also holds between the corresponding points of time $\mathsf{f}\ t_1$ and $\mathsf{f}\ t_2$ on the concrete time-scale.

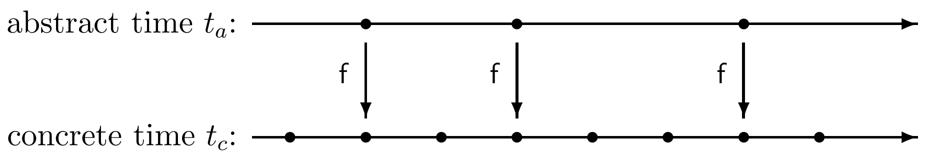


Figure 6.1: A Typical Mapping between Time-scales.

6.1.1 Constructing Mappings between Time Scales

The first step in the formulation of a correctness statement that involves temporal abstraction by sampling is to define an appropriate mapping from the abstract time-scale used in the specification to the concrete time-scale used in the model. In general, the points of concrete time that correspond to points of abstract time may depend on the behaviour of the device itself. In this case, a fixed mapping from abstract time to concrete time—for example a function that maps successive points of abstract time to every tenth point of concrete time—is not possible.

Consider, for example the correspondence between time-scales shown below in Figure 6.2. Here, successive points of abstract time correspond to the points of concrete time at which there is a rising edge of the clock signal *ck*. The precise correspondence between concrete time and abstract time depends on the behaviour of this clock signal, and the mapping f must be defined in such a way as to reflect this dependence. This can be done formally by *constructing* the function f shown in Figure 6.2 from the predicate 'Rise *ck*', which identifies those points of time on the concrete time-scale at which the rising edges of the clock *ck* occur.

In general, any time mapping can be defined formally by means of a predicate that specifies which points of time on the concrete time-scale are to correspond to points of time on the abstract time-scale. The idea is to define this predicate such that it is true of precisely those selected points of concrete time which are to be in the image of the mapping from abstract time to concrete time. The free variables in the model can themselves be used as parameters to this predicate. In synchronous systems, for example, the appropriate points of concrete time can often be identified by the value of a clock signal *ck*. (In asynchronous systems, handshaking signals might be used for the same purpose.) The required time mapping can then be constructed from the values given by this predicate on concrete time. This allows the mapping from abstract time to concrete time used in a correctness statement to reflect the time-dependent behaviour of the device itself (as described by the model). The time mapping does not assign a fixed point

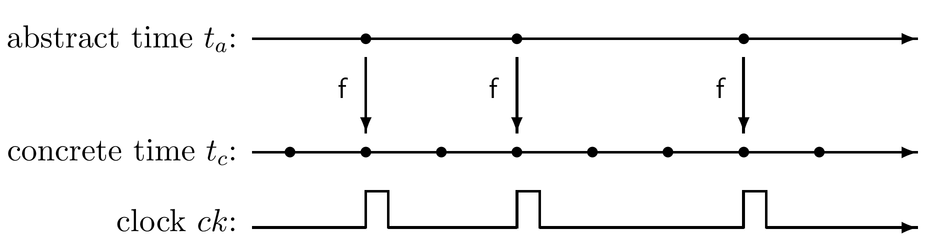


Figure 6.2: A Time Mapping which Depends on a Clock *ck*.

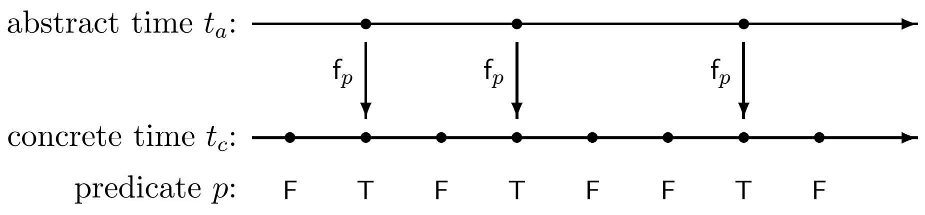


Figure 6.3: Constructing a Time Mapping f_p from a Predicate p.

of concrete time to each point of abstract time, but establishes a correspondence between time-scales that covers the entire range of time-dependent behaviour described by the model.

To construct a time mapping in this way, it sufficient to define a predicate $p{:}num{\to}bool$ that is true of precisely those points of concrete time which are to correspond to points of abstract time. Given such a predicate p, it is possible to construct a mapping f_p which assigns each point of abstract time n to the nth point of concrete time at which p is true, as shown in Figure 6.3 above. If for any predicate p and abstract time n the term '$\mathsf{Timeof}\ p\ n$' denotes the point of concrete time at which p is true for the nth time, then the mapping between time-scales f_p shown in this diagram can be defined by:

$$\vdash \mathsf{f}_p\ n = \mathsf{Timeof}\ p\ n \qquad\qquad \text{(i.e.}\quad \vdash \mathsf{f}_p = \mathsf{Timeof}\ p\text{)}$$

It remains to define the function Timeof formally in higher order logic.

6.1.2 Defining the Function Timeof

The term '$\mathsf{Timeof}\ p\ n$', as informally described above, may in fact be undefined for some values of p and n. If the predicate p true at only a finite number of points of concrete time, then there will be some number N such that for all $n > N$ there is no concrete time at which p is true for the nth time. The value of $\mathsf{Timeof}\ p\ n$ is therefore 'undefined' for these values of n, and the function $\mathsf{Timeof}\ p$ is itself a partial function.

In higher order logic, however, all functions must be total functions. The higher order function Timeof will therefore be defined to be a total function whose value is only partially specified. This will be done by using the primitive constant ε to define Timeof such that '$\mathsf{Timeof}\ p$' denotes the required mapping between time-scales when the predicate p is true infinitely often, and denotes a mapping about which nothing can be proved when p is true only finitely often.

6.1.2.1 The Relation $\mathsf{Istimeof}$

The formal definition of Timeof is based on a relation $\mathsf{Istimeof}$, defined such that the term '$\mathsf{Istimeof}\ p\ n\ t$' has the meaning '$p$ is true for the nth time at time t'. The definition of this relation is done by primitive recursion on the natural number n. When n is zero, the defining equation is:

$$\vdash \mathsf{Istimeof}\ p\ 0\ t = p\ t \wedge \forall t'.\, t' < t \supset \neg(p\ t')$$

That is, the predicate p is true for the first (i.e. the 0th) time at concrete time t if p is true at time t and false at every point of time prior to time t. For the (n+1)th time at which p is true, the defining equation is:

$$\vdash \mathsf{Istimeof}\ p\ (\mathsf{Suc}\ n)\ t = \exists t'.\, \mathsf{Istimeof}\ p\ n\ t' \wedge \mathsf{Next}\ t'\ t\ p$$

where the auxiliary predicate Next is defined by:

$$\vdash \mathsf{Next}\ t_1\ t_2\ p = t_1 < t_2\ \wedge\ p\ t_2\ \wedge\ \forall t.\, (t_1 < t \wedge t < t_2) \supset \neg p\ t$$

In this case, the defining equation for $\mathsf{Istimeof}$ states that p is true for the (n+1)th time at concrete time t if there exists a point of concrete time t' prior to time t at which p is true for the nth time, and t is the next time after t' at which p is true. To summarize, the primitive recursive definition of the relation $\mathsf{Istimeof}$ is given by the two theorems shown below.

$$\begin{aligned} &\vdash \mathsf{Istimeof}\ p\ 0\ t &&= p\ t\ \wedge\ \forall t'.\, t' < t \supset \neg p\ t' \\ &\vdash \mathsf{Istimeof}\ p\ (\mathsf{Suc}\ n)\ t &&= \exists t'.\, \mathsf{Istimeof}\ n\ p\ t' \wedge \mathsf{Next}\ t'\ t\ p \end{aligned}$$

The formal justification for this recursive definition follows from the primitive recursion theorem for the natural numbers, using the method of deriving recursive defining equations discussed in Section 2.1.6 of Chapter 2.

6.1.2.2 A Theorem about $\mathsf{Istimeof}$

This primitive recursive definition of $\mathsf{Istimeof}\ p\ n\ t$ captures the idea that the predicate p is true for the nth time at concrete time t. There is no guarantee, however, that such a time t exists for all values of p and n. In order to use the relation $\mathsf{Istimeof}$ to define the function Timeof, it is necessary to show that if the predicate p is true infinitely often, then for all n there is a unique time t at which p is true for the nth time. That is, if p is true infinitely often, then the relation $\mathsf{Istimeof}\ p\ n\ t$ defines a unique value t for each value of n, and therefore in fact represents well-defined total function that maps p and n to t.

The condition that p must be true at an infinite number of points of concrete time is stated formally by the predicate Inf defined below.

$$\vdash \mathsf{Inf}\ p = \forall t.\ \exists t'.\ t' > t \wedge p\ t'$$

Given this predicate, it is straightforward to show that if p is true infinitely often, then for all n there exists a unique time t at which p is true for the nth time:

$$\vdash \mathsf{Inf}\ p \supset \forall n.\ \exists!\, t.\ \mathsf{Istimeof}\ p\ n\ t \tag{6.1}$$

The formal proof of this theorem proceeds by proving the existence and uniqueness parts separately. The existence of t follows by induction on n, using the well ordering property of natural numbers:

$$\vdash \forall p.\ \exists t. p\ t \supset \exists t.\ p\ t \wedge \forall t'. t' < t \supset \neg p\ t'$$

to infer from the assumption $\mathsf{Inf}\ p$ that there is always a smallest *next* time at which the predicate p is true. The uniqueness of t also follows by induction on n.

6.1.2.3 Using Istimeof to Define Timeof

Given theorem (6.1), the relation Istimeof can be used to define the function Timeof as follows. Using the primitive constant ε, the function Timeof can be defined formally by the equation shown below.

$$\vdash \mathsf{Timeof}\ p\ n = \varepsilon\ (\mathsf{Istimeof}\ p\ n)$$

This equation defines the term Timeof $p\ n$ to denote some time, t say, such that Istimeof $p\ n\ t$ is true. If no such time exists, then Timeof $p\ n$ denotes an arbitrary natural number. Using the primitive constant ε, this definition makes the term 'Timeof p' always denote a total function. The term 'Timeof $p\ n$' denotes *some* natural number for all values of n and p, even when the predicate p is true at only a finite number of points of concrete time.

If, however, the predicate p is true infinitely often, then for all n there will exist a unique time t such that Istimeof $p\ n\ t$ is true. Thus, if Inf p holds, then Timeof $p\ n$ will in fact denote the unique time at which p is true for the nth time, as desired. More formally, an immediate consequence of the existence part of theorem (6.1) is:

$$\vdash \mathsf{Inf}\ p \supset \mathsf{Istimeof}\ p\ n\ (\mathsf{Timeof}\ p\ n)$$

from which it follows immediately that:

$$\vdash \mathsf{Inf}\ p \supset p(\mathsf{Timeof}\ p\ n)$$
$$\vdash \mathsf{Inf}\ p \supset \forall t.\ (t < (\mathsf{Timeof}\ p\ 0)) \supset \neg p\ t$$

That is, if the predicate p is true infinitely often, then $\mathsf{Timeof}\ p\ n$ always denotes a point of concrete time at which p is in fact true, and $\mathsf{Timeof}\ p$ maps 0 to the first time at which p is true. From the uniqueness part of theorem (6.1) it also follows that $\mathsf{Timeof}\ p$ denotes an increasing function from abstract to concrete time, and that this function does not skip any points of concrete time identified by the predicate p:

$$\vdash \mathsf{Inf}\ p \supset \forall n.\,(\mathsf{Timeof}\ p\ n) < (\mathsf{Timeof}\ p\ (n{+}1))$$
$$\vdash \mathsf{Inf}\ p \supset \forall n\ t.\,(\mathsf{Timeof}\ p\ n) < t \wedge t < (\mathsf{Timeof}\ p\ (n{+}1)) \supset \neg p\ t$$

These lemmas about Timeof show that if the predicate p is true infinitely often, then the term '$\mathsf{Timeof}\ p$' is a well defined total function and denotes the desired mapping from abstract time to selected points of concrete time. The function $\mathsf{Timeof}\ p$ maps each point of abstract time n to the point of concrete time at which p is true for the nth time, as required.

6.1.3 Using Timeof to Formulate Correctness

Having formally defined the function Timeof, and shown that it constructs a well defined time mapping for any predicate p that is true at an infinite number of points of concrete time, it is possible to use Timeof to formulate correctness theorems based on temporal abstraction by sampling. The time mapping required for such a correctness theorem just an increasing function f of type $num{\rightarrow}num$. Any such function can be defined using Timeof and an appropriate predicate p which indicates the points of concrete time that are to correspond to points of abstract time. Formally, the property that a function f is strictly increasing is logically equivalent to the assertion that f can be constructed from a predicate p for which $\mathsf{Inf}\ p$ holds:

$$\vdash \forall f.\,\mathsf{Incr}\ f = \exists p.\,\mathsf{Inf}\ p \wedge f{=}\mathsf{Timeof}\ p$$

This theorem follows from the definition of the constant Incr and the properties of Timeof discussed above in Section 6.1.2.3.

If p is an appropriate predicate that indicates which points of concrete time correspond to points of abstract time, then a correctness theorem that relates a detailed design model $M[c_1,\ldots,c_n]$ to an abstract specification $S[a_1,\ldots,a_n]$ can be formulated in logic as shown below:

$$\vdash M[c_1,\ldots,c_n] \supset S[c_1 \circ (\mathsf{Timeof}\ p),\ldots,c_n \circ (\mathsf{Timeof}\ p)]$$

This correctness theorem states that whenever the signals $c_1\ \ldots,\ c_n$ satisfy the model, the abstract signals constructed by sampling $c_1,\ \ldots,\ c_n$ when the predicate p is true will satisfy the temporally abstract specification. In the general case, the predicate p can be defined in terms of the variables $c_1, \ldots, c_n$, in order to make the times at which the values in the model are sampled depend on the behaviour of the device itself.

If when[1] is an infix constant defined formally as follows:

$$\vdash s \,\mathsf{when}\, p = s \circ (\mathsf{Timeof}\ p)$$

then this correctness statement can be written:

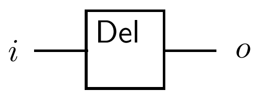
$$\vdash M[c_1, \ldots, c_n] \supset S[c_1 \,\mathsf{when}\, p, \ldots, c_n \,\mathsf{when}\, p]$$

Since every mapping from abstract to concrete time can be constructed using 'Timeof' from an appropriate predicate p, any correctness relationship based on temporal abstraction by sampling can be expressed in this form.

The example given in the next section shows how the when operator defined above can be used to formulate the correctness of a D-type flip flop with respect to the abstract specification of one-bit unit delay register.

6.2 A Simple Example

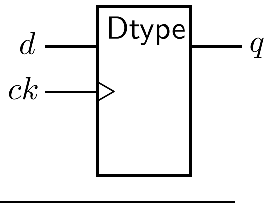
A commonly used register-transfer level device is the unit delay, described formally by the specification shown below.



$$\vdash \mathsf{Del}(i, o) = \forall t.\, o(t{+}1) = i\ t$$

This specification simply states that the value on the output o is equal to the value on the input i delayed by one unit of discrete time.

The unit delay device described by this specification is an abstraction—there are many circuits that can implement the abstract behaviour described by the specification $\mathsf{Del}(i, o)$. One possible implementation is the rising edge triggered D-type flip flop discussed in Chapter 3. The sequential behaviour of this device is modelled in logic by the term $\mathsf{Dtype}(ck, d, q)$ defined below:



$$\vdash \mathsf{Rise}\ ck\ t = \neg ck(t) \wedge ck(t{+}1)$$

$$\vdash \mathsf{Dtype}(ck, d, q) = \forall t.\, q(t{+}1) = (\mathsf{Rise}\ ck\ t \Rightarrow d\ t \mid q\ t)$$

[1] In an early account of this work, the function when was called 'ABS' [62]. The mnemonically superior name 'when' was suggested by the LUSTRE operator of the same name [38].

Informally, the D-type device shown above implements a unit delay by sampling the input value d when the clock rises and holding this value on the output q until the next rise of the clock. In this way the D-type delays by one clock period the sequence of values consisting of the values present on the input d at successive rising edges of the clock ck. This suggests that the time mapping used to relate the model $\mathsf{Dtype}(ck, d, q)$ to the abstract specification $\mathsf{Del}(i, o)$ should map successive points of abstract time to the points of concrete time at which the clock rises.

Using the constant Rise, the required time mapping is given by the term '$\mathsf{Timeof}\ (\mathsf{Rise}\ ck)$'. Given the mapping between time-scales denoted by this term, a correctness statement that relates the D-type model to the unit delay specification can then formulated as shown below:

$$\vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \supset (\mathsf{Dtype}(ck, d, q) \supset \mathsf{Del}(d\ \mathsf{when}\ (\mathsf{Rise}\ ck), q\ \mathsf{when}\ (\mathsf{Rise}\ ck)))$$

This correctness theorem states that if the sequences given by the variables ck, d, and q satisfy the model, then the abstract sequences obtained by sampling d and q at successive rising edges of the clock ck will satisfy the abstract specification for a unit delay device. Here, there is a *family* of sampling functions used to relate the model to the abstract specification. For each value of the clock ck, the term $\mathsf{Rise}\ ck$ denotes an appropriate predicate that identifies the points of concrete time which at which the clock rises. The infix when operator is then used to sample the sequences d and q whenever this predicate is true.

The assumption that the clock rises infinitely often is a *validity condition* on the abstraction relationship expressed by this correctness statement. The theorem asserts that the specification represents a valid abstract view of the behaviour of a D-type device only if the validity condition '$\mathsf{Inf}(\mathsf{Rise}\ ck)$' is satisfied. This validity condition on the clock must be met by the environment in which the D-type flip flop is placed. The condition itself is as unrestrictive as possible: the clock ck is not required to be regular or have a minimum period. The liveness condition on the clock input expressed by $\mathsf{Inf}(\mathsf{Rise}\ ck)$ is sufficient for the D-type device to function correctly as specified by $\mathsf{Del}(i, o)$

The proof of this correctness theorem is straightforward. The main step is an induction on the number of time steps between adjacent rises of the clock, showing that the value on the input d that is sampled at each rising edge of the clock ck is held on the output q until the next rising edge. The correctness theorem then follows easily from this result and the properties of Timeof proved above in Section 6.1.2.3.

This proof provides a very simple example of a common type of temporal abstraction, where contiguous intervals of concrete time correspond to successive units of abstract time. Examples involving detailed timing information or several different time mappings in the same correctness statement are typically much more complex than the simple example given here. But—as far as the abstraction relationship itself is concerned—more complex examples of temporal abstraction by sampling involve the same general approach as illustrated by this example.

6.3 The T-ring: A Case Study

The T-ring is a very simple ring communication network, designed and built by D. Gaubatz and M. Burrows at the University of Cambridge [27]. It was designed to provide a simplified ring network which could be specified and proved correct as a prelude to attempting the much more difficult formal verification of the Cambridge Fast Ring network [52]. (The verification of the Cambridge Fast Ring network was never actually attempted, but an ECL chip used as a part of the Fast Ring network was verified by Herbert in [48].) The T-ring was also used by members of the hardware verification group in Cambridge to explore timing and documentation issues.

This section outlines the main results in a proof of correctness for the design of the T-ring, in which the constructs for defining time mappings discussed in the preceding sections are used. All the lemmas and intermediate correctness results concerned with temporal abstraction in this correctness proof for the T-ring were proved formally using the HOL theorem prover. The top-level correctness theorem stated in Section 6.3.6, however, was proved manually. But neither the details of the HOL proofs done for this example (which take over 3000 lines of ML source code to generate) nor the details of the additional proofs which were done by hand will be given here. The aim of this section is to give an overview of relatively complex example of temporal abstraction, without burdening the reader with the details of the highly intricate (but generally shallow) formal proofs in higher order logic that were involved.

6.3.1 Informal Description of the T-ring

The T-ring has three major components: the *transmitter*, the *receiver*, and the *monitor*. These components are connected together to form a data transmission network in the shape of circular ring, as shown in Figure 6.4. The transmitter sends messages (clockwise) around the ring to the receiver. Each message sent by the transmitter contains only one bit of information. No source or destination address is included in a message, since in the T-ring network there is only one transmitter and one receiver.

Storage for messages in a ring network is normally provided by delay in the transmission wires which run between the devices connected to it. In the TTL implementation of T-ring, however, these transmission wires are very short, and they impose virtually no delay between the components connected to the ring. Storage is therefore ‘artificially’ supplied in the T-ring by three delay devices

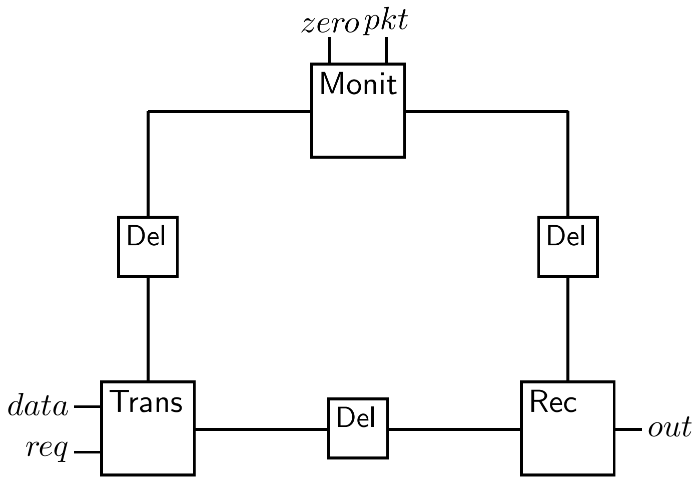


Figure 6.4: The T-ring.

inserted between each pair of the other three components in the ring. These are labelled Del in Figure 6.4. Each of these delay devices is simply a shift register which imposes 8 bits of delay between its input and its output. Together, these shift registers supply the T-ring network with the storage for circulating messages which in a more realistic ring network is provided by delay in the transmission wires. The transmitter, receiver, and monitor themselves contribute no delay to the ring, so the T-ring network has a total of 24 bits of storage.

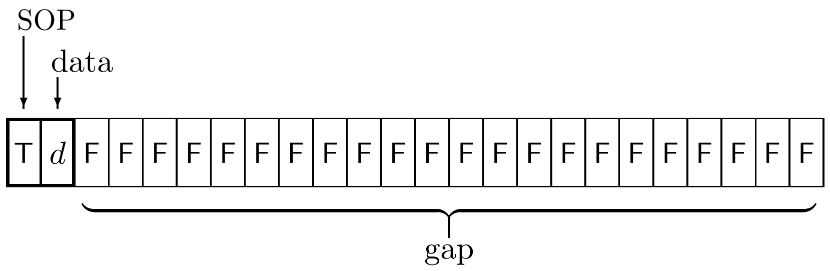


Figure 6.5: The T-ring Slot Structure.

Messages are transmitted serially around the ring encoded in a simple format called a *packet*. A packet consists of only two bits: a start-of-packet (SOP) bit followed by a data bit. The start-of-packet bit is always a boolean T. There is only one such packet circulating in the ring at any time. The other 22 bits of storage in the ring constitute a *gap*, each bit of which has the value F. This pattern of bits—a single packet followed by 22 bits of gap—is called the *slot structure* of the ring (see Figure 6.5). The circulating packet is a *slot* in the circular bit pattern, where data can be inserted into the ring and read from the ring. The devices connected to the ring detect the presence of the packet by sensing a transition from F (the end of the gap) to T (the SOP bit).

The monitor is used to create the slot structure of the T-ring. The monitor has two inputs: *zero* and *pkt*. When the *zero* input is activated, a boolean F is inserted into the bit pattern on the ring at the monitor's output. The first step in creating the slot structure is to activate *zero* long enough to set each bit in the ring to F. The *pkt* input is then activated once, and a single start-of-packet bit is inserted into the bit pattern on the ring. This creates one packet on the ring, and completes the creation of the T-ring slot structure.

Once the slot structure exists, the ring is ready to carry one-bit messages from the transmitter to the receiver. The transmitter has two inputs: *req* and *data*. A request to transmit data is made by activating the *req* input. Once a request has been made, subsequent requests are ignored until the pending request has been serviced. When there is a pending request to transmit data, the transmitter waits until the circulating packet arrives on its input from the ring, and then inserts the bit which is currently on the *data* input into the packet just after the SOP bit. Because the *data* input is not read until the packet arrives at the transmitter, the data bit which is sent to the receiver will not necessarily be the value on the *data* line at the time of the request. The receiver simply reads the data bit in the circulating packet each time it comes around, and makes this value available on the output wire *out*.

6.3.2 The Specification

To write a formal specification of the T-ring behaviour informally described above, the two auxiliary predicates During and After are used to express conditions that must be satisfied by the T-ring inputs *pkt* and *zero* in order to initialize the slot structure of the ring. These two predicates have the formal definitions shown below:

$$\vdash \mathsf{During}\ t_1\ t_2\ v\ w = \forall t.\,(t_1 \leq t \wedge t \leq t_2) \supset w\ t = v$$

$$\vdash \mathsf{After}\ t\ w\ v = \forall t'.\,(t < t') \supset w\ t' = v$$

The predicate During expresses the condition that an input wire w has a constant value v during an interval of time from time t_1 to time t_2 (inclusive). The predicate After expresses the condition that an input wire w has a constant value v after a given time t_1.

Using these two predicates, a formal specification that describes the behaviour of the T-ring network is given by the term 'Tring *zero pkt data req out*' defined below:

$$
\begin{array}{l}
\vdash \mathsf{Tring}\ \mathit{zero}\ \mathit{pkt}\ \mathit{data}\ \mathit{req}\ \mathit{out} = \\
\quad \mathsf{During}\ t\ (t{+}26{+}n{+}m)\ \mathsf{F}\ \mathit{pkt}\ \wedge \\
\quad \mathsf{During}\ t\ (t{+}25{+}n)\ \mathsf{F}\ \mathit{zero}\ \wedge \\
\quad \mathit{pkt}(t{+}27{+}n{+}m)\ \wedge \\
\quad \mathsf{After}(t{+}28{+}n{+}m)\ \mathsf{F}\ \mathit{pkt}\ \wedge \\
\quad \mathsf{After}(t{+}29{+}n{+}m)\ \mathsf{T}\ \mathit{zero}\ \supset \\
\quad \forall t'.\, t' \geq (t{+}21{+}n{+}m) \wedge \mathit{req}\ t' \supset \\
\qquad \exists n.\, (0 \leq n \wedge n \leq 24) \wedge \mathit{out}(t'{+}17{+}n) = \mathit{data}(t'{+}n{+}1)
\end{array}
$$

The specification is stated in the form of an implication. The antecedent of the implication describes the sequences of values that must appear on the monitor inputs *pkt* and *zero* in order to create the T-ring slot structure described in the previous section. The consequent of the implication describes the transmission of data through the T-ring network once the slot structure have been initialized. These two parts of the specification are explained briefly in the next two sections.

6.3.2.1 Initialization

The initialization sequence described by the T-ring specification defined above consists of the following steps:

1. Starting at some time t, activate the *zero* input for at least 25 units of time in order to clear the entire bit pattern in the ring. While clearing the ring, do not activate the *pkt* input. After the bit pattern in the ring has been cleared, the *pkt* input can remain inactive as long as desired. The *zero* input is active *low* and the *pkt* input is active *high*, so the required input sequence is given by:

   $$\mathsf{During}\ t\ (t{+}25{+}n)\ \mathsf{F}\ \mathit{zero}\ \wedge\ \mathsf{During}\ t\ (t{+}26{+}n{+}m)\ \mathsf{F}\ \mathit{pkt}$$

2. Activate the *pkt* input once to create a single a packet in the circulating bit pattern in the ring. This creates the T-ring slot structure. The *pkt* input can be activated any time after the ring has been cleared:

   $$\mathit{pkt}(t{+}27{+}n{+}m)$$

3. Do not activate the *pkt* and *zero* inputs again after the T-ring slot structure has been created:

   $$\mathsf{After}(t{+}28{+}n{+}m)\ \mathsf{F}\ \mathit{pkt}\ \wedge\ \mathsf{After}(t{+}29{+}n{+}m)\ \mathsf{T}\ \mathit{zero}$$

The particular numerical constants which appear in this initialization sequence (e.g. '25' and '26' in step 1) were derived in the course of the correctness proof for the design of the T-ring. These numbers represent the shortest periods of time for which the *pkt* and *zero* inputs must have the required values to initialize the ring.

The variable n stands for an arbitrarily long (but, of course, finite) additional period of time during which the *zero* input can remain active once the ring is cleared. Likewise, the variable m stands for an arbitrarily long period of time between the time at which the ring is cleared and the time at which the *pkt* input is activated.

6.3.2.2 Data Transmission

The consequent of the formal specification shown above describes the transfer of data from the transmitter to the receiver once the T-ring has been initialized. The ring is able to receive requests on the *req* input to transmit data as early as 6 time units before the packet is inserted during the initialization sequence (time $(t{+}21{+}n{+}m)$ in the specification). If *req* input is activated at some time t' on or after this time, the value on the *data* input will be read within 24 time units of time t' and this value will appear on the output *out* 16 time units after it is read. Formally, if *req* is true at time t', then the following relationship will hold:

$$\exists n.\, (0 \leq n \wedge n \leq 24) \wedge \mathit{out}(t'{+}17{+}n) = \mathit{data}(t'{+}n{+}1)$$

That is, the data bit present on the transmitter *data* input will be sent to the receiver output *out*. The transmission delay is 16 bits between *data* and *out*.

The formal specification defined above is only a *partial* specification of the behaviour of the T-ring. It leaves unspecified, for example, the behaviour of the T-ring when uninitialized. Furthermore, it covers only the most basic requirement for the correct operation of the device—namely, the ability to transmit one bit of data from the receiver to the transmitter whenever requested to do so. This specification therefore represents a *behavioural* abstraction of the more detailed behaviour given by the T-ring design model.

The correctness results which were proved to show that the T-ring design model satisfies the abstract specification defined above are discussed in the sections that follow. The proof was structured into a two-level hierarchy of correctness results. These are considered in 'bottom up' order in the following sections. At the lower level of the hierarchy—the *timing level*—a correctness theorem is proved for each of the four kinds of components in the T-ring network: the delay devices, the receiver, the transmitter, and the monitor. The circuit designs for these components, and the correctness result for each designs, are discussed in Section 6.3.5. The next level of the proof is the *register-transfer* level, at which the abstract specifications for each component are composed to obtain a model for the entire T-ring, and this model verified with respect to the top-level T-ring specification discussed above. The correctness result at this level of the proof is discussed in Section 6.3.6. Finally, in Section 6.3.7, a correctness theorem that relates the timing level design to the top-level specification is derived from the correctness results obtained at each of the two levels of the proof. The discussion begins with an overview of the timing scheme for the TTL implementation of T-ring network, and a description of the primitive hardware components used in the design of the T-ring.

6.3.3 T-ring Timing

The T-ring is a synchronous TTL system driven by a single master clock. Values on the ring change on the falling edge of the clock and are sampled by the transmitter, receiver and monitor on the rising edge of the clock. The idea of this timing scheme is to ensure that the value on the ring is sampled by the transmitter, the receiver, and the monitor when it is most likely to be stable—half-way between two falling edges of the clock. This timing scheme is implemented by having the primitive components that *read* from the ring triggered on the rising edges of the clock and the components that *write* to the ring triggered on the falling edges of the clock. A consequence of this approach is that the T-ring correctness proof involves temporal abstraction by sampling on both rising and falling edges of the clock.

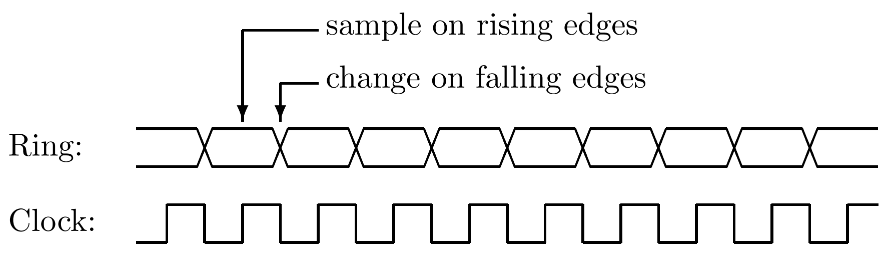
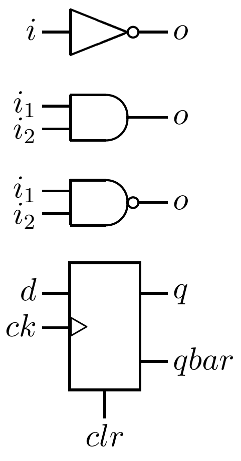


Figure 6.6: T-ring Timing.

6.3.4 Specifications of the Primitives

There are three combinational primitives used in the T-ring: the inverter, the AND-gate, and the NAND-gate. Their specifications in logic are simple, and are shown in Figure 6.7. The only primitive device with state which is used in the T-ring is the rising edge triggered D-type flop flop with asynchronous clear, also shown in Figure 6.7. The formal specification of this device is similar to that of the D-type discussed earlier in this chapter. The only differences are the addition of an active-low 'clear' input *clr*, and an inverted output *qbar*.

The D-type flip flop shown in Figure 6.7 is sometimes used with the *clr* input disabled. When this asynchronous clear input is not used, it is driven by the power source defined below:



$\vdash \mathsf{Not}(i,o) = \forall t.\, o\,t = \neg(i\ t)$

$\vdash \mathsf{And}(i_1,i_2,o) = \forall t.\, o\,t = (i_1\ t \wedge i_2\ t)$

$\vdash \mathsf{Nand}(i_1,i_2,o) = \forall t.\, o\,t = \neg(i_1\ t \wedge i_2\ t)$

$$\begin{aligned}&\vdash \mathsf{Dtype}(ck, clr, d, q, qbar) = \\ &\quad \forall t.\, q(t{+}1) = (clr(t{+}1) \Rightarrow (\mathsf{Rise}\,ck\ t \Rightarrow d\ t \mid q\ t) \mid \mathsf{F}) \wedge \\ &\quad \forall t.\, qbar\ t = \neg(q\ t)\end{aligned}$$

Figure 6.7: Primitive Devices used in the T-ring.

$\vdash \mathsf{Pwr}\ p = \forall t.\, p\ t = \mathsf{T}$

In this case, the D-type specification simplifies to:

$$\begin{aligned}&\vdash (\exists p.\, \mathsf{Pwr}\ p \wedge \mathsf{Dtype}(ck, p, d, q, qbar)) = \\ &\quad \forall t.\, q(t{+}1) = (\mathsf{Rise}\ ck\ t \Rightarrow d\ t \mid q\ t)\ \wedge \forall t.\, qbar\ t = \neg(q\ t)\end{aligned}$$

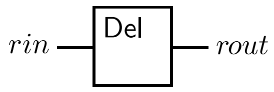
For clarity, the *clr* inputs of D-type flip flops will not be shown in circuit diagrams when they are disabled in this way. Similarly, if the *qbar* output of a flip flop is not used, it will also be omitted. The power source defined above is sometimes used to drive other internal wires in the design of the T-ring. This will be indicated in the circuit diagrams shown in the following sections by writing ‘T’ beside these wires.

6.3.5 Correctness of the T-ring Components

In the next four sections, the circuit design for each of the four components in the T-ring is described, the abstract specification device is defined, and an overview is given of the correctness results proved for each T-ring component.

6.3.5.1 The Delay Devices

The simplest devices in the T-ring are the shift registers inserted between the other three components connected to the ring. These provide data storage for the ring by imposing a delay of 8 bits between their inputs and outputs. The register transfer level specification for these devices is shown below.



$\vdash \mathsf{Del}\ rin\ rout = \forall t.\, rout(t{+}8) = rin\ t$

Each of the three delay devices used in the T-ring is implemented by a shift register composed of eight flip flops driven by an inverted clock signal, as shown in Figure 6.8. A model '$\mathsf{Delay}\ ck\ rin\ rout$' that describes this 8-bit shift register is straightforward to define using the primitives defined above in Section 6.3.4. The model is constructed in the usual way, by applying the operations of composition ('$\wedge$') and hiding ('$\exists$') to the primitive components in the design, and the formal definition of the model need not be given here.

Once the model given by Delay has been defined, it is possible to show that the shift register shown in Figure 6.8 correctly implements the abstract specification for the 8-bit delay device:

$$\begin{array}{l} \vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \supset \\ \quad \mathsf{Delay}\ ck\ rin\ rout \supset \\ \quad \mathsf{Del}\ (rin\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (rout\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \end{array}$$

Here, the input and output values rin and $rout$ are sampled on falling edges of the clock ck. The correctness theorem states that these sampled values satisfy the temporally abstract specification for an 8-bit delay device. The function Fall, used to construct the time mapping in this theorem, is analogous to the function Rise. Its formal definition is shown below.

$$\vdash \mathsf{Fall}\ ck\ t = ck(t) \wedge \neg ck(t{+}1)$$

The proof of the correctness theorem for the delay device follows by induction on the length of the interval of concrete time between successive falling edges of the clock, in much the same way as unit delay correctness theorem discussed in Section 6.2 is proved by induction on the number of time steps between two rising edges.

Although the correctness theorem shown above relates the 8-bit shift register model to the abstract specification for an 8-bit delay, it is *not* an appropriate correctness statement for the delay devices used in the T-ring. The transmitter, the receiver, and the monitor sample their inputs from the ring on the *rising* edge of the clock. The register transfer level abstractions of these inputs will

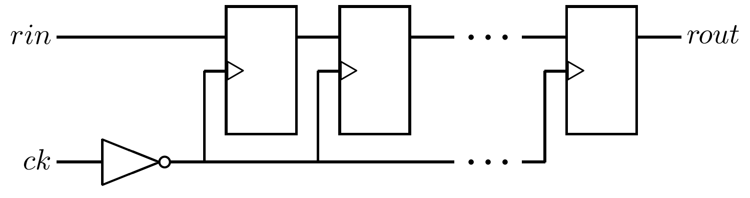


Figure 6.8: Design of the Delay Device.

therefore be sampled sequences of the form '$in\ \mathsf{when}\ (\mathsf{Rise}\ ck)$'. These inputs are driven by the outputs of the delay devices connected to them, and in the correctness theorem shown above the output of a delay device is a sampled signal of the form '$rout\ \mathsf{when}\ (\mathsf{Fall}\ ck)$'. But when the delay devices are connected to the other components of the T-ring, the abstractions must correspond. It is therefore necessary to sample the output of the delay device on the rising edges of the clock, rather than the falling edges.

The correctness theorem for the 8-bit delay devices used in the T-ring should therefore have the form shown below.

$$\begin{array}{l}\mathsf{Inf}(\mathsf{Rise}\ ck) \supset \\ \mathsf{Delay}\ ck\ rin\ rout \supset \\ \mathsf{Del}\ (rin\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (rout\ \mathsf{when}\ (\mathsf{Rise}\ ck))\end{array}$$

Here, the input rin is sampled on the falling edges of the clock, and the output $rout$ is sampled on the rising edges of the clock. The sampled delay device output $rout\ \mathsf{when}\ (\mathsf{Rise}\ ck)$ will therefore match the abstract inputs of the receiver, transmitter, and monitor that are driven by this output.

The implication shown above, however, does not hold. Because $rout$ is driven by a D-type device, it is possible to show (by induction) that it remains stable until just after each fall of the clock. The value of this output at the time of the nth *rise* of the clock is therefore the same as its value at the time of the next fall of the clock after the nth rise. But because the clock may start out either high or low at time '0', it is not known if this next fall of the clock will be the nth falling edge or the $(n{+}1)$th falling edge.[2] If the next fall is the nth, then the output of a delay device is the same when sampled on the either the rising or the falling edges of the clock:

$$rout(\mathsf{Timeof}\ (\mathsf{Rise}\ ck)\ n) = rout(\mathsf{Timeof}\ (\mathsf{Fall}\ ck)\ n)$$

In this case, the implication shown above holds. But if the next falling edge after the nth rise is the $(n{+}1)$th, then it follows that:

$$rout(\mathsf{Timeof}\ (\mathsf{Rise}\ ck)\ n) = rout(\mathsf{Timeof}\ (\mathsf{Fall}\ ck)\ (n{+}1))$$

in which case the implication proposed as a correctness statement for the delay device will not be true for all values of $rout$.

To get around this problem, it is sufficient to ensure that the output $rout$ is sampled on those falling edges of the clock which are *preceded by rising clock edges*. This can be done simply by 'choosing the origin' of time such that the clock starts out low at time 0. Imposing the condition that $\neg(ck\ 0)$ holds of the clock makes it possible to show that the nth falling edge occurs after the nth rising edge:

[2]This is a major inconvenience associated with modelling time by the natural numbers.

$$\vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \mathsf{Timeof}\ (\mathsf{Rise}\ ck)\ n < \mathsf{Timeof}\ (\mathsf{Fall}\ ck)\ n$$
$$\vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \mathsf{Timeof}\ (\mathsf{Fall}\ ck)\ n < \mathsf{Timeof}\ (\mathsf{Rise}\ ck)\ (n{+}1)$$

Because the last D-type flip flop in the delay device keeps the output *rout* stable between falling edges of the clock, it follows from these two theorems that:

$$\begin{array}{l} \vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \\ \quad \mathsf{Delay}\ ck\ rin\ rout \supset \\ \quad rout\ \mathsf{when}\ (\mathsf{Fall}\ ck) = rout\ \mathsf{when}\ (\mathsf{Rise}\ ck) \end{array}$$

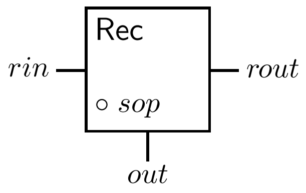
From this, a correctness theorem for the delay devices can be proved in which the time mapping on the output *rout* will match the abstraction used on the inputs of the other components to which this output will be connected:

$$\begin{array}{l} \vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \\ \quad \mathsf{Delay}\ ck\ rin\ rout \supset \\ \quad \mathsf{Del}\ (rin\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (rout\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \end{array} \tag{6.2}$$

This is the final form of correctness for the 8-bit shift registers in the T-ring.

6.3.5.2 The Receiver

At the register transfer level, the receiver looks like:



and the abstract specification for the receiver is given by:

$$\begin{array}{l} \vdash \mathsf{Rec}\ sop\ rin\ rout\ out = \\ \qquad \forall t.\, sop(t{+}1) \ = (rin(t{+}1) \wedge \neg sop\ t)\ \wedge \\ \qquad \forall t.\, rout(t{+}1) = rin(t{+}1)\ \wedge \\ \qquad \forall t.\, out(t{+}1) \ = (sop\ t \Rightarrow rin(t{+}1) \mid out\ t) \end{array}$$

This specification describes the receiver as a device with one boolean input *rin* and three boolean outputs *rout*, *out*, and *sop*. The value on each output at time $t{+}1$ is specified in terms of the input at time $t{+}1$ and the outputs at time t.

The variables *rin* and *rout* model the input from the ring and the output to the ring respectively. The receiver does not change the value on the ring, and there is no delay between the ring input *rin* and the ring output *rout*. The relationship between *rin* and *rout* at time 0 is unimportant to the correct operation of the T-ring, and is left unspecified by the equation for *rout*. The input-output relationship at time 0 *could* be specified here, but it is instead left unspecified in order to simplify the proof of correctness.

The variable *sop* represents an output that indicates whether the bit currently on the ring input *rin* is the start-of-packet bit. The equation for *sop* specifies that if the value currently being read from the ring is T and bit on the ring at the previous point of time was not the SOP bit, then the current input bit is the start-of-packet bit.

The variable *out* represents the data output of the receiver. The equation for *out* specifies that if the bit on the ring at the previous moment of time was the SOP bit, then the data output *out* becomes the bit currently being read from the ring. If the previous bit in the ring input was not the SOP bit, then value on the output *out* stays unchanged.

The circuit diagram for the receiver is shown in Figure 6.9. The device contains two D-type flip flops, which are used as registers to store the *sop* and *out* values between clock cycles. The *sop* flip flop (the D-type flip flop on the left in the diagram) is driven by the clock signal *ck*, and is therefore triggered by the rising edges of the clock. The output flip flip, however, is not driven by the clock signal *ck*, but by the inverted *sop* signal. This flop flop therefore samples the data present on the ring input *rin* whenever there is a falling edge of the *sop* signal. A formal model '$\mathsf{Receiver}\ sop\ ck\ out\ rin\ rout$' which describes the receiver circuit shown in Figure 6.9 is straightforward to define in the usual way. Its formal definition will therefore not be given here.

The correctness statement for the receiver involves the concept of the value on a wire being stable during the interval of time between two falling edges of the clock. To express this in logic, the predicate Stable is defined as follows.

$$\begin{array}{l} \vdash \mathsf{Stable}\ w\ ck = \\ \quad \forall n\, t.\, ((\mathsf{Timeof}\ n\ (\mathsf{Fall}\ ck)) < t \wedge t \leq (\mathsf{Timeof}\ (n{+}1)\ (\mathsf{Fall}\ ck))) \supset \\ \qquad w\ t = w(\mathsf{Timeof}\ (n{+}1)\ (\mathsf{Fall}\ ck))) \end{array}$$

This definition states that the value on a wire w is stable between each pair of adjacent falling edges of the clock ck. The outputs of D-type flip flops in the

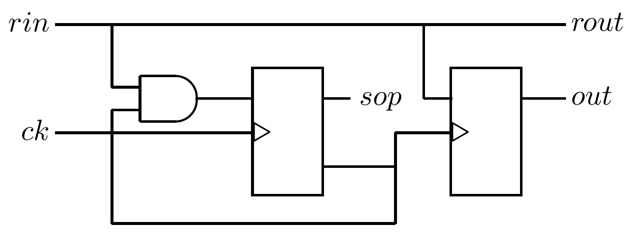


Figure 6.9: Design of the Receiver.

T-ring that are clocked on the falling edges of the clock (i.e. the D-type devices in the shift registers) are stable in the way stipulated by the Stable predicate defined above. The values on these outputs change just after a falling edge of the clock, and remain stable until the next falling edge.

Given the predicate Stable, the correctness statement for the receiver circuit shown in Figure 6.9 is written as follows:

$$\begin{array}{l} \vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \wedge \mathsf{Stable}\ rin\ ck \supset \\ \quad \mathsf{Receiver}\ sop\ ck\ out\ rin\ rout \supset \\ \quad \mathsf{Rec}\ (sop\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\ \qquad\quad (out\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\ \qquad\quad (rin\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \\ \qquad\quad (rout\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \end{array}$$

This theorem expresses a relationship of temporal abstraction between the design model for the receiver and its abstract specification. The outputs *sop*, *rout*, and *out* are sampled on the falling edges of the clock. The input *rin* is sampled on the rising edges of the clock, in agreement with the T-ring timing scheme explained in Section 6.3.3. The validity conditions $\mathsf{Inf}(\mathsf{Rise}\ ck)$ and $\neg(ck\ 0)$ are the same as those in the correctness theorem for the shift registers discussed in the previous section. The additional validity condition on the input *rin* expressed by the Stable predicate is needed allow the output *rout* that drives the ring to be sampled on the falling edges of the clock. In the receiver circuit, this output is wired directly to the input *rin*, and the input-output relationship stipulated by the abstract specification for the receiver is:

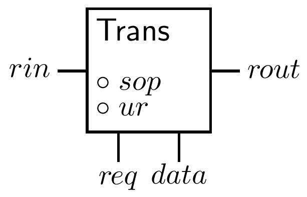
$$\forall t.\ (rout\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (t{+}1) = (rin\ \mathsf{when}\ (\mathsf{Rise}\ ck))\ (t{+}1)$$

The stability condition on the input *rin* ensures that this equation holds when *rin* is sampled on the rising edges of the clock and *rout* (which is directly connected to *rin*) is sampled on the falling edges of the clock. This stability assumption can be seen as a *timing condition* which must hold for the temporal abstraction expressed by the correctness statement shown above to be valid. The environment in which the receiver is placed is expected to satisfy this condition.

Full details of the formal proof of the correctness theorem shown above will not be given here. One of the main steps in the proof is again an induction on the length of a clock cycle between two rising edges of the clock. The fact that the two D-type devices in the receiver keep their outputs stable between rising edges is used to shift output values sampled on rising edges to output values sampled on falling edges, to obtain the abstraction relationship shown above. The proof is complicated by the somewhat tricky clocking arrangement whereby the right hand D-type flip flop shown in Figure 6.9 is triggered by the inverted *sop* signal. This makes it the impossible to consider each of the two D-type devices in the receiver separately during the proof—for example, by showing that each one implements an abstract device with unit delay, and then composing the two devices at the more abstract level.

6.3.5.3 The Transmitter

At the abstract level, the transmitter looks like:



The abstract specification for the transmitter is defined below:

$$
\begin{aligned}
&\vdash \mathsf{Trans}\ sop\ ur\ data\ req\ rin\ rout = \\
&\qquad \forall t.\, sop(t{+}1) \ = (rin(t{+}1) \wedge \neg sop\ t) \ \wedge \\
&\qquad \forall t.\, ur(t{+}1) \ \ \ = ((sop\ t \wedge ur\ t) \Rightarrow \mathsf{F} \mid (req\ t \Rightarrow \mathsf{T} \mid ur\ t)) \ \wedge \\
&\qquad \forall t.\, rout(t{+}1) = ((sop\ t \wedge ur\ t) \Rightarrow data(t{+}1) \mid rin(t{+}1))
\end{aligned}
$$

Like the receiver, the transmitter has an *sop* output that indicate the presence of the start-of-packet bit. The defining equation for this output is the same as that used in the abstract specification for the receiver.

The variable *ur* represents a state which is true when there is a *pending* user request to transmit data. The state equation for *ur* specifies that if there is a pending request at time t and the SOP bit arrives at time t, then *ur* will become false at time $t{+}1$, indicating that the request has been serviced by the incoming packet. Otherwise, the next state of *ur* depends on the value of the request input *req*. If there is a request to transmit data at time t, then *ur* becomes true at time $t{+}1$. If there is no request, then the value of *ur* remains the same.

The equation for the output *rout* specifies how data from the *data* input is inserted into data part of the packet on the ring. If there is a pending request to transmit data when the SOP bit arrives at some time t, then the value on the *data* input will be passed to the ring output *rout* at time $t{+}1$, replacing the data bit of the incoming packet. When there is no pending request to transmit data, or the packet has not yet arrived at the transmitter, the output value on *rout* is the same as the input value on *rin*.

The design of the transmitter is shown in Figure 6.10. The transmitter circuit is considerably more complex than the receiver circuit discussed in the previous section. The circuitry used to detect the start-of-packet bit is the same as in the receiver. But the output flip flop (the D-type on the right at the top) in the transmitter drives a multiplexer, which selects between the *data* and *rin* inputs. Furthermore, this output D-type is clocked by an inverted clock signal. The

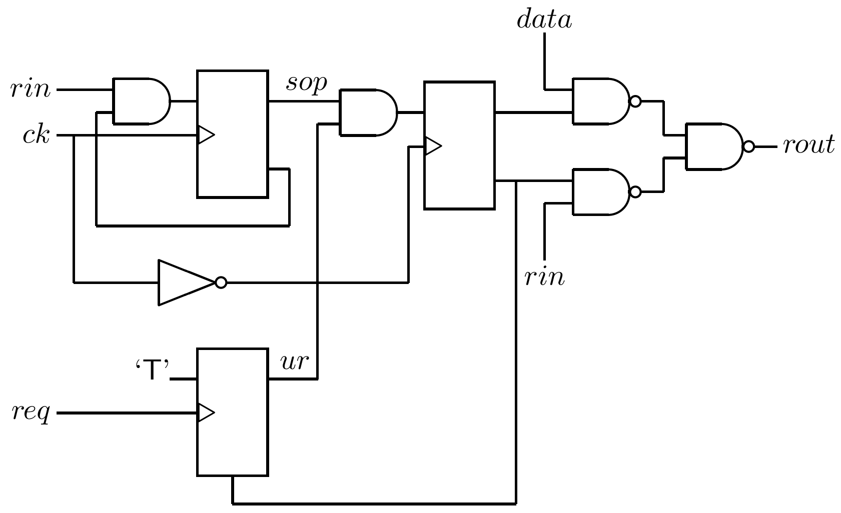


Figure 6.10: Design of the Transmitter.

request input *req* drives the *clock* input of the lower D-type flip flop, which is used to generate and store the pending request signal *ur*. The lower D-type is cleared on the falling edge of the clock, by a signal coming from the *qbar* output of the D-type on the right. This tricky style of TTL design caused considerable difficulties in the formal correctness proof for this device. The intricate temporal relationships between events in this circuit made the proof intricate as well, and—as was also the case in the receiver proof—the circuit could not easily be broken down into several parts that could be verified separately.

The most interesting aspect of the transmitter correctness proof, however, is that it requires a new kind of temporal abstraction. The request wire *req* drives the clock input of the lower D-type shown in Figure 6.10. This means that a request to transmit data in fact consists of a *rising edge* on the input *req*. This rising edge on the *req* input may occur asynchronously—at any time during a clock cycle, so the sampling operator when cannot be used to construct a register transfer level abstraction of the *req* input. A new abstraction operator, between, is therefore defined to deal with this problem.

The between operator has the following formal definition:

$$\vdash (s \ \mathsf{between}\ p)\ n = \exists t.\ (\mathsf{Timeof}\ n\ p) \leq t\ \wedge\ t < (\mathsf{Timeof}\ (n{+}1)\ p)\ \wedge\ s\ t$$

The predicate p in this definition identifies points of detailed time which are of interest at the abstract level—in the same way that the predicate p is used in the sampling construct 's when p'. As in temporal abstraction by sampling, this predicate defines the sequence of contiguous intervals of concrete time that correspond to successive units of abstract time. The variable s in this definition stands for a sequence of values *num*→*bool* at the concrete level of abstraction from which the abstract signal s between p is constructed. The definition states that the abstract signal s between p will be true at some abstract time n if the signal s is true at some intermediate point of detailed time between the nth and the (n+1)th time p is true. This 'synchronizes' the asynchronous signal s with the abstract time-scale.

For the request input *req* of the transmitter, an appropriate abstract signal constructed using the between operator is given by:

$$(\mathsf{Rise}\ req)\ \mathsf{between}\ (\mathsf{Fall}\ ck)$$

This *abstract* request signal is true at some abstract time n if there is a rising edge on the signal *req* at any time during the nth clock cycle—that is, at any time between the nth falling edge of the clock and the $(n+1)$th falling edge of the clock. Thus, a request to transmit data at the abstract level corresponds to the event of a rising edge on the physical *req* input at any intermediate point of time during an interval of concrete time between falling edges of the clock.

Using the new kind of temporal abstraction expressed by between, it is possible to state the correctness of the transmitter design as follows:

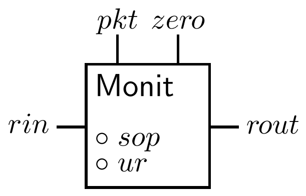
$$\begin{array}{l}
\vdash \mathsf{Inf}(\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \wedge \mathsf{Stable}\ rin\ ck \supset \\
\quad \mathsf{Transmitter}\ ck\ sop\ ur\ req\ data\ rin\ rout \supset \\
\quad \mathsf{Trans}\ (sop\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (ur\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (data\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad ((\mathsf{Rise}\ req)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (rin\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \\
\qquad\qquad (rout\ \mathsf{when}\ (\mathsf{Fall}\ ck))
\end{array}$$

The validity conditions are the same as those in the receiver correctness statement. In particular, the stability condition on the ring input *rin* is again needed, since the *rin* value is sometimes passed transparently through the output multiplexer to the output *rout*, which is itself sampled in the falling edges of the clock. The ring input *rin* is again sampled on the rising edges of the clock, and all the other signals except the request signal *req* are sampled on the falling edges of the clock. The abstract request signal is constructed using the between operator introduced above. The model of the transmitter design given by the function Transmitter is defined in the usual way.

The formal proof of this correctness theorem is (as was mentioned above) complicated by the tightly-linked relationships between events that occur in the circuit. As with all the correctness proofs of the T-ring components, induction on the duration of a clock cycle is a major component of the proof. But there is also much intricate reasoning about the timing of events by symbolic manipulation of the expressions that define the values of both internal and external wires of the design at rising and falling edges of the clock.

6.3.5.4 The Monitor

The monitor is very similar to the transmitter:



and has a similar, but slightly more complex, abstract specification:

$$
\begin{aligned}
&\vdash \mathsf{Monit}\ sop\ ur\ zero\ pkt\ rin\ rout = \\
&\quad \forall t.\, sop(t{+}1) \ = (rin(t{+}1) \land \neg sop\ t)\ \land \\
&\quad \forall t.\, ur(t{+}1) \ \ = ((\neg sop\ t \land ur\ t) \Rightarrow \mathsf{F} \mid (pkt\ t \Rightarrow \mathsf{T} \mid ur\ t))\ \land \\
&\quad \forall t.\, rout(t{+}1) = ((\neg sop\ t \land ur\ t) \Rightarrow \mathsf{T} \mid (\neg zero(t{+}1) \Rightarrow \mathsf{F} \mid rin(t{+}1)))
\end{aligned}
$$

The *sop* output for the monitor is defined in the same way as the *sop* outputs of the transmitter and receiver.

Like the transmitter, the monitor also has a *ur* output that indicates if there is a pending request from the user. In the monitor, however, this indicates a request to insert a start-of-packet bit into the ring. If a start-of-packet bit is *not* present on the ring input *rin* at time t, and there is a pending request to insert an SOP onto the ring, then a start-of-packet bit will be inserted at time $t{+}1$. The *ur* flag will therefore be cleared at time $t{+}1$. Otherwise, the next state of *ur* depends on the value on the *pkt* input. If the *pkt* is true at time t, then there is a request to create a packet on the ring, and *ur* becomes true at time $t{+}1$. If there is no request on *pkt*, the value of *ur* stays the same.

The equation for the output to the ring *rout* shows that a start-of-packet bit will be inserted into the bit pattern in the ring when the previous incoming bit was not an SOP bit and there is a pending request. If an SOP bit is not being inserted onto the ring, then the *zero* input can be activated to clear the ring. (The fact that a pending request on *ur* blocks the *zero* input explains the first step in the initialization sequence discussed on page 6.3.2.1.) When the *zero* is low, the ring output has the boolean value F. Otherwise, the output value *rout* will be the same as that of the ring input *rin*. The complexity of the monitor suggests that it was intended to allow the creation of a T-ring slot structure with several packets. It can be seen from this equation for *rout*, however, that the monitor cannot reliably create a new packet on the ring without the danger of overwriting the *data* bit of an already existing packet.

The monitor circuit design, shown in Figure 6.11, is a minor variation on the design of the transmitter. Again, the request input (in this case *pkt*) is used to trigger a D-type flip flop, and the pending request flag *ur*, which is generated by

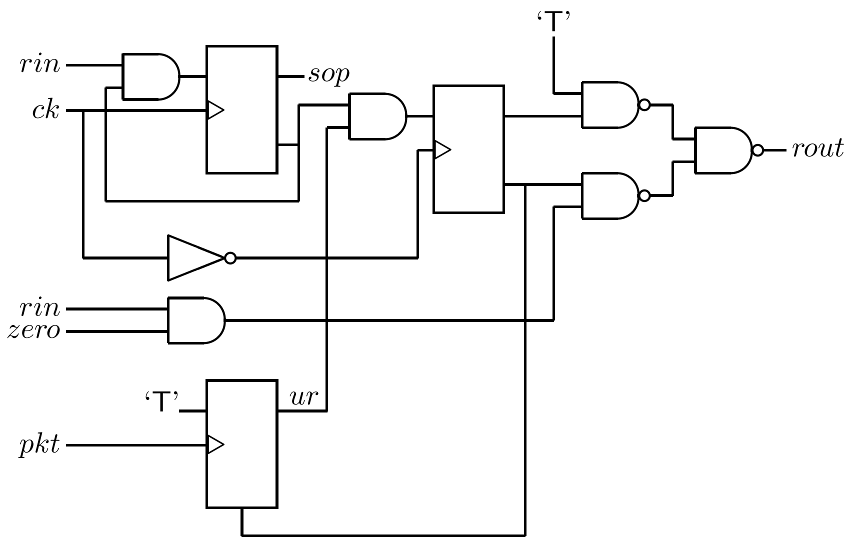


Figure 6.11: Design of the Monitor.

this flip flop, is cleared via its asynchronous clear input. The main differences are the addition of an AND-gate to implement the active-low clear input *zero*, and the presence of 'T' on the upper input of the multiplexer that drives the ring, to implement the insertion of an SOP bit into the bit-pattern on the ring.

The correctness theorem for the monitor also similar to the correctness statement for the transmitter:

$$
\begin{array}{l}
\vdash \mathsf{Inf}\ (\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \wedge \mathsf{Stable}\ rin\ ck \supset \\
\quad \mathsf{Monitor}\ ck\ sop\ ur\ pkt\ zero\ rin\ rout \supset \\
\quad \mathsf{Monit}\ (sop\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (ur\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (zero\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad ((\mathsf{Rise}\ pkt)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (rin\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \\
\qquad\qquad (rout\ \mathsf{when}\ (\mathsf{Fall}\ ck))
\end{array}
$$

Again, the input *rin* is sampled on the rising edges of the clock, the abstract request signal is constructed using the between operator, and all other values are sampled on the falling edges of the clock. The proof of this correctness theorem is similar to the proof of correctness for the transmitter.

6.3.6 Correctness of the Register Transfer Design

Having completed the correctness proofs for each of the four different kinds of components used the design of the T-ring—the delay devices, the receiver, the transmitter, and the monitor—the next step in the T-ring correctness proof is to the use the abstract specifications for these devices to verify the correctness of entire system with respect to the top-level specification described in Section 6.3.2.

Formal proofs of all the correctness results presented in the previous sections have been done in the HOL system. The correctness statement for the next level of the proof, however, has not been proved formally in the HOL system, but was proved manually (i.e. 'on paper'). The following sections give a very brief overview of the correctness result for the T-ring at this level.

6.3.6.1 The Model

The top-level correctness proof begins with a definition of a model of the entire T-ring network at the register transfer level of abstraction. This model describes the network shown in Figure 6.4, is defined formally in logic as follows:

$$\begin{array}{l} \vdash \mathsf{TringRt}\ \mathit{zero}\ \mathit{pkt}\ \mathit{data}\ \mathit{req}\ \mathit{out} = \\ \quad \exists \mathit{sop}_1\ \mathit{sop}_2\ \mathit{sop}_3\ \mathit{ur}_1\ \mathit{ur}_2\ l_1\ l_2\ l_3\ l_4\ l_5\ l_6. \\ \qquad \mathsf{Trans}\ \mathit{sop}_1\ \mathit{ur}_1\ \mathit{data}\ \mathit{req}\ l_6\ l_1 \wedge \mathsf{Del}\ l_1\ l_2 \wedge \\ \qquad \mathsf{Monit}\ \mathit{sop}_2\ \mathit{ur}_2\ \mathit{zero}\ \mathit{pkt}\ l_2\ l_3 \wedge \mathsf{Del}\ l_3\ l_4 \wedge \\ \qquad \mathsf{Rec}\ \mathit{sop}_3\ \mathit{out}\ l_4\ l_5 \wedge \mathsf{Del}\ l_5\ l_6 \end{array}$$

The model is constructed from the *abstract* specifications of the six components in the T-ring network, rather than from the models that describe the designs of these components. It is this concise model of the entire ring that is to be proved correct with respect to the top level specification. The correctness theorem based on this abstract model of the T-ring will later be combined with the correctness results for the each of the six components in the ring to obtain a correctness theorem for the completely 'concrete' T-ring design.

6.3.6.2 The Correctness Statement

The top-level correctness statement for the T-ring is simple:

$$\vdash \mathsf{TringRt}\ \mathit{zero}\ \mathit{pkt}\ \mathit{data}\ \mathit{req}\ \mathit{out} \supset \mathsf{Tring}\ \mathit{zero}\ \mathit{pkt}\ \mathit{data}\ \mathit{req}\ \mathit{out} \qquad (6.3)$$

This correctness theorem simply states that the T-ring specification defined in Section 6.3.2 is a behavioural abstraction of the model defined in Section 6.3.6.1.

The proof of this top-level correctness statement will not be given here. It involves showing that the circulating bit pattern created by setting up the slot structure using the initialization sequence explained in Section 6.3.2 can be used to store a packet in the ring for an indefinite period of time between data transmission requests. This is done essentially by 'simulating' the transmission of a packet through one complete cycle around the ring, and thereby obtaining an assertion about the values on the transmission wires which is invariant every 24 units of time. The data transmission part of the correctness statement then follows from this assertion, together with some simple facts about modulo arithmetic. The proof has been done in enough detail that it should be straightforward (but tedious) to conduct a fully formal proof using the HOL system.

6.3.7 Putting the Proof Together

In the preceding sections, the verification of the T-ring was structured into a two-level hierarchy of correctness results. In Section 6.3.5, correctness theorems for each of the T-ring components were discussed. And in Section 6.3.6, a correctness theorem was given for the entire T-ring network—based on the *abstract* specifications of its components. In order to complete the proof, it is necessary to put these results together to show that the T-ring circuit design at the *timing level*—i.e. a model of the entire T-ring constructed from the primitive specifications for gates and flip flops—is correct with respect to the *top level* specification. This must be done to justify using only the abstract specifications of the T-ring components when proving that the top-level specification is satisfied.

The complete T-ring design is described formally by the predicate TringIMP defined by the equation shown below:

$$
\begin{array}{l}
\vdash \mathsf{TringIMP}\ ck\ zero\ pkt\ data\ req\ out = \\
\quad \exists sop_1\ sop_2\ sop_3\ ur_1\ ur_2\ l_1\ l_2\ l_3\ l_4\ l_5\ l_6. \\
\quad\quad \mathsf{Transmitter}\ ck\ sop_1\ ur_1\ req\ data\ l_6\ l_1\ \wedge \\
\quad\quad \mathsf{Delay}\ ck\ l_1\ l_2\ \wedge \\
\quad\quad \mathsf{Monitor}\ ck\ sop_2\ ur_2\ pkt\ zero\ l_2\ l_3\ \wedge \\
\quad\quad \mathsf{Delay}\ ck\ l_3\ l_4\ \wedge \\
\quad\quad \mathsf{Receiver}\ sop_3\ ck\ out\ l_4\ l_5\ \wedge \\
\quad\quad \mathsf{Delay}\ ck\ l_5\ l_6
\end{array}
$$

In this definition, the timing level models for the transmitter, the monitor, the receiver, and the three delay devices are simply combined using composition (‘$\wedge$’) to obtain a model of the entire design. The data transmission wires $l_1, \ldots, l_6$ that connect these components together into a ring are treated as internal wires to the design, and are therefore hidden (‘$\exists$’). The three start-of-packet state variables sop_1, sop_2, and sop_3 are also hidden from the external environment, as are the two user request flags ur_1 and ur_2.

Using the correctness theorems shown in the previous sections, and the rules for putting together hierarchical proofs discussed in Chapter 4, it is straightforward to derive a correctness statement for this concrete design model of the entire T-ring system. The first step is to use the rule $\wedge$-MONO discussed in Chapter 4 to combine the correctness theorems for each of the six components used in the design of the T-ring together into one correctness theorem for the composition of these components. This yields the theorem shown below:

$$
\begin{array}{l}
\vdash \mathsf{Inf}\ (\mathsf{Rise}\ ck) \wedge \neg(ck\ 0)\ \supset \\
\quad \mathsf{Transmitter}\ ck\ sop_1\ ur_1\ req\ data\ l_6\ l_1\ \wedge\ \mathsf{Delay}\ ck\ l_1\ l_2\ \wedge \\
\quad \mathsf{Monitor}\ ck\ sop_2\ ur_2\ req\ zero\ l_2\ l_3\ \wedge\ \mathsf{Delay}\ ck\ l_3\ l_4\ \wedge \\
\quad \mathsf{Receiver}\ sop_3\ ck\ out\ l_4\ l_5\ \wedge\ \mathsf{Delay}\ ck\ l_5\ l_6 \supset \\
\quad (\mathsf{Stable}\ l_6\ ck \wedge \mathsf{Stable}\ l_2\ ck \wedge \mathsf{Stable}\ l_4\ ck) \supset \\
\quad \mathsf{Trans}\ (sop\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (ur\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (data\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad ((\mathsf{Rise}\ req)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (l_6\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \\
\qquad\qquad (l_1\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ \wedge \\
\quad \mathsf{Del}\ (l_1\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (l_2\ \mathsf{when}\ (\mathsf{Rise}\ ck))\ \wedge \\
\quad \mathsf{Monit}\ (sop\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (ur\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (zero\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad ((\mathsf{Rise}\ pkt)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad (l_2\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \\
\qquad\qquad (l_3\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ \wedge \\
\quad \mathsf{Del}\ (l_3\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (l_4\ \mathsf{when}\ (\mathsf{Rise}\ ck))\ \wedge \\
\quad \mathsf{Rec}\ (sop\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\quad (out\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\quad (l_4\ \mathsf{when}\ (\mathsf{Rise}\ ck)) \\
\qquad\quad (l_5\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ \wedge \\
\quad \mathsf{Del}\ (l_5\ \mathsf{when}\ (\mathsf{Fall}\ ck))\ (l_6\ \mathsf{when}\ (\mathsf{Rise}\ ck))
\end{array}
$$

The stability conditions on the wires l_6, l_2 and l_4 are in fact satisfied by the Delay devices which drive them. That is, it is possible to prove (again by induction on the time between rising edges of the clock) that:

$$\vdash \mathsf{Inf}\ (\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \mathsf{Delay}\ ck\ in\ out \supset \mathsf{Stable}\ out\ ck$$

This theorem allows the stability conditions on l_6, l_2 and l_4 in the large theorem shown above to be dropped, since these conditions are already ensured by the delay devices which drive these wires.

The next step is to existentially quantify the hidden wires in the both the antecedent and the consequent of the correctness theorem shown above, using the rule $\exists$-EXT. Abbreviating the resulting theorem using the definitions of TringRt and TringIMP yields:

$$
\begin{array}{l}
\vdash \mathsf{Inf}\ (\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \\
\quad \mathsf{TringIMP}\ ck\ zero\ pkt\ data\ req\ out \supset \\
\quad \mathsf{TringRt}\ (zero\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad\quad ((\mathsf{Rise}\ pkt)\ \mathsf{between}(\mathsf{Fall}\ ck)) \\
\qquad\qquad\quad (data\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad\quad ((\mathsf{Rise}\ req)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\
\qquad\qquad\quad (out\ \mathsf{when}\ (\mathsf{Fall}\ ck))
\end{array}
$$

Using the transitivity of implication (i.e. the rule sat-TRANS) this theorem and the top-level correctness theorem (6.3), give a correctness statement for the entire T-ring design:

$$\begin{array}{l} \vdash \mathsf{Inf}\ (\mathsf{Rise}\ ck) \wedge \neg(ck\ 0) \supset \\ \quad \mathsf{TringIMP}\ ck\ zero\ pkt\ data\ req\ out \supset \\ \quad \mathsf{Tring}\ (zero\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\ \qquad\quad ((\mathsf{Rise}\ pkt)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\ \qquad\quad (data\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \\ \qquad\quad ((\mathsf{Rise}\ req)\ \mathsf{between}\ (\mathsf{Fall}\ ck)) \\ \qquad\quad (out\ \mathsf{when}\ (\mathsf{Fall}\ ck)) \end{array}$$

This completes the T-ring verification.

6.4 Related Work

Temporal Abstraction in LCF_LSM

Gordon's LCF_LSM formalism [32] includes a special-purpose inference rule for temporal abstraction by sampling. Gordon defines an operator on state machines which merges state transitions, yielding a machine which runs at a 'coarser' grain of discrete time. The operator uses a predicate on the outputs of a state machine to mark which sequences of state transition are to be coalesced into single state transitions. Starting in a particular state, subsequent state transitions are merged until this predicate becomes true of the state machine's output values. There is an implicit assumption that all inputs remain stable during the merged cycles. The temporal abstraction mechanism provided by this operator was implemented in the LCF_LSM theorem prover and used in the formal verification of a simple microcoded computer [33]. The mechanized version of the inference rule for this operator requires a device to have a 'done' output to mark the end of a sequence of merged state transitions.

Projection in Interval Temporal Logic

Moszkowski [72], writing on future research directions for interval temporal logic, defines a *projection* operator for describing the behaviour of digital hardware over intervals of time consisting of only selected points of time. Predicates containing 'marker' variables are used to identify the states, or points of time, which are of interest. The role of these predicates is analogous to the role of the predicate p in the sampling construct '$s\ \mathsf{when}\ p$' defined in this chapter. In Tempura [71], the projection operator is given a slightly different definition, in which the predicates mentioned above describe a series of consecutive subintervals of time. This version of the ITL projection operator can executed by the Tempura interpreter. The dual of this executable projection operator (which is not itself executable) can be used to describe the behaviour of a device over subintervals of time corresponding to a series of clock cycles. The main emphasis of the discussion of temporal projection given by Moszkowski in [72,71] is the abstract *description* of device behaviour at selected points of time, rather than the formal verification of a detailed design model with respect to such a description.

Temporal projection in ITL and Tempura is also discussed by Hale in [39,40]. In [39], Hale shows how the T-ring network discussed in this chapter can be specified formally in ITL, and shows how the ITL specification of the T-ring can be executed in Tempura to simulate the operation of T-ring network. The formulation of a correctness statement in ITL for the receiver node of the T-ring is also discussed. The Tempura version of the temporal projection operator is used to state the correctness of the receiver with respect to a specification at a higher level of temporal abstraction. The receiver circuit design in this correctness statement differs from the receiver circuit verified in this chapter. In particular, Hale avoids the complexity introduced by the timing scheme discussed above in Section 6.3.3 by having values on the ring change on the rising edges of the clock, rather than on the falling edges of the clock. The correctness of the monitor and transmitter designs are not considered in [39].

Higher Order Logic: HOL

The work most closely related to the particular formulation of temporal abstraction by sampling developed in this chapter is the work on temporal abstraction reported by Herbert in [48]. Herbert defines a higher order predicate '$\mathsf{UP\_OF}\ ck\ n\ t$' which is equivalent to the following instance of the more general $\mathsf{Istimeof}$ predicate defined above in Section 6.1.2.1: '$\mathsf{Istimeof}(\mathsf{Rise}\ ck)\ n\ t$'. Herbert also defines a sampling function ABS by the equation:

$$\vdash \mathsf{ABS}\ select\ sig\ n = sig(\varepsilon\ \ \lambda t.\, select\ n\ t)$$

and uses this function to construct an abstract signal '$\mathsf{ABS}(\mathsf{UP\_OF}\ ck)\ sig$' by sampling the detailed signal sig on the rising edges of the clock ck. This is construction is equivalent to sampling the signal sig using the when operator defined in Section 6.1.3 as follows: '$sig\,\mathsf{when}\ (\mathsf{Rise}\ ck)$'.

The main difference between these two formulations is that all the constructs for temporal abstraction by sampling defined in this chapter (e.g. $s\,\mathsf{when}\,p$) are parameterized by a predicate p that identifies the points of time at which a signal is to be sampled, while the $\mathsf{UP\_OF}$ construct which forms the basis of Herbert's work is a specialized predicate for sampling on only the rising edges of a clock. Herbert mentions [48, page 128] that constructs analogous to $\mathsf{UP\_OF}$ can be defined for sampling on the falling edges of a clock, but considers only examples in which sampling in fact occurs on the rising edges of a clock. (In the examples presented by Herbert, every flip flop is triggered on the rising edges of a global clock.) In short: the two approaches are very similar, but the constructs for temporal abstraction defined in the present work (Timeof, when, etc.) are more general than those defined by Herbert in [48]. Furthermore, temporal abstraction relationship of the kind involving the between operator defined Section 6.3.5.3 are not considered in Herbert's work.

There are also differences of emphasis between the examples given by Herbert in [48] and the T-ring example presented in this chapter. The examples presented by Herbert involve a detailed analysis of the timing conditions that must hold for a temporal abstraction to be valid (e.g. conditions on clock periods, set-up and hold times, etc.). A formal approach to timing analysis is one of the main contributions of Herbert's work. By contrast, details about system timing were for the most part ignored in the T-ring example. The emphasis of the present work is on finding general and widely applicable constructs (e.g. 'when') for relating signals at different levels temporal abstraction, rather than on detailed timing analysis. The aim of the T-ring example was to show how these constructs can be used when an abstraction relationship involves more than just sampling on one kind of event (e.g. rising edges of the clock).

The when operator and the associated functions Istimeof and Timeof defined in this chapter have been adopted in the work on hardware verification using higher order logic discussed by Dhingra in [21] and by Joyce in (for example) [54,56].

Chapter 7

Abstraction between Models

In this chapter, an example is given to illustrate the idea of a relationship of abstraction between two models of hardware behaviour which was introduced in Section 4.2 of Chapter 4. The two models considered in this example are the threshold switching model of transistor behaviour defined in Chapter 5 and the simpler switch model of transistor behaviour defined in Chapter 2.

Both of these formal models of the behaviour of CMOS circuit designs are, of course, abstractions of the physical reality they represent, and both models are therefore bound to be inaccurate in some respects. But the switch model of transistor behaviour is also an abstraction of the threshold switching model, in the sense that both models describe the same set of primitive components—power, ground, N-type and P-type transistors—but the switch model presents a more abstract view of the behaviour of these components than the threshold switching model. The threshold switching model reflects the fact that real CMOS transistors do not pass both logic levels equally well. But in the more abstract (and therefore simpler) switch model, this aspect of transistor behaviour is ignored: transistors are modelled as if they were ideal boolean switches.

The switch model is therefore a less accurate formal model of CMOS behaviour than the threshold switching model. A circuit that can be proved correct using the switch model may in fact be incorrect according to the more accurate threshold switching model. For certain circuits, however, the two models are effectively equivalent. For *these* circuits, a proof of correctness in the switch model amounts to a proof of correctness in the threshold switching model; the switch model is an adequate basis for verification of these circuits, and the extra accuracy of the more complex threshold switching model is not needed.

In this chapter, the abstraction relationship between these two formal models of CMOS circuit behaviour is formalized in logic by means of semantic functions defined on a concrete recursive type *circ*. This recursive type is an instance of the general class of concrete types discussed in Chapter 5, and can be defined formally (and automatically) using the method for defining types explained in Appendix A. The type *circ* is used to formulate a theorem that describes the conditions under which correctness results obtained in the simple switch model of transistor behaviour are effectively equivalent to correctness results obtained in the more complex threshold model of transistor behaviour.

7.1 Representing the Structure of MOS Circuits

In this section, a specially-defined recursive type '*circ*' is introduced to provide an explicit representation in logic for the *structure* of the class of all CMOS circuit designs. The motivation for introducing this recursive type is that it makes it possible for assertions about the abstraction relationship between the two CMOS transistor models considered in this chapter to be stated and proved as *theorems* of higher order logic, rather than meta-theorems about provability in the logic. The advantage of this approach is that it allows the formal proofs of these theorems to be checked mechanically using the HOL theorem prover—all the theorems in this chapter have been proved completely formally using the HOL system.

Formally, a transistor model is just set of logical terms, each of which is intended to describe one of the primitive components from which CMOS circuits are built. The model of any particular circuit *design* is also a logical term, constructed by applying the syntactic operations of composition $\wedge$ and hiding $\exists$ to instances of these primitives. The syntactic structure of such a model mirrors the structure of CMOS circuit it describes: where the circuit contains two parts wired together, the model contains a subterm of the form '$tm_1 \wedge tm_2$'; and where the circuit contains an internal wire, the model contains a subterm of the form '$\exists w.\, tm$'.

The set of all CMOS design models is a set of logical terms—i.e. a subset of the formal language of terms in higher order logic. For example, the set of all design models constructed from the switch model primitives defined in Chapter 2 is the smallest set of logical terms T such that: T contains every instance of the primitive specifications '$\mathsf{Pwr}\, p$', '$\mathsf{Gnd}\, g$', '$\mathsf{Ntran}(g, s, d)$' and '$\mathsf{Ptran}(g, s, d)$', and T is closed under the syntactic operations of conjunction '$\wedge$' and existential quantification '$\exists$'. Similarly, the set of all design models in the threshold switching model of CMOS behaviour is the smallest set of logical terms that can be built up using $\wedge$ and $\exists$ from instances of the threshold switching primitives defined in Chapter 5.

The idea proposed in this chapter is to use a specially-defined concrete recursive type *circ* to embed the syntax of these subsets of higher order logic within the logic itself. This allows metalinguistic quantification over the set of all design models to be replaced by explicit quantification *within* the logic over values of type *circ*. The set of values denoted by *circ* provides formal representation in logic of the syntax of a language of *circuit terms*, whose expressions describe how circuits are built up from their constituent parts.[1] A circuit term is either a primitive expression that denotes a basic component (i.e. power, ground, an N-type or P-type transistor), or a composite expression that denotes a circuit built up by the operations of composition and hiding. The concrete recursive type *circ* defined below denotes the set of all such circuit terms.

[1]The type *circ* defined in this section is a formalization in higher order logic of the algebraic approach used by Cardelli [12] and Winskel [86] to model circuit designs.

7.1.1 The Type of Ascii Character Strings

With the direct approach to modelling hardware behaviour in logic, where the model of a hardware design is just a boolean-valued logical term, the external wires of a circuit design are simply represented by free variables in the model of its behaviour. The first step in the formal definition of the type *circ*, which embeds the syntax of design models as a set of values within the logic, is to define a type of ascii character strings to represent the names of the wires in a CMOS device. This type is a straightforward instance of the class of concrete recursive types discussed in Chapter 5, and can be defined as follows.

Using the notation for describing concrete types introduced in Section 5.1.2 of Chapter 5, a logical type *ascii* which denotes the set of all 7-bit ascii character codes can be defined (informally) by the equation shown below.

$$ascii \quad ::= \quad \mathsf{Ascii}\ bool\ \ bool\ \ bool\ \ bool\ \ bool\ \ bool\ \ bool$$

The type *ascii* described by this equation provides a formal representation in logic for the set of all 7-bit ascii character codes. Each character is represented by a value of type *ascii* obtained by applying the function Ascii to the seven boolean values in its ascii character code. The letter 'a', for example, has the 7-bit ascii code '1100001' and is represented formally by the term 'Ascii T T F F F F T'. The 7-bit ascii code for any other ascii character can likewise be represented in logic by a value of type *ascii* constructed using the function Ascii.

Given this representation of 7-bit ascii character codes, a logical type of ascii character *strings* can be defined informally by the equation shown below.

$$str \quad ::= \quad \mathsf{Empty} \quad | \quad \mathsf{String}\ \ ascii\ \ str$$

The concrete recursive type *str* described by this equation is similar to the type of finite lists discussed in Chapter 5. Every value of type *str* consists of a finite sequence of ascii character codes constructed using the function String from the empty string represented by the constant Empty. For example, the character string 'ab' is represented by the term:

$$\mathsf{String}\ (\mathsf{Ascii\ T\ T\ F\ F\ F\ F\ T})\ \ (\mathsf{String}\ (\mathsf{Ascii\ T\ T\ F\ F\ F\ T\ F})\ \ \mathsf{Empty})$$

Any finite-length string of ascii characters can be represented formally in logic by a value of type *str* constructed using the functions String, Ascii, and the constant Empty in a similar way.

To provide a concise notation for writing terms that represent ascii character strings, the following notation for *string constants* is introduced. A string constant is a constant of type *str* written between single quotes as follows: $\mathtt{'}\mathsf{c}_1 \ldots \mathsf{c}_n\mathtt{'}$. Such a constant should be regarded as an object language abbreviation for the value of type *str* that represents the ascii character string '$c_1 \ldots c_n$'. String constants written in this notation are simply defined constants introduced to abbreviate logical terms of type *str* constructed using the boolean constants T and F, and the constructors String, Ascii, and Empty. For example, the string constant 'ab' is defined by:

$$\vdash \text{'ab'} = \mathsf{String}\ (\mathsf{Ascii}\ \mathsf{T}\ \mathsf{T}\ \mathsf{F}\ \mathsf{F}\ \mathsf{F}\ \mathsf{F}\ \mathsf{T})\ \ (\mathsf{String}\ (\mathsf{Ascii}\ \mathsf{T}\ \mathsf{T}\ \mathsf{F}\ \mathsf{F}\ \mathsf{F}\ \mathsf{T}\ \mathsf{F})\ \ \mathsf{Empty})$$

and abbreviates the term of type *str* which represents the string 'ab'.

The theorems of higher order logic which characterize the defined types *ascii* and *str* have the form discussed in Section 5.1.2 of Chapter 5:

$$\vdash \forall f.\, \exists! fn.\, \forall b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1.\, fn(\mathsf{Ascii}\ b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1) = f\ b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1$$

$$\vdash \forall e\, f.\, \exists! fn.\, (fn\ \mathsf{Empty} = e) \wedge (\forall c\, s.\, fn(\mathsf{String}\ c\ s) = f\ \ (fn\ s)\ c\ s)$$

These theorems can be derived from appropriate formal definitions of the type constants *ascii* and *tri* using the method explained in Appendix A. (In the HOL implementation of this method, these theorems are proved automatically.) The standard properties discussed in Section 5.1.2.2 of Chapter 5 follow from these abstract characterizations of the types *ascii* and *str*. In particular, it follows from the two theorems shown above that the functions Ascii and String are one-to-one, and that every value of type *str* can be obtained using these functions and the constant Empty. Two values of type *str* are therefore equal exactly when they represent the same finite sequence of ascii characters. This property was used in the formal proofs of the theorems about particular CMOS circuit designs given in subsequent sections of this chapter.

7.1.2 The Type of Circuit Terms

Given the type *str* defined above, the syntax of circuit terms can be represented formally in logic by the recursive type *circ* defined (using the informal notation introduced in Chapter 5) by the following equation:

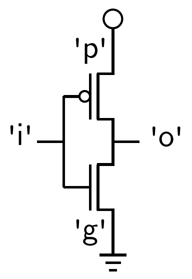
$$\begin{array}{lll} circ & ::= & \mathsf{Pwr}\ str \mid \mathsf{Gnd}\ str \mid \ \mathsf{Ntran}\ str\ str\ str \mid \\ & & \mathsf{Ptran}\ str\ str\ str \mid \ \mathsf{Join}\ circ\ circ \mid \mathsf{Hide}\ str\ circ \end{array}$$

This equation defines a recursive type *circ* with six constructors, corresponding to the six different syntactic constructs in the abstract syntax of the language it represents. The first four constructors represent the primitive CMOS devices power, ground, N-type transistors, and P-type transistors. These four constructors are simply functions that map wire names (modelled by strings) to values of type *circ*. The constructor Pwr:$str{\to}circ$, for example, maps a value s of type *str* to a circuit term 'Pwr s' which represents a power node. Similarly, the constructor for N-type transistors, Ntran:*str*→*str*→*str*→*circ*, maps three strings to a value of type *circ* that represents an N-type transistor. For example, a circuit term of the form 'Ntran $g\ s\ d$' stands for an N-type transistor with gate labelled by the string g, source labelled by the string s, and drain labelled by the string d.

The other two constructors, Join and Hide, represent composition and hiding operations which are used to construct circuit terms that model composite CMOS circuit designs. These two constructors are recursive functions that map values of type *circ* to values of type *circ*. The function Join:*circ*→*circ*→*circ* represents the composition operation on circuit terms. If c_1 and c_2 are two values of type *circ*, then the circuit term 'Join $c_1\ c_2$' represents the composition of the two circuits represented by c_1 and c_2. The function Hide:*str*→*circ*→*circ* represents the hiding operation on circuit terms. If c is a circuit term and s is a string, then the circuit term 'Hide $s\ c$' represents the circuit obtained by hiding the wire labelled s in the circuit represented by c.

The recursive type *circ* provides an explicit representation in logic for the structure of the class of all CMOS circuit designs. A circuit term that models the structure of any particular circuit can built up using the six constructors discussed above. For example, a circuit term Inv which models the structure of the CMOS inverter considered in Chapter 3 can be defined formally in logic as shown in Figure 7.1. The circuit term shown in this diagram is structurally isomorphic to the *logical terms* used to model the CMOS inverter design in Chapters 3 and 5. The input and output wires of the inverter circuit are modelled by the strings 'i' and 'o', and the two internal wires are modelled by the strings 'p' and 'g'. The circuit term denoted by Inv is built up from the circuit terms for the primitive components in the inverter design using the constructors Join and Hide. The circuit term for any other CMOS design can also be constructed (in the obvious way) using the functions Pwr, Gnd, Ntran, Ptran, Join, and Hide.

As was discussed in Section 5.1.2 of Chapter 5, the logical type *circ* described informally by the equation shown above can be defined formally in higher order logic (and automatically in HOL) using the method for defining concrete types



```
⊢ Inv = Hide 'p' (Hide 'g'
              (Join (Join (Pwr 'p') (Ntran 'i' 'g' 'o'))
                    (Join (Gnd 'g') (Ptran 'i' 'p' 'o'))))
```

Figure 7.1: The Circuit Term for a CMOS Inverter.

explained in Appendix A. The abstract characterization for the concrete recursive type *circ* derived by this method is a primitive recursion theorem of the kind discussed in Chapter 5:

$$\begin{array}{l} \vdash \forall f_1\, f_2\, f_3\, f_4\, f_5\, f_6.\, \exists! fn. \\ \quad \forall p.\, fn(\mathsf{Pwr}\; p) = f_1\; p\; \wedge \\ \quad \forall g.\, fn(\mathsf{Gnd}\; g) = f_2\; g\;\; \wedge \\ \quad \forall g\, s\, d.\, fn(\mathsf{Ntran}\; g\; s\; d) = f_3\; g\; s\; d\; \wedge \\ \quad \forall g\, s\, d.\, fn(\mathsf{Ptran}\; g\; s\; d) = f_4\; g\; s\; d\;\; \wedge \\ \quad \forall c_1\, c_2.\, fn(\mathsf{Join}\; c_1\; c_2) = f_5\; (fn\; c_1)\; (fn\; c_2)\; c_1\; c_2\;\; \wedge \\ \quad \forall s\; c.\, fn(\mathsf{Hide}\; s\; c) = f_6\; (fn\; c)\; s\; c \end{array} \tag{7.1}$$

This theorem asserts the unique existence any primitive recursive function defined by cases on the six constructors of the concrete recursive type *circ*. As was discussed in Chapter 5, this property can be used to justify the introduction of function constants that denote primitive recursive functions on *circ*. The principle of structural induction on *circ* also follows formally from this theorem. These properties are used in the formal proofs of the theorems discussed in the sections that follow.

7.2 Defining the Semantics of CMOS Circuits

The recursive type *circ* defined in the previous section denotes a set of values whose structure mirrors the way in which CMOS circuits are built up from their primitive components. This provides an embedded language of circuit terms in higher order logic for modelling the purely structural aspect of the class of all CMOS circuit designs. The following sections show how the behaviour of this class of circuit designs can also be modelled in logic, by defining a formal semantics for this language. The semantics of circuit terms will be defined in two different ways. One of these corresponds to the switch model of transistor behaviour defined in Chapter 3, and the other corresponds to the threshold switching model of transistor behaviour defined in Chapter 5.

For both transistor models, the semantics of circuit terms will be based on the idea of an *environment*. An environment is a function $e{:}str{\rightarrow}ty$ that maps wire names (modelled by strings) to values. Such a function assigns a value 'es' to every external wire s of a device, and thus describes a possible pattern of communication with the 'environment' in which a device operates. In the switch model, values on the wires of a device are represented by booleans. An environment in this model is therefore a function $e{:}str{\rightarrow}bool$ which assigns a value '$e\,s$' of type *bool* to every wire name s. This associates a boolean logic level with every external wire of a CMOS device. In threshold switching model, the values present on the wires of a device are modelled by the three-valued type *tri* introduced in Section 5.2.2 of

Chapter 5. In this model, an environment is a function of type $str{\rightarrow}tri$, which assigns a value of type tri to each external wire of a device.

Using this idea of an environment, a denotational semantics can be given to circuit terms by defining by a 'meaning' function M that maps circuit terms to predicates on environments. The precise definition of this function will depend on the model of transistor behaviour which is used, but the basic idea is to define a function M such that for every circuit term c, the application '$\mathsf{M}\ c$' denotes a predicate which is satisfied by only those environments that represent allowable configurations of values on the wires of the circuit represented by c. For any environment e, the expression $\mathsf{M}\ c\ e$ will then be true exactly when e represents a configuration of externally observable values that could occur on the wires of the CMOS circuit represented by the circuit term c.

7.2.1 The Switch Model Semantics

The semantic function Sm for the switch model is defined by primitive recursion on circuit terms. This function has logical type $circ{\rightarrow}((str{\rightarrow}bool){\rightarrow}bool)$. When applied to a circuit term c, it yields a predicate $\mathsf{Sm}\ c$ on environments of type $str{\rightarrow}bool$. The primitive recursive definition of Sm is:

$$\begin{array}{lcl}
\vdash \mathsf{Sm}\ (\mathsf{Pwr}\ p)\ e & = & (e\ p = \mathsf{T}) \\
\vdash \mathsf{Sm}\ (\mathsf{Gnd}\ g)\ e & = & (e\ g = \mathsf{F}) \\
\vdash \mathsf{Sm}\ (\mathsf{Ntran}\ g\ s\ d)\ e & = & e\ g \supset (e\ d = e\ s) \\
\vdash \mathsf{Sm}\ (\mathsf{Ptran}\ g\ s\ d)\ e & = & \neg(e\ g) \supset (e\ d = e\ s) \\
\vdash \mathsf{Sm}\ (\mathsf{Join}\ c_1\ c_2)\ e & = & \mathsf{Sm}\ c_1\ e \wedge \mathsf{Sm}\ c_2\ e \\
\vdash \mathsf{Sm}\ (\mathsf{Hide}\ s\ c)\ e & = & \exists b.\, \mathsf{Sm}\ c\ \lambda st.\, (st{=}s \Rightarrow b \mid e\ st)
\end{array}$$

The validity of this primitive recursive definition is justified formally by the characterization of the defined type $circ$ given by theorem (7.1) on page 145.

The first four equations in this recursive definition of the semantic function Sm define the semantics of primitive devices: power, ground, N-type transistors, and P-type transistors. Each equation states what must be true of an environment in which the corresponding component is operating. The equation for Ntran, for example, states what must be true in any environment in which an N-type transistor with gate g, source s, and drain d is placed. This equation imposes the constraint that any environment e which assigns the value T to g must also assign equal values to d and s. The three other equations define the semantics of power, ground, and P-type transistors in a similar way.

The last two equations shown above define the semantics of composition and hiding. The semantic equation for the constructor Join states that an environment e is a possible assignment of values to the wires in a composition of two circuits exactly when it is a possible assignment of values to the wires of *both* subcircuits. The equation for Hide uses existential quantification to isolate the hidden wire from the environment. It states that e is a possible environment for the circuit represented by '$\mathsf{Hide}\ s\ c$' exactly when there exists *some* environment which is allowed by the semantics of c and which differs from e only in the boolean value it assigns to the string s.

The theorems shown above define the value of the semantic function Sm for any circuit term c and environment e, and thereby constitute a semantics for the class of all CMOS circuit design. The switch model semantics of any particular circuit design can derived formally using these recursive defining equations for the function Sm. For example, one can prove the following theorem about the semantics of the circuit term Inv defined above in Figure 7.1:

$$\vdash \mathsf{Sm}\ \mathsf{Inv}\ e = (e\ \texttt{'o'} = \neg(e\ \texttt{'i'}))$$

This theorem is exactly analogous to the inverter correctness theorem proved in Section 3.4.4 of Chapter 3. It can be proved by first computing the switch model semantics of Inv using the defining equations for Sm, and then deriving the correctness result by a sequence of steps which are very similar to those used in the proof given in Chapter 3. The theorem states that in the switch model of CMOS behaviour given by the meaning function Sm, the behaviour of the circuit design represented by the circuit term Inv is indeed that of an inverter—in every environment e, the boolean value on the output $\texttt{'o'}$ is the negation of the boolean value on the input $\texttt{'i'}$.

7.2.2 The Threshold Model Semantics

In the threshold switching model, signals on circuit nodes are modelled by values of type *tri*. The semantics of *circ* for the threshold switching model will therefore be given by a function Tm from *circ* to predicates on environments of type $str{\rightarrow}tri$. The function Tm is defined by primitive recursion as follows:

$$\begin{array}{lcl}
\vdash \mathsf{Tm}\ (\mathsf{Pwr}\ p)\ e & = & (e\ p = \mathsf{Hi}) \\
\vdash \mathsf{Tm}\ (\mathsf{Gnd}\ g)\ e & = & (e\ g = \mathsf{Lo}) \\
\vdash \mathsf{Tm}\ (\mathsf{Ntran}\ g\ s\ d)\ e & = & (e\ g = \mathsf{Hi}) \supset ((e\ d = \mathsf{Lo}) = (e\ s = \mathsf{Lo})) \\
\vdash \mathsf{Tm}\ (\mathsf{Ptran}\ g\ s\ d)\ e & = & (e\ g = \mathsf{Lo}) \supset ((e\ d = \mathsf{Hi}) = (e\ s = \mathsf{Hi})) \\
\vdash \mathsf{Tm}\ (\mathsf{Join}\ c_1\ c_2)\ e & = & \mathsf{Tm}\ c_1\ e \wedge \mathsf{Tm}\ c_2\ e \\
\vdash \mathsf{Tm}\ (\mathsf{Hide}\ s\ c)\ e & = & \exists v.\, \mathsf{Tm}\ c\ \lambda st.\, (st{=}s \Rightarrow v \mid e\ st)
\end{array}$$

This definition is similar to the recursive definition of the semantic function for the switch model semantic of circuit terms. The difference is that the function Tm defined here is defined for environments of type $str{\rightarrow}tri$, and the threshold switching model of CMOS behaviour is used in the defining equations for the primitive devices Pwr, Gnd, Ntran, and Ptran. The semantics of composition and hiding are the same as in the switch model.

Like the defining equations for the semantic function Sm, the equations shown above for the semantic function Tm can be used to derive assertions about the behaviour of any CMOS circuit design. For example, one can prove from the defining equations for Tm that the inverter circuit represented by the circuit term Inv has the threshold model semantics given by the following theorem:

$$\vdash \mathsf{Tm}\ \mathsf{Inv}\ e = ((e\ \texttt{'i'} = \mathsf{Hi}) \supset (e\ \texttt{'o'} = \mathsf{Lo})) \wedge ((e\ \texttt{'i'} = \mathsf{Lo}) \supset (e\ \texttt{'o'} = \mathsf{Hi}))$$

Here, the model of CMOS circuit behaviour used is the threshold switching model discussed in Chapter 5. The theorem shows that in any environment e in which the input $\texttt{'i'}$ has either the value Hi or the value Lo, the output $\texttt{'o'}$ must have the value Lo or the value Hi, respectively.

7.3 Defining Satisfaction

Given the two semantic functions defined in the preceding sections, it is possible to formulate an assertion in the logic that describes the conditions under which a correctness result obtained in the switch model of CMOS designs amount to a correctness correctness result in the threshold switching model of CMOS designs. The first step in formulating this assertion is to define what it means for a circuit design to satisfy a specification of required behaviour in a given transistor model.

In the following definition of satisfaction, c is a circuit term representing a CMOS circuit design, M stands for a semantic function on circuit terms, S is an abstract specification of required behaviour, a is a data abstraction function, and C is a validity condition on the abstraction relationship between the semantics of the circuit design given by '$M\ c$' and the abstract specification S:

$$\vdash \mathsf{Sat}\ M\ c\ C\ a\ S = \forall e{:}str\to\alpha.\ C\ e \supset (M\ c\ e \supset S(a \circ e)) \tag{7.2}$$

This definition formalizes correctness as a relationship of data abstraction between the design model for the CMOS circuit represented by the circuit term c and the abstract specification of required behaviour S. Expressed in the notation used in Chapter 4, an abstraction relationship of this kind would be written:

$$\vdash C[c_1, \ldots, c_n] \supset M[c_1, \ldots, c_n] \underset{F}{\mathsf{sat}} S[a_1, \ldots, a_n] \qquad \text{where}\ \ F = a$$

The type variable α in definition (7.2) stands for the type used to model the values on the wires of a circuit. This type would be *bool* in the switch model semantics given by Sm and *tri* in the threshold switching semantics given by Tm. The variable S stands for an abstract specification of intended behaviour, and ranges over predicates on environments of type $str\to\beta$. The variable $a{:}\alpha\to\beta$ is a data abstraction function that maps values of the concrete type α used in the model of circuit behaviour to values of the abstract type β. The variable C is a predicate on environments of type $str{\rightarrow}\alpha$, and represents a validity condition on the correctness relationship expressed by definition (7.2).

The definition of Sat states that in a transistor model M, the CMOS circuit c is correct with respect to an abstract specification S if for every environment e that is allowed by the semantics $M\ c$ and satisfies the validity condition C, the abstract environment $a \circ e$ satisfies the abstract specification S. Composition on the left with the data abstraction function translates an environment e of type $str{\rightarrow}\alpha$, which assigns a concrete value $e\ s$ of type α to each string s, into a corresponding environment $a \circ e$ of type $str{\rightarrow}\beta$, which assigns the corresponding abstract value $a(e\ s)$ to each string.

7.4 Correctness in the two Models

In the switch model, a specification of required behaviour is simply predicate on environments of type $str{\rightarrow}bool$. A formal specification Not for the inverter circuit discussed above, for example, can be defined by the equation shown below.

$$\vdash \mathsf{Not}\ e = (e\ \texttt{'o'} = \neg(e\ \texttt{'i'}))$$

Here, the function Not is simply a predicate on switch model environments which describes what must hold of the environment e of correctly functioning inverter with input 'i' and output 'o'. Using the satisfaction relation Sat defined in the previous section, a switch model correctness theorem for the inverter circuit represented the circuit term Inv defined on page 144 can be written:

$$\vdash \mathsf{Sat\ Sm\ Inv\ true\ id\ Not} \qquad \text{where} \ \vdash \mathsf{true}\ e = \mathsf{T} \ \text{ and } \ \vdash \mathsf{id}\ b = b$$

Here, the validity condition true is simply the condition which is true of every switch model environment e, and the data abstraction function id is simply the identity function on *bool*. The correctness theorem shown above simply states that the inverter circuit Inv is correct in the switch model Sm with respect to the specification Not. The data abstraction function and validity condition are trivial, and the correctness theorem is in fact equivalent to the assertion:

$$\vdash \forall e.\, \mathsf{Sm\ Inv}\ e \supset (e\ \texttt{'o'} = \neg(e\ \texttt{'i'}))$$

In the analysing the formal relationship between the switch model defined by Sm and the threshold switching model defined by Tm, only switch model correctness results of the kind illustrated by this example will be considered, where correctness is stated using the identity data abstraction function id and the trivially-satisfied validity condition true.

In the threshold model, a specification of required behaviour is predicate on environments of type $str{\rightarrow}tri$. To formulate the relationship between the two transistor models, correctness results in the switch model must be related to equivalent correctness results in the threshold switching model. This can be done by using the threshold model validity condition def and the data abstraction function abs defined formally by the equations shown below.

$$\vdash \mathsf{def}\ e = \forall s.\ \neg(e\ s = \mathsf{X})$$
$$\vdash (\mathsf{abs}\ \mathsf{Hi} = \mathsf{T}) \wedge (\mathsf{abs}\ \mathsf{Lo} = \mathsf{F})$$

The validity condition def states that a threshold switching model environment $e{:}str{\rightarrow}tri$ assigns only the strongly-driven value Hi or the strongly-driven value Lo to every string s. That is, no wire s in the environment e has the degenerate logic level modelled by the third value 'X' of type tri. The function $\mathsf{abs}{:}tri{\rightarrow}bool$ is the data abstraction function defined formally in Chapter 5.

Using these two constants, any switch model correctness result of the kind discussed above can be reformulated as a correctness result with respect to an *abstract* specification in the threshold switching model. Consider, for example, the switch model correctness result for the inverter circuit discussed above. The correctness of the inverter circuit represented by Inv with respect to the abstract specification Not can be expressed in the threshold model Tm by the theorem shown below.

$$\vdash \mathsf{Sat}\ \mathsf{Tm}\ \mathsf{Inv}\ \mathsf{def}\ \mathsf{abs}\ \mathsf{Not}$$

Expanding this theorem with the definitions of the satisfaction relation Sat, the validity condition def, and the specification of required behaviour Not gives the following equivalent threshold model correctness theorem:

$$\vdash \forall e.\ (\forall s.\neg(e\ s = \mathsf{X})) \supset (\mathsf{Tm}\ \mathsf{Inv}\ e \supset (\mathsf{abs}(e\ \texttt{'o'}) = \neg(\mathsf{abs}(e\ \texttt{'i'}))))$$

This theorem states that in a well-behaved environment e, where no external wire has the degenerate value X, the inverter circuit modelled by Inv is correct with respect to the abstract specification of intended behaviour for an inverter.

In general, any switch model correctness assertion '$\mathsf{Sat}\ \mathsf{Sm}\ c\ \mathsf{true}\ \mathsf{id}\ S$', where $c{:}circ$ is a circuit term and $S{:}(str{\rightarrow}bool){\rightarrow}bool$ is a switch model specification of required behaviour, can be expressed in the threshold model by a correctness assertion of the form '$\mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S$', where S is treated as an abstract specification of required behaviour. This translation of switch model correctness statements into equivalent threshold model correctness statements is the basis for the formulation of the abstraction relationship between the two models discussed in the sections that follow.

7.5 Relating the Models

Using the formalization of satisfaction and correctness introduced above, it is possible to formulate assertions about the abstraction relationship between the two CMOS transistor models defined by the semantic functions Sm and Tm. In particular, it is possible to express the idea that for some condition on circuit terms Wb, the two models of transistor behaviour agree on the correctness results that can be proved about circuits that satisfy this condition:

$$\vdash \forall c.\, \mathsf{Wb}\ c \supset \forall S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ \mathsf{true}\ \mathsf{id}\ S = \mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S \qquad (7.3)$$

Informally, this theorem states if circuit term c satisfies the condition Wb, then for any specification S, the CMOS circuit design represented by c is correct with respect to S in the switch model exactly when it is correct with respect to S in the threshold model. The simple switch model is therefore an adequate basis for correctness proofs of circuits which satisfy the condition Wb. For these circuits, there is no point in using the more complex threshold switching model, since the two models agree on the abstract specifications that these circuits satisfy.

Theorem (7.3) is an explicit and rigorous formulation in higher order logic of the notion that the switch model of CMOS behaviour introduced in Chapter 2 is an abstraction of the threshold switching model of CMOS behaviour introduced in Chapter 5. That the switch model is an abstraction (i.e. a simplification) of the threshold model is expressed formally by the fact that the theorem shown above asserts only that some of the correctness results provable in the detailed threshold model—namely correctness theorems based on the data abstraction function abs and qualified by the validity condition def—are also provable in the more abstract switch model. The predicate Wb expresses a validity condition on this abstraction relationship between the two models. For circuits that satisfy this validity condition, any proposition about correctness in the switch model is equivalent to the corresponding proposition about correctness in the threshold model. For the class of circuits that satisfy the condition Wb, the switch model is a *valid* abstraction of the more detailed threshold model, in the sense that it cannot be used to prove a correctness result that does not also hold in the more accurate threshold model.

The formal definition of the validity condition Wb in theorem (7.3) and an overview of the formal proof of this theorem are given in the two sections that follow. The proof is done in two parts. It is first shown in Section 7.5.1 that for any CMOS circuit design, a correctness result in the threshold model implies a correctness result in the simpler switch model. The conditional equivalence given by theorem (7.3) is then derived in Section 7.5.2 by defining the condition Wb and proving that the converse implication holds for circuits that satisfy this condition.

7.5.1 Correctness in Tm implies Correctness in Sm

Theorem (7.3) states that propositions about correctness in the two models of CMOS behaviour are equivalent for the class of circuit designs that satisfy the condition on circuit terms Wb. In fact, for *any* circuit term c, a correctness result in the threshold model of CMOS behaviour implies a correctness result in the simpler switch model of CMOS behaviour:

$$\vdash \forall c\, S.\ \mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S \supset \mathsf{Sat}\ \mathsf{Sm}\ c\ \mathsf{true}\ \mathsf{id}\ S \tag{7.4}$$

It is only necessary to impose the condition Wb on circuit terms in order to prove the converse implication. This result is exactly what one would expect. If a circuit can be proved correct using the detailed threshold model, then it must also be correct according to the more abstract—but less accurate—switch model. The problem comes in proving the converse implication: that correctness in the simple switch model implies correctness in the more accurate threshold model. Because the switch model ignores threshold effects, the converse of implication (7.4) does hold for every circuit term c, and it is therefore necessary to impose the condition Wb on circuit terms in order to make the models agree on correctness. This is considered in Section 7.5.2.

The first step in the formal proof theorem (7.4) is to define a representation function which is the right inverse of the data abstraction function abs. The required function $\mathsf{rep}{:}\mathit{bool}{\rightarrow}\mathit{tri}$ is straightforward to define by cases on *bool* such that it has the following property:

$$\vdash (\mathsf{rep}\ \mathsf{T} = \mathsf{Hi}) \wedge (\mathsf{rep}\ \mathsf{F} = \mathsf{Lo})$$

Given this representation function, it is possible to formulate the following key lemma about the satisfaction of an abstract specification S in the threshold model of CMOS behaviour given by Tm:

$$\vdash \mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S = \forall e.\ \mathsf{Tm}\ c\ (\mathsf{rep} \circ e) \supset S\ e \tag{7.5}$$

This lemma shows that satisfaction of an abstract specification S in the threshold model is equivalent to the assertion that for every threshold model representation '$\mathsf{rep} \circ e$' of a switch model environment 'e', if '$\mathsf{rep} \circ e$' is allowed by the threshold model semantics, then the switch model environment e is also allowed by the abstract specification S. The proof of this lemma is straightforward, and follows immediately from the fact that the data representation and abstraction functions abs and rep have the following properties:

$$\vdash \mathsf{def}(\mathsf{rep} \circ e) \qquad \vdash \mathsf{abs} \circ \mathsf{rep} \circ e = e \qquad \vdash \mathsf{def}\ e \supset \mathsf{rep} \circ \mathsf{abs} \circ e = e$$

These theorems about abs and rep show that there is an isomorphism between abstract environments of type *str→bool* and the set of all concrete environments of type *str→tri* that satisfy the validity condition def. Satisfaction of an abstract specification S in the threshold model qualified by the validity condition def is therefore equivalent to satisfaction of S by an abstract environment e for which the corresponding concrete environment $\mathsf{rep} \circ e$ is allowed by the threshold model semantics.

Given this lemma, the formal proof of theorem (7.4) reduces to proving the following equivalent theorem about the relationship between satisfaction in the threshold model and satisfaction in the switch model:

$$\vdash \forall c\, S.\ (\forall e.\ \mathsf{Tm}\ c\ (\mathsf{rep} \circ e) \supset S\ e) \supset (\forall e.\ \mathsf{Sm}\ c\ e \supset S\ e)$$

The variable S in this theorem stands for an arbitrary predicate on switch model environments of type *str→bool*. It is straightforward to show that it is sufficient to consider only the strongest predicate that satisfies the left hand side of the implication—namely the predicate denoted by $\lambda e.\ \mathsf{Tm}\ c\ (\mathsf{rep} \circ e)$. The proof of theorem (7.4) can therefore be further reduced to proving the equivalent assertion:

$$\vdash \forall c\, e.\ \mathsf{Sm}\ c\ e \supset \mathsf{Tm}\ c\ (\mathsf{rep} \circ e) \tag{7.6}$$

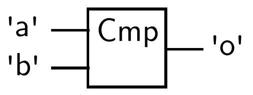
This theorem states that for every environment e which is allowed by switch model semantics, the corresponding environment $\mathsf{rep} \circ e$ in which the boolean values T and F are represented by the values Hi and Lo, is allowed by the threshold model semantics. This final lemma follows easily by structural induction on the circuit term c, and completes the formal proof of theorem (7.4).

7.5.2 Conditional Equivalence of the two Models

The preceding section has shown that a correctness result in the threshold model implies a correctness result in the simpler switch model. The converse implication, however, does not always hold. Some CMOS circuit designs which can be proved correct with respect to a specification S in the switch model are not correct with respect to S in the threshold switching model. Formally, one can prove that:

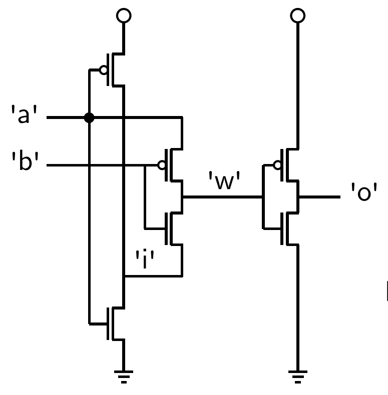
$$\vdash \neg\forall c\, S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ \mathsf{true}\ \mathsf{id}\ S \supset \mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S \tag{7.7}$$

The CMOS circuit shown Figure 7.2 on page 154 provides a counterexample by which this negative result can be proved. This circuit is intended to be an implementation of the one-bit comparator specified by:



$$\vdash \mathsf{Cmp}\ e = (e\ \texttt{'out'} = (e\ \texttt{'a'} = e\ \texttt{'b'}))$$

This specification describes a device with two boolean inputs 'a' and 'b' and one



⊢ Cmpr = Hide 'w' (Join Xor (Inv 'w' 'o'))

⊢ Xor = Hide 'i' (Join (Inv 'a' 'i')
(Join (Ptran 'a' 'b' 'w')
(Ntran 'i' 'b' 'w')))

⊢ Inv i o = Hide 'p' (Hide 'g'
(Join (Join (Pwr 'p') (Ntran i 'g' o))
(Join (Gnd 'g') (Ptran i 'p' o))))

Figure 7.2: An Incorrect CMOS Comparator.

boolean output 'o'. The device compares the values on the input wires 'a' and 'b'. If these input values are equal, then the value on the output 'o' is *true*. Otherwise the value on the output 'o' is *false*.

The CMOS circuit shown in Figure 7.2 is intended to implement the comparator behaviour specified by the predicate Cmp defined above, but is in fact composed of an *incorrect* exclusive-or gate Xor connected to an inverter by the internal wire 'w'. The exclusive-or circuit modelled by the circuit term Xor shown in this diagram is given in [11] as an example of a CMOS design which can be proved correct using the switch model of transistors, but which is in fact incorrect due to the threshold switching behaviour of its transistors.

According to the simple switch model of CMOS circuit behaviour, the circuit shown in Figure 7.2 a correct implementation of the one-bit comparator specified by the predicate Cmp defined above. It is straightforward to prove the following correctness result for this circuit in the switch model:

⊢ Sat Sm Cmpr true id Cmp

This theorem states that the circuit modelled by the circuit term Cmpr is correct with respect to the specification Cmp. According to the more accurate threshold switching model, however, the circuit represented by Cmpr is not correct with respect to the abstract specification Cmp. Formally:

⊢ ¬Sat Tm Cmpr def abs Cmp

That is, the circuit design represented by the circuit term Cmpr does not satisfy the abstract specification Cmp under the validity condition def.

The problem with the comparator circuit is that, for certain input values, the value on the internal wire 'w' can be the degraded logic level X because of threshold effects. If the input 'a' is Lo and the input 'b' is Hi then the value on the hidden wire 'w' can be either Hi or X. This means that the voltage on 'w' may be too low to drive the gates of the transistors in the output inverter. In the threshold model, the output 'o' is not forced to be the correct value Lo in this case, and the circuit therefore fails to satisfy the abstract specification of required behaviour given by Cmp. The problem is, of course, completely invisible to the switch model of CMOS behaviour, and the device is (incorrectly) regarded as correct in this simpler model.

The problem with the incorrect comparator circuit shown in Figure 7.2 can be detected in the threshold model because there is an environment e which satisfies the validity condition def e and the constraint Tm Cmpr e, but which does not satisfy the constraint Cmp $(\mathsf{abs} \circ e)$ imposed by the specification of required behaviour. In particular, there is a threshold model environment e that satisfies both the validity condition and the model, and makes the following assignment of values to the external wires of the device:

$$e\ \texttt{'a'} = \mathsf{Lo},\ \ e\ \texttt{'b'} = \mathsf{Hi},\ \ \text{and}\ \ e\ \texttt{'o'} = \mathsf{Hi}.$$

For this environment, it is not only the case that the threshold model semantics *allows* the internal wire 'w' to have the value X, but that the threshold model semantics *forces* the internal wire 'w' to have the value X.

This observation motivates the following recursive definition of a predicate on circuit terms which rules out circuits with internal wires that can be forced to have the value X, and therefore expresses a condition which is sufficient to make the two transistor models agree on correctness results. For the circuit terms that model primitive devices, and for circuit terms constructed using the function Join, the defining equations for the condition Wb are:

$$\begin{array}{lcl}
\vdash \mathsf{Wb}\ (\mathsf{Pwr}\ p) & = & \mathsf{T} \\
\vdash \mathsf{Wb}\ (\mathsf{Gnd}\ p) & = & \mathsf{T} \\
\vdash \mathsf{Wb}\ (\mathsf{Ntran}\ g\ so\ dr) & = & \mathsf{T} \\
\vdash \mathsf{Wb}\ (\mathsf{Ptran}\ g\ so\ dr) & = & \mathsf{T} \\
\vdash \mathsf{Wb}\ (\mathsf{Join}\ c_1\ c_2) & = & \mathsf{Wb}\ c_1 \wedge \mathsf{Wb}\ c_2
\end{array}$$

These equations simply state that the primitive devices satisfy the condition Wb, and that a composite circuit design satisfies Wb if its subcomponents do. To rule out internal wires whose value must be X for some external environment, the defining equation for Wb is:

$$\begin{array}{l}
\vdash \mathsf{Wb}\ (\mathsf{Hide}\ s\ c) = \mathsf{Wb}\ c\ \wedge\ \forall e.\,(\mathsf{Tm}\ c\ e \wedge \forall st.\,\neg(st{=}s) \supset \neg(e\ st{=}\mathsf{X})) \supset \\
\qquad\qquad\qquad\qquad\qquad \exists v.\,\neg(v{=}\mathsf{X}) \wedge \mathsf{Tm}\ c\ (\lambda st.\,(st{=}s \Rightarrow v \mid e\ st))
\end{array}$$

This equation states that for a circuit '$\mathsf{Hide}\ s\ c$' to satisfy the condition Wb it must be the case that the circuit c satisfies Wb, and whenever c is in an environment in which every external wire except for s does not have the value X, it is possible for the wire s not to have the value X as well. In other words, there is no well-behaved *external* environment e that satisfies the validity condition '$\mathsf{def}\ e$' but forces the internal wire s to have the degenerate value X.

If the condition Wb is defined as shown above, then the following theorem about satisfaction in the two models of transistor behaviour can be proved:

$$\vdash \forall c.\, \mathsf{Wb}\ c \supset \forall S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ \mathsf{true}\ \mathsf{id}\ S \supset \mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S \tag{7.8}$$

This theorem states that for circuit terms c that satisfy Wb, a correctness result proved in the simple switch model implies an equivalent correctness result in the more complex threshold switching model. This expresses the fact that the simple switch model is a valid abstraction of the more detailed threshold model only for a particular class of circuit designs.

Given the lemmas about satisfaction in the threshold model discussed in Section 7.5.1 and the primitive recursive defining equations shown above for the condition Wb, the formal proof of theorem (7.8) is straightforward. By lemma (7.5), the proof of this theorem reduces to showing that:

$$\vdash \forall c.\, \mathsf{Wb}\ c \supset \forall S.\ (\forall e.\, \mathsf{Sm}\ c\ e \supset S\ e) \supset (\forall e.\, \mathsf{Tm}\ c\ (\mathsf{rep} \circ e) \supset S\ e)$$

Again, it is sufficient to consider the strongest specification S that satisfies the left hand side of the implication. So the proof of theorem (7.8) further reduces to proving the simpler theorem:

$$\vdash \forall c.\, \mathsf{Wb}\ c \supset \forall e.\, \mathsf{Tm}\ c\ (\mathsf{rep} \circ e) \supset \mathsf{Sm}\ c\ e \tag{7.9}$$

This theorem can be proved by structural induction on circuit terms c. Only the step case for the constructor Hide is at all difficult. When the circuit term c has the form $\mathsf{Hide}\ s\ c$, the defining equation for $\mathsf{Wb}(\mathsf{Hide}\ s\ c)$ is needed to ensure that the hidden wire s is not forced to be the value X. If this is the case, then there is always a boolean internal values in the switch model such that the implication stated in theorem (7.9) holds. The defining equation for $\mathsf{Wb}(\mathsf{Hide}\ s\ c)$ was in fact discovered during the proof of the step case for hiding in the proof of theorem (7.9) by structural induction.

From theorem (7.8) above, and theorem (7.4) proved in Section 7.5.1, it follows immediately that correctness results in the two transistor models formalized by the semantic functions Tm and Sm are effectively equivalent for circuit terms that satisfy the condition Wb, as stated formally by theorem (7.3). The predicate Wb therefore states a condition on circuit terms which is sufficient to ensure that the two models of CMOS design behaviour agree on correctness. For circuit designs that satisfy Wb, the simple switch model is an adequate basis for verification, since the threshold switching model can not be used to detect design errors in these circuits which will not also be found by using the simpler switch model.

7.6 Summary and Discussion

This chapter has shown how a specially-defined concrete recursive type *circ* can be used to formulate and prove assertions about the relative accuracy of two formal models of hardware behaviour. These assertions are essentially statement about the class of all design models built up using the alternative primitive specifications in each model of hardware behaviour. Using the recursive type *circ* to embed the syntax of these design models within the logic itself allows these assertions to be formulated as *theorems* of the logic, and therefore proved formally using the HOL theorem prover for higher order logic.

The aim of this chapter was both to demonstrate the feasibility of this approach to formal reasoning about a class of circuit designs, and to give a simple example of the idea of a relationship of abstraction between two formal models of hardware behaviour. There are two ways in which the particular result obtained in this chapter about the formal relationship between the threshold model and the switch model might be improved. These are briefly discussed below.

A Syntactic Condition to Replace Wb

The predicate Wb defined in Section 7.5.2 is a condition on CMOS circuit designs which is sufficient to ensure that they can be verified using the simple switch model rather than the more accurate (but also more complex) threshold model. The predicate Wb, however, was not defined in a way that makes it useful in practice for determining when the simpler switch model can be used. In the defining equations for Wb, the *semantic* function Tm is used to state the condition that hidden wires are not forced to have the value X. This means that for any particular circuit term c it is necessary to carry out a proof in the threshold model of CMOS behaviour in order to determine if the condition '$\mathsf{Wb}\ c$' holds. But this may be (and, in the author's experience, typically is) just as much work as simply proving a threshold model correctness theorem for the circuit represented by c. For an equivalence result of the kind stated by theorem (7.3) to be useful in practice, a condition is needed that can be checked purely *syntactically*.

A syntactic condition on circuit terms that makes the two transistor models agree on correctness could be seen as a 'design rule' for CMOS circuits which ensures that they are *verifiable* using the simple switch model. One example of such a syntactic condition might be a predicate FC which is true of a circuit term c exactly when c represents a fully complementary CMOS circuit design. If such a predicate is defined formally and shown to satisfy the implication:

$$\forall c.\ \mathsf{FC}\ c \supset \forall S.\ \mathsf{Sat}\ \mathsf{Sm}\ c\ \mathsf{true}\ \mathsf{id}\ S = \mathsf{Sat}\ \mathsf{Tm}\ c\ \mathsf{def}\ \mathsf{abs}\ S$$

then a correctness proof for any circuit that satisfies the syntactic condition FC can be safely done using the simpler switch model semantics. A result of this kind would show that the switch model is adequate for fully complementary CMOS logic, and if circuits are designed using this conservative CMOS design style, the extra complexity of the threshold switching model is not needed to model them. Furthermore, if FC is a purely syntactic condition on circuit terms—i.e. a condition that describes only the *structure* of fully complementary circuit designs—then checking whether the simple switch model can be used for the correctness proof of any particular circuit design can be done without having to reason about its behaviour in the more complex threshold model.

Weakening the Validity Condition def

A second way in which the result proved in this chapter could be improved is by weakening the validity condition def introduced in Section 7.4. This condition was defined in order to translate an arbitrary switch model correctness statement of the form '$\forall e.\,\mathsf{Sm}\ c\ e \supset S\ e$' into an equivalent threshold model correctness statement of the form:

$$\forall e.\,\mathsf{def}\ e \supset (\mathsf{Tm}\ c\ e \supset S\ (\mathsf{abs} \circ e))$$

The problem with a correctness statement of this kind is that the constraint imposed on the environment e by the validity condition '$\mathsf{def}\ e$' is too strong. The predicate def was defined in Section 7.4 by the equation:

$$\vdash \mathsf{def}\ e = \forall s.\,\neg(e\ s = \mathsf{X})$$

This means that in a threshold model correctness statement of the general form shown above it is assumed that the environment e assigns only a strongly-driven value (i.e. either Hi or Lo) to *every* external wire of the CMOS circuit represented by the circuit term c. In particular, the validity condition '$\mathsf{def}\ e$' not only ensures that the input wires of the circuit represented by c are strongly-driven, but also makes the *assumption* that the degenerate value X will never appear on the *output* wires of the device represented by c.

The effect of making this assumption is that a threshold model correctness statement of the form given by the implication shown above fails to distinguish between circuits whose output wires are always strongly driven and circuits that can have the degraded logic level X on their output wires. A correctness theorem provable in the switch model may therefore be translated into a threshold model correctness statement which is also provable—but only by virtue of the *assumption* that all the values that appear on the outputs of a device are strongly driven. For example, the threshold model correctness statement

$$\vdash \forall e.\,\mathsf{def}\ e \supset (\mathsf{Tm}\ \mathsf{Xor}\ e \supset (\mathsf{abs}(e\ \texttt{'w'}) = \neg(\mathsf{abs}(\ e\texttt{'a'}) = \mathsf{abs}(e\ \texttt{'b'}))))$$

states the correctness of the *incorrect* exclusive-or circuit shown in Figure 7.2, and can be proved in the threshold model even though the exclusive-or circuit whose behaviour is modelled by '$\mathsf{Tm\ Xor}$' does not itself ensure that the output wire 'w' is strongly driven for all input values. The threshold switching problems with the exclusive-or circuit represented by the circuit term Xor become apparent only when the output wire 'w' is hidden—as in the comparator circuit discussed above in Section 7.5.2, for example.

To overcome this problem, it is necessary to weaken the validity condition used to express threshold model correctness statements, so that it constrains only the *inputs* of a CMOS circuit to have the strongly-driven values Hi and Lo. This would improve the translation of switch model correctness statements into corresponding threshold model correctness statements by removing the built-in assumption that only strongly-driven values appear on the output wires of a device.

One way of formulating such a validity condition is to define a condition '$\mathsf{def}\ c\ e$' which is parameterized by a circuit term c and has the meaning 'the environment e assigns only strongly-driven values to the external wires of c which are directly connected to the gates of transistors'. A formal definition of this validity condition is given by the following primitive recursive definition on circuit terms.

$$\begin{array}{lcl}
\vdash \mathsf{def}\ (\mathsf{Pwr}\ p)\ e & = & \mathsf{T} \\
\vdash \mathsf{def}\ (\mathsf{Gnd}\ g)\ e & = & \mathsf{T} \\
\vdash \mathsf{def}\ (\mathsf{Ntran}\ g\ s\ d)\ e & = & \neg(e\ g = \mathsf{X}) \\
\vdash \mathsf{def}\ (\mathsf{Ptran}\ g\ s\ d)\ e & = & \neg(e\ g = \mathsf{X}) \\
\vdash \mathsf{def}\ (\mathsf{Join}\ c_1\ c_2)\ e & = & \mathsf{def}\ c_1\ e \wedge \mathsf{def}\ c_2\ e \\
\vdash \mathsf{def}\ (\mathsf{Hide}\ s\ c)\ e & = & \mathsf{def}\ c\ \lambda st.\ (st{=}s \Rightarrow \mathsf{Hi} \mid e\ st)
\end{array}$$

For every circuit term c, these equations define a validity condition on threshold model environments '$\mathsf{def}\ c$. The defining equations make the condition $\mathsf{def}\ c\ e$ impose the constraint that that all the *externally driven* inputs connected to the gates of N-type and P-type transistor in the circuit represented by c must be strongly driven by the environment e. This expresses a validity condition which constrains the values only on external wires that are known to be inputs to the circuit represented by the circuit term c.

For example, one can prove from this definition the following theorem:

$$\vdash \mathsf{def\ Xor}\ e = \neg(e\ \texttt{'a'} = \mathsf{X}) \wedge \neg(e\ \texttt{'b'} = \mathsf{X})$$

This theorem shows that the new validity condition for the exclusive-or circuit discussed above constrains only the input wires 'a' and 'b' to be strongly driven. In this case, the validity condition does not constrain the value that appears on the output wire 'w', and therefore does not make the unjustified assumption that this value is not X. With this new validity condition, it is *not* possible to prove the correctness of the incorrect exclusive-or gate in the threshold switching model. This overcomes the problem with the simpler (and provable) threshold switching model correctness statement for the Xor circuit discussed above—the correctness statement for Xor no longer *begs the question* by being based on the assumption that the output `'w'` has either the value Hi or the value Lo.

Given this new validity condition for expressing correctness statements in the threshold switching model, an assertion can be formulated about the equivalence of correctness results in the two models of transistor behaviour which does not contain the built-in assumption that only the values Hi and Lo will be present on the output wires of a device—unlike the conditional equivalence between the two models expressed by theorem (7.3). This assertion would have the form:

$$\forall c.\, \mathsf{Wb}\ c \supset ((\forall e.\, \mathsf{Sm}\ c\ e \supset S\ e) = (\forall e.\, \mathsf{def}\ c\ e \supset (\mathsf{Tm}\ c\ e \supset S\ (\mathsf{abs} \circ e))))$$

for some condition on circuit terms Wb. This condition on circuit terms would have to be stronger than the predicate Wb defined above in Section 7.5.2, since CMOS circuits whose outputs are not strongly driven would now (correctly) be considered incorrect in the threshold switching model. A theorem of this kind would give an improved statement of the abstraction relationship between the simple switch model of CMOS behaviour and the more detailed threshold switching model of CMOS behaviour.

7.7 Related Work

The general approach taken to the formalization of abstraction between models in this chapter is strongly influenced by the categorical ideas used by Winskel in [86] to relate two formal models of CMOS transistor behaviour. Winskel uses the notion of adjunction between partial order categories to relate his static-configuration model of CMOS behaviour [85] to the switch model of transistor behaviour described by Camilleri *et al.* in [11]. This categorical approach provided the basic framework for organization of the proof of theorem (7.3) given in this chapter. In particular, lemma (7.5) discussed in Section 7.5.1 was inspired by the adjunction between satisfaction in the two models described by Winskel in [86].

Chapter 8

Conclusions and Future Work

This chapter provides a summary of the main contributions of the work reported in this dissertation and suggests some areas for future development of this research.

8.1 Summary

One of the main aims of this dissertation was to give a motivated account of the role and importance of abstraction in hardware verification, to give a reasonably general account of the use of abstraction in reasoning about hardware correctness and assessing the accuracy of formal models of hardware behaviour, and to explain some fundamental techniques for expressing certain abstraction relationships in higher order logic. In addition to providing this general account of basic principles, this dissertation has also given detailed examples to show how these principles can be applied in practice.

Chapter 1 began with a general account of the importance of abstraction to hardware verification, motivated by a discussion of some fundamental limitations to the scope of hardware verification by formal proof. The logical formalism and associated hardware verification methodology adopted in this work were then introduced in Chapters 2 and 3. The formalization in higher order logic of two fundamental kinds of abstraction was then discussed in Chapter 4. In Section 4.1, it was shown how correctness can be expressed formally in higher order logic by using abstraction mechanisms to relate detailed design models to more abstract specifications of required behaviour. Three fundamental types of abstraction were discussed: behavioural abstraction, data abstraction, and temporal abstraction. Some general issues having to do with abstraction were discussed in this section, and an account was given of the role of abstraction in hierarchical verification. Section 4.2 then introduced the idea of a relationship of abstraction between models of hardware behaviour, which was further developed and illustrated by a worked example presented in Chapter 7.

The remaining chapters provided concrete examples to illustrate the ideas about abstraction which were introduced in general terms in Chapters 1 and 4. Detailed examples of the use of data and temporal abstraction in reasoning about hardware correctness were given in Chapters 5 and 6, and an example of an abstraction relationship between two models of CMOS transistor behaviour was given in Chapter 7. A summary of the main specific contributions of the research reported in Chapters 6–7 is given below:

- Chapter 5 showed how any instance of a wide class of concrete recursive data types can be characterized formally in higher order logic and gave an overview of some basic formal properties of these types.

- An example was then given to show how a simple three-valued enumerated type could be used to formulate a threshold switching model of CMOS transistor behaviour which is more slightly more accurate than the simple switch model described in Chapter 3. An example was given to show how a data abstraction function could be used to relate a simple CMOS circuit description based on this model to an abstract specification of required behaviour. The example was used to illustrate some basic issues relating to the formalization of data abstraction in higher order logic.

- A second application of concrete types was then given in Section 5.3, where the use of a special-purpose recursive type of lists to provide an unambiguous formal representation in logic for bit-vectors was discussed. It was shown how this special-purpose defined type avoids certain problems associated with a commonly-used representation of bit-vectors based on the type $num \rightarrow bool$.

- An example of data abstraction was then given in which a class of tree-shaped circuit designs with n-bit inputs was proved correct with respect to an abstract specification. The example illustrated a new technique for representing the structure of circuit designs using a concrete recursive type of binary trees. It also showed a practical application of the facility for defining primitive recursive functions provided by the theorems used to characterize concrete recursive types in higher order logic.

- In Chapter 6, three functions were defined for expressing correctness in logic by relationships of temporal abstraction: Timeof, when, and $\mathsf{between}$. The Timeof function allows a time mapping to be defined from any predicate that identifies points of time at which the behaviour expressed by a detailed design model is also of interest at a more abstract level of description. The when and $\mathsf{between}$ functions can be used to abstract away from irrelevant details about time-dependent behaviour by relating temporally detailed signals to more abstract ones.

- A case study was then presented in which these general-purpose operators were used to relate two different levels of temporal abstraction in a hierarchically-structured correctness proof for the design of a simple ring communication network. The T-ring case study showed some of the complexities that can arise in a reasonably realistic example. In particular, it showed that the formal verification of a hardware device can become complex when tricky clocking strategies are used, but that temporal abstraction can help to control this complexity.

- In Chapter 7, a new technique was developed for modelling the behaviour of CMOS circuits by means of semantic functions defined on a concrete recursive type *circ*. By providing an explicit representation in logic for the structure of the class of all CMOS circuit designs, this recursive type made it possible to formulate assertions about the abstraction relationship between two different transistor models as *theorems* of the logic, rather than meta-theorems. These assertions could therefore be proved formally using the HOL theorem prover.

- The example given in Chapter 7 showed how the abstract switch model of transistor behaviour could be related formally to the more detailed threshold switching model. A theorem was derived which states a condition under which correctness results obtained in the two models are equivalent.

The logical basis for all the work on abstraction discussed in Chapters 5 and 7 is the systematic method for defining arbitrary concrete types explained in detail in Appendix A. One of the main *practical* contributions of this research is the automation of this method in the HOL theorem prover. The mechanized theorem proving tools developed in HOL for defining recursive types allow a user of the system to introduce new types quickly and easily. Neither a fully detailed account of this HOL implementation of recursive type definitions nor an account of the associated tools developed to support formal reasoning about recursive types using the HOL system are not given in this thesis. But a brief overview of these tools is given in Appendix B, together with an example that shows how they were used to prove one of the theorems about hardware discussed in Chapter 5.

8.2 Future Work

The following sections outline some directions for future research suggested by aspects of work discussed in this dissertation.

8.2.1 Type Definitions

The method for automating formal type definitions in higher order logic could be extended in a number useful ways. One possibility is to extend the method for defining concrete recursive types to include mutually recursive concrete types. Given the basis for constructing representations provided by the logical type of labelled trees defined in Section A.4.2 of Appendix A, it should be straightforward to develop a systematic method for defining mutually recursive types in logic. The automation of such a method in the HOL theorem prover would proceed very much along the lines of the present implementation of type definitions in HOL.

A more complex task would be to extend the method and associated HOL tools to deal with the formal definition in logic of *abstract* types specified by equational constraints. This extension would be of significant practical value, since there are many natural applications for abstract types in describing hardware behaviour at higher levels of data abstraction. At present, type definitions for any abstract types that may be needed to specify hardware behaviour must be done manually in the HOL system. Again, the automation of a method for defining abstract types could follow the lines of the present HOL tools for automating the formal definitions of concrete types. Abstract types described informally as a set of named operations together with some equational constraints on the values given by these operations could be defined formally in higher order logic by taking quotients of the existing representations for concrete types.

8.2.2 Combining Forms for Defining Models

One disappointing feature of the approach to representing bit-vectors developed in Chapter 5 is the complexity of the formal definitions for the models of n-bit circuit designs based on this representation. This was illustrated by the models defined in both the multiplexer and the test-for-zero examples given in Chapter 5. In both cases, it was necessary to include what was essentially explicit wiring information in the formal definitions of models. A more elegant approach might be to define models using a collection of higher-order 'combining forms' of the kind used to construct models in Sheeran's design language Ruby [77]. This would require a somewhat more elaborate formal representation for bit-vectors than the straightforward list representation proposed in Chapter 5, but would also make the definitions of models much simpler.

8.2.3 Higher-level Rules for Reasoning about Timing

Although a few general properties of the when and between functions defined in Chapter 6 were derived for the correctness proofs for the T-ring components, more work needs to be done to provide higher-level rules for reasoning about timing relationships using these functions. Because of the tricky clocking schemes used in the T-ring, the formal verification of this device was not an ideal vehicle for the discovery of generally applicable properties the when and between operators. In many of the formal proofs in the T-ring verification, for example, intricate proofs by induction on the lengths of intervals of time at the detailed level of description were necessary to derive appropriate relationships between the values which were to be sampled using the when operator.

There are, however, many natural applications for the functions when and between in expressing relationships between various different levels of temporal abstraction. Dhingra, for example, in his work on formalizing a circuit design style in higher order logic [21] used the version of the when operator published in [63] to formulate correctness statements for devices driven by a 2-phase clock. The version of when defined in [63] was also used by Joyce [56] to express relationships between timing at the microcode level and timing at the macroinstruction level in reasoning about the correctness of a simple microprocessor.[1] Some general principles for reasoning about timing relationships using the when and between functions would therefore be widely applicable in hardware verification. Some preliminary work has already been done on this, but more examples need to be considered to develop a widely-applicable collection of general theorems about the properties of when and between.

8.2.4 Abstraction between Models

Some suggestions for future development of the research reported in Chapter 7 have already been discussed in Section 7.6.

8.2.5 Synthesis

Abstraction mechanisms of the kind discussed in this dissertation may have a role to play in the development of formal methods for the *synthesis* of designs. This work has concentrated on the use of abstraction mechanisms to relate detailed formal descriptions of existing hardware designs to more abstract specifications. In essence, the process of abstraction is used here to *suppress* detailed information about hardware designs. Synthesis involves the opposite process—adding detail to formal specifications until an implementable design description is obtained. The possibility of a relationship between the abstraction mechanisms used here for post-design verification and heuristics for hardware design synthesis from formal specifications should be investigated.

[1]In fact, Joyce used 'Timeof'—i.e. the definition of when—in [56].

Appendix A

Defining Concrete Types

This appendix describes a systematic method for defining an arbitrary concrete recursive type[1] in higher order logic. A full implementation of this method has been done in the HOL system. The automatic theorem-proving tools based on the work described in this appendix are included in the 1988–89 release of the HOL system ('HOL88'). The concrete types used in the hardware verification examples in Chapters 5 and 7 were defined formally in the HOL system using these tools.

Outline of the Appendix

The main aim of the work reported in this appendix was to provide a *practical* tool for defining types automatically in the HOL system. It was therefore considered essential for this tool to be reasonably efficient. The strategy used to achieve this aim was to develop a method which reduced to a minimum the amount of inference that had to be done to define an arbitrary concrete type formally in logic. This was done by: (1) defining a collection of basic types which could be used to construct an appropriate representation for an arbitrary concrete recursive type, and (2) proving a general theorem which states that any type whose representation is constructed from these basic types is characterized by an abstract 'axiom' of the desired form. This allowed the process of defining a concrete recursive type and proving an abstract 'axiomatization' for it to be done automatically, without the system having to do much inference at 'run time'. (For a general discussion of this efficiency strategy, see Section 2.2.3.)

This appendix, which gives full details of the logical basis for this approach to automating type definitions, is divided into two main parts. The first part consists of Sections A.2–A.4, in which formal definitions are given for all the types used as 'building blocks' to construct representations for the values of an arbitrary concrete recursive type. The second part (Sections A.5–A.6) describes the method for defining concrete recursive types in general and provides a very brief overview of the HOL theorem-proving tools based on this method.

[1]The types definable by the method described here will be referred to as 'concrete recursive types', or just 'concrete types'. Use of the former term is not intended to *exclude* non-recursive examples, but merely to emphasize that this class of types *includes* recursive ones.

A.1 Representation and Abstraction Functions

This section provides an overview of some preliminary definitions which are used in every type definition done in later parts of this appendix.

The primitive rule of definition which allows new types to be introduced in higher order logic was explained in Section 2.1.7.1. This rule allows a new type to be defined by postulating a *type definition axiom* of the general form shown below.

$$\vdash \exists f{:}ty_P{\rightarrow}ty.\,(\forall a_1\,a_2.f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\,P\ r = (\exists a.\,r = f\ a))$$

An axiom of this form defines a new type ty_P by asserting that there exists an isomorphism f between ty_P and the subset of an existing type ty defined by the predicate P. A type definition axiom of this form merely asserts the *existence* of such an isomorphism—to formulate abstract axioms for a new type, it is necessary to have constants which actually *denote* such an isomorphism and its inverse. These constants are needed to make it possible to define operations on the values of a new type in terms of operations on values of the representing type.

A constant which denotes an isomorphism between a defined type and the subset of an existing type which represents it can be defined as follows. A type definition axiom asserts the existence of a function f which maps each value of the defined type ty_P to a corresponding value of type ty. By the method described in Section 2.1.5 of Chapter 2, the primitive constant ε can be used to define a constant $\mathsf{REP}{:}ty_P{\rightarrow}ty$ which denotes this function. Using ε, the constant REP can be defined such that:

$$\vdash \forall a_1\,a_2.\,\mathsf{REP}\ a_1 = \mathsf{REP}\ a_2 \supset a_1 = a_2 \wedge \forall r.\,P\ r = (\exists a.\,r = \mathsf{REP}\ a)$$

The resulting theorem has the same form as the type definition axiom for ty_P; all that has been done is to give the name ‘REP’ to the function which the type definition axiom asserts to exist. This constant is called a *representation* function, and the theorem shown above asserts that it is one-to-one and onto the subset of ty given by P. That is, the constant REP denotes an isomorphism between the new type ty_P and the representing subset of ty.

Once REP has been defined, the primitive constant ε can be used to define an inverse *abstraction* function: $\mathsf{ABS}{:}ty{\rightarrow}ty_P$. The formal definition of this function is shown below.

$$\vdash \mathsf{ABS}\ r = (\varepsilon\ \ \lambda a.\,r = \mathsf{REP}\ a).$$

It is straightforward to prove that the function defined by this equation is one-to-one (for the values of type ty that satisfy P) and onto the new type ty_P. These two properties are stated formally by the two theorems shown below.

$$\vdash \forall r_1\, r_2.\, P\ r_1 \supset (P\ r_2 \supset (\mathsf{ABS}\ r_1 = \mathsf{ABS}\ r_2 \supset r_1 = r_2))$$
$$\vdash \forall a.\, \exists r.\, (a = \mathsf{ABS}\ r) \wedge P\ r$$

It also follows from the definitions shown above that the abstraction function ABS is the left inverse of the representation function REP. For values of type *ty* that satisfy P, it also follows that REP is the left inverse of ABS. These two properties are stated formally by:

$$\vdash \forall a.\, \mathsf{ABS}(\mathsf{REP}\ a) = a$$
$$\vdash \forall r.\, P\ r = (\mathsf{REP}(\mathsf{ABS}\ r) = r)$$

Abstraction and representation functions of this kind are used in every type definition described in later sections of this appendix. These functions are always defined in the way outlined above, and details of their definitions will therefore be omitted. Theorems corresponding to those shown above for ABS and REP are used in the proofs of abstract axioms for each new type defined, but the proofs of these theorems will not be given.

A.2 Three Simple Type Definitions

Three examples are given in this section which illustrate the general approach to defining new types discussed in Section 2.1.7.2 of Chapter 2. In each example, a logical type is defined, and its properties axiomatized, in three distinct steps. In the first step, an appropriate subset of an existing type is found to represent the values of the new type, and a predicate is defined to specify this subset. The second step is to postulate a type definition axiom for the new type, and define abstraction and representation functions of the kind described in Section A.1. Finally, an abstract axiomatization is formulated for the new type, and derived by formal proof from the properties of its definition. This axiomatization describes the essential properties of the newly-defined type, but does so without reference to the way it is represented and defined. In the examples given below, some basic theorems about the types which are defined are proved from their abstract axiomatizations.

The three simple types defined below are used as basic 'building blocks' in the general method explained in Section A.5.3 for finding appropriate representations for arbitrary concrete recursive types.

A.2.1 The Type Constant *one*

This section describes the definition and axiomatization of the simplest (and the smallest) type possible in higher order logic: a type constant *one*, which denotes a set having exactly one element.

A.2.1.1 The Representation

To represent the type *one*, any singleton subset of an existing type will do. In the type definition given below, the subset of *bool* containing only the truth-value T will be used. This subset can be specified by the predicate $\lambda b{:}bool.\, b$, which denotes the identity function on *bool*. The set of booleans satisfying this predicate clearly has the property that the new type *one* is expected to have, namely the property of having exactly one element.

A.2.1.2 The Type Definition

As was discussed in Chapter 2, the primitive rule for type definitions requires the representing subset for a new type to be non-empty, and a theorem which states this must be proved before a definition for the type *one* can be made. In the present case, the representing subset is specified by the predicate $\lambda b.\, b$, and it is trivial to prove that this predicate specifies a non-empty set of booleans. The theorem $\vdash \exists x.\, (\lambda b.b)x$ follows immediately from $\vdash (\lambda b.b)\mathsf{T}$, which is itself equivalent to $\vdash \mathsf{T}$.

Once it has been shown that $\lambda b.b$ specifies a non-empty set of booleans, the type constant *one* can be defined by postulating the type definition axiom shown below.

$$\vdash \exists f{:}one{\to}bool.\, (\forall a_1\, a_2. f\ a_1{=}f\ a_2 \supset a_1{=}a_2) \wedge (\forall r.\, (\lambda b.b)\ r{=}(\exists a.\, r{=}f\ a))$$

Using this axiom for *one*, a representation function $\mathsf{REP\_one}{:}one{\to}bool$ can be defined by the method outlined in Section A.1. This representation function is one-to-one:

$$\vdash \forall a_1\, a_2.\, \mathsf{REP\_one}\ a_1 = \mathsf{REP\_one}\ a_2 \supset a_1 = a_2 \tag{A.1}$$

and onto the subset of *bool* defined by $\lambda b.b$:

$$\vdash \forall r.\, (\lambda b.b)\ r = (\exists a.\, r = \mathsf{REP\_one}\ a)$$

This theorem can be simplified by performing the β-reduction $\vdash (\lambda b.b)\ r = r$. This immediately yields the theorem shown below.

$$\vdash \forall r.\, r = (\exists a.\, r = \mathsf{REP\_one}\ a) \tag{A.2}$$

The theorems (A.1) and (A.2) about $\mathsf{REP\_one}{:}one{\to}bool$ are used in the proof given in the following section of the abstract axiomatization of *one*. An inverse abstraction function $\mathsf{ABS\_one}{:}bool{\to}one$ is not needed in this simple example.[2]

[2]In fact, the axiomatization of *one* can be derived directly from its definition. The constant $\mathsf{REP\_one}$ is defined here merely to simplify the presentation of the proof that follows.

A.2.1.3 Deriving the Axiomatization of *one*

The axiomatization of the type *one* will consist of the following single theorem:

$$\vdash \forall f{:}\alpha{\to}one.\, \forall g{:}\alpha{\to}one.\, (f = g)$$

This theorem states that any two functions f and g mapping values of type α to values of type *one* are equal. From this, it follows that there is only one value of type *one*, since if there were more than one such value it would be possible to define two different functions of type $\alpha{\to}one$. This theorem is therefore an abstract characterization of the type *one*; it expresses the essential properties of this type, but does so without reference to the way it is defined.

The abstract axiom for *one* shown above follows from the properties of $\mathsf{REP\_one}$ given by theorems (A.1) and (A.2). Specializing the variable r in (A.2) to the term $\mathsf{REP\_one}(f\ x)$ yields:

$$\vdash \mathsf{REP\_one}(f\ x) = (\exists a.\, \mathsf{REP\_one}(f\ x) = \mathsf{REP\_one}\ a)$$

The right hand side of this equation is equal to T, and it can therefore be simplified to $\vdash \mathsf{REP\_one}(f\ x)$. Similar reasoning yields the theorem $\vdash \mathsf{REP\_one}(g\ x)$, from which it follows that:

$$\vdash \mathsf{REP\_one}(f\ x) = \mathsf{REP\_one}(g\ x)$$

From this, and theorem (A.1) stating that the function $\mathsf{REP\_one}$ is one-to-one, it follows that $\vdash f\ x = g\ x$ and therefore that $\vdash \forall f\ g.\, (f = g)$, as desired.

A.2.1.4 A Theorem about *one*

Once the axiom for *one* has been proved, it is straightforward to prove a theorem which states explicitly that there is only one value of type *one*. This is done by defining a constant one to denote the single value of type *one*. Using the primitive constant ε, the constant one can be defined formally by:

$$\vdash \mathsf{one} = \varepsilon(\lambda x{:}one.\, \mathsf{T})$$

From the axiom for *one*, it follows that $\vdash \lambda x{:}\alpha.\, v = \lambda x{:}\alpha.\, \mathsf{one}$. Applying both sides of this equation to $x{:}\alpha$, and doing a β-reduction, gives $\vdash v = \mathsf{one}$. Generalizing v yields $\vdash \forall v{:}one.\, v = \mathsf{one}$, which states that every value v of type *one* is equal to the constant one, i.e. there is only one value of type *one*.

A.2.2 The Type Operator *prod*

In this section, a binary type operator *prod* is defined to denote the cartesian product operation on types. If ty_1 and ty_2 are types, then the type $(ty_1, ty_2)prod$ will be the type of ordered pairs whose first component is of type ty_1 and whose second component is of type ty_2.

A.2.2.1 The Representation

The type $(\alpha,\beta)\mathit{prod}$ can be represented by a subset of the polymorphic primitive type $\alpha{\to}\beta{\to}\mathit{bool}$. The idea is that an ordered pair $\langle a{:}\alpha,\ b{:}\beta\rangle$ will be represented by the function

$$\lambda x\, y.\, (x{=}a) \wedge (y{=}b)$$

which yields the truth-value T when applied to the two components a and b of the pair, and yields F when applied to any other two values of types α and β.

Every pair can be represented by a function of the form shown above, but not every function of type $\alpha{\to}\beta{\to}\mathit{bool}$ represents a pair. The functions that do represent pairs are those which satisfy the predicate $\mathsf{Is\_pair\_REP}$ defined by:

$$\vdash \mathsf{Is\_pair\_REP}\ f = \exists v_1\, v_2.\, f = \lambda x\, y.\, (x{=}v_1) \wedge (y{=}v_2),$$

i.e. those functions f which have the form $\lambda x\, y.\, (x{=}v_1) \wedge (y{=}v_2)$ for some pair of values v_1 and v_2. This will be the subset predicate for the representation of $(\alpha,\beta)\mathit{prod}$. As will be shown below, the set of functions satisfying $\mathsf{Is\_pair\_REP}$ has exactly the standard properties of the cartesian product of types α and β.

A.2.2.2 The Type Definition

To introduce a type definition axiom for *prod*, one must first show that the predicate $\mathsf{Is\_pair\_REP}$ defines a non-empty subset of $\alpha{\to}\beta{\to}\mathit{bool}$. This is easy, since it is the case that $\vdash \forall a\, b.\, \mathsf{Is\_pair\_REP}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))$ and therefore $\vdash \exists f.\, \mathsf{Is\_pair\_REP} f$. Once this theorem has been proved, a type definition axiom of the usual form can be introduced for the type operator *prod*:

$$\begin{array}{l} \vdash \exists f{:}(\alpha,\beta)\mathit{prod}{\to}(\alpha{\to}\beta{\to}\mathit{bool}). \\ \qquad (\forall a_1\, a_2. f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_pair\_REP}\ r = (\exists a.\, r = f\ a)) \end{array}$$

This defines the compound type $(\alpha,\beta)\mathit{prod}$ to be isomorphic to the subset of $\alpha{\to}\beta{\to}\mathit{bool}$ defined by $\mathsf{Is\_pair\_REP}$. Since the type variables α and β in this axiom can be instantiated to any two types, it has the effect of giving a representation not only for the particular type '$(\alpha,\beta)\mathit{prod}$', but also for the product of *any* two types. For example, instantiating both α and β to *bool* yields a type definition axiom for the cartesian product $(\mathit{bool},\mathit{bool})\mathit{prod}$. As will be shown below, the abstract axiomatization of *prod* derived from the type definition axiom given above is also formulated in terms of the compound type $(\alpha,\beta)\mathit{prod}$. It therefore also holds for any substitution instance of $(\alpha,\beta)\mathit{prod}$—i.e. for the product of any two types.

The abstract axiomatization of *prod* derived in the following section will use the abstraction and representation functions:

$$\begin{array}{l} \mathsf{ABS\_pair}{:}(\alpha{\to}\beta{\to}\mathit{bool}){\to}(\alpha,\beta)\mathit{prod} \quad \text{and} \\ \mathsf{REP\_pair}{:}(\alpha,\beta)\mathit{prod}{\to}(\alpha{\to}\beta{\to}\mathit{bool}) \end{array}$$

which relate pairs to the functions of type $\alpha{\to}\beta{\to}\mathit{bool}$ which represent them. These representation and abstraction functions are defined formally as described above in Section A.1. A set of theorems stating that $\mathsf{Abs\_pair}$ and $\mathsf{Rep\_pair}$ are isomorphisms can also be proved as outlined in Section A.1. These theorems will be used in the proof of the axiom for *prod* given in the next section.

For notational convenience, the infix type operator '$\times$' will now be used for the product of two types. In what follows, a type expression of the form $ty_1 \times ty_2$ should be read as a metalinguistic abbreviation for $(ty_1, ty_2)\mathit{prod}$.

A.2.2.3 Deriving the Axiomatization of *prod*

To formulate the axiomatization of $(\alpha \times \beta)$, two constants will be defined:

$$\mathsf{Fst}{:}(\alpha \times \beta){\to}\alpha \quad \text{and} \quad \mathsf{Snd}{:}(\alpha \times \beta){\to}\beta.$$

These denote the usual *projection* functions on pairs; the function Fst extracts the first component of a pair, and the function Snd extracts the second component of a pair. The definitions of these functions are:

$$\begin{array}{l} \vdash \mathsf{Fst}\, p \;= \varepsilon\ \lambda x.\, \exists y.\, (\mathsf{REP\_pair}\, p)\ x\ y \\ \vdash \mathsf{Snd}\, p = \varepsilon\ \lambda y.\, \exists x.\, (\mathsf{REP\_pair}\, p)\ x\ y \end{array}$$

These definitions first use the representation function $\mathsf{REP\_pair}$ to map a pair p to the function which represents it. They then 'select' the required component of the pair using ε. From the definitions of Fst and Snd, it is possible to show that

$$\begin{array}{l} \vdash \mathsf{Fst}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) \;= a \\ \vdash \mathsf{Snd}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) = b \end{array} \tag{A.3}$$

by using the fact that $\mathsf{Rep\_pair}$ is the left inverse of $\mathsf{ABS\_pair}$ for functions which satisfy the subset predicate $\mathsf{Is\_pair\_REP}$. Once these theorems have been proved, the axiomatization of the cartesian product of two types can be derived without further reference to the way Fst and Snd are defined.

Using the functions Fst and Snd, the axiomatization of the cartesian product of two types can be formulated based on the categorical notion of a *product*. With this approach, the following theorem will be the single axiom for the product of two types:

$$\vdash \forall f{:}\gamma{\to}\alpha.\ \forall g{:}\gamma{\to}\beta.\ \exists!\, h{:}\gamma{\to}(\alpha \times \beta).\ (\mathsf{Fst} \circ h = f) \wedge (\mathsf{Snd} \circ h = g)$$

This theorem states that for all functions f and g, there is a unique function h such that the diagram

$$\begin{tikzcd}
 & \gamma \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[dd, "h"] & \\
\alpha & & \beta \\
 & (\alpha \times \beta) \arrow[ul, "\mathsf{Fst}"] \arrow[ur, "\mathsf{Snd}"'] &
\end{tikzcd}$$

is commutative—i.e. such that $\forall x.\, \mathsf{Fst}(h\ x){=}f\ x$ and $\forall x.\, \mathsf{Snd}(h\ x){=}g\ x$. As noted above, this theorem is proved for the polymorphic type $(\alpha \times \beta)$. It therefore characterizes the product of any two types, since the type variables α and β in this theorem can be instantiated to any two types of the logic to yield an axiom for their product.

An outline of the proof of the axiom shown above is as follows. Given two functions $f{:}\gamma{\to}\alpha$ and $g{:}\gamma{\to}\beta$, define the function $h{:}\gamma{\to}(\alpha \times \beta)$ as follows:

$$h\ v = \mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}f\ v) \wedge (y{=}g\ v))$$

Using the theorems (A.3) above, it follows that $\mathsf{Fst} \mathbin{\mathsf{o}} h = f$ and $\mathsf{Snd} \circ h = g$. To show that h is unique, suppose that there is also a function h' such that $\mathsf{Fst} \mathbin{\mathsf{o}} h' = f$ and $\mathsf{Snd} \circ h' = g$. Suppose v is some value of type γ. Since $\mathsf{ABS\_pair}$ is onto $(\alpha \times \beta)$, there exist a and b such that $h'\ v = \mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))$. Thus,

$$\begin{array}{lllllll}
f\ v & = & \mathsf{Fst}(h'\ v) & = & \mathsf{Fst}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) & = & a \qquad \text{and} \\
g\ v & = & \mathsf{Snd}(h'\ v) & = & \mathsf{Snd}(\mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}a) \wedge (y{=}b))) & = & b
\end{array}$$

which means that

$$h'\ v \ = \ \mathsf{ABS\_pair}(\lambda x\, y.\, (x{=}f\ v) \wedge (y{=}g\ v)) \ = \ h\ v$$

and therefore that $h' = h$.

A.2.2.4 Theorems about *prod*

Using the axiom for products proved in the previous section, an infix operator $\otimes$ can be defined such that for all functions $f{:}\gamma{\to}\alpha$ and $g{:}\gamma{\to}\beta$ the expression $f \otimes g$ denotes the unique function of type $\gamma{\to}(\alpha \times \beta)$ which the axiom asserts to exist. This operator can be defined using ε as follows:

$$\vdash \forall f\, g.\, (f \otimes g) = \varepsilon\ \lambda h.\, (\mathsf{Fst} \circ h = f) \wedge (\mathsf{Snd} \circ h = g)$$

It follows immediately from this definition and the derived abstract axiom for products that $(f \otimes g)$ denotes a function which makes the diagram shown above commute:

$$\vdash \mathsf{Fst} \circ (f \otimes g) = f \quad \text{and} \quad \vdash \mathsf{Snd} \circ (f \otimes g) = g.$$

It can also be shown that for all f and g, the term $f \otimes g$ denotes the unique function with this property:

$$\vdash \forall f\, g\, h.\, (\mathsf{Fst} \circ h = f) \wedge (\mathsf{Snd} \circ h = g) \supset (h = (f \otimes g)).$$

Using the operator $\otimes$, an infix pairing function ',' can be defined to give the usual syntax for pairs, with (a, b) denoting the ordered pair having first component a and second component b. The definition is:

$$\vdash \forall a\, b.(a, b) = ((\mathsf{K}\ a) \otimes \mathsf{I})\ b \qquad \text{where } \mathsf{K} = \lambda a\, b.a \text{ and } \mathsf{I} = \lambda a.a.$$

With this definition of pairing, the three theorems about the cartesian product type listed below, which were discussed in Section 2.1.7.2, can be proved.

$$\vdash \forall a\, b.\, \mathsf{Fst}(a, b) = a$$
$$\vdash \forall a\, b.\, \mathsf{Snd}(a, b) = b$$
$$\vdash \forall p.\, p = (\mathsf{Fst}\ p, \mathsf{Snd}\ p)$$

The first two of these theorems follow from the definition of the infix pairing operator ',' and the fact that $\vdash \mathsf{Fst} \circ ((\mathsf{K}\ a) \otimes \mathsf{I}) = \mathsf{K}\ a$ and $\vdash \mathsf{Snd} \circ ((\mathsf{K}\ a) \otimes \mathsf{I}) = \mathsf{I}$. The third theorem follows from the uniqueness of functions defined using $\otimes$.

A.2.3 The Type Operator *sum*

The final example in this section is the definition and axiomatization of a binary type operator *sum* to denote the disjoint sum operation on types. The set that will be denoted by the compound type $(ty_1, ty_2)sum$ can be thought of as the union of two disjoint sets: a copy of the set denoted by ty_1, in which each element is labelled as coming from ty_1; and a copy of the set denoted by ty_2, in which each element is labelled as coming from ty_2. Thus each value of type $(ty_1, ty_2)sum$ will correspond either to a value of type ty_1 or to a value of type ty_2. Furthermore, each value of type ty_1 and each value of type ty_2 will correspond to a unique value of type $(ty_1, ty_2)sum$.

A.2.3.1 The Representation

One way of representing a value v of type $(\alpha, \beta)sum$ would be to use a triple (a, b, f) of type $\alpha \times \beta \times bool$, where f is a boolean 'flag' which indicates whether v corresponds to the value a of type α or the value b of type β. With this representation, each value a of type α would correspond to a triple $(a, \mathsf{d}_\beta, \mathsf{T})$ in the representation, where d_β is some fixed 'dummy' value of type β. Likewise, each value b of type β would have a corresponding triple $(\mathsf{d}_\alpha, b, \mathsf{F})$ in the representation, where d_α is a dummy value of type α. Using this representation, every value in the representing subset of $\alpha \times \beta \times bool$ would correspond either to a value of type α labelled by T or to a value of type β labelled by F.

The representation of values of type $(\alpha, \beta)sum$ can be both simplified and made independent of the product type operator by noting that a triple $(a, \mathsf{d}_\beta, \mathsf{T})$, for example, can itself be represented by the function:

$$\lambda x\, y\, fl.\, (x{=}a) \wedge (y{=}\mathsf{d}_\beta) \wedge (fl{=}\mathsf{T})$$

This function is true exactly when applied to the value a, the dummy value d_β and the truth-value T. Every function of this form corresponds to unique value of type α, and every value of type α corresponds to a function of this form.

This property, however, is shared by functions of a somewhat simpler form:

$$\lambda x\, y\, fl.\, (x{=}a) \wedge (fl{=}\mathsf{T})$$

The dummy value d_β is therefore not necessary in representing the values of the disjoint sum. A value of type $(\alpha, \beta)sum$ which corresponds to a value a of type α can be represented by a function of the simpler form shown above. A value of type $(\alpha, \beta)sum$ which corresponds to a value b of type β can likewise be represented by a function of the form: $\lambda x\, y\, fl.\, (y{=}b) \wedge (fl{=}\mathsf{F})$.

The type $(\alpha, \beta)sum$ can therefore be represented by the subset of functions of type $\alpha{\to}\beta{\to}bool{\to}bool$ that satisfy the predicate Is_sum_REP defined by:

$$\begin{aligned} \vdash \mathsf{Is\_sum\_REP}\ f = {} & (\exists v_1. f = \lambda x\, y\, fl.\, (x{=}v_1) \wedge (fl{=}\mathsf{T})) \vee \\ & (\exists v_2. f = \lambda x\, y\, fl.\, (y{=}v_2) \wedge (fl{=}\mathsf{F})) \end{aligned}$$

The set of functions satisfying Is_sum_REP contains exactly one function for each value of type α and exactly one function for each value of type β. It therefore represents the disjoint sum of the set of values of type α and the set of values of type β.

A.2.3.2 The Type Definition

The type definition axiom for *sum* is introduced in exactly the same way as the defining axioms for *one* and *prod*. The first step is to prove a theorem stating that $\mathsf{Is\_sum\_REP}$ is true of at least one value in the representing set: $\vdash \exists f.\, \mathsf{Is\_sum\_REP}\ f$. A type definition axiom of the usual form can then be introduced:

$$\begin{aligned}&\vdash \exists f{:}(\alpha,\beta)\mathit{sum}{\to}(\alpha{\to}\beta{\to}\mathit{bool}).\\ &\qquad (\forall a_1\, a_2. f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_sum\_REP}\ r = (\exists a.\, r = f\ a))\end{aligned}$$

and the abstraction and representation functions

$$\begin{aligned}&\mathsf{ABS\_sum}{:}(\alpha{\to}\beta{\to}\mathit{bool}{\to}\mathit{bool}){\to}(\alpha,\beta)\mathit{sum} \quad \text{and}\\ &\mathsf{REP\_sum}{:}(\alpha,\beta)\mathit{sum}{\to}(\alpha{\to}\beta{\to}\mathit{bool}{\to}\mathit{bool})\end{aligned}$$

defined in the usual way. As outlined in Section A.1, the definitions of $\mathsf{Abs\_sum}$ and $\mathsf{REP\_sum}$ and the type definition axiom for *sum* yield the usual isomorphism theorems about such abstraction and representation functions. These theorems will be used in the derivation of the abstract axiom for *sum*.

For notational clarity, an infix type operator '+' will now be used for the disjoint sum of two types. In what follows, the metalinguistic abbreviation $ty_1 + ty_2$ will be used to stand for $(ty_1, ty_2)\mathit{sum}$.

A.2.3.3 Deriving the Axiomatization of *sum*

The axiomatization of $(\alpha + \beta)$ will use two constants:

$$\mathsf{Inl}{:}\alpha{\to}(\alpha+\beta) \quad \text{and} \quad \mathsf{Inr}{:}\beta{\to}(\alpha+\beta)$$

defined by:

$$\begin{aligned}&\vdash \mathsf{Inl}\ a = \mathsf{ABS\_sum}(\lambda x\, y\, fl.\, (x{=}a) \wedge (fl{=}\mathsf{T}))\\ &\vdash \mathsf{Inr}\ b = \mathsf{ABS\_sum}(\lambda x\, y\, fl.\, (y{=}b) \wedge (fl{=}\mathsf{F}))\end{aligned}$$

The constants Inl and Inr denote the left and right *injection* functions for sums. Every value of type $(\alpha + \beta)$ is either a left injection $\mathsf{Inl}\ a$ for some value $a{:}\alpha$ or a right injection $\mathsf{Inr}\ b$ for some value $b{:}\beta$.

The form of the axiom for $(\alpha + \beta)$ is based on the categorical notion of a *coproduct*. The axiom for $(\alpha + \beta)$ is:

$$\vdash \forall f{:}\alpha{\to}\gamma.\ \forall g{:}\beta{\to}\gamma.\ \exists!\, h{:}(\alpha+\beta){\to}\gamma.\ (h \circ \mathsf{Inl} = f) \wedge (h \circ \mathsf{Inr} = g)$$

This theorem asserts that for all functions f and g there is a unique function h such that the diagram shown below is commutative.

$$\begin{tikzcd}
 & \gamma & \\
\alpha \arrow[ur, "f"] \arrow[dr, "\mathsf{Inl}"'] & & \beta \arrow[ul, "g"'] \arrow[dl, "\mathsf{Inr}"] \\
 & (\alpha + \beta) \arrow[uu, "h"] &
\end{tikzcd}$$

The proof of the axiom for sums is similar to the one outlined in the previous section for products. The proof will therefore not be given in full here. The existence of h follows simply by defining

$$h\ s = ((\exists v_1.\, x = \mathsf{Inl}\ v_1) \Rightarrow f(\varepsilon\ \ \lambda v_1.\, x = \mathsf{Inl}\ v_1) \,|\, g(\varepsilon\ \ \lambda v_2.\, x = \mathsf{Inr}\ v_2))$$

for given f and g. The uniqueness of h follows from the fact that Inl and Inr are one-to-one, and from the fact that $\mathsf{ABS\_sum}$ is onto.

A.2.3.4 Theorems about *sum*

Using the axiom for sums, it is possible to define an operator $\oplus$ which is analogous to the operator $\otimes$ defined above for products. The definition of $\oplus$ is:

$$\vdash \forall f\, g.\, (f \oplus g) = \varepsilon\ \ \lambda h.\, (h \circ \mathsf{Inl} = f) \wedge (h \circ \mathsf{Inr} = g)$$

From the axiom for sums, it follows that for all functions f and g the term $(f \oplus g)$ denotes a function that makes the diagram for sums commute:

$$\vdash (f \oplus g) \circ \mathsf{Inl} = f \quad \text{and} \quad \vdash (f \oplus g) \circ \mathsf{Inr} = g$$

and that $(f \oplus g)$ denotes the unique function with this property:

$$\vdash \forall f\, g\, h.\, (h \circ \mathsf{Inl} = f) \wedge (h \circ \mathsf{Inr} = g) \supset (h = (f \oplus g)).$$

Using $\oplus$, it is possible to define two *discriminator* functions $\mathsf{Isl}{:}(\alpha + \beta){\rightarrow}\mathit{bool}$ and $\mathsf{Isr}{:}(\alpha + \beta){\rightarrow}\mathit{bool}$ as follows:

$$\vdash \mathsf{Isl} = (\mathsf{K}\ \mathsf{T}) \oplus (\mathsf{K}\ \mathsf{F}) \quad \text{and} \quad \vdash \mathsf{Isr} = (\mathsf{K}\ \mathsf{F}) \oplus (\mathsf{K}\ \mathsf{T})$$

From these definitions, and the properties of $\oplus$ shown above, it follows that every value of type $(\alpha + \beta)$ satisfies either Isl or Isr:

$$\vdash \forall s{:}(\alpha + \beta).\, \mathsf{Isl}\ s \vee \mathsf{Isr}\ s$$

and that Isl is true of left injections and Isr is true of right injections:

$$\begin{array}{ll} \vdash \forall a.\, \mathsf{Isl}(\mathsf{Inl}\ a) & \vdash \forall b.\, \neg\mathsf{Isl}(\mathsf{Inr}\ b) \\ \vdash \forall b.\, \mathsf{Isr}(\mathsf{Inr}\ b) & \vdash \forall a.\, \neg\mathsf{Isr}(\mathsf{Inl}\ a) \end{array}$$

The operator $\oplus$ can also be used to define *projection* functions $\mathsf{Outl}{:}(\alpha+\beta)\to\alpha$ and $\mathsf{Outr}{:}(\alpha+\beta)\to\beta$ that map values of type $(\alpha+\beta)$ to the corresponding values of type α or β. Their definitions are:

$$\vdash \mathsf{Outl} = \mathsf{I} \oplus (\mathsf{K}\ \varepsilon\ \ \lambda b.\, \mathsf{F}) \quad \text{and} \quad \vdash \mathsf{Outr} = (\mathsf{K}\ \varepsilon\ \ \lambda a.\, \mathsf{F}) \oplus \mathsf{I}$$

where $\varepsilon\ \lambda a.\, \mathsf{F}$ and $\varepsilon\ \lambda b.\, \mathsf{F}$ denote ‘arbitrary’ values of type α and β respectively. From these definitions, it follows that the projection functions Outl and Outr have the properties:

$$\begin{array}{ll} \vdash \forall a.\, \mathsf{Outl}(\mathsf{Inl}\ a) = a & \vdash \forall s.\, \mathsf{Isl}\ s \supset \mathsf{Inl}(\mathsf{Outl}\ s) = s \\ \vdash \forall a.\, \mathsf{Outr}(\mathsf{Inr}\ a) = a & \vdash \forall s.\, \mathsf{Isr}\ s \supset \mathsf{Inr}(\mathsf{Outr}\ s) = s \end{array}$$

A.3 Two Recursive Types: Numbers and Lists

This section outlines the definition of two recursive types: *num* (the type natural numbers) and (α)*list* (the polymorphic type of lists). Both *num* and (α)*list* are simple examples of the kind of recursive types which can be defined using the general method that will be described in Section A.5. Their definitions are given here as examples to introduce the idea of defining recursive types in higher order logic. They also provide examples of the general form of abstract axiomatization that will be used in Section A.5 for such types.

Both *num* and (α)*list* will be used in Section A.4 to construct representations for two logical types of trees. Along with the basic building blocks: *one*, *prod* and *sum*, these types of trees will then be used in Section A.5.3 to construct representations for arbitrary concrete recursive types.

A.3.1 The Natural Numbers

The construction of the natural numbers described in this section is based on the definition of the type *num* outlined by Gordon in [29]. The type *num* of natural numbers is defined using a subset of the primitive type *ind* of individuals. This primitive type is characterized by a single axiom, the *axiom of infinity* shown below:

$$\vdash \exists f{:}\mathit{ind}\to\mathit{ind}.\, (\forall x_1\, x_2.\, (f\ x_1 = f\ x_2) \supset (x_1 = x_2)) \wedge \neg(\forall y.\, \exists x.\, y = f x)$$

This theorem is one of the basic axioms of higher order logic. It asserts the existence of a function f from *ind* to *ind* which is one-to-one but not onto.

From this axiom, it follows that there are at least a countably infinite number of distinct values of type *ind*. Informally, this follows by observing that there is at least one value of type *ind* which is not in the image of f. Call this value i_0. Now define i_1 to be $f(i_0)$. Since i_1 is in the image of the function f and i_0 is not, it follows that they are distinct values of type *ind*. Now, define i_2 to be $f(i_1)$. By the same argument as given above for i_1, it is clear that i_2 is not equal to i_0. Furthermore, i_2 is also not equal to i_1, since from the fact that f is one-to-one it follows that if $i_2 = i_1$ then $f(i_1) = f(i_0)$ and so $i_1 = i_0$. So i_2 is distinct from both i_1 and i_0. Defining i_3 to be $f(i_2)$, i_4 to be $f(i_3)$, etc. gives—by the same reasoning—an infinite sequence of distinct values of type *ind*. This infinite sequence can be used to represent the natural numbers.

A.3.1.1 The Representation and Type Definition

As was outlined informally above, it follows from the axiom of infinity that there exists a function which can be used to 'generate' an infinite sequence of distinct values of type *ind*. This axiom merely asserts the existence of such a function; the first step in representing the natural numbers is to define a constant $\mathsf{S}{:}\mathit{ind}{\rightarrow}\mathit{ind}$ which in fact *denotes* this function. This can be done using ε, as discussed in Section 2.1.5. The result is the following theorem about S:

$$\vdash (\forall x_1\, x_2.\, (\mathsf{S}\ x_1 = \mathsf{S}\ x_2) \supset (x_1 = x_2)) \wedge \neg(\forall y.\, \exists x.\, y = \mathsf{S} x)$$

Once S has been defined, a constant $\mathsf{Z}{:}\mathit{ind}$ can be defined which denotes a value not in the image of S. From this value Z, an infinite sequence of distinct individuals can then be generated by repeated application of S. The definition of Z simply uses the primitive constant ε to choose an arbitrary value not in the image of S:

$$\vdash \mathsf{Z} = \varepsilon\ \lambda y{:}\mathit{ind}.\, \forall x.\, \neg(y = \mathsf{S}\ x)$$

From this definition of Z, and the theorem about S shown above, it follows that Z is not in the image of S and that S is one-to-one. Formally:

$$\begin{aligned} &\vdash \forall i.\, \neg(\mathsf{S}\ i = \mathsf{Z}) \\ &\vdash \forall i_1\, i_2.\, (\mathsf{S}\ i_1 = \mathsf{S}\ i_2) \supset (i_1 = i_2) \end{aligned} \tag{A.4}$$

By the informal argument given in the introduction to this section, these two theorems imply that the individuals denoted by Z, $\mathsf{S}(\mathsf{Z})$, $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$, $\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{Z})))$, ... form an infinite sequence of distinct values, and can therefore be used to represent the type *num* of natural numbers.

To make a type definition for *num*, a predicate $\mathsf{N}{:}\mathit{ind}{\rightarrow}\mathit{bool}$ must be defined which is true of just those individuals in this infinite sequence. This can be done by defining N to be true of the values of type *ind* in the *smallest* subset of individuals which contains Z and is closed under S. The formal definition of N is:

$$\vdash \mathsf{N}\ i = \forall P{:}\mathit{ind}{\rightarrow}\mathit{bool}.\, P\ \mathsf{Z} \wedge (\forall x.\, P\ x \supset P(\mathsf{S}\ x)) \supset P\ i$$

This definition states that N is true of a value $i{:}\mathit{ind}$ exactly when i is an element of *every* subset of *ind* which contains Z and is closed under S. This means that the subset of *ind* defined by N is the *smallest* such set and therefore contains just those individuals obtainable from Z by zero or more applications of S.

From the definition of N, it is easy to prove the following three theorems:

$$\begin{array}{l} \vdash \mathsf{N}\ \mathsf{Z} \\ \vdash \forall i.\, \mathsf{N}\ i \supset \mathsf{N}(\mathsf{S}\ i) \\ \vdash \forall P.\, (P\ \mathsf{Z} \wedge \forall i.\, (P\ i \supset P(\mathsf{S}\ i)) \supset \forall i.\, \mathsf{N}\ i \supset P\ i \end{array} \tag{A.5}$$

The first two of these theorems state that the subset of *ind* defined by N contains Z and is closed under the function S. The third theorem states that the subset of *ind* defined by N is the smallest such set. That is, any set of individuals containing Z and closed under S has the set of individuals specified by N as a subset.

Using the predicate N, the type constant *num* can be defined by introducing a type definition axiom of the usual form. From the theorem $\vdash \mathsf{N}\ \mathsf{Z}$, it follows immediately that $\vdash \exists i.\, \mathsf{N}\ i$. The following type definition axiom for the type *num* can therefore be introduced:

$$\vdash \exists f{:}\mathit{num}{\rightarrow}\mathit{ind}.(\forall a_1\, a_2. f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{N}\ r = (\exists a.\, r = f\ a))$$

and the usual abstraction and representation functions

$$\mathsf{ABS\_num}{:}\mathit{ind}{\rightarrow}\mathit{num} \quad \text{and} \quad \mathsf{REP\_num}{:}\mathit{num}{\rightarrow}\mathit{ind}$$

for mapping between values of type *num* and their representations of type *ind* can defined as described in Section A.1.

A.3.1.2 Deriving the Axiomatization of *num*

The natural numbers are often axiomatized by the Peano's postulates. The five theorems labelled (A.4) and (A.5) in the previous section amount to a formulation of Peano's postulates for the natural numbers *represented by individuals.* It is therefore straightforward to derive Peano's postulates for the type *num* from these corresponding theorems about the subset of *ind* specified by N.

The first step in deriving the Peano postulates for *num* is to define the two constants:

$$0{:}num \quad \text{and} \quad \mathsf{Suc}{:}num{\to}num,$$

which denote the number zero and the successor function on natural numbers. Using the abstraction and representation functions ABS_num and REP_num, the constants 0 and Suc can be defined as follows:

$$\vdash 0 = \mathsf{ABS\_num\ Z}$$
$$\vdash \mathsf{Suc}\ n = \mathsf{ABS\_num}(\mathsf{S}(\mathsf{REP\_num}\ n))$$

From these definitions, the five theorems labelled (A.4) and (A.5), and the fact that the abstraction and representation functions ABS_num and REP_num are isomorphisms, it is easy to prove the abstract axiomatization of *num*, consisting of the three Peano postulates shown below:

$$\vdash \forall n.\, \neg(\mathsf{Suc}\ n = 0)$$
$$\vdash \forall n_1\, n_2.\, \mathsf{Suc}\ n_1 = \mathsf{Suc}\ n_2 \supset n_1 = n_2$$
$$\vdash \forall P.\, (P\ 0 \wedge \forall n.\, P\ n \supset P(\mathsf{Suc}\ n)) \supset \forall n.\, P\ n$$

The first of Peano's postulates shown above states that zero is not the successor of any natural number. This theorem follows immediately from the corresponding theorem $\vdash \forall i.\, \neg(\mathsf{S}\ i = \mathsf{Z})$ derived in the previous section for the representing values of type *ind*. Likewise, the second of Peano's postulates, which states that Suc is one-to-one, follows from the corresponding theorem about S. The third postulate states the validity of mathematical induction on natural numbers; it follows from the last of three theorems (A.5) derived in the previous section.

A.3.1.3 The Primitive Recursion Theorem

Once Peano's postulates have been proved, all the usual properties of the natural numbers can be derived from them. One important property is that functions can be uniquely defined on the natural numbers by primitive recursion. This is stated by the primitive recursion theorem, shown below:

$$\vdash \forall x f.\, \exists! fn.\, (fn\ 0 = x) \wedge \forall n.\, fn\ (\mathsf{Suc}\ n) = f\ (fn\ n)\ n \tag{A.6}$$

This theorem states that a function $fn{:}num{\to}\alpha$ can be *uniquely* defined by primitive recursion—i.e. by specifying a value for x to define the value of $fn(0)$ and an expression f to define the value of $fn(\mathsf{Suc}\ n)$ recursively in terms of $fn(n)$ and n. The proof of this theorem will not be given here, but an outline of the proof can be found in Gordon's paper [29]. The proof of a similar theorem for a logical type of *trees* is given in Section A.4.1.3.

An important property of primitive recursion theorem is that it is equivalent to Peano's postulates for *num*. The single theorem (A.6) can therefore be used as the abstract axiomatization of the defined type *num*, instead of the three separate theorems expressing Peano's postulates. In Section A.5.2, it will be shown how any concrete recursive type can be axiomatized in higher order logic by a similar 'primitive recursion' theorem.

As discussed in Section 2.1.6.1 of Chapter 2, any function definition by primitive recursion on the natural numbers can be justified formally in logic by appropriately specializing the variables x and f in theorem (A.6).

A.3.2 Finite-length Lists

This section describes the definition of a recursive type $(\alpha)list$ of lists containing values of type α. In principle, it is possible to represent this type by a subset of some *primitive* compound type. In practice, however, it is easier to use the defined type constant *num* and the type operator $\times$ (defined above in Section A.2.2). The representation using *num* and $\times$ described below is based on the construction of lists in [29].

A.3.2.1 The Representation and Type Definition

Lists are simply finite sequences of values, all of the same type. A list with n values of type α will be represented by a pair (f, n), where f is a function of type $num{\to}\alpha$ and n is a value of type *num*. The idea is that the function f will give the sequence of values in the list; $f(0)$ will be the first value, $f(1)$ will be the second value, and so on. The second component of a pair (f, n) representing a list will be a number n giving the length of the list represented.

The set of values used to represent lists can not be simply the set of all pairs of type $(num{\to}\alpha) \times num$. The pairs used must be restricted so that each list has a *unique* representation. The one-element list '[42]', for example, will be represented by a pair $(f, 1)$, where $f(0){=}42$. But there are an infinite number of different functions $f{:}num{\to}num$ that satisfy the equation $f(0){=}42$. To make the representation of '[42]' unique, some 'standard' value must be chosen for the value of $f(m)$ when $m > 0$. The predicate $\mathsf{Is\_list\_REP}$ defined below uses the standard value $(\varepsilon\ \lambda x{:}\alpha.\mathsf{T})$ to specify a set of pairs which contains a unique representation for each list:

$$\vdash \mathsf{Is\_list\_REP}(f, n) = \forall m.\, m \geq n \supset (f\ m = \varepsilon\ \ \lambda x{:}\alpha.\mathsf{T})$$

With this representation, there is exactly one pair (f, n) for each finite-length list of values of type α. If such a pair satisfies $\mathsf{Is\_list\_REP}$, then for $m < n$ the value of $f(m)$ will be the corresponding element of the list represented. For $m \geq n$, the value of $f(m)$ will be the standard value $(\varepsilon\ \lambda x.\mathsf{T})$.

It is easy to prove that $\vdash \exists f\, n.\, \mathsf{Is\_list\_REP}(f, n)$, since $\mathsf{Is\_list\_REP}$ holds of the pair $((\lambda n.\, \varepsilon\lambda x.\mathsf{T}), 0)$. A type definition axiom of the usual form can therefore be introduced for the type $(\alpha)list$:

$$\begin{array}{l}\vdash \exists f{:}(\alpha)list{\rightarrow}((num{\rightarrow}\alpha) \times num). \\ \qquad (\forall a_1\, a_2. f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_list\_REP}\ r = (\exists a.\, r = f\ a))\end{array}$$

and the abstraction and representation functions:

$$\begin{array}{l}\mathsf{ABS\_list}{:}((num{\rightarrow}\alpha) \times num){\rightarrow}(\alpha)list \quad \text{and} \\ \mathsf{REP\_list}{:}(\alpha)list{\rightarrow}((num{\rightarrow}\alpha) \times num)\end{array}$$

can be defined based on the type definition axiom in the usual way.

A.3.2.2 Deriving the Axiomatization of $(\alpha)list$

The abstract axiomatization of lists will be based on two constructors:

$$\mathsf{Nil} : (\alpha)list \qquad \text{and} \qquad \mathsf{Cons} : \alpha{\rightarrow}(\alpha)list{\rightarrow}(\alpha)list.$$

The constant Nil denotes the empty list. The function Cons constructs lists in the usual way: if h is a value of type α and t is a list then Cons h t denotes the list with head h and tail t. The definition of Nil is simple:

$$\vdash \mathsf{Nil} = \mathsf{ABS\_list}((\lambda n{:}num.\, \varepsilon\ \lambda x{:}\alpha.\, \mathsf{T}), 0)$$

This equation simply defines Nil to be the list whose representation is the pair $(f, 0)$, where $f(n)$ has the value $(\varepsilon\ \lambda x.\mathsf{T})$ for all n.

The constructor Cons can be defined by first defining a function Cons_REP which 'implements' the Cons-operation on list representations. The definition is:

$$\vdash \mathsf{Cons\_REP}\ h\ (f, n) = ((\lambda m.(m{=}0 \Rightarrow h \mid f(m-1))),\ n+1)$$

The function Cons_REP takes a value h and pair (f, n) representing a list and yields the representation of the result of inserting 'h at the head of the represented list. This result is a pair whose first component is a function yielding value h when applied to 0 (the head of the resulting list) and the value given by $f(m{-}1)$ when applied to m for all $m{>}0$ (the tail of the resulting list). The second component is the length $n{+}1$, one greater than the length of the input list representation.

Once Cons_REP has been defined, it is easy to define Cons. The definition is:

$$\vdash \mathsf{Cons}\ h\ t = \mathsf{ABS\_list}(\mathsf{Cons\_REP}\ h\ (\mathsf{REP\_list}\ t))$$

The function Cons defined by this equation simply takes a value h and a list t, maps t to its representation, computes the representation of the desired result using $\mathsf{Cons\_REP}$, and then maps that result back to the corresponding abstract list.

Once Nil and Cons have been defined, the following abstract axiom for lists can be derived by formal proof:

$$\vdash \forall x\, f.\, \exists! fn.\, (fn(\mathsf{Nil}) = x) \wedge (\forall h\, t.\, fn(\mathsf{Cons}\; h\; t) = f\; (fn\; t)\; h\; t) \tag{A.7}$$

This axiom is analogous to the primitive recursion theorem for natural numbers, and is an example of the general form of the theorems which will be used in Section A.5 to characterize all concrete recursive types. Once this theorem has been derived from the type definition axiom for lists and the definitions of Cons and Nil, all the usual properties of lists follow without further reference to the way lists are defined.

The axiom (A.7) for lists follows from the type definition for $(\alpha)list$. Full details will not be given here, but the proof is comparatively simple. The existence of the function fn in theorem (A.7) follows by demonstrating the existence of a corresponding function on list representations. This function can be defined by primitive recursion on the length component of the representation by using the primitive recursion theorem (A.6) for natural numbers. The uniqueness of the function fn in the abstract axiom for lists can then be proved by mathematical induction on the length component of list representations.

A.3.2.3 Theorems about $(\alpha)list$

Once the abstract axiom (A.7) for lists has been proved, the following theorems about lists can be derived from it:

$$\vdash \forall h\, t.\, \neg(\mathsf{Nil} = \mathsf{Cons}\; h\; t)$$
$$\vdash \forall h_1\, h_2\, t_1\, t_2.\, (\mathsf{Cons}\; h_1\; t_1 = \mathsf{Cons}\; h_2\; t_2) \supset ((h_1 = h_2) \wedge (t_1 = t_2))$$
$$\vdash \forall P.\, (P\; \mathsf{Nil} \wedge \forall t.\, P\; t \supset \forall h.\, P(\mathsf{Cons}\; h\; t)) \supset \forall l.\, P\; l$$

These three theorems are analogous to Peano's postulates for the natural numbers, which were derived in Section A.3.1.2. The first theorem states that Nil is not equal to any list constructed by Cons; the second theorem states that Cons is one-to-one; and the third theorem asserts the validity of structural induction on lists.

A.4 Two Recursive Types of Trees

This section describes the formal definitions of two different logical types which denote sets of trees. First, a type *tree* is defined which denotes the set of all trees whose nodes can branch any (finite) number of times. This type is then used to define a second logical type of trees, $(\alpha)Tree$, which denotes the set of *labelled* trees. These have the same sort of structure as values of type *tree*, but they also have a label of type α associated with each node.

The type $(\alpha)Tree$ defined in this section is of interest because each logical type in the class of recursive types discussed in Section A.5 can be represented by some subset of it. Once the type of labelled trees has been defined, it can be used (along with the type *one* and the type operators $\times$ and $+$) to construct systematically a representation for any concrete recursive type. This avoids the problem of having to find an *ad hoc* representation for each recursive type, and so makes it possible to mechanize efficiently the formal definition of such types.

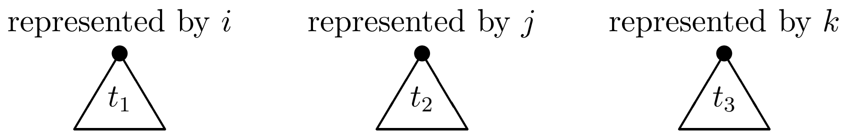
A.4.1 The Type of Trees: *tree*

Values of the logical type *tree* defined in this section will be finite trees whose internal nodes can branch any finite number of times. These trees will be *ordered*. That is, the relative order of each node's immediate subtrees will be important, and two similar trees which differ only in the order of their subtrees will be considered to be different trees.

A.4.1.1 The Representation and Type Definition

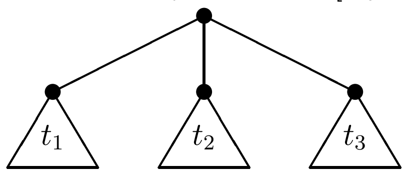
Trees will be represented by coding them as natural numbers; each tree will be represented by a unique value of type *num*. The smallest possible tree consists of a single leaf node with no subtrees; it will be represented by the number 0. To represent a tree with one or more subtrees, a function node_REP:$(num)list{\rightarrow}num$ will be defined which computes the natural number representing such a tree from a list of the numbers which represent its subtrees. The function node_REP will take as an argument a list l of numbers. If each of the numbers in the list represents a tree, then node_REP l will represent the tree whose subtrees are represented by the numbers in l.

Consider, for example, a tree with three subtrees: t_1, t_2, and t_3. Suppose that the three subtrees t_1, t_2, and t_3 are represented by the natural numbers i, j, and k respectively:



The number representing the tree which has t_1, t_2, and t_3 as subtrees will then be denoted by node_REP$[i;j;k]$:

represented by $\mathsf{node\_REP}[i;j;k]$



where the conventional list notation $[i;j;k]$ is a syntactic abbreviation for the list denoted by $\mathsf{Cons}\ i\ (\mathsf{Cons}\ j\ (\mathsf{Cons}\ k\ \mathsf{Nil}))$.

Since node_REP takes a *list* of numbers as arguments, it can be used to compute the code for a tree with any finite number of immediate subtrees. Thus, using node_REP, the natural number representing a tree of any shape can be computed recursively from the natural numbers representing its subtrees. The only property that node_REP must have for this to work is the property of being a one-to-one function on lists of numbers:

$$\vdash \forall l_1\, l_2.\, (\mathsf{node\_REP}\ l_1 = \mathsf{node\_REP}\ l_2) \supset (l_1 = l_2) \tag{A.8}$$

This theorem asserts that if node_REP computes the same natural number from two lists l_1 and l_2, then these lists must be equal and therefore must consist of the same finite sequence of numbers. If node_REP has this property, then it can be used to compute a *unique* numerical representation for every possible tree. It remains to define the function node_REP such that theorem (A.8) holds.

One way of formally defining node_REP is to use the well-known coding function $(n, m) \longmapsto (2n + 1) \times 2^m$ which codes a pair of natural numbers by a single natural number. Using this coding function, node_REP can be defined by recursion on lists such that the following two theorems hold:

$$\begin{array}{lcl} \vdash \mathsf{node\_REP}\ \mathsf{Nil} & = & 0 \\ \vdash \mathsf{node\_REP}\ (\mathsf{Cons}\ n\ t) & = & ((2 \times n) + 1) \times (2\ \mathsf{Exp}\ (\mathsf{node\_REP}\ t)) \end{array} \tag{A.9}$$

These two equations define the value of $\mathsf{node\_REP}\ l$ by 'primitive recursion' on the list l. When l is the empty list Nil, the result is 0. When l is a non-empty list with head n and tail t, the result is computed by coding as a single natural number the pair consisting of n and the result of applying node_REP recursively to t. Primitive recursive definitions of this kind can be justified by formal proof using the abstract axiom (A.7) for lists derived in Section A.3.2.2. The two theorems (A.9) can be derived from an appropriate instance of this axiom and a non-recursive definition of the constant node_REP.

Theorem (A.8) stating that node_REP is one-to-one can be derived from the two theorems (A.9) which define node_REP by primitive recursion. The proof is done by structural induction on the lists l_1 and l_2 using the theorem shown in Section A.3.2.3 stating the validity of proofs by induction on lists.

The function node_REP can be used to compute a natural number to represent any finitely branching tree. To make a type definition for the type constant *tree*, a predicate on natural numbers $\mathsf{Is\_tree\_REP}{:}num{\to}bool$ must be defined which is true of just those numbers representing trees. This predicate will be defined in the same way as the corresponding predicate was defined in Section A.3.1.1 for the representation of numbers by individuals: $\mathsf{Is\_tree\_REP}\ n$ will be true if the number n is in the smallest set of natural numbers closed under node_REP.

The formal definition of Is_tree_REP uses the auxiliary function Every, defined recursively on lists as follows:

$$\begin{aligned} &\vdash \mathsf{Every}\ P\ \mathsf{Nil} &&= \mathsf{T} \\ &\vdash \mathsf{Every}\ P\ (\mathsf{Cons}\ h\ t) &&= (P\ h) \wedge \mathsf{Every}\ P\ t \end{aligned}$$

These two theorems define $\mathsf{Every}\ P\ l$ to mean that the predicate P holds of every element of the list l. Using Every, the predicate Is_tree_REP is defined as follows:

$$\vdash \mathsf{Is\_tree\_REP}\ n = \forall P.\, (\forall tl.\, \mathsf{Every}\ P\ tl \supset P(\mathsf{node\_REP}\ tl)) \supset P\ n$$

This definition states that a number n represents a tree exactly when it is an element of every subset of *num* which is closed under node_REP. It follows that the set of numbers for which Is_tree_REP is true is the smallest set closed under node_REP. This set contains just those natural numbers which can be computed using node_REP and therefore contains only those numbers which represent trees.

To use Is_tree_REP to define a new type, the theorem $\vdash \exists n.\, \mathsf{Is\_tree\_REP}\ n$ must first be proved. This theorem follows immediately from the fact that Is_tree_REP is true of 0, i.e. the number denoted by node_REP Nil. Once this theorem has been proved, a type definition axiom of the usual form can be introduced:

$$\begin{aligned} &\vdash \exists f{:}tree{\to}num. \\ &\qquad (\forall a_1\, a_2. f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_tree\_REP}\ r = (\exists a.\, r = f\ a)) \end{aligned}$$

along with the usual abstraction and representation functions:

$$\mathsf{ABS\_tree}{:}num{\to}tree \quad \text{and} \quad \mathsf{REP\_tree}{:}tree{\to}num.$$

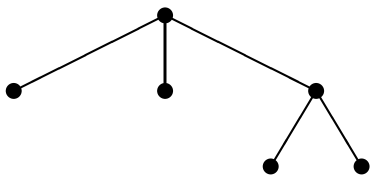
A.4.1.2 The Axiomatization of *tree*

The abstract axiom for *tree* will be based on the constructor:

$$\mathsf{node}{:}(tree)list{\to}tree$$

The function node builds trees from smaller trees. If $tl{:}(tree)list$ is a list of trees, then the term node tl denotes the tree whose immediate subtrees are the trees in the list tl. If tl is the empty list of trees, then node tl denotes the tree consisting of a single leaf node. Using node, it is possible to construct a tree of any shape.

For example, the tree:



is denoted by the expression: $\mathsf{node}[\mathsf{node\ Nil};\ \mathsf{node\ Nil};\ \mathsf{node}[\mathsf{node\ Nil};\ \mathsf{node\ Nil}]\,]$.

An auxiliary function Map will be used in the definition of the constructor node. The function Map is the usual mapping function for lists; it takes a function $f{:}\alpha{\to}\beta$ and a list $l{:}(\alpha)list$ and yields the result of applying f to each member of l in turn. The recursive definition of Map is:

$$\begin{array}{lcl} \vdash \mathsf{Map}\ f\ \mathsf{Nil} & = & \mathsf{Nil} \\ \vdash \mathsf{Map}\ f\ (\mathsf{Cons}\ h\ t) & = & \mathsf{Cons}\ (f\ h)\ (\mathsf{Map}\ f\ t) \end{array}$$

Using Map and the function $\mathsf{node\_REP}{:}(num)list{\to}num$ defined in the previous section, the formal definition in logic of node is:

$$\vdash \mathsf{node}\ tl = (\mathsf{ABS\_tree}(\mathsf{node\_REP}(\mathsf{Map}\ \ \mathsf{REP\_tree}\ \ tl)))$$

The constructor node defined by this equation takes a list of trees tl, applies node_REP to the corresponding list of numbers representing the trees in tl, and then maps the result to the corresponding abstract tree.

The following two important theorems follow from the formal definition of node given above; they are analogous to the Peano postulates for the natural numbers, and are used to prove the abstract axiom for the type *tree*:

$$\begin{array}{l} \vdash \forall tl_1\ tl_2.\,(\mathsf{node}\ tl_1 = \mathsf{node}\ tl_2) \supset (tl_1 = tl_2) \\ \vdash \forall P.\,(\forall tl.\ \mathsf{Every}\ P\ tl \supset P\ (\mathsf{node}\ tl)) \supset \forall t.\,P\ t \end{array}$$

The first of these theorems states that the constructor node is one-to-one. This follows directly from theorem (A.8), which states that the corresponding function node_REP is one-to-one. The second theorem shown above asserts the validity of induction on trees, and can be used to justify proving properties of trees by structural induction. This theorem can be proved from the definitions of node and Is_tree_REP and the fact that ABS_tree and REP_tree are isomorphisms relating trees and the numbers that represent them.

The abstract axiomatization of the defined type *tree* consists of the single theorem shown below:

$$\vdash \forall f.\,\exists! fn.\,\forall tl.\,fn(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fn\ tl)\ tl \tag{A.10}$$

This theorem is analogous to the primitive recursion theorem (A.6) for natural numbers and the abstract axiom (A.7) for lists. It asserts the unique existence of functions defined recursively on trees. The universally quantified variable f ranges over functions that map a list of values of type α and a list of trees to a value of type α. For any such function, there is a unique function $fn{:}tree{\to}\alpha$ that satisfies the equation $fn(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fn\ tl)\ tl$. For any tree $(\mathsf{node}\ tl)$, this equation defines the value of $fn(\mathsf{node}\ tl)$ recursively in terms of the result of applying fn to each of the immediate subtrees in the list tl.

A.4.1.3 An Outline of the Proof of the Axiom for *tree*

It is straightforward to prove the uniqueness part of the abstract axiom for trees; the uniqueness of the function fn in theorem (A.10) follows by structural induction on trees using the induction theorem for the defined type *tree*. The existence part of theorem (A.10) is considerably more difficult to prove. It follows from a slightly weaker theorem in which the list of subtrees tl is not an argument to the universally quantified function f:

$$\vdash \forall f.\ \exists fn.\ \forall tl.\ fn(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fn\ tl) \tag{A.11}$$

This weaker theorem can be proved by first defining a height function Ht on trees and then proving that, for any number n, there exists a function fun which satisfies the desired recursive equation for trees whose height is bounded by n:

$$\vdash \forall f\, n.\ \exists fun.\ \forall tl.\ (\mathsf{Ht}(\mathsf{node}\ tl) \leq n) \supset (fun(\mathsf{node}\ tl) = f\ (\mathsf{Map}\ fun\ tl))$$

The main step in the proof of this theorem is an induction on n.

This theorem can be used to define a higher order function fun which yields approximations of the function fn whose existence is asserted by theorem (A.11). For any n and f, the term $(\mathsf{fun}\ n\ f)$ denotes an approximation of fn which satisfies the recursive equation in theorem (A.11) for trees whose height is no greater than n. This is stated formally by the following theorem:

$$\begin{aligned}\vdash \forall f\, n\, tl.\ &(\mathsf{Ht}(\mathsf{node}\ tl) \leq n) \supset \\ &(\mathsf{fun}\ n\ f\ (\mathsf{node}\ tl) = f\ (\mathsf{Map}\ (\mathsf{fun}\ n\ f)\ tl))\end{aligned} \tag{A.12}$$

The approximations constructed by fun have the following important property: for any two numbers n and m, the corresponding functions constructed by fun behave the same for trees whose height is bounded by both n and m. This property follows by structural induction on trees, and is stated formally by the theorem shown below.

$$\vdash \forall t\ n\ m\ f.\ (\mathsf{Ht}\ t){<}n \wedge (\mathsf{Ht}\ t){<}m \supset (\mathsf{fun}\ n\ f\ t = \mathsf{fun}\ m\ f\ t) \tag{A.13}$$

Theorem (A.11) asserts the existence of a function fn for any given f. The higher order function fun can be used to construct this function explicitly from the given function f. For any f, the term $\lambda t.\, \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}\ [t]))\ f\ t$ denotes the function which satisfies the desired recursive equation. An outline of the proof of this is as follows. Specializing f, n, and tl in theorem (A.12) to f, $\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl])$, and tl respectively yields the following implication:

$$\begin{aligned}\vdash\ & \mathsf{Ht}(\mathsf{node}\ tl) \leq \mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]) \supset \\ & \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f\ (\mathsf{node}\ tl) = f(\mathsf{Map}\ (\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f)\ tl)\end{aligned}$$

The height function Ht has the property: $\vdash \forall t.\, \mathsf{Ht}\ t \leq \mathsf{Ht}(\mathsf{node}\ [t])$. The antecedent of the implication shown above is therefore always true, and the theorem can be simplified to:

$$\vdash \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f\ (\mathsf{node}\ tl) = f(\mathsf{Map}\ (\mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f)\ tl)$$

The property of fun expressed by theorem (A.13) implies that the above theorem is equivalent to:

$$\vdash \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[\mathsf{node}\ tl]))\ f\ (\mathsf{node}\ tl) = f(\mathsf{Map}\ (\lambda t.\, \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[t]))\ f\ t)\ tl)$$

which is itself equivalent (by β-reduction) to:

$$\vdash (\lambda t.\, \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[t]))f\ t)(\mathsf{node}\ tl) = f(\mathsf{Map}\ (\lambda t.\, \mathsf{fun}\ (\mathsf{Ht}(\mathsf{node}[t]))\ f\ t)\ tl)$$

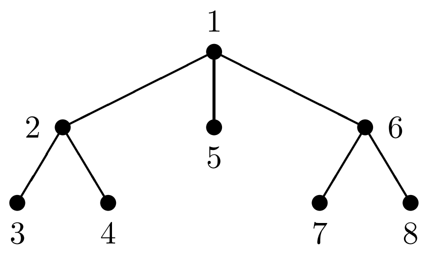
Theorem (A.11) follows immediately from this last result. The slightly stronger abstract axiom for *tree*, theorem (A.10), then follows from theorem (A.11) by a relatively straightforward formal proof.

A.4.2 The Type of Labelled Trees: $(\alpha)Tree$

This section outlines the definition of the type $(\alpha)Tree$ which denotes the set of labelled trees. Labelled trees of the kind defined in this section have the same sort of general structure as values of the logical type *tree* defined in the previous section. The only difference is that a tree of type $(\alpha)Tree$ has a value or 'label' of type α associated with each of its nodes. It is therefore comparatively simple to define the type $(\alpha)Tree$, since the values of the structurally similar type *tree* can be used in its representation.

A.4.2.1 The Representation and Type Definition

The representation of a labelled tree of type $(\alpha)Tree$ will be a pair (t, l), where t is a value of type *tree* giving the shape of the tree being represented and l is a list of type $(\alpha)list$ containing the values associated with its nodes. The values in the list l will occur in the sequence which corresponds to a *preorder traversal* of the labelled tree being represented. Consider, for example, the labelled tree shown below:



This tree has a natural number associated with each node and can be represented by a pair (t, l) of type $tree \times (num)list$. The first component t of this pair will be the value of type *tree* whose structure corresponds to the above picture. The second component l will be a list of length eight containing the numbers associated with the nodes of the corresponding labelled tree. The numbers in this list will occur in the order $[1; 2; 3; 4; 5; 6; 7; 8]$, corresponding to a preorder traversal of the labelled tree being represented.

Any α-labelled tree can be similarly represented by a pair of type $tree \times (\alpha)list$, but not every such pair represents a tree. For a pair (t, l) to represent a labelled tree, the length of the list l must be the same as the number of nodes in the tree t. This can be expressed in logic by defining two functions:

$$\mathsf{Length}{:}(\alpha)list{\rightarrow}num \quad \text{and} \quad \mathsf{Size}{:}tree{\rightarrow}num$$

which compute the length of a list and the number of nodes in a tree, respectively. The function Length can be defined recursively by using the abstract axiom (A.7) for lists to derive the following two equations:

$$\begin{aligned} &\vdash \mathsf{Length\ Nil} &&= 0 \\ &\vdash \mathsf{Length\ (Cons}\ h\ t) &&= (\mathsf{Length}\ t) + 1 \end{aligned}$$

The function Size can be defined by first defining a recursive function on lists $\mathsf{Sum}{:}(num)list{\rightarrow}num$ which computes the sum of a list of natural numbers:

$$\begin{aligned} &\vdash \mathsf{Sum\ Nil} &&= 0 \\ &\vdash \mathsf{Sum\ (Cons}\ n\ l) &&= n + (\mathsf{Sum}\ l) \end{aligned}$$

and then using the abstract axiom (A.10) for the defined type *tree* to derive the following recursive definition of Size:

$\vdash \mathsf{Size}(\mathsf{node}\ tl) = (\mathsf{Sum}(\mathsf{Map}\ \mathsf{Size}\ tl)) + 1$

Using the functions Length and Size, the values of type $tree \times (\alpha)list$ that represent labelled trees can be specified by the predicate Is_Tree_REP defined as follows:

$\vdash \mathsf{Is\_Tree\_REP}(t, l) = (\mathsf{Length}\ l = \mathsf{Size}\ t)$

This predicate is true of just those pairs (t, l) where the number of nodes in the tree t equals the length of the list l. It is therefore true of precisely those values of type $tree \times (\alpha)list$ which can be used to represent labelled trees.

For any value $v{:}\alpha$, the predicate Is_Tree_REP holds of the pair: ($\mathsf{node}\ \mathsf{Nil}$, $[v]$). From this, it immediately follows that $\vdash \exists p.\, \mathsf{Is\_Tree\_REP}\ p$. The following type definition axiom can therefore be introduced to define $(\alpha)Tree$:

$$\begin{aligned}&\vdash \exists f{:}(\alpha)Tree{\to}(tree \times (\alpha)list).\\ &\qquad (\forall a_1\, a_2. f\ a_1 = f\ a_2 \supset a_1 = a_2) \wedge (\forall r.\, \mathsf{Is\_Tree\_REP}\ r = (\exists a.\, r = f\ a))\end{aligned}$$

The associated abstraction and representation functions:

$\mathsf{ABS\_Tree}{:}(tree \times (\alpha)list){\to}(\alpha)Tree$ and
$\mathsf{REP\_Tree}{:}(\alpha)Tree{\to}(tree \times (\alpha)list)$

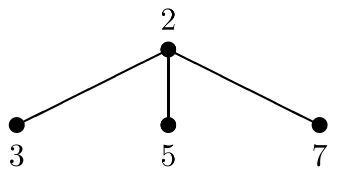
can then be defined in the usual way (as described in Section A.1).

A.4.2.2 Deriving the Axiomatization of $(\alpha)Tree$

The abstract axiom for $(\alpha)Tree$ is based on the constructor

$\mathsf{Node}{:}\alpha{\to}((\alpha)Tree)list{\to}(\alpha)Tree$

which is analogous to the constructor node for $tree$. If v is a value of type α, and l is a list of labelled trees, then the term $(\mathsf{Node}\ v\ l)$ denotes the labelled tree whose immediate subtrees are those occurring in l and whose root node is labelled by the value v. The function Node can be used to construct labelled trees of any shape. For example, the tree:



is denoted by the term: $\mathsf{Node}\ 2\ [\mathsf{Node}\ 3\ \mathsf{Nil};\ \mathsf{Node}\ 5\ \mathsf{Nil};\ \mathsf{Node}\ 7\ \mathsf{Nil}]$.

The definition of Node uses an auxiliary function $\mathsf{Flat}{:}((\alpha)list)list{\to}(\alpha)list$, which takes a list of lists and yields the result of appending them all together into a single list. The recursive definition of Flat is:

$$\begin{aligned} &\vdash \mathsf{Flat\ Nil} &&= \mathsf{Nil} \\ &\vdash \mathsf{Flat}\ (\mathsf{Cons}\ h\ t) &&= \mathsf{Append}\ h\ (\mathsf{Flat}\ t) \end{aligned}$$

where Append is defined (also recursively) by:

$$\begin{aligned} &\vdash \mathsf{Append\ Nil}\ l &&= l \\ &\vdash \mathsf{Append}\ (\mathsf{Cons}\ h\ l_1)\ l_2 &&= \mathsf{Cons}\ h\ (\mathsf{Append}\ l_1\ l_2) \end{aligned}$$

Using Flat, and the function Map defined above in Section A.4.1.2, the formal definition of the constructor Node is given by the following theorem:

$$\begin{aligned} \vdash \mathsf{Node}\ v\ l = \mathsf{ABS\_Tree}((&\mathsf{node}(\mathsf{Map}\ (\mathsf{Fst\ o\ REP\_Tree})\ l)), \\ &((\mathsf{Cons}\ v\ (\mathsf{Flat}(\mathsf{Map}\ (\mathsf{Snd\ o\ REP\_Tree})\ l))))) \end{aligned}$$

This definition uses REP_Tree to obtain the representation of each labelled tree in the list l. This yields a list of pairs representing labelled trees. The function node is then used to construct a new tree whose subtrees are the tree components in this list of pairs, and Flat is used to construct the corresponding list of node-values. The result is then mapped back to an abstract labelled tree using the abstraction function ABS_Tree.

Using the constructor Node $v\ l$ defined above, the abstract axiom for $(\alpha)Tree$ can be written:

$$\vdash \forall f.\, \exists! fn.\, \forall v\, tl.\, fn(\mathsf{Node}\ v\ tl) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl \tag{A.14}$$

This theorem is of the same general form as theorem (A.10), the abstract axiom for the defined type *tree*. It states the uniqueness of functions defined by 'primitive recursion' on labelled trees. The proof of this theorem is straightforward, but it requires some tricky (and uninteresting) lemmas involving the partitioning of lists. Details of the proof will therefore not be given here. The general strategy of the proof is to use the abstract axiom for values of type *tree* to define a recursive function on *representations* which 'implements' the function *fn* asserted to exist by the axiom (A.14).

A.5 Automating Recursive Type Definitions

This section outlines a method for formally defining any simple concrete recursive type in higher order logic. This method has been used to implement an efficient derived inference rule in HOL which defines such recursive types automatically. The input to this derived rule is a user-supplied informal[3] specification of the recursive type to be defined. This type specification is written in a notation

[3]In this context, *informal* means not in the language of higher order logic.

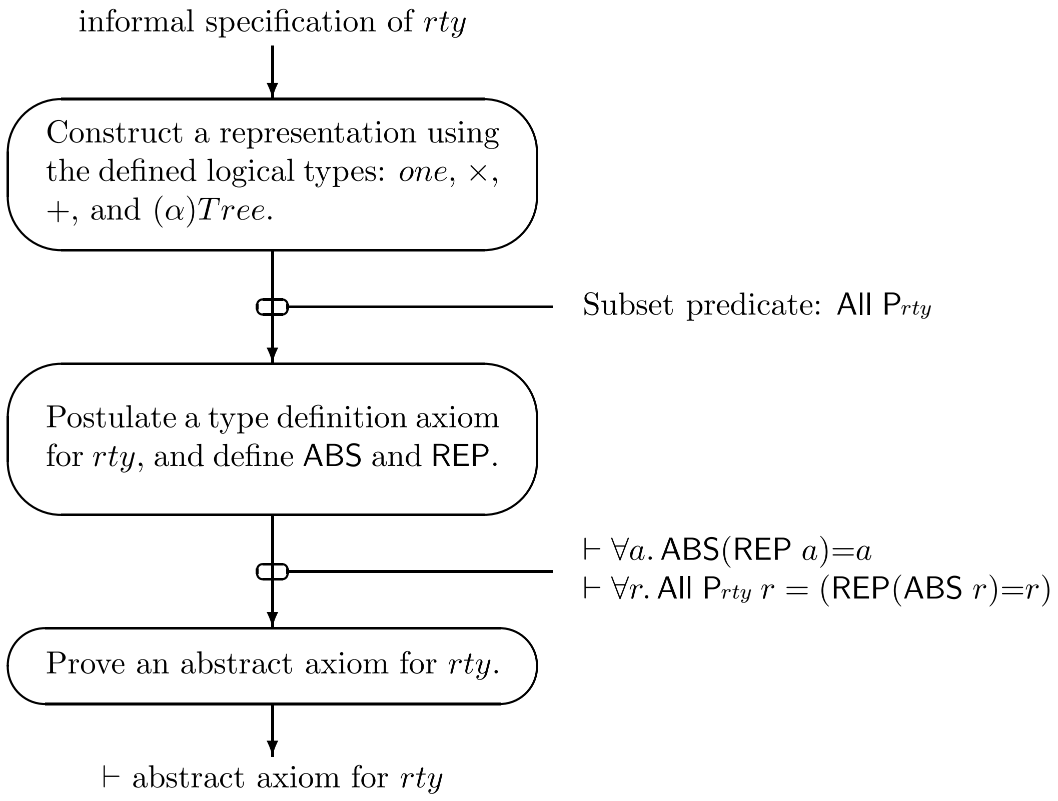

informal specification of *rty*
Construct a representation using the defined logical types: *one*, ×, +, and (α)*Tree*.
Subset predicate: All P_{rty}
Postulate a type definition axiom for *rty*, and define ABS and REP.
$\vdash \forall a.\ \mathsf{ABS}(\mathsf{REP}\ a)=a$
$\vdash \forall r.\ \mathsf{All}\ \mathsf{P}_{rty}\ r = (\mathsf{REP}(\mathsf{ABS}\ r)=r)$
Prove an abstract axiom for *rty*.
⊢ abstract axiom for *rty*

Figure A.1: Defining a Recursive Type *rty*.

which resembles a data type declaration in functional programming languages like Standard ML [46]. It simply states the names of the new type's constructors and the logical types of their arguments. The output is a theorem of higher order logic which abstractly characterizes the properties of the desired recursive type—i.e. a derived 'abstract axiomatization' of the type.

An overview of the algorithm used by this programmed inference rule to define a recursive type is shown Figure A.1. A concrete recursive type *rty* is defined and axiomatized by this algorithm in three steps. In the first step, an appropriate representation is found for the values of the recursive type *rty* to be defined. This representation is always some subset of a substitution instance of $(\alpha)Tree$—i.e. a subset of some type $(ty)Tree$ of general trees labelled by values of type *ty*. The type *ty* of labels for these trees is built up systematically using the type constant *one* and the type operators × and +. The output of this stage is a 'subset predicate' which defines the set of labelled trees used to represent values of the new type *rty*. This predicate has the standard form: '$\mathsf{All}\ \mathsf{P}_{rty}$', where P_{rty} is a predicate whose exact form is determined by the specification of the type to be defined. (The meaning of 'All' is explained below in Section A.5.3.2.) No logical inference needs to be done in this step, so the ML code which implements it in the HOL system is quite fast.

In the second step, a type definition axiom is introduced for the new type, based on the subset predicate All P_{rty}. The associated abstraction and representation functions ABS and REP are then defined and proved to be isomorphisms between the new type *rty* and the set of values specified by All P_{rty}. The output of this stage consists of the two theorems about ABS and REP shown in Figure A.1. The proofs done in this step are easy and routine (see Section A.1), and their mechanization in HOL is therefore efficient and straightforward.

In the final step, an abstract axiom for the new type *rty* is derived by formal proof from the definition of the subset predicate All P_{rty} and the two theorems about ABS and REP proved in the previous stage. This is the only step in the algorithm where a non-trivial amount of logical inference has to be done. The ML implementation of this step therefore uses the 'optimization' strategy for HOL derived inference rules discussed in Section 2.2.3: a pre-proved general theorem about recursive types is used to reduce to a minimum the amount of inference that has to be done at 'run time' to derive the desired result. This pre-proved theorem has the form shown below:

$$\vdash \forall P.\ \cdots\ \langle \beta \text{ is isomorphic to 'All } P\text{'}\rangle \supset \langle \text{abstract axiom for } \beta \rangle$$

Informally, this theorem states that *any* type β which is represented by a set of labelled trees 'All P' satisfies an abstract axiomatization of the required form. By specializing P in this theorem to the predicate P_{rty} constructed in the first step, the abstract axiom for *rty* follows simply by modus ponens (using the theorems about ABS and REP derived in the second step) and a relatively small amount of straightforward simplification.

A detailed description of the HOL implementation of this algorithm for defining recursive types is beyond the scope of this appendix, but the sections which follow give an overview of the logical basis for this implementation. In Section A.5.1, the syntax of informal type specifications is described, and some simple examples are given of type specifications written in this notation. Section A.5.2 describes the general form of the abstract axioms that are derived by the system, and Section A.5.3 explains how appropriate representations for these types can be systematically constructed from their informal type specifications. Finally, in Section A.5.4 a general theorem is given which states that any recursive type represented in the way described in Section A.5.3 satisfies an abstract axiom of the form shown in Section A.5.2. An example of the application of this theorem is also given.

A.5.1 Informal Type Specifications[4]

Every logical type which can be defined by the method outlined in the following sections can be described informally by a *type specification* of the following general form:

$$(\alpha_1,\ldots,\alpha_n)rty \quad ::= \quad \mathsf{C}_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \quad \cdots \quad | \quad \mathsf{C}_m\ ty_m^1\ \ldots\ ty_m^{k_m}$$

where each ty_i^j is either an existing logical type (not containing rty) or is the type expression $(\alpha_1,\ldots,\alpha_n)rty$ itself. This equation specifies a type $(\alpha_1,\ldots,\alpha_n)rty$ with n type variables $\alpha_1,\ldots,\alpha_n$ where $n \geq 0$. If $n = 0$ then rty is a type constant; otherwise rty is an n-ary type operator. The type specified has m distinct constructors C_1, ..., C_m where $m \geq 1$. Each constructor C_i takes k_i arguments, where $k_i \geq 0$; and the types of these arguments are given by the type expressions ty_i^j for $1 \leq j \leq k_i$. If one or more of the type expressions ty_i^j is the type $(\alpha_1,\ldots,\alpha_n)rty$ itself, then the equation specifies a *recursive* type. In any specification of a recursive type, at least one constructor must be non-recursive—i.e. all its arguments must have types which already exist in the logic.

The logical type specified by the equation shown above denotes the set of all values which can be finitely generated using the constructors C_1, ..., C_m, where each constructor is one-to-one and any two different constructors yield different values. Every value of this logical type is denoted by some term of the form:

$$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$$

where x_i^j is a term of logical type ty_i^j for $1 \leq j \leq k_i$. In addition, any two terms:

$$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i} \qquad \text{and} \qquad \mathsf{C}_j\ x_j^1\ \ldots\ x_j^{k_j}$$

denote equal values exactly when their constructors are the same (i.e. $i = j$) and these constructors are applied to equal arguments (i.e. $x_i^n = x_j^n$ for $1 \leq n \leq k_i$).

A.5.1.1 Some Examples of Type Specifications

The two simple recursive types *num* and $(\alpha)list$ which were defined in Section A.3 are both examples of types that can be described by type specifications of the general form described above.

The specification of the type *num* of natural numbers is the simple equation shown below:

$$num \quad ::= \quad 0 \quad | \quad \mathsf{Suc}\ num$$

[4]Some of the notation used in this section is adapted from Bird and Wadler's clear description of the syntax of type definitions in their excellent book [5] on functional programming.

This equation specifies the type constant *num* to have two constructors: $0{:}num$ and $\mathsf{Suc}{:}num\to num$. The type *num* which is described by this type specification denotes the smallest set of values generated from the constant 0 by zero or more applications of the constructor Suc—i.e. the set of values denoted by terms of the form: 0, Suc(0), Suc(Suc(0)), ... etc.

The type specification for the type $(\alpha)list$ of finite lists is similar to the one given above for *num*. It is:

$$(\alpha)list \quad ::= \quad \mathsf{Nil} \quad | \quad \mathsf{Cons}\ \alpha\ \ (\alpha)list$$

This equation states that the type $(\alpha)list$ denotes the set of all values generated by the two constructors: $\mathsf{Nil}{:}(\alpha)list$ and $\mathsf{Cons}{:}\alpha\to(\alpha)list\to(\alpha)list$.

A slightly more complex example is the recursive type *btree*, described by the type specification shown below:

$$btree \quad ::= \quad \mathsf{Leaf}\ \ num \quad | \quad \mathsf{Tree}\ \ btree\ \ btree$$

This equation specifies a type of *binary trees* whose leaf nodes (but not internal nodes) are labelled by natural numbers. When defined formally, this type will have two constructors: $\mathsf{Leaf}{:}num\to btree$ and $\mathsf{Tree}{:}btree\to btree\to btree$. The function Leaf constructs leaf nodes; if n is a value of type *num*, then ($\mathsf{Leaf}\ n$) denotes a leaf node labelled by n. The constructor Tree builds binary trees from smaller binary trees; if t_1 and t_2 are binary trees then ($\mathsf{Tree}\ t_1\ t_2$) denotes the binary tree with left subtree t_1 and right subtree t_2.

In addition to recursive types, simple enumerated and 'record' types can also be specified by equations of the form described above. For example, the type constant *one* and the two type operators *prod* and *sum*, whose formal definitions were given in Section A.2, can be informally specified by the three equations shown below:

$$\begin{array}{rcl} one & ::= & \mathsf{one} \\ (\alpha,\beta)prod & ::= & \mathsf{pair}\ \ \alpha\ \ \beta \\ (\alpha,\beta)sum & ::= & \mathsf{Inl}\ \ \alpha \quad | \quad \mathsf{Inr}\ \ \beta \end{array}$$

The first of these specifications simply states that *one* is the enumerated type with exactly one value: the value denoted by the constant one. The second specification states that every value of type $(\alpha,\beta)prod$ is denoted by some term of the form ($\mathsf{pair}\ a\ b$), i.e. an ordered pair with first component $a{:}\alpha$ and second component $b{:}\beta$. The third equation states that every value of type $(\alpha,\beta)sum$ is either a left injection constructed by Inl or a right injection constructed by Inr.

Many more examples of types—both recursive and non-recursive—which can be specified by equations of the form discussed in this section can be found in books on functional programming. (See, for example, Chapter 8 of [5].)

A.5.2 Formulating Abstract Axioms for Recursive Types

The input to the HOL programmed inference rule which defines types is, in general, an informal specification of the form:

$$(\alpha_1, \ldots, \alpha_n)rty \quad ::= \quad \mathsf{C}_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \quad \cdots \quad | \quad \mathsf{C}_m\ ty_m^1\ \ldots\ ty_m^{k_m}$$

Each type $(\alpha_1, \ldots, \alpha_n)rty$ specified by an equation of this form can be abstractly characterized by a single theorem of higher order logic. This theorem is the output of the HOL derived rule for defining types and has the following general form:

$$\begin{array}{l} \vdash \forall f_1\ \cdots\ f_m.\ \exists! fn{:}(\alpha_1, \ldots, \alpha_n)rty{\to}\beta. \\ \quad \forall x_1^1 \cdots x_1^{k_1}.\ fn(\mathsf{C}_1\ x_1^1\ \ldots\ x_1^{k_1}) = f_1\ (fn\ x_1^1)\ \ldots\ (fn\ x_1^{k_1})\ x_1^1\ \ldots\ x_1^{k_1}\ \wedge \\ \qquad\qquad\qquad \vdots \\ \quad \forall x_m^1 \cdots x_m^{k_m}.\ fn(\mathsf{C}_m\ x_m^1 \ldots x_m^{k_m}) = f_m\ (fn\ x_m^1) \ldots (fn\ x_m^{k_m})\ x_m^1 \ldots x_m^{k_m} \end{array} \tag{A.15}$$

where the right hand sides of the equations include recursive applications $(fn\ x_i^j)$ of the function fn only for variables x_i^j of type $(\alpha_1, \ldots, \alpha_n)rty$.

Theorem (A.15) states that for any m functions f_1, …, f_m there is a *unique* function fn which satisfies the 'primitive recursive' definition whose exact form is determined by the functions f_1, …, f_m. This is an abstract characterization of the type $(\alpha_1, \ldots, \alpha_n)rty$: it states the essential properties of the type, but does so without reference to the way it is represented. It follows from this theorem that every value of type $(\alpha_1, \ldots, \alpha_n)rty$ is constructed by one of the constructors C_1, …, C_m, that each of these constructors is one-to-one, and that different constructors yield different values. The proof that theorem (A.15) implies these properties of $(\alpha_1, \ldots, \alpha_n)rty$ and the constructors C_1, …, C_m can be outlined as follows.

The fact that every value of type $(\alpha_1, \ldots, \alpha_n)rty$ is constructed by one of the functions $\mathsf{C}_1, \ldots, \mathsf{C}_m$ follows from the uniqueness part of theorem (A.15). Suppose there is some value, v say, such that $v \neq (\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i})$ for $1 \leq i \leq m$. I.e. v is not constructed by any C_i. One could then define two functions f and g of type $(\alpha_1, \ldots, \alpha_n)rty{\to}bool$ which yield the boolean T for all values constructed by any constructor C_i:

$$\forall x_i^1 \cdots x_i^{k_i}.\ f(\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = g(\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}) = \mathsf{T} \qquad \text{for } 1 \leq i \leq m$$

and when applied to v yield different results: $f\ v = \mathsf{T}$ and $g\ v = \mathsf{F}$. If f and g are defined this way then $f \neq g$, since $f\ v \neq g\ v$. But from the uniqueness part of theorem (A.15) it follows that if f and g have the property shown above, then $f = g$. Therefore no such value v exists, and every value of type $(\alpha_1, \ldots, \alpha_n)rty$ is constructed by some C_i.

The fact that the constructors C_1, …, C_m are one-to-one can be proved by using theorem (A.15) to define a 'destructor' function D_i for each C_i such that:

$$\vdash \mathsf{D}_i(\mathsf{C}_i\, x_i^1\ \ldots\ x_i^{k_i}) = (x_i^1, \ldots, x_i^{k_i})$$

For each constructor C_i, the corresponding destructor function D_i can be defined by appropriately specializing the quantified variable f_i in theorem (A.15). From the property of the destructor D_i shown above, it is then easy to prove that:

$$\vdash (\mathsf{C}_i\, x_i^1\ \ldots\ x_i^{k_i}) = (\mathsf{C}_i\, y_i^1\ \ldots\ y_i^{k_i}) \supset (x_i^1 = y_i^1 \wedge \cdots \wedge x_i^{k_i} = y_i^{k_i})$$

which states that C_i is one-to-one, as desired.

Finally, the fact that different constructors yield different values can be proved by appropriately specializing the universally quantified functions $f_1, \ldots, f_m$ in theorem (A.15) to obtain a theorem asserting the proposition shown below:

$$\vdash \exists fn. \forall x_i^1 \cdots x_i^{k_i}.\, fn(\mathsf{C}_i\, x_i^1\ \ldots\ x_i^{k_i}) = i \qquad \text{for } 1 \leq i \leq m$$

This states the existence of a function fn which yields the natural number i when applied to values constructed by the ith constructor. This means that any two different constructors C_i and C_j yield different values of type $(\alpha_1, \ldots, \alpha_n)rty$, since applying fn to these values gives different natural numbers.

Using theorems of the form illustrated by (A.15) to axiomatize recursive types is closely related to the initial algebra approach to the theory of abstract data types [28]. This approach is very elegant from a theoretical point of view, but it is also of *practical* value in the HOL mechanization of recursive type definitions. Each recursive type is characterized by a single theorem, and all the theorems which characterize such types have the same general form. This uniform treatment of recursive types is the basis for the *efficient* automation of their construction in HOL. It allows the axiom for any recursive type to be quickly derived from a pre-proved theorem stating that axioms of this kind hold for *all* such types. Furthermore, it makes it possible to derive useful standard properties of recursive types (e.g. structural induction) in a uniform way, with relatively short formal proofs and therefore by efficient programmed inference rules.

A.5.3 Constructing Representations for Recursive Types

This section outlines a method by which a representation can be found for any type specified informally by an equation of the form described in Section A.5.1. Each representation is an appropriately-defined subset of a type constructed using the type constant *one*, the type operators $\times$ and $+$, and the type $(\alpha)Tree$. A simple example is first given in Sections A.5.3.1 and A.5.3.2; the method for finding representations in general is then outlined in Section A.5.3.3.

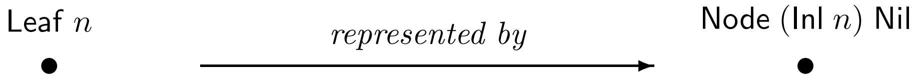
A.5.3.1 An Example: the Representation of Binary Trees

Consider the type *btree* described above in Section A.5.1.1. This type was specified informally by:

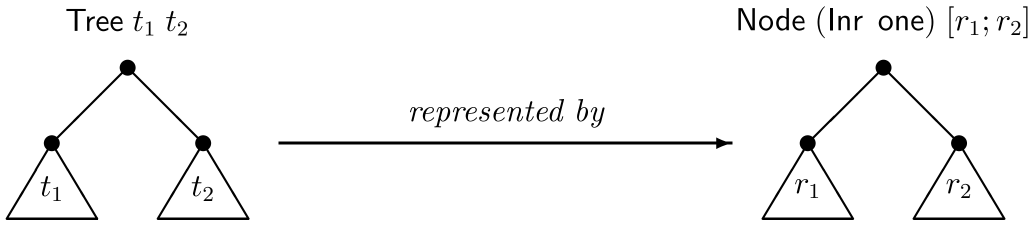
$$btree \quad ::= \quad \mathsf{Leaf}\ num \quad | \quad \mathsf{Tree}\ btree\ btree$$

The type *btree* specified by this equation can be represented in higher order logic by a subset of the set denoted by the compound type $(num + one)Tree$. This type denotes the set of all trees (of any shape) whose nodes are labelled either by a value of type *num* or by the single value one of type *one*. The idea of this representation is that each binary tree t of type *btree* is represented by a corresponding tree of type $(num + one)Tree$ which has both the same shape as t and the same labels on its nodes as t.

Consider, for example, the binary tree ($\mathsf{Leaf}\ n$), consisting of a single leaf node labelled by the natural number n. This binary tree will be represented by a leaf node of type $(num + one)Tree$ labelled by the left injection ($\mathsf{Inl}\ n$):



A binary tree ($\mathsf{Tree}\ t_1\ t_2$) which is not a leaf node, but has two subtrees t_1 and t_2, will be represented by a tree of type $(num + one)Tree$ which also has two subtrees and is labelled by the right injection ($\mathsf{Inr}\ \mathsf{one}$):



where r_1 and r_2 are the representations of the two binary trees t_1 and t_2. The 'dummy' value ($\mathsf{Inr}\ \mathsf{one}$) is used here to label the root node of the representation, since the binary tree which is being represented has no value associated with its root node.

A.5.3.2 Defining the Subset Predicate for *btree*

To introduce a type definition axiom for *btree*, a predicate Is_btree_REP must first be defined which is true of just those values of type $(num + one)Tree$ which represent binary trees using the scheme outlined above. This predicate is defined formally by building it up from two auxiliary predicates: Is_Leaf and Is_Tree. These two auxiliary predicates correspond to the two kinds of binary trees which will be represented, and each one states what the representation of the corresponding kind of binary tree looks like.

The predicates $\mathsf{Is\_Leaf}$ and $\mathsf{Is\_Tree}$ are defined as follows. Every value in the representation is a tree of the form ($\mathsf{Node}\ v\ tl$), where v is a label of logical type ($num + one$) and tl is a list of subtrees. If such a tree represents a leaf node ($\mathsf{Leaf}\ n$), then the label v must be the value ($\mathsf{Inl}\ n$) and the list tl must be empty. These conditions are expressed formally by the predicate $\mathsf{Is\_Leaf}$, defined as follows:

$$\vdash \mathsf{Is\_Leaf}\ v\ tl = (\exists n.\, v = \mathsf{Inl}\ n \wedge \mathsf{Length}\ tl = 0)$$

If ($\mathsf{Node}\ v\ tl$) represents a binary tree ($\mathsf{Tree}\ t_1\ t_2$) with two subtrees, then the list of subtrees tl must have length two, and the label v must be the value ($\mathsf{Inr\ one}$). The definition of $\mathsf{Is\_Tree}$ is therefore:

$$\vdash \mathsf{Is\_Tree}\ v\ tl = (v = \mathsf{Inr\ one} \wedge \mathsf{Length}\ tl = 2)$$

The two predicates $\mathsf{Is\_Leaf}$ and $\mathsf{Is\_Tree}$ state what kind of values v and tl must be for the tree ($\mathsf{Node}\ v\ tl$) to be the *root* node of legal binary-tree representation. But if a general tree of type $(num + one)Tree$ in fact represents a binary tree, then not only its root node but every node it contains (i.e. all its subtrees) must also satisfy either $\mathsf{Is\_Leaf}$ or $\mathsf{Is\_Tree}$. This can be expressed formally in logic by first defining a higher order function All recursively on trees as follows:

$$\vdash \mathsf{All}\ P\ (\mathsf{Node}\ v\ tl) = P\ v\ tl \wedge \mathsf{Every}\ (\mathsf{All}\ P)\ tl$$

Using All, the predicate $\mathsf{Is\_btree\_REP}$ can then be defined such that it is true of a tree t exactly when the label and subtree list of every node in t satisfies either $\mathsf{Is\_Leaf}$ or $\mathsf{Is\_Tree}$. The definition of $\mathsf{Is\_btree\_REP}$ is simply:

$$\vdash \mathsf{Is\_btree\_REP}\ t = \mathsf{All}\ \ (\lambda v.\, \lambda tl.\, \mathsf{Is\_Leaf}\ v\ tl\ \vee\ \mathsf{Is\_Tree}\ v\ tl)\ \ t$$

This predicate exactly specifies the subset of $(num + one)Tree$ whose values represent binary trees, and can therefore be used to introduce a type definition axiom for the new type *btree* in the usual way. All the predicates which specify representations of recursive type are defined using All in exactly the way shown above for $\mathsf{Is\_btree\_REP}$.

A.5.3.3 Finding Representations in General

The representation of binary trees by a subset of $(num+one)Tree$ illustrates the general method for finding representations of any type specified by an equation of the form described in Section A.5.1. In general, a recursive type specified by an equation of this kind denotes a set of labelled *trees* with a fixed number of different kinds of nodes. Any such type can therefore be represented by a subset of values denoted by some instance of the defined type $(\alpha)Tree$ of general trees.

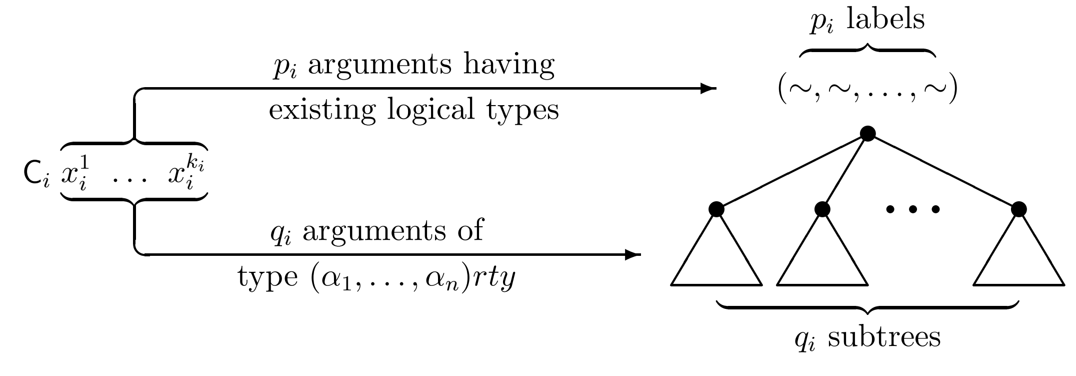

p_i labels
$(\sim, \sim, \ldots, \sim)$
p_i arguments having existing logical types
$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$
q_i arguments of type $(\alpha_1, \ldots, \alpha_n)rty$
q_i subtrees

Figure A.2: The Representation of a value $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$.

Suppose, for example, that $(\alpha_1, \ldots, \alpha_n)rty$ is specified by:

$$(\alpha_1, \ldots, \alpha_n)rty \quad ::= \quad \mathsf{C}_1\ ty_1^1\ \ldots\ ty_1^{k_1} \quad | \quad \cdots \quad | \quad \mathsf{C}_m\ ty_m^1\ \ldots\ ty_m^{k_m}$$

This equation specifies a type with m different kinds of values, corresponding to the m constructors C_1, ..., C_m. When this type is defined formally in higher order logic, each of its values will be denoted by some term of the form:

$$\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$$

where C_i is a constructor and each argument x_i^j is a value of type ty_i^j for $1 \leq j \leq k_i$. In the general case of a recursive type, some of the k_i arguments to C_i will have existing logical types and some will have the type $(\alpha_1, \ldots, \alpha_n)rty$ itself. Let p_i be the number of arguments which have existing logical types and let q_i be the number of arguments which have type $(\alpha_1, \ldots, \alpha_n)rty$, where $k_i = p_i + q_i$. The abstract value of type $(\alpha_1, \ldots, \alpha_n)rty$ denoted by $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$ can be represented by a tree which has q_i subtrees and p_i values associated with its root node.

This representation scheme is illustrated the diagram shown in Figure A.2. In the general case illustrated by this diagram, the tree representing $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$ is labelled by p_i-tuple of values. Each of these values is one of the p_i arguments to C_i which are not of type $(\alpha_1, \ldots, \alpha_n)rty$ but have types which already exist in the logic. When $p_i = 0$, the representing tree is labelled not by a tuple but by the constant one (as was done for the constructor Tree of *btree*). And when $p_i = 1$ the representing tree is labelled simply by a single value of the appropriate type (as was done for the constructor Leaf of *btree*). The q_i subtrees shown in the diagram are the representations of the arguments to C_i which have the type $(\alpha_1, \ldots, \alpha_n)rty$. If $q_i = 0$ then the representing tree has no subtrees.

Each of the m kinds of values constructed by C_1, ..., C_m can be represented by a tree using the scheme outlined above. In general, a value obtained using the

ith constructor C_i will be represented by a tree labelled by a tuple of p_i values. The representing type for $(\alpha_1, \ldots, \alpha_n)rty$ will therefore be a type expression of the form:

$$(\overbrace{(\underbrace{ty \times \cdots \times ty}_{\text{product of } p_1 \text{ types}}) + \cdots + (\underbrace{ty \times \cdots \times ty}_{\text{product of } p_m \text{ types}})}^{\text{sum of } m \text{ products}})Tree$$

where the ty's are the existing logical types occurring in the equation which specifies the new type $(\alpha_1, \ldots, \alpha_n)rty$ being defined.

Using this scheme, a predicate $\mathsf{Is\_rty\_REP}$ can be defined to specify a set of trees to represent $(\alpha_1, \ldots, \alpha_n)rty$ in exactly the same way as the predicate $\mathsf{Is\_btree\_REP}$ was defined for the representation of *btree*. The definition of $\mathsf{Is\_rty\_REP}$ will have the form:

$$\vdash \mathsf{Is\_rty\_REP}\ t = \mathsf{All}\ \ (\lambda v.\, \lambda tl.\, \mathsf{Is\_C}_1\ v\ tl\ \vee\ \cdots\ \vee\ \mathsf{Is\_C}_m\ v\ tl)\ \ t$$

where each $\mathsf{Is\_C}_i$ is an auxiliary predicate specifying which trees represent values constructed by the corresponding constructor C_i. The ith auxiliary predicate $\mathsf{Is\_C}_i$ is defined as follows. When $i \neq m$, the definition is:

$$\vdash \mathsf{Is\_C}_i\ v\ tl = \exists x_1 \ldots x_{p_i}.\, v = \mathsf{Inl}(\underbrace{\mathsf{Inr}\cdots(\mathsf{Inr}}_{i-1\ \mathsf{Inr}\text{'s}}(x_1, \ldots, x_{p_i}))\cdots) \wedge \mathsf{Length}\ tl = q_i$$

where p_i is the number of arguments to C_i which have existing logical types, and q_i is the number of arguments of type $(\alpha_1, \ldots, \alpha_n)rty$. This definition states that if a tree ($\mathsf{Node}\ v\ tl$) represents a value $\mathsf{C}_i\ x_i^1\ \ldots\ x_i^{k_i}$ then it must have the right number subtrees in tl and its label v must be an appropriate injection of some p_i-tuple (of the right logical type, of course). When $i = m$, the definition is similar:

$$\vdash \mathsf{Is\_C}_m\ v\ tl = \exists x_1 \ldots x_{p_m}.\, v = (\underbrace{\mathsf{Inr}\cdots(\mathsf{Inr}}_{m-1\ \mathsf{Inr}\text{'s}}(x_1, \ldots, x_{p_m}))\cdots) \wedge \mathsf{Length}\ tl = q_m$$

The only difference is that the last injection applied is Inr, not Inl.

A.5.4 Deriving Abstract Axioms for Recursive Types

The *uniform* treatment of representations for recursive types makes it possible to write an efficient HOL derived inference rule which proves abstract axioms for them. Every representation is some subset '$\mathsf{All}\ P$' of an instance of $(\alpha)Tree$. A general theorem can therefore be formulated stating that an abstract axiom of the required form holds for *any* recursive type represented this way. This theorem can then be simply 'instantiated' to obtain an abstract axiom for any particular recursive type.

The theorem stating that every recursive type satisfies an abstract axiom of the desired form is shown below:

$$\begin{array}{l}\vdash \forall P.\ \forall Abs{:}(\alpha)Tree{\rightarrow}\beta.\ \forall Rep{:}\beta{\rightarrow}(\alpha)Tree. \\ \quad (\forall a.\ Abs(Rep\ a){=}a \wedge \forall r.\ \mathsf{All}\ P\ r = (Rep(Abs\ r){=}r)) \supset \\ \quad \forall f.\ \exists!\ fn.\ \forall v\ tl.\ P\ v\ (\mathsf{Map}\ Rep\ tl) \supset \\ \qquad\qquad\qquad fn(Abs(\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{array} \tag{A.16}$$

Informally, this theorem states that any type β which is represented by (i.e. is isomorphic to) a set '$\mathsf{All}\ P$' of trees satisfies an abstract axiom of the form described in Section A.5.2. Theorem A.16 makes this assertion in form of an implication:

$$\vdash \forall P.\ \cdots\ \langle\beta \text{ is isomorphic to '}\mathsf{All}\ P\text{'}\rangle \supset \langle\text{abstract axiom for } \beta\rangle$$

where the antecedent of this implication is written formally as follows:

$$\forall a.\ Abs(Rep\ a){=}a \wedge \forall r.\ \mathsf{All}\ P\ r = (Rep(Abs\ r){=}r)$$

This simply says that β is isomorphic to the set of trees of type $(\alpha)Tree$ which satisfy $\mathsf{All}\ P$. The type variable β stands for the new recursive type which is represented by $\mathsf{All}\ P$, and the variables Abs and Rep are the abstraction and representation functions for β.

The conclusion of theorem (A.16) states that functions can be uniquely defined by 'primitive recursion' on the structure which β inherits from $\mathsf{All}\ P$. That is, for any f, there is a unique function $fn{:}\beta{\rightarrow}\gamma$ which satisfies the recursive equation:

$$fn(Abs(\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl$$

whenever the condition $P\ v\ (\mathsf{Map}\ Rep\ tl)$ holds of v and tl. This condition on v and tl restricts the recursive equation shown above to apply only to 'well-constructed' values of type β. If $P\ v\ (\mathsf{Map}\ Rep\ tl)$ holds, then $\mathsf{All}\ P$ is true of the value $\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl)$ on the left hand side of the equation. The corresponding *abstract* value, denoted by:

$$Abs(\mathsf{Node}\ v\ (\mathsf{Map}\ Rep\ tl)),$$

will then be a correctly-represented value of type β. The example which is given in Section A.5.4.1 below shows how the form of the predicate P in the condition $P\ v\ (\mathsf{Map}\ Rep\ tl)$ determines the final 'shape' of the resulting axiom.

Theorem (A.16) illustrates the expressive power which higher-order variables and type polymorphism give to higher order logic. The variable P in this theorem ranges (essentially) over all predicates on $(\alpha)Tree$. And the two type variables α and β can be instantiated to any two logical types. Theorem (A.16) therefore asserts that an abstract axiom holds for *any* recursive type, since any such type is isomorphic to an appropriate subset $\mathsf{All}\ P$ of some instance of (α)Tree. Because general results like theorem (A.16) can be formulated as theorems in the logic, they can be used to make programmed inference rules in HOL efficient. Derived inference rules can use such pre-proved general theorems to avoid having to do costly 'run time' inference. Theorem (A.16) is used in this way by the derived rule which automates recursive type definitions.

The example given in the following section shows how this derived rule uses the general theorem (A.16) to prove the abstract axiom for a particular recursive type.

A.5.4.1 Example: Deriving the Axiom for *btree*

The example given in this section is the proof of the abstract axiom for *btree*, the type whose representation was described in Section A.5.3.2. The following is the sequence of main steps which the HOL system carries out to define *btree* and derive an abstract axiom for it:

(1) Define the subset predicate $\mathsf{Is\_btree\_REP}$, introduce a type definition axiom for *btree*, and define the associated abstraction and representation functions $\mathsf{ABS}{:}(num + one)Tree{\to}btree$ and $\mathsf{REP}{:}btree{\to}(num + one)Tree$.

This is done as outlined in Sections A.5.3.2 and A.1. The result of this step is the two theorems shown below:

$$\begin{array}{l}\vdash \forall a.\, \mathsf{ABS}(\mathsf{REP}\ a) = a \\ \vdash \forall r.\, \mathsf{All}\ \mathsf{Is\_btree\_REP}\ r = (\mathsf{REP}(\mathsf{ABS}\ r) = r)\end{array}$$

These theorems simply state that the newly-introduced type constant *btree* denotes a set of values which is isomorphic to the subset of $(num + one)Tree$ defined by $\mathsf{All\ Is\_btree\_REP}$.

(2) Use theorem (A.16) to obtain an (unsimplified) abstract axiom for *btree*.

If the type variables α and β in theorem (A.16) are instantiated to the types $(num + one)$ and *btree* respectively, then the universally quantified variables P, Abs, and Rep can be specialized to the terms $\mathsf{Is\_btree\_REP}$, ABS, and REP. The resulting instance of theorem (A.16) is an implication whose antecedent matches the two theorems about ABS and REP derived in the previous step. The theorem shown below therefore follows simply by modus ponens (and rewriting, with the definition of $\mathsf{Is\_btree\_REP}$):

$$\begin{array}{l}\vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, (\mathsf{Is\_Leaf}\ v\ (\mathsf{Map\ REP}\ tl) \vee \mathsf{Is\_Tree}\ v\ (\mathsf{Map\ REP}\ tl)) \supset \\ \qquad\qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{array}$$

This theorem expresses the essence of the desired abstract axiom for *btree*. The remaining steps carried out by the system are sequence of straightforward simplifications of this theorem which put it into the desired final form.

(3) Remove the disjunction: $\mathsf{Is\_Leaf}\ v\ (\mathsf{Map\ REP}\ tl) \vee \mathsf{Is\_Tree}\ v\ (\mathsf{Map\ REP}\ tl)$.

The theorem derived in the previous step contains a term which has the form $\forall v\, tl.\, (P \vee Q) \supset R$. By a simple proof in predicate calculus, this term is equivalent to the conjunction: $(\forall v\, tl.\, P \supset R) \wedge (\forall v\, tl.\, Q \supset R)$. The theorem derived in the previous step is therefore equivalent to:

$$\begin{array}{l} \vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, \mathsf{Is\_Leaf}\ v\ (\mathsf{Map\ REP}\ tl) \supset \\ \qquad\qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\ \ \wedge \\ \qquad\qquad \forall v\ tl.\, \mathsf{Is\_Tree}\ v\ (\mathsf{Map\ REP}\ tl) \supset \\ \qquad\qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl \end{array}$$

In the general case of a type with m constructors, the subset predicate will be a disjunction of the general form:

$$\mathsf{Is\_C}_1\ v\ (\mathsf{Map}\ \mathit{Rep}\ tl)\ \vee\ \cdots\ \vee\ \mathsf{Is\_C}_m\ v\ (\mathsf{Map}\ \mathit{Rep}\ tl)$$

When this step is done, it will introduce a conjunction of m implications in the body of the abstract axiom, each of which corresponds to one of the m constructors C_1, ..., C_m.

(4) Rewrite with the definitions of $\mathsf{Is\_Leaf}$ and $\mathsf{Is\_Tree}$. This yields:

$$\begin{array}{l} \vdash \forall f.\, \exists! fn.\, \forall v\ tl.\, (\exists n.\, v = \mathsf{Inl}\ n \wedge \mathsf{Length}(\mathsf{Map\ REP}\ tl) = 0) \supset \\ \qquad\qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\ \ \wedge \\ \qquad\qquad \forall v\ tl.\, (v = \mathsf{Inr\ one} \wedge \mathsf{Length}(\mathsf{Map\ REP}\ tl) = 2) \supset \\ \qquad\qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map\ REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl \end{array}$$

Note: In the HOL implementation, the predicates $\mathsf{Is\_Leaf}$ and $\mathsf{Is\_Tree}$ are not actually defined as new constants; they are instead written using λ-terms. This step therefore does not need to be done in the HOL implementation.

(5) Simplify terms of the form: $\mathsf{Length}(\mathsf{Map\ REP}\ tl) = m$.

A term of the form $\mathsf{Length}(\mathsf{Map\ REP}\ tl) = m$ is equivalent to a simplified term of the form $\mathsf{Length}\ tl = m$. This in turn is equivalent to saying that tl is equal to some list of m values: $\exists t_1 \ldots t_m.\, tl = [t_1; \ldots; t_m]$. The terms involving Length in the previous theorem can therefore be simplified, resulting in the following theorem:

$$\begin{array}{l}\vdash \forall f.\, \exists! fn.\, \forall v\; tl.\, (\exists n.\, v = \mathsf{Inl}\ n \wedge tl = \mathsf{Nil}) \supset \\ \qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map}\ \mathsf{REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\ \ \wedge \\ \qquad \forall v\; tl.\, (v = \mathsf{Inr\ one} \wedge \exists t_1\, t_2.\, tl = [t_1; t_2]) \supset \\ \qquad\qquad fn(\mathsf{ABS}(\mathsf{Node}\ v\ (\mathsf{Map}\ \mathsf{REP}\ tl))) = f\ (\mathsf{Map}\ fn\ tl)\ v\ tl\end{array}$$

This step introduces the variables t_1 and t_2. They range over values of type *btree* and occur in the axiom for *btree* in its final form.

(6) Remove equations of the form: $v = \cdots$ and $tl = \cdots$.

The antecedents of the two logical implications in the previous theorem both contain equations giving values for v and tl. These can be removed by using (a generalization of) the fact that in predicate calculus a term of the form $\forall y.\, (\exists x.\, y = tm_1[x]) \supset tm_2[y]$ is equivalent to $\forall x.\, tm_2[tm_1[x]]$. The result of removing the equations for v and tl is:

$$\begin{array}{l}\vdash \forall f.\, \exists! fn.\, \forall n.\, fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inl}\ n)(\mathsf{Map}\ \mathsf{REP}\ \mathsf{Nil}))) \\ \qquad\qquad\qquad = f\ (\mathsf{Map}\ fn\ \mathsf{Nil})\ (\mathsf{Inl}\ n)\ \mathsf{Nil} \quad \wedge \\ \qquad \forall t_1\, t_2.\, fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inr\ one})\ (\mathsf{Map}\ \mathsf{REP}\ [t_1; t_2]))) \\ \qquad\qquad\qquad = f\ (\mathsf{Map}\ fn\ [t_1; t_2])\ (\mathsf{Inr\ one})\ [t_1; t_2]\end{array}$$

The body of the theorem now consists of two equations. These define the value of fn for the two different kinds of binary trees.

(7) Rewrite with the definition of Map. This yields:

$$\begin{array}{l}\vdash \forall f.\, \exists! fn.\, \forall n.\, fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inl}\ n)\ \mathsf{Nil})) \\ \qquad\qquad\qquad = f\ \mathsf{Nil}\ (\mathsf{Inl}\ n)\ \mathsf{Nil} \quad \wedge \\ \qquad \forall t_1\, t_2.\, fn(\mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inr\ one})\ [\mathsf{REP}\ t_1; \mathsf{REP}\ t_2])) \\ \qquad\qquad\qquad = f\ [fn\ t_1; fn\ t_2]\ (\mathsf{Inr\ one})\ [t_1; t_2]\end{array}$$

(8) Define the abstract constructors Leaf and Tree as follows:

$$\begin{array}{lcl}\vdash \mathsf{Leaf}\ n & = & \mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inl}\ n)\ \mathsf{Nil}) \\ \vdash \mathsf{Tree}\ t_1\ t_2 & = & \mathsf{ABS}(\mathsf{Node}\ (\mathsf{Inr\ one})\ [\mathsf{REP}\ t_1; \mathsf{REP}\ t_2])\end{array}$$

The constructors Leaf and Tree defined by these equations first use Node to construct the representations of the required values and then use ABS to obtain the corresponding values of type *btree*. Rewriting the theorem derived in the previous step with these definitions yields:

$$\begin{array}{l}\vdash \forall f.\, \exists! fn.\, \forall n.\, fn(\mathsf{Leaf}\ n) = f\ \mathsf{Nil}\ (\mathsf{Inl}\ n)\ \mathsf{Nil}\ \ \wedge \\ \qquad \forall t_1\, t_2.\, fn(\mathsf{Tree}\ t_1\ t_2) = f\ [fn\ t_1; fn\ t_2]\ (\mathsf{Inr\ one})\ [t_1; t_2]\end{array}$$

(9) Introduce two functions f_1 and f_2 in place of f.

With an appropriate choice of value for the universally quantified variable f, two functions f_1 and f_2 can be introduced for the right hand sides of the two equations. These define the value of fn separately for the two constructors Leaf and Tree. Specializing f to the appropriate function, and simplifying, gives:

$$\begin{aligned}\vdash \forall f_1\, f_2.\, \exists! fn.\, \forall n.\, fn(\mathsf{Leaf}\ n) = f_1\ n\ \wedge \\ \forall t_1\, t_2.\, fn(\mathsf{Tree}\ t_1\ t_2) = f_2\ (fn\ t_1)\ (fn\ t_2)\ t_1\ t_2\end{aligned}$$

This theorem is the abstract axiom for *btree*—in its final form.

The HOL derived rule which automates recursive type definitions carries out the sequence of steps shown above for each informal type specification entered by the user. An appropriate instance of theorem (A.16) yields an 'unsimplified' abstract axiom for the type being defined. This axiom is then systematically transformed into the form described in Section A.5.2 by the sequence of simple equivalence-preserving steps shown above. The amount of actual logical inference that must be carried out is relatively small, and each step is a straightforward transformation of the theorem derived in the previous step. The HOL implementation of this procedure is therefore both efficient and robust.

A.6 Concluding Remarks

The method for defining recursive types described in this appendix is the logical basis for a set of efficient theorem-proving tools which have been implemented in the HOL system. In addition to the derived inference rule which automates recursive type definitions, a number of related tools have been implemented in HOL for generating proofs involving recursive types. These include:

1. a derived inference rule for proving the validity of structural induction on concrete recursive types, and related tools for generating proofs by structural induction (e.g. a general structural induction tactic),
2. a set of rules which automate the inference necessary to define functions by 'primitive recursion' on recursive types,
3. derived rules which prove that the constructors of recursive types are one-to-one and yield distinct values, and
4. tools for generating interactive proofs by case analysis on the constructors of recursive types.

An example HOL session, which illustrates the use of these tools, is given in Appendix B. Preliminary work is underway to extend these tools to deal with types defined by mutual recursion, and types with equational constraints (i.e. simple 'abstract' data types).

Defining a logical type in HOL is rarely the primary goal of the user of the system, but often a necessary part of some more interesting proof. The efficient automation of type definitions in HOL is therefore of significant practical value, since defining types 'by hand' in the system is tedious and tricky. The method for automating type definitions described here allows new recursive types to be introduced by the HOL user quickly and easily. This is made possible by the systematic construction of representations for these types, the uniform treatment of abstract axioms for them (using essentially the initial algebra approach to data type specifications), and the expressive power of higher order logic itself.

Appendix B

An Example HOL Session

This appendix contains an example interactive HOL session that shows how the theorem proving tools developed for reasoning about concrete recursive types can be used to prove some of the theorems about hardware given in Chapter 5. Considerable knowledge of the HOL system is needed for a complete understanding of this session, but a full explanation of HOL is not given here. The reader who is interested in the details should consult [30] and the HOL manual.

The session has been edited slightly to help make it comprehensible to a reader who is not familiar with the HOL system. In the machine-readable syntax for terms used in HOL, for example, the universal and existential quantifiers are written '`!`' and '`?`' respectively. In the edited session given below, these ascii symbols for quantifiers have been replaced by the usual non-alphabetic symbols '$\forall$' and '$\exists$'. Other changes made to the actual session conducted with the system are likewise restricted to purely cosmetic matters.

The interactions with the HOL system that follow should be understood as having taken place in sequence. Input typed by the user is preceded by the prompt '`#`' and terminated by two semicolons '`;;`'. The remaining lines show the response from the system. Logical terms are enclosed within double quotes: `"`*<term>*`"`. Formal theorems generated during the session are preceded by the turnstile symbol '`|-`'. The ML `let` construct is used to bind theorems and other ML objects to variables for future use. For example, typing `let v = "x < y";;` binds the ML variable `v` to the term `"x < y"`. A brief commentary is given to explain the interactions that occur with the system. The session itself begins on the next page.

Defining a concrete type automatically

The ML function for defining a concrete type automatically is `define_type`. This function takes two ML strings as arguments. The first is just a name under which the results of the definition are stored on disk. The second is a user-supplied 'grammar' for the logical type which is to be defined. In the example given below, the type of binary trees discussed in Chapter 5 is defined automatically using this function. The result returned by the call to `define_type` is an abstract characterization for the defined type `btree`, in the form of a primitive recursion theorem for the type of binary trees. This is proved automatically by `define_type` from an automatically-constructed formal definition of the type constant `btree`. The theorem is saved for future use by binding it to the ML variable `btree_thm`.

```
#let btree_thm = define_type `btree` `btree = LEAF | NODE btree btree`;;
btree_thm =
|- ∀e f.
    ∃!fn. (fn LEAF = e) ∧ (∀b b'. fn(NODE b b') = f(fn b)(fn b')b b')
```

The three-valued type used in Chapter 5 can also be defined automatically using the ML function `define_type`:

```
#let tri_thm = define_type `tri` `tri = Hi | Lo | X`;;
tri_thm =
|- ∀e0 e1 e2. ∃!fn. (fn Hi = e0) ∧ (fn Lo = e1) ∧ (fn X = e2)
```

Any instance of the class of concrete types discussed in Chapter 5 can be defined automatically from a user-supplied 'grammar' in exactly the same way.

Proving a Peano-like 'axiomatization' for a concrete type

Given the theorem bound above to `btree_thm`, a Peano-like axiomatization for the newly-defined type `btree` can be proved automatically as follows:

```
#let distinct = prove_constructors_distinct btree_thm;;
distinct = |- ∀b b'. ¬(LEAF = NODE b b')

#let one_one = prove_constructors_one_one btree_thm;;
one_one =
|- ∀b b' b'' b'''.
    (NODE b b' = NODE b'' b''') = (b = b'') ∧ (b' = b''')

#let induction = prove_induction_thm btree_thm;;
induction =
|- ∀P. P LEAF ∧ (∀b b'. P b ∧ P b' ⊃ P(NODE b b')) ⊃ (∀b. P b)
```

The argument to each of the three ML functions used above is the primitive recursion theorem for binary trees ('`btree_thm`'). Each function proves one of the three theorems which make up a Peano-type ‘axiomatization’ for the recursive type `btree`. What each function does is obvious from its name and the resulting theorem it returns. These three ML functions generate their output theorems by purely formal proof (as do all the functions used in this appendix). They can be used to prove similar Peano-type axioms for any concrete type definable using `define_type`. For example the two theorems about *tri* shown on page 84 can be proved automatically, as follows:

```
#let tri_distinct = prove_constructors_distinct tri_thm;;
tri_distinct = |- ¬(Hi = Lo) ∧ ¬(Hi = X) ∧ ¬(Lo = X)

#let tri_induct = prove_induction_thm tri_thm;;
tri_induct = |- ∀P. P Hi ∧ P Lo ∧ P X ⊃ (∀t. P t)
```

Each of the constructors for `tri` is a constant, so there is no theorem stating that these constructors are one-to-one.

Deriving a cases theorem for a concrete type

Given an induction theorem, the ML function `prove_cases_theorem` automatically proves a ‘cases’ theorem for any concrete type. For example:

```
#let btree_cases = prove_cases_thm induction;;
btree_cases = |- ∀b. (b = LEAF) ∨ (∃b'' b'. b = NODE b'' b')

#let tri_cases = prove_cases_thm tri_induct;;
tri_cases = |- ∀t. (t = Hi) ∨ (t = Lo) ∨ (t = X)
```

Each of the two theorems proved here states that every value of the corresponding concrete type is obtainable using one of its constructors. This reasonably easy consequence of induction, but weaker than it.

Defining functions on concrete types

The ML function `new_recursive_definition` automates the inferences necessary to justify any given primitive recursive definition on a concrete recursive type. It takes four arguments. The first is a boolean flag which indicates if the function to be defined will be an infix. The second is the primitive recursion theorem for the concrete type in question (i.e. a theorem obtained from `define_type`). The third argument is a name under which the resulting definition will be saved on disk. And the fourth argument is a term giving the desired primitive recursive definition. The value returned by `new_recursive_definition` is a theorem which states the primitive recursive definition requested by the user. This theorem is derived by formal proof from an instance of the general primitive recursion theorem given as the second argument.

For example, the recursive function Leaves defined in Chapter 5 can be defined in the HOL system as shown below:

```
#let Leaves =
    new_recursive_definition false btree_thm `Leaves`
      "(Leaves LEAF = 1) ∧
       (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))";;
Leaves =
|- (Leaves LEAF = 1) ∧
   (∀t1 t2. Leaves(NODE t1 t2) = (Leaves t1) + (Leaves t2))
```

The result of the call to `new_recursive_definition` is a theorem which states that the constant `Leaves` satisfies the primitive recursive defining equations requested by the user. This theorem is derived automatically from an instance of the general primitive recursion theorem for binary trees (`btree_thm`) and an appropriate non-recursive definition of the constant `Leaves`.

The function `new_recursive_definition` can also be used to define functions by cases on enumerated types. The data abstraction function abs discussed in Chapter 5, for example, can be defined as follows:

```
#let abs =
    new_recursive_definition false tri_thm `abs`
      "(abs Hi = T) ∧ (abs Lo = F)";;
abs = |- (abs Hi = T) ∧ (abs Lo = F)
```

As a final example, the recursively-defined model of an OR-gate tree defined in Chapter 5 can be defined in the system as follows:

```
#let Ortree =
    new_recursive_definition false btree `Ortree`
      "(Ortree LEAF in o = (in = [o])) ∧
       (Ortree (NODE t1 t2) in o =
               ∃i1 i2 o1 o2. (in = i1 ++ i2) ∧ (o = o1 ∨ o2) ∧
                             Ortree t1 i1 o1 ∧ Ortree t2 i2 o2)";;
Ortree =
|- (∀in o. Ortree LEAF in o = (in = [o])) ∧
   (∀t1 t2 in o.
     Ortree(NODE t1 t2)in o =
     (∃i1 i2 o1 o2.
       (in = i1 ++ i2) ∧
       (o = o1 ∨ o2) ∧
       Ortree t1 i1 o1 ∧
       Ortree t2 i2 o2))
```

It is assumed here that the infix append function '`++`' has already been defined (this is also done using the ML program `new_recursive_definition`).

An example proof

The following interactions show a goal-directed proof of the consistency theorem for the test-for-zero model in Chapter 5. The proof is done using Paulson's interactive goal management package, which can be found described in [74]. Only very brief comments are given to accompany the proof, as a full explanation would also involve a detailed account of the HOL theorem prover. Again, the reader unfamiliar with HOL is referred to [30], or the HOL system manual.

It is assumed that interactions which appear below are part of the same HOL session as the preceding ones, and that the bindings to ML variables made above are therefore still in force. It is furthermore assumed that the append function `++` and the length function `Length` have been defined recursively on lists, and are bound to ML identifiers as shown below:

```
#Length;;
|- (Length[] = 0) ∧ (∀h t. Length(CONS h t) = (Length t) + 1)

#Append;;
|- (∀l. [] ++ l = l) ∧ (∀h t l. (CONS h t) ++ l = CONS h(t ++ l))
```

The first step is to define the test-for-zero model, set up to goal to be proved, and immediately rewrite with the definition of the model.

```
#let Tfztree =
     new_definition
     (`Tfztree`, "Tfztree t in out = ∃o. Ortree t in o ∧ (out = ¬o)");;
Tfztree =
|- ∀t in out. Tfztree t in out = (∃o. Ortree t in o ∧ (out = ¬o))

#set_goal([],"∀t in. ¬(in = []) ⊃
             ((∃out. Tfztree t in out) = (Leaves t = Length in))");;

"∀t in.
  ¬(in = []) ⊃ ((∃out. Tfztree t in out) = (Leaves t = Length in))"

#expand(REWRITE_TAC [Tfztree]);;
OK..
"∀t in.
  ¬(in = []) ⊃
  ((∃out o. Ortree t in o ∧ (out = ¬o)) = (Leaves t = Length in))"
```

The system responds '`OK..`' and reduces the goal to be proved to the proposition shown above.

The proof now proceeds by structural induction on the binary tree `t`, using the induction theorem bound to `induction`. The tactic `INDUCT_THEN` takes an induction theorem for any concrete type and breaks a goal down into subgoals which correspond to the base and step cases of a structural induction. The second argument to `INDUCT_THEN` indicates what is to be done with the induction hypotheses. In the present case, they are made into assumptions using `STRIP_ASSUME_TAC`:

```
#expand(INDUCT_THEN induction STRIP_ASSUME_TAC);;
OK..
2 subgoals
"∀in.
  ¬(in = []) ⊃
  ((∃out o. Ortree(NODE t1 t2)in o ∧ (out = ¬o)) =
   (Leaves(NODE t1 t2) = Length in))"
    [ "∀in.
       ¬(in = []) ⊃
       ((∃out o. Ortree t1 in o ∧ (out=¬o))=(Leaves t1 = Length in))" ]
    [ "∀in.
       ¬(in = []) ⊃
       ((∃out o. Ortree t2 in o ∧ (out = ¬o)) =
        (Leaves t2 = Length in))" ]

"∀in.
  ¬(in = []) ⊃
  ((∃out o. Ortree LEAF in o ∧ (out = ¬o)) = (Leaves LEAF = Length in))"
```

There are now two subgoals, corresponding to the two possible cases `t=LEAF` and `t=NODE t1 t2`. The second subgoal shown above (the base case in the induction) is the next one to prove, and the first step is to rewrite with the recursive definitions of `Ortree` and `Leaves`:

```
#expand(REWRITE_TAC [Ortree;Leaves]);;
OK..
"∀in.
  ¬(in = []) ⊃ ((∃out o. (in = [o]) ∧ (out = ¬o)) = (1 = Length in))"
```

The proof of this subgoal now proceeds by case analysis on the list `in`. The proof requires a 'cases' theorem for (boolean) lists, a theorem which states that all lists constructed using `CONS` are distinct from the empty list `[]`, and a theorem which states that the function `CONS` is one-to-one:

```
list_cases;;
|- ∀l. (l = []) ∨ (∃t h. l = CONS h t)

#NOT_CONS_NIL;;
|- ∀h t. ¬(CONS h t = [])

#CONS_11;;
|- ∀h t h' t'. (CONS h t = CONS h' t') = (h = h') ∧ (t = t')
```

Using these theorems, the current subgoal can be reduced to a consideration of the single case `in`=`CONS h t`, since the antecedent of the subgoal states that `in` is not the empty list:

```
expand(GEN_TAC THEN
       REPEAT_TCL STRIP_THM_THEN
       SUBST1_TAC (SPEC "in" list_cases) THEN
       REWRITE_TAC [NOT_CONS_NIL;CONS_11]);;
OK..
"(∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o)) = (1 = Length(CONS h t))"
```

Using the definition of `Length`, the goal can be further reduced as follows:

```
#expand(REWRITE_TAC [Length]);;
OK..
"(∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o)) = (1 = (Length t) + 1)"
```

The goal is now a boolean equation, which can be split up into two implications using `EQ_TAC`:

```
#expand(EQ_TAC);;
OK..
2 subgoals
"(1 = (Length t) + 1) ⊃ (∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o))"

"(∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o)) ⊃ (1 = (Length t) + 1)"
```

The newly-generated current goal is straightforward to prove using the definition of `Length`, the assumption that the list `t` is empty, and a few trivial facts about addition (‘`ADD_CLAUSES`’):

```
#expand(STRIP_TAC THEN ASM_REWRITE_TAC [ADD_CLAUSES;Length]);;
OK..
goal proved
|- (∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o)) ⊃
   (1 = (Length t) + 1)

Previous subproof:
"(1 = (Length t) + 1) ⊃ (∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o))"
```

The remaining subgoal is a bit more difficult, but it can be reduced by straightforward arithmetic (using some built-in theorems) to:

```
#expand(REWRITE_TAC [INV_SUC_EQ;ADD_CLAUSES;num_CONV "1"]);;
OK..
"(0 = Length t) ⊃ (∃out o. ((h = o) ∧ (t = [])) ∧ (out = ¬o))"
```

The task is now to show that if the list `t` has length 0, then `t=[]`. Another case split on the list `t`, rewriting with the definition of `Length`, and the use of some elementary built-in arithmetic yields:

```
#expand(REPEAT_TCL STRIP_THM_THEN
       SUBST1_TAC (SPEC "t" list_cases) THEN
       REWRITE_TAC [Length;ADD_CLAUSES;num_CONV "1";
                    NOT_EQ_SYM (SPEC_ALL NOT_SUC)]);;
OK..
"∃out o. (h = o) ∧ (out = ¬o)"
```

The current subgoal is now trivial, and can be proved as follows:

```
#expand(MAP_EVERY EXISTS_TAC ["¬h";"h"] THEN
       REWRITE_TAC []);;
OK..
goal proved
|- ∃out o. (h = o) ∧ (out = ¬o)
   ⋮
|- ∀in.
    ¬(in = []) ⊃
    ((∃out o. Ortree LEAF in o ∧ (out = ¬o)) =
     (Leaves LEAF = Length in))
```

The system responds '`Goal proved`' and prints a summary of the subgoals proved on this branch of the goal tree (some of which are here omitted, being replaced by dots). This completes the base case in the main induction on binary trees. But there still remains the step case (`t=NODE t1 t2`). The system goes on to print the remaining subgoal:

```
⋮
Previous subproof:
"∀in.
  ¬(in = []) ⊃
  ((∃out o. Ortree(NODE t1 t2)in o ∧ (out = ¬o)) =
   (Leaves(NODE t1 t2) = Length in))"
    [ "∀in.
       ¬(in = []) ⊃
       ((∃out o. Ortree t1 in o ∧ (out=¬o))=(Leaves t1 = Length in))" ]
    [ "∀in.
       ¬(in = []) ⊃
       ((∃out o. Ortree t2 in o ∧ (out=¬o))=(Leaves t2 = Length in))" ]
```

The proof of this remaining subgoal is considerably longer than the proof of the base case, and a fully commented proof will therefore not be given here. Only some main lemmas needed to complete the proof will be shown.

The first few lemmas needed to finish the proof concern the the infix append function ++. These are all straightforward to prove by induction on lists:

```
#let Length_Append =
    TAC_PROOF(([], "∀l1 l2. Length(l1++l2) = (Length l1)+(Length l2)"),
        INDUCT_THEN list_INDUCT STRIP_ASSUME_TAC THEN
        ASM_REWRITE_TAC [Length;Append;ADD_CLAUSES;num_CONV "1"]);;
Length_Append = |- ∀l1 l2. Length(l1 ++ l2) = (Length l1) + (Length l2)


#let Append_ASSOC =
    TAC_PROOF(([], "∀l1 l2 l3. l1 ++ (l2 ++ l3) = ((l1 ++ l2) ++ l3)"),
        LIST_INDUCT_TAC THEN ASM_REWRITE_TAC [Append]);;
Append_ASSOC = |- ∀l1 l2 l3. l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3


#let Append_Not_Nil =
    TAC_PROOF(([],"∀l. ¬(l ++ [x] = [])"),
        INDUCT_THEN list_INDUCT ASSUME_TAC THEN
        REWRITE_TAC [Append;NOT_CONS_NIL]);;
Append_Not_Nil = |- ∀l. ¬(l ++ [x] = [])


#let Append_empty =
    TAC_PROOF(([], "∀l1 l2. ([] = l1 ++ l2) = ((l1 = []) ∧ (l2 = []))"),
              INDUCT_THEN list_INDUCT STRIP_ASSUME_TAC THEN
              REWRITE_TAC [Append;NOT_CONS_NIL;NOT_NIL_CONS] THEN
              GEN_TAC THEN MATCH_ACCEPT_TAC EQ_SYM_EQ);;
Append_empty = |- ∀l1 l2. ([] = l1 ++ l2) = (l1 = []) ∧ (l2 = [])
```

A slightly more complicated lemma about lists is also needed. This relates a list of length n+2 to a list of length n+1:

```
#let list_Length2 =
    TAC_PROOF(([],"∀l n. ((SUC(SUC n)) = Length l) =
                         (∃h:α. ∃l'. ((SUC n) = Length l') ∧
                                     (l = (CONS h l')))"),
        INDUCT_THEN list_INDUCT ASSUME_TAC THENL
        [REWRITE_TAC [Length;INV_SUC_EQ;NOT_SUC;NOT_NIL_CONS];
         REWRITE_TAC [Length;INV_SUC_EQ;CONS_11;
                      num_CONV "1";ADD_CLAUSES] THEN
         REPEAT (STRIP_TAC ORELSE EQ_TAC) THENL
         [EXISTS_TAC "h:α" THEN EXISTS_TAC "l:(α)list" THEN
          ASM_REWRITE_TAC []; ASM_REWRITE_TAC []]]);;
list_Length2 =
|- ∀l n.
    (SUC(SUC n) = Length l) =
    (∃h l'. (SUC n = Length l') ∧ (l = CONS h l'))
```

The main step is again an induction on lists.

The following lemma about lists is also needed. It simply states that a list of length n+1 cannot be empty:

```
#let Length_lemma =
    TAC_PROOF(([], "∀l. (SUC n = Length l) ⊃ ¬(l = [])"),
        INDUCT_THEN list_INDUCT ASSUME_TAC THEN
        REWRITE_TAC [Length;NOT_CONS_NIL;ADD_CLAUSES;
                     num_CONV "1";NOT_SUC]);;
Length_lemma = |- ∀l. (SUC n = Length l) ⊃ ¬(l = [])
```

Another list which cannot be empty is the input to an OR-gate tree. The smallest tree is a single inverter, with an input word of length 1. It therefore follows that:

```
#let Ortree_lemma =
    TAC_PROOF(([], "Ortree t i o ⊃ ¬(i = [])"),
        CONV_TAC (ONCE_DEPTH_CONV CONTRAPOS_CONV) THEN
        REWRITE_TAC [] THEN
        DISCH_THEN SUBST1_TAC THEN
        SPEC_TAC ("o:bool","o:bool") THEN
        SPEC_TAC ("t:btree","t:btree") THEN
        INDUCT_THEN induction STRIP_ASSUME_TAC THEN
        REWRITE_TAC [Ortree] THENL
        [REWRITE_TAC [NOT_NIL_CONS];
         REWRITE_TAC [Append_empty] THEN
         REPEAT STRIP_TAC THEN
         MAP_EVERY POP_ASSUM [MP_TAC;MP_TAC;K ALL_TAC] THEN
         REPEAT (POP_ASSUM SUBST_ALL_TAC) THEN
         ASM_REWRITE_TAC []]);;
Ortree_lemma = |- Ortree t i o ⊃ ¬(i = [])
```

Finally, it is necessary to prove that the function `Leaves` computes a non-zero number for each tree:

```
#let Leaves_non_zero =
    TAC_PROOF(([], "∀tr. ∃n. Leaves tr = SUC n"),
        INDUCT_THEN induction STRIP_ASSUME_TAC THENL
        [EXISTS_TAC "0" THEN REWRITE_TAC [Leaves;num_CONV "1"];
         REWRITE_TAC [Leaves] THEN REPEAT (POP_ASSUM SUBST1_TAC) THEN
         EXISTS_TAC "SUC (n+n')" THEN REWRITE_TAC [ADD_CLAUSES]]);;
Leaves_non_zero = |- ∀tr. ∃n. Leaves tr = SUC n
```

This theorem is proved by structural induction on the tree `t`.

The main lemma now needed to complete the proof is the following:

```
#let Main_lemma =
    TAC_PROOF(([], "∀l t1 t2.
             ((Leaves t1) + (Leaves t2) = Length l) ⊃
             ∃l1 l2. (l = l1 ++ l2) ∧ ¬(l1 = []) ∧ ¬(l2 = []) ∧
                     (Leaves t1 = Length l1) ∧
                     (Leaves t2 = Length l2)"),
    REPEAT GEN_TAC THEN
    STRIP_THM_THEN SUBST1_TAC (SPEC "t1:btree" Leaves_non_zero) THEN
    STRIP_THM_THEN SUBST1_TAC (SPEC "t2:btree" Leaves_non_zero) THEN
    REWRITE_TAC [ADD_CLAUSES] THEN
    MAP_EVERY (SPEC_TAC o (λt.t,t)) ["n':num";"l:(α)list";"n:num"] THEN
    INDUCT_TAC THEN REWRITE_TAC [ADD_CLAUSES] THENL
    [INDUCT_THEN list_INDUCT ASSUME_TAC THENL
     [REWRITE_TAC [Length;NOT_SUC];
      REWRITE_TAC [Length;INV_SUC_EQ] THEN REPEAT STRIP_TAC THEN
      MAP_EVERY EXISTS_TAC ["[h:α]";"l:(α)list"] THEN
      POP_ASSUM MP_TAC THEN
      REWRITE_TAC [Length;Append;num_CONV "1"] THEN
      REWRITE_TAC [NOT_CONS_NIL;INV_SUC_EQ;ADD_CLAUSES] THEN
      STRIP_TAC THEN IMP_RES_TAC lemma1 THEN ASM_REWRITE_TAC []];
     PURE_ONCE_REWRITE_TAC [SYM(el 4 (CONJUNCTS ADD_CLAUSES))] THEN
     REPEAT STRIP_TAC THEN RES_TAC THEN
     POP_ASSUM (MP_TAC o REWRITE_RULE [list_Length2]) THEN
     POP_ASSUM_LIST (K ALL_TAC) THEN REPEAT STRIP_TAC THEN
     MAP_EVERY EXISTS_TAC ["l1 ++ [h:α]";"l':(α)list"] THEN
     ASM_REWRITE_TAC[Append;SYM(SPEC_ALL Append_ASSOC)] THEN
     ASM_REWRITE_TAC[Length_Append;Length;ADD_CLAUSES;num_CONV "1"] THEN
     IMP_RES_TAC lemma1 THEN ASM_REWRITE_TAC [Append_Not_Nil]]);;
Main_lemma =
|- ∀l t1 t2.
    ((Leaves t1) + (Leaves t2) = Length l) ⊃
    (∃l1 l2.
      (l = l1 ++ l2) ∧
      ¬(l1 = []) ∧
      ¬(l2 = []) ∧
      (Leaves t1 = Length l1) ∧
      (Leaves t2 = Length l2))
```

This theorem is used to 'split' the list `l` into two nonempty sublists `l1` and `l2`. The lengths of these sublists match the number of leaves in the two subtrees `t1` and `t2`. This allows the two induction hypotheses in the remaining subgoal (shown on page 217) to be used.

Given Main_lemma, and some of the other lemmas proved above, the remaining subgoal can be proved as follows:

```
#expand(REWRITE_TAC [Ortree;Leaves] THEN
       REPEAT (STRIP_TAC ORELSE EQ_TAC) THENL
       [ASM_REWRITE_TAC [Length_Append] THEN
        SUBGOAL_THEN
          "(Leaves t1 = Length (i1:(bool)list)) ∧
           (Leaves t2 = Length (i2:(bool)list))"
           (λth. REWRITE_TAC [th]) THEN
       IMP_RES_TAC Ortree_lemma THEN RES_TAC THEN
       STRIP_TAC THEN FIRST_ASSUM MATCH_MP_TAC THENL
       [MAP_EVERY EXISTS_TAC ["¬o1:bool";"o1:bool"];
        MAP_EVERY EXISTS_TAC ["¬o2:bool";"o2:bool"]] THEN
        STRIP_TAC THEN (REFL_TAC ORELSE FIRST_ASSUM ACCEPT_TAC);
        HOL_IMP_RES_THEN MP_TAC Main_lemma THEN
        POP_ASSUM (K ALL_TAC) THEN POP_ASSUM (K ALL_TAC) THEN
        REPEAT STRIP_TAC THEN POP_ASSUM MP_TAC THEN
        POP_ASSUM MP_TAC THEN RES_TAC THEN
        REPEAT (DISCH_THEN (ANTE_RES_THEN STRIP_ASSUME_TAC)) THEN
        MAP_EVERY EXISTS_TAC ["¬(o ∨ o')";"o ∨ o'"] THEN
        CONJ_TAC THENL [ALL_TAC; REFL_TAC] THEN
        MAP_EVERY EXISTS_TAC ["l1:(bool)list";"l2:(bool)list"] THEN
        MAP_EVERY EXISTS_TAC ["o:bool";"o':bool"] THEN
        ASM_REWRITE_TAC []]);;
#OK..
goal proved
.. |- ∀in.
       ¬(in = []) ⊃
       ((∃out o. Ortree(NODE t1 t2)in o ∧ (out = ¬o)) =
        (Leaves(NODE t1 t2) = Length in))
⋮
|- ∀t in.
    ¬(in = []) ⊃ ((∃out. Tfztree t in out) = (Leaves t = Length in))

Previous subproof:
goal proved
```

This finishes the proof of the last remaining subgoal. The system prints a trace of the subgoals which have been proved. The final theorem is the desired consistency theorem for the test-for-zero model:

```
|- ∀t in.
    ¬(in = []) ⊃ ((∃out. Tfztree t in out) = (Leaves t = Length in))
```

Q.E.D

Summary: HOL tools for reasoning about concrete types

A summary of the main HOL tools developed by the author for reasoning about concrete types is given in the table shown below.

Infix Abbreviations for Types	
ML Function	*Description*
`define_type`	defines an arbitrary concrete type
`prove_constructors_distinct`	proves constructors yield different values
`prove_constructors_one_one`	proves that constructors are one-to-one
`prove_induction_thm`	derives structural induction
`prove_cases_thm`	proves a cases theorem
`new_recursive_definition`	justifies primitive recursive definitions
`INDUCT_THEN`	general structural induction tactic

The logical basis for all these tools is the method for defining concrete types explained in Appendix A. Examples of the use of these programs were given in the interactive session shown above.

References

[1] Andrews, P.B., *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*, Computer Science and Applied Mathematics Series (Academic Press, 1986).

[2] Barros, J.C. and B.W. Johnson, 'Equivalence of the Arbiter, the Synchronized, the Latch and the Inertial Delay', *IEEE Transactions on Computers*, Vol. C-32, No. 7 (July 1983), pp. 603–614.

[3] Barrow, H.G., 'VERIFY: A Program for Proving the Correctness of Digital Hardware Designs', *Artificial Intelligence*, Vol. 24, No. 1–3 (December 1984), pp. 437–491.

[4] Berthet, C. and E. Cerny, 'An Algebraic Model for Asynchronous Circuits Verification', *IEEE Transactions on Computers*, Vol. 37, No. 7 (July 1988), pp. 242–254.

[5] Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice Hall International Series in Computer Science (Prentice Hall, 1988).

[6] Birtwistle, G., B. Graham, T. Melham, and R. Schediwy, 'Hardware Verification by Formal Proof', *Proceedings of the Canadian Conference on Electrical and Computer Engineering, Vancouver, November 3–4, 1988*, edited by V. J. Bhargava, pp. 379–384.

[7] Birtwistle, G., J. Joyce, B. Liblong, T. Melham, and R. Schediwy, 'Specification and VLSI Design', in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G.J. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 83–97.

[8] Bochmann, G.V., 'Hardware Specification with Temporal Logic: An Example', *IEEE Transactions on Computers*, Vol. C-31, No. 3 (March 1982), pp. 223–231.

[9] Boyer, R.S. and J.S. Moore, *A Computational Logic*, ACM Monograph Series (Academic Press, 1979).

[10] Camilleri, A.J., 'Executing Behavioural Definitions in Higher Order Logic', Ph.D. dissertation, University of Cambridge (1988).

[11] Camilleri, A., M. Gordon, and T. Melham, 'Hardware Verification using Higher-Order Logic', in: *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference, Grenoble, September 1986*, edited by D. Borrione (North-Holland, 1987), pp. 43–67.

[12] Cardelli, L., ‘An Algebraic Approach to Hardware Description and Verification’, Ph.D. dissertation, The University of Edinburgh (April 1982).

[13] Chin, S., E.P. Stabler, and K.J. Greene, ‘Using Higher Order Logic to Synthesize Correct Designs’, Technical Report no. 8805, CASE center, Syracuse University (March 1988).

[14] Church, A., ‘A Formulation of the Simple Theory of Types’, *The Journal of Symbolic Logic*, Vol. 5 (1940), pp. 56–68.

[15] Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, (Springer-Verlag, 1981).

[16] Cohn, A., ‘A Proof of Correctness of The VIPER Microprocessor: The First Level’, in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, 1988), pp. 27–71.

[17] Cohn, A., ‘Correctness Properties of the Viper Block Model: The Second Level’, in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam, (Springer-Verlag, 1989), pp. 1–91.

[18] Cohn, A., ‘The Notion of Proof in Hardware Verification’, to appear in: *Journal of Automated Reasoning*, Vol. 5, No. 2 (1989).

[19] Cousineau, G., G. Huet, and L. Paulson, *The ML Handbook* (INRIA, 1986).

[20] Cullyer, W.J., ‘Implementing Safety-Critical Systems: The VIPER Microprocessor’, in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, 1988), pp. 1–25.

[21] Dhingra, I.S., ‘Formalising an Integrated Circuit Design Style in Higher Order Logic’, Ph.D. dissertation, University of Cambridge (1988).

[22] Dill, D.L. and E.M. Clarke, ‘Automatic verification of asynchronous circuits using temporal logic’, *IEE Proceedings*, Vol. 133, Part E, No. 5 (September 1986), pp. 276–282.

[23] Eveking, H., ‘The Application of CHDL’s to the Abstract Specification of Hardware’, in: *Computer Hardware Description Languages and their Applications: Proceedings of the IFIP WG 10.2 Seventh International Conference, Tokyo, August 1985*, edited by C.J. Koomen and T. Moto-oka (North-Holland, 1985), pp. 167–178.

[24] Eveking, H., ‘Formal Verification of Synchronous Systems’, in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G.J. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 137–151.

[25] Fusaoka, A., H. Seki, and K. Takahashi, 'Description and Reasoning of VLSI Circuit in Temporal Logic', *New Generation Computing*, Vol. 2, (1984), pp. 79–90.

[26] Gallier, J.H., *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Computer Science and Technology Series (Harper and Row, 1986).

[27] Gaubatz, D. and M. Burrows, 'TRING', unpublished internal report, Computer Laboratory, University of Cambridge (October 1984).

[28] Goguen, J.A., J.W. Thatcher, and E.G. Wagner, 'An initial algebra approach to the specification, correctness, and implementation of abstract data types', in: *Current Trends in Programming Methodology*, edited by R.T. Yeh (Prentice-Hall, 1978), Vol. IV, pp. 80–149.

[29] Gordon, M., 'HOL: A Machine Oriented Formulation of Higher Order Logic', Technical Report no. 68, Computer Laboratory, University of Cambridge, revised version (July 1985).

[30] Gordon, M.J.C., 'HOL: A Proof Generating System for Higher-Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, 1988), pp. 73–128.

[31] Gordon, M., 'How to Specify and Verify Hardware using Higher Order Logic', unpublished lecture notes, University of Cambridge (Autumn, 1984).

[32] Gordon, M., 'LCF_LSM: A System for Specifying and Verifying Hardware', Technical Report no. 41, Computer Laboratory, University of Cambridge (1983).

[33] Gordon, M., 'Proving a Computer Correct with the LCF_LSM Hardware Verification System', Technical Report no. 42, Computer Laboratory, University of Cambridge (1983).

[34] Gordon, M., 'Why higher-order logic is a good formalism for specifying and verifying hardware', in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G.J. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 153–177.

[35] Gordon, M.J., A.J. Milner, and C.P. Wadsworth, 'Edinburgh LCF: A Mechanised Logic of Computation', Lecture Notes in Computer Science, Vol. 78 (Springer-Verlag, 1979).

[36] Gries, D., *The Science of Programming*, Texts and Monographs in Computer Science (Springer-Verlag, 1981).

[37] Guttag, J.V., E. Horowitz, and D.R. Musser, 'Abstract Data Types and Software Validation', *Communications of the ACM*, Vol. 21, No. 12 (December 1978), pp. 1048–1064.

[38] Halbwachs, N., A. Lonchampt, and D. Pilaud, 'Describing and Designing Circuits by means of a Synchronous Declarative Language', in: *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference, Grenoble, September 1986*, edited by D. Borrione (North-Holland, 1987), pp. 255–268.

[39] Hale, R.W.S., 'Modelling a Ring Network in Interval Temporal Logic', in: *Microcomputers, Usage and Design: Proceedings of the eleventh EUROMICRO Symposium on Microprocessing and Microprogramming*, edited by K. Waldschmidt and B. Myhrhaug (North-Holland, 1985), pp. 77–84.

[40] Hale, R.W.S., 'Programming in Temporal Logic', Ph.D. dissertation, University of Cambridge (1988).

[41] Hanna, F.K., 'Overview of the Veritas Project', unpublished internal report, University of Kent, (1983).

[42] Hanna, F.K. and N. Daeche, 'Purely Functional Implementation of a Logic', in: *Proceedings of the 8th International Conference on Automated Deduction*, edited by J.H. Siekmann, Lecture Notes in Computer Science, Vol. 230 (Springer-Verlag, 1986), pp. 598–607.

[43] Hanna, F.K. and N. Daeche, 'Specification and verification of digital systems using higher-order predicate logic', *IEE Proceedings*, Vol. 133, Part E, No. 5 (September 1986), pp. 242–254.

[44] Hanna, F.K. and N. Daeche, 'Specification and Verification using Higher-Order Logic: A Case Study', in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G.J. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 179–213.

[45] Hanna, F.K., N. Daeche, and M. Longley, 'VERITAS$^+$: a Specification Language based on Type Theory', in: *Proceedings of the Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Cornell University, July 5–7 1989, (Proceedings to appear in Lecture Notes in Computer Science).

[46] Harper, R., D. MacQueen, and R. Milner, 'Standard ML', Report no. ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh (March 1986).

[47] Hatcher, W.S., *The Logical Foundations of Mathematics*, Foundations and Philosophy of Science and Technology Series (Pergamon Press, 1982).

[48] Herbert, J.M.J., 'Application of Formal Methods to Digital System Design', Ph.D. dissertation, University of Cambridge (1986).

[49] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science (Prentice-Hall, 1985).

[50] Hoare, C.A.R., ‘Proof of Correctness of Data Representations’, *Acta Informatica*, Vol. 1, No. 4 (1972), pp. 271–281.

[51] Hoare, C.A.R. and M.J.C. Gordon, ‘Partial Correctness of C-MOS Switching Circuits: An Exercise in Applied Logic’, in: *Proceedings of the Third Annual Symposium on Logic in Computer Science*, (Computer Society Press, 1988), pp. 28–36.

[52] Hopper, A. and R.M. Needham, ‘The Cambridge Fast Ring Networking System’, Technical Report no. 90, Computer Laboratory, University of Cambridge (1986).

[53] Hunt, W.A., ‘FM8501: A Verified Microprocessor’, Ph.D. dissertation, The University of Texas at Austin (December 1985).

[54] Joyce, J.J., ‘Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic’, Technical Report no. 136, Computer Laboratory, University of Cambridge (June 1988).

[55] Joyce, J.J., ‘Formal Specification and Verification of Microprocessor Systems’, in: *EUROMICRO 88: Proceedings of the 14th Symposium on Microprocessing and Microprogramming*, edited by S. Winter and H. Schumny, (North-Holland, 1988), pp. 371–378.

[56] Joyce, J.J., ‘Formal Verification and Implementation of a Microprocessor’, in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, 1988), pp. 129–157.

[57] Joyce, J.J., ‘Generic Structures in the Formal Specification and Verification of Digital Circuits’, in: *The Fusion of Hardware Design and Verification: Proceedings of the IFIP WG 10.2 Working Conference*, edited by G.J. Milne (North-Holland, 1988), pp. 51–75.

[58] Joyce, J., G. Birtwistle, and M. Gordon, ‘Proving a Computer Correct in Higher Order Logic’, Technical Report no. 100, Computer Laboratory, University of Cambridge (December 1986).

[59] Leeser, M.E., ‘Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic’, Ph.D. dissertation, University of Cambridge (December, 1987).

[60] Leisenring, A.C., *Mathematical Logic and Hilbert’s ε-Symbol*, University Mathematical Series (Macdonald & Co., 1969).

[61] Martin-Löf, P, ‘Constructive Mathematics and Computer Programming’, *Philosophical Transactions of the Royal Society of London*, Series A, Vol. 312 (1983), pp. 501–518.

[62] Melham, T.F., ‘A Signal Abstraction Function’, unpublished internal report, Computer Laboratory, University of Cambridge, (April 1985).

[63] Melham, T.F., 'Abstraction Mechanisms for Hardware Verification', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, 1988), pp. 267–291.

[64] Melham, T.F., 'Automating Recursive Type Definitions in Higher Order Logic', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam, (Springer-Verlag, 1989), pp. 341–386.

[65] Milne, G.J., 'Simulation and Verification: Related Techniques for Hardware Analysis', in: *Computer Hardware Description Languages and their Applications: Proceedings of the IFIP WG 10.2 Seventh International Conference, Tokyo, August 1985*, edited by C.J. Koomen and T. Moto-oka (North-Holland, 1985), pp. 404–417.

[66] Milne, G.J., 'CIRCAL: A calculus for circuit description', *Integration*, Vol. 1, Nos. 2 and 3 (1983), pp. 121–160.

[67] Milne, G. and R. Milner, 'Concurrent Processes and Their Syntax', *Journal of the ACM*, Vol. 26 (1979), pp. 302-321.

[68] Milner, R., 'A Theory of Type Polymorphism in Programming', *Journal of Computer and System Sciences*, No. 17 (1978), pp. 348–375.

[69] Milner, R., 'How to derive inductions in LCF', unpublished internal report, The University of Edinburgh (August 1980).

[70] Monahan, B.Q., 'Data Type Proofs Using Edinburgh LCF', Ph.D. dissertation, The University of Edinburgh (1985).

[71] Moszkowski, B.C., *Executing Temporal Logic Programs*, (Cambridge University Press, 1986).

[72] Moszkowski, B.C., 'Reasoning about Digital Circuits', Ph.D. dissertation, Stanford University (April 1983).

[73] Narendran, P. and J. Stillman, 'Formal Verification of the Sobel Image Processing Chip', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam, (Springer-Verlag, 1989), pp. 92–127.

[74] Paulson, L.C., *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).

[75] Piloty, R. and D. Borrione, 'The Conlan Project: Concepts, Implementations, and Applications', *Computer*, Vol. 18, No. 2 (February 1985), pp. 81–92.

[76] Sheeran, M., 'Design and verification of regular synchronous circuits', *IEE Proceedings*, Vol. 133, Part E, No. 5 (September 1986), pp. 295–304.

[77] Sheeran, M., 'Retiming and Slowdown in Ruby', in: *The Fusion of Hardware Design and Verification: Proceedings of the IFIP WG 10.2 Working Conference*, edited by G.J. Milne (North-Holland, 1988), pp. 289–308.

[78] Stavridou, V., H. Barringer and D.A. Edwards, 'Formal Specification and Verification of Hardware: A Comparative Case Study', in: *Proceedings of the* ACM IEE *25th Design Automation Conference*, (Computer Society Press, 1988), pp. 197–204.

[79] Subrahmanyam, P.A., 'Towards a framework for dealing with system timing in Very High Level Silicon Compilers', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science, SECS35 (Kluwer Academic Publishers, 1988), pp. 159–215.

[80] Traub, N., 'A Formal Approach to Hardware Analysis', Ph.D. dissertation, The University of Edinburgh (March 1987).

[81] Turner, D., 'An Overview of Miranda', *Sigplan Notices*, Vol. 21, No. 12 (December 1986), pp. 158–166.

[82] Whitehead, A.N. and B. Russell, *Principia Mathematica*, 3 volumes (Cambridge University Press, 1910-3).

[83] Wagner, T.J., 'Hardware Verification', Ph.D. dissertation, Stanford University, (September 1977).

[84] Weste, N.H.E. and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, VLSI Systems Series (Addison-Wesley, 1985).

[85] Winskel, G., 'Models and logic of MOS circuits', in: *Logic of Programming and Calculi of Discrete Design: International Summer School Directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, edited by M. Broy, NATO ASI Series, Series F, Computer and Systems Sciences, Vol. 36 (Springer-Verlag, 1987), pp. 367–413.

[86] Winskel, G., 'Relating two models of hardware', in: *Category Theory and Computer Science*, edited by D.H. Pitt, A. Poigné, and D.E. Rydeheard, Lecture Notes in Computer Science, Vol. 283 (Springer-Verlag, 1987), pp. 98–113.

[87] Wojick, A.S., 'Formal Design Verification of Digital Systems', in: *Proceedings of the* ACM IEE *20th Design Automation Conference*, (Computer Society Press, 1983), pp. 228–234.