Machine-Assisted Theorem-Proving for Software Engineering

Andrew Martin
Pembroke College

Technical Monograph PRG-121
ISBN 0-902928-95-3
July 1996

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
United Kingdom

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Machine-Assisted Theorem-Proving for Software Engineering

D.Phil. Thesis

Andrew Martin
Pembroke College

Michaelmas 1994

Abstract

The thesis describes the production of a large prototype proof system for Z, and a tactic language in which the proof *tactics* used in a wide range of systems (including the system described here) can be discussed.

The details of the construction of the tool—using the $\mathcal{W}$ logic for Z, and implemented in 2OBJ—are presented, along with an account of some of the proof tactics which enable $\mathcal{W}$ to be applied to typical proofs in Z. A case study gives examples of such proofs. Special attention is paid to soundness concerns, since it is considerably easier to check that a program such as this one produces sound proofs, than to check that each of the impenetrable proofs which it creates is indeed sound. As the first such encoding of $\mathcal{W}$, this helped to find bugs in the published presentations of $\mathcal{W}$, and to demonstrate that $\mathcal{W}$ makes proof in Z tractable.

The second part of the thesis presents a tactic language, with a formal semantics (independent of any particular tool) and a set of rules for reasoning about tactics written in this language. A small set of these rules is shown to be complete for the finite (non-recursive) part of the language. Some case studies are included, as are some ideas on how this tactic language can give rise to lightweight implementations of theorem proving tools. The tool described in some detail is another theorem-prover for Z, this time based on LittleZ.

Contents

Note to the reader

ORIGINALLY, this monograph was prepared as the author's D.Phil. thesis. With the benefit of hindsight (a year after the thesis was finalized), perhaps some more comments on the use of 2OBJ are warranted. When the work described in the first part of thesis was begun, 2OBJ was under active development in Oxford, and showed some promise of becoming a useful logical framework. That project ended however, leaving the tool unsupported, and causing some frustration. The work described here is almost certainly the largest case study undertaken with 2OBJ; it was not entirely sufficient for the task—see Chapter 4.

Since this work was completed, others have followed a similar path [KB95], encoding the W logic using the *Isabelle* system. The paper cited contains a comparison of that work and this, remarking that Isabelle is much more flexible and powerful than 2OBJ. The encoding described here is rather more *faithful* than that in Isabelle, but at the price of a considerable loss of efficiency.

Acknowledgements

THE work described here has been carried out under a research studentship from EPSRC (formerly, SERC). Support has come from many members of the PRG, to whom I am most grateful. Special thanks are due to my supervisor, Jim Woodcock, for advice and guidance (on both style and content); also to Stephen Brien, Andrew Stevens and Keith Hobley, all of whom have been long-suffering as I have attempted to understand and apply their work. The 'Attic Crowd' helped to provide a congenial and occasionally productive working environment.

Thanks, too, to those who have read and commented-on various drafts of material presented here. Paul Gardiner has studied numerous different versions of Chapter 5, and never tires of suggesting improvements. Anonymous referees have made helpful comments on Chapters 2 and 5; Bernard Sufrin and Philipp Heuberger have also helped with some valuable comments on earlier drafts of parts of the thesis. Peter Baumann, Ina Kraan, Ian Toyn and Jon Hall have read with sufficient diligence to spot some subtle errors; my thanks to them. My examiners, Bill Roscoe and Will Harwood, also deserve thanks for helping me to spot, and to iron out, some further infelicities.

Finally, all this would never have come about without support from many friends—particularly in Pembroke MCR and St. Matthew's Church—and my family, most chiefly my parents.

Andrew Martin
24th May 1996

Dominus illuminatio mea et salus mea

Chapter 1

Introduction

MODERN SOFTWARE ENGINEERING relies to an increasingly large extent upon so-called 'formal methods' of program design and development. To date, much of the work on formal methods has concentrated on the use of formal methods for *design* work (earning some methods the more accurate description of *formal description techniques*). Validation of that design, and subsequent development work are often accomplished by more traditional techniques of coding and testing.

One of the reasons for this is that formal development and validation necessarily involves considerable effort in *proof*. In VDM [Jon90], it is necessary to prove that operation specifications do not break any global invariants that have been declared on the state variables. Likewise, in the usual style of using Z [Spi92a, WD96] (which is used in this thesis), the specification should include a demonstration that its declared initial state satisfies all state invariant properties (Chapter 3 presents an example of such a proof).

Development calls for similar proofs, about operation and/or data refinement, whether the development is *constructive* (that is, the methodology used supplies a proof of correctness 'for free' as the development proceeds), or it calls for *post hoc* proof (producing a program, and then proving that it meets its specification).[1] Such proofs, though potentially very similar to those proposed by mathematicians for centuries, are generally quite unwieldy.

These proofs present problems for a number of reasons:

- The nature of formal descriptions tends to give rise to very formal proof requirements. Proofs undertaken are correspondingly of a very formal nature—quite unlike anything usually produced in mathematics (except in elementary logic textbooks, where formal proof exercises serve merely to demonstrate that detailed formal proof is possible (but undesirable)).

- The datatypes used in computing tend to be much 'larger' than those typically present in mathematics. This leads to proofs in which there are multiple cases to

[1] The latter is, in general, much harder than the former, and tends to be avoided.

be considered, and many small details and side-conditions to be checked; proofs in which none of the steps is hard to follow, but the sheer number makes the reader uncertain that he would spot any but the most glaring omissions.

- Any new, ground-breaking proof in mathematics will be published in the literature, and subject to considerable peer-review; hence its correctness is likely to be checked by numerous professional mathematicians. The proof that a given program meets its specification will, even with the best engineering quality assurance schemes, be looked at by only a handful of colleagues, most of whom will not share detailed knowledge of the problem area.[DMLP79]

These problems make the development activity of proof ripe for machine support. Computers tend to be good at keeping track of great levels of detail, recording large structures faithfully, and checking without complaint the most tedious of calculations. Of course, involving computers also itself adds to the level of formality needed: with today's technology, we are unable to give a computer the intuition which a mathematician might bring to a problem, and so proof search tends to be fairly naïve (though many useful procedures and heuristics have been discovered—see below; this remains an active area of research in the logic community). This, in turn, makes the proofs even harder to understand.

1.1 Theorem-Proving Tools

Many machine proof tools have been proposed and implemented in the last forty years. Initially, much emphasis was placed on tools which would accomplish proofs of mathematical theorems (from group theory, for example). Such tools are generally *automatic*. Either they implement complete algorithms ('uniform proof procedures') guaranteed to find a proof if one exists within their logic; or heuristics ('non-uniform proof procedures') which seek to limit the combinatorial explosion thus obtained, by restricting their search to 'likely' proofs. In either case, the proof is accomplished (or fails) without significant user intervention.

Gradually such methods began to be applied to problems more related to computer science. Tools were implemented which allowed the user significant control over the activity of searching for proofs—via *tactic programming*. Some of the highlights are as follows:

- The Boyer-Moore theorem prover (NQTHM) [BS79] was one of the first to apply theorem proving techniques to program verification tasks (and continues to be developed). It is an automatic theorem-prover, working with a quantifier-free first-order logic, and was also one of the first such tools to have a general induction principle built-in. Guidance to the tool is achieved by having the user propose lemmas which are likely to be useful—the system proves them and then attempts to use them in the construction of its main proof.

- Edinburgh LCF is a British contribution which also dates from the '70s [GMW79]. It was LCF which introduced the notion of a *tactic* as a program for directing the theorem prover, and *tactical* as a higher-order function for combining tactics. LCF is less automatic than NQTHM; after presenting the system with a goal to prove, the user supplies a tactic which directs the system on how to

find the proof. Edinburgh LCF was based on Scott's logic of computable functions and was useful for reasoning about denotational semantics and functional programming. Cambridge LCF [Pau87] extends the logic of Edinburgh LCF and is well-suited to reasoning about domain theory. LCF also introduced the notion of guaranteeing soundness via a *safe datatype*. In LCF objects which represent proofs belong to a datatype `Proof`, and the *only* way to construct proofs is via functions which construct them using primitive axioms and rules—thus ensuring that only sound proofs are created.

- HOL is another tool which has grown out of the work on LCF [Gor88]. HOL uses the same implementation technology (based on ML—which was initially designed as a metalanguage for LCF), but implements a different logic. It has found particular application in the proof of correctness for digital circuits [BGH+92].

- A more recent system is PVS [ORSvH93], which is much more closely targeted upon specification and proof for computer systems. By having a closely-coupled language, type-checker and proof checker (type-checking, for example, may entail some theorem proving), it aims to offer a higher degree of automation than is present in LCF, but to give the user more control over the proof than is possible in NQTHM.

Other tools have been implemented to support particular software development techniques.

- The **B** [Abr91] tool was initially a configurable and extendible proof system, but is now part of a **B** development method, based on *abstract machine notation*. The theorem-proving ability of the tool is mainly used in the 'auto-prover' which checks the soundness of refinement steps.

- *mural* [JJLM91] is a tool which supports the proof activity required by a VDM development. It has a user-interface which is tailored to VDM (displaying the VDM text as it wold appear in a typeset document). Proof may be conducted interactively or via tactic programs—which are structured by arranging rules and tactics into *theories* for dealing with particular datatypes and particular classes of proof.

These tools differ in their implementation technologies and the degree of automation which they offer. Most are configurable and extendible, to enable them to adapt to a user's problem domain, but each supplies a logic and methodology of its own (in most cases, the efficiency of the heuristics offered to the user is heavily dependent on the logic which is implemented).

A more recent development in this field is the idea of a *logical framework*. This is a tool whose only built-in notions are *metalogical* ideas of what it is to make an inference, what comprises a proof, how proofs are constructed from primitive inference rules, what operations on proofs are sound, and so on. These tools are generally supported by an extensive formal (generic) proof theory. Before attempting proofs, the the user must first supply (or select) an *object logic*—an account of a logical system, its syntax and rules of inference.

- The Edinburgh Logical Framework (LF) [HHP91], for example, is based on a typed λ-ealculus, in which logics are represented via a 'judgements as types' principle, whereby each judgement is identified with the type of its proofs.

- Isabelle [Pau89, Pau90] arose out of Paulson's work on LCF—as a more generic approach to theorem proving. It is supplied with many pre-defined object logics, including classical first-order logic, many-sorted first-order logic, Zermelo-Fraenkel set theory, the logic of LCF, etc.

- 2OBJ[GSHH92] has been developed in Oxford (in a joint project with RHBNC), and has not penetrated very far beyond this, maybe due to implementation difficulties. 2OBJ uses a safe datatype, in the style of LCF, but instead of making proofs the safe objects, in 2OBJ the sort (type) of *tactics* is the protected one—tactics can be built only from primitive rules and pre-defined tacticals.

1.2 Tools Applied to Formal Methods

Some of the popular formal methods have had, from the outset, well-defined underlying semantics which has given rise to a workable proof theory and proof tools.

VDM is based on a three-valued logic, and has a domain-theoretic semantics. The standard text on VDM [Jon90] describes how to construct VDM proofs, and so provided the essential groundwork for the implementation of mural—see above. CSP has a number of semantic models (the choice of which being dictated by the power of the results which the user wishes to bring to bear on problems), and two of these (the *traces* model and the *failures/divergences* model) are described in the standard text on CSP [Hoa85]. A model-checking tool (FDR [For92]) based on the latter has recently been implemented.

The focus of the work in this thesis is on Z, which is in (relatively) widespread use for system specification, but for which deductive systems are still being explored. Spivey gave Z a detailed formal semantics [Spi88] after Z had already been in use for some time. More recent work has provided Z with a simplified semantics [GLW91, BN+92] and this has given rise to a reasoning system for Z, named $\mathcal{W}$ in [WB92].

Z is presented as a broad-spectrum formal method. It is envisaged that it should be possible to specify the functional aspects of any computer system in Z, prove that the specification is self-consistent, and refine it (in formal fashion) into executable code. As such, most of the proofs which a Z user is called upon to undertake will not involve deep properties of the specification; they will be unlikely to involve inductive arguments, and they should, therefore, be highly automatable. Moreover, when a proof task is not so straightforward, there remains a significant amount of book-keeping to be done (checking, for example, that partial functions are always applied within their domains); and in this a proof assistant can be very valuable.

Machine support for (and automation of) such proofs is the subject of this thesis. We shall be interested in software engineering concerns (tactic programming, in particular—tactics permit proof re-use, and the re-creation of proofs following specification changes), as well as means of guaranteeing the soundness of the proof tools produced.

A large prototype proof system for Z has been produced. The details of the construction of this tool—using the $\mathcal{W}$ logic, and implemented in 2OBJ—are presented in this thesis. Special attention is paid to soundness concerns, since it is considerably easier to check that a program such as this one produces sound proofs, than to check that each of the impenetrable proofs which it creates is indeed sound. As the first such encoding of $\mathcal{W}$, this helped to find bugs in the presentation in [WB92], and to

demonstrate that $\mathcal{W}$ makes proof in Z tractable (though in this implementation, only just—see Chapter 4).

Two proof tools for Z are commercially available—ProofPower and *Zola*—these will be discussed in Chapter 4, where each is compared to the approach taken in this thesis. Both permit a similar range of proofs to be constructed to those produced by the tool described here. ProofPower is constructed using a version of HOL (see above); and as such, produces proofs which are inherently sound with respect to HOL's logic; *Zola* implements a logic which is closer to $\mathcal{W}$.

One of the problematic areas in the construction of the tool mentioned above was the construction of tactics—the programs which direct proofs. Any that are more complex than simply instructions to the tool to apply a few proof rules in sequence, they become hard to comprehend. As a broad goal, the user seeks very general tactics—ones which will prove a large class of theorems without intervention, and without undue inefficiency. A way to reason about such programs—their semantics, how to transform them without changing their effect, how to specify and re-use them etc.—was needed, and so the second part of the thesis presents such a language, with a formal semantics (independent of any particular tool) and a set of rules for reasoning about tactics written in this language. It includes some case studies, and some ideas on how this tactic language can give rise to lightweight implementations of theorem proving tools. The tool described in some detail is another theorem-prover for Z, this time based on LittleZ [BHW94].

1.3 Using Proof Tools

Several factors affect the usability and value of a proof tool, such as:

- soundness: this has been a large concern in the work presented here. Its importance is relative to the amount of confidence which will be placed in the tool's output, and the extent to which that output will be checked by others.
- user interface: in particular, this includes
  - the interactive component, and
  - the tactic language.

  The first of these will receive little attention here; the thesis attempts to show that the second can be treated in much the same way as any other programming language.
- efficiency of implementation: clearly, there is little value in producing an interactive tool if the user must wait for many minutes between mouse clicks. Conversely, even a highly-automated tool will not be very useful if it can be run only as an overnight batch job.
- the extent of the tactic library: a user wishing to construct proofs about specifications does not wish to spend time repeating proofs of basic laws from the mathematical toolkit—nor proofs of laws that 'should' be in the toolkit but are not. Moreover, the user may reasonably expect to be provided with a set of well-understood proof search procedures—and an adequate specification of their function.

- the level at which the basic laws sit (i.e. whether they deal with points, sets, predicates, functions, schema operations, etc.): this issue can largely be hidden by an extensive tactic library (provided the tool is fast enough), but at some point most users will need to come into contact with the basic laws—which must be comprehensible in the context of a given specification.

These issues are strongly inter-related. The greater the sophistication of the basic laws, the more efficient the implementation—but the harder it is to prove that those laws are sound (both because they are far removed from the semantic definitions and because as they become more specialized, more are required). The nature of the tactic language will affect the way in which the tactic library can be utilized in users' tactics. In the second part of this thesis, a tactic language is described which aims to permit reasoning about tactics via a collection of algebraic laws—thus promoting re-use and refinement.

1.4 Outline

As indicated above, the thesis is presented in two parts. The first describes the production of a proof tool for Z by *encoding* $\mathcal{W}$ in 2OBJ. The second describes a tactic language—a general language, not specific to any particular tool.

Chapter 2 describes the encoding of $\mathcal{W}$ in 2OBJ, and goes on to describe some uses of 2OBJ's tactic programming facilities in enlarging the granularity of proof steps which can be undertaken in $\mathcal{W}$. Chapter 3 uses these rules and tactics to discharge the proof obligations arising in a 'typical' Z specification. Chapter 4 discusses the benefits and difficulties involved in using 2OBJ, and the extent to which $\mathcal{W}$ is appropriate for the business of proof in Z. It also compares the approach taken here with that taken in ProofPower and *Zola*.

Chapter 5 presents the tactic language, mentioned above and gives its formal semantics. It then lists a *complete* set of laws for *manipulating finite* (non-recursive) tactics, before going on to consider how recursion is to be modelled and incorporated in the rule system. The addition of *structural combinators* to the language permits a more succinct expression of certain tactics than is generally possible. Chapter 6 uses this tactic notation and the laws of tactic transformation to describe some tactic case studies, and to demonstrate some of the properties of the tactics presented. In Chapter 7 this tactic semantics is used a basis for a new proof tool for Z, implemented directly in a lazy functional language. This implementation is very much more efficient than that presented in Part I.

Part I

Work with 2OBJ

Chapter 2

Encoding $\mathcal{W}$ in 2OBJ

THIS CHAPTER presents details of the construction of a prototype theorem-proving tool for Z. The tool (named JigsaW) is based on the deductive system (which has been called $\mathcal{W}$ [WB92]) contained in the draft Z standard [BN+92], and is implemented using the 2OBJ metalogical theorem prover [GSHH92]. Much of the material in this chapter has appeared in a paper at FME'93: *Industrial Strength Formal Methods* [Mar93a].

The following sections give an outline of $\mathcal{W}$ and of 2OBJ. Section 2.3 discusses how Z's syntax has been encoded in OBJ3, and Section 2.4 shows how the basic predicate calculus rules are expressed in 2OBJ. Sections 2.5 and 2.6 consider two proof-structuring devices; *rule-lifting* and *tactics*. In Section 2.7 the encoding of $\mathcal{W}$'s rules for reasoning about expressions is explained, and in Section 2.8 the rules which enable Z's specification constructs to be used in proof are considered. The next four sections describe some larger tactics—for dealing automatically with propositional calculus (Section 2.9), expression axioms (Section 2.10) and the mathematical toolkit (Section 2.11). These are brought together in a more general tactic in Section 2.12. The final section (2.13) describes the means by which tactics are used to apply binding substitution rules with care.

2.1 $\mathcal{W}$: A Logic for Z

As Z has grown in popularity, various logics have been proposed for reasoning within it. One such logic is $\mathcal{W}$ [WB92]. $\mathcal{W}$ has the great benefit of having (largely) been proven sound with respect to the semantics of standard Z [BN+92].[1] The logic is a sequent calculus in the style of Gentzen; but since it is for reasoning in Z, it is a *typed* logic. Thus the sequents take the following form:

Declarations | Predicates ⊢ Predicates .

[1] The latter document gives a new presentation of the logic, but this account remains based upon the former.

The sequent is said to be *valid* iff, in an environment augmented by the Declarations, by assuming all the Predicates on the left-hand side (the *antecedents*) it is possible to prove one of those on the right (the *consequents*). Any (or all) of these parts of the sequent may be empty.[2] If there are no predicates or no declarations on the left, the bar is omitted. Rules in $\mathcal{W}$ are written

$$\frac{\text{premisses}}{\text{conclusion}}\ (\text{name})$$

where the conclusion is a sequent, and the premisses consist of zero or more sequents. The rule may also have a side-condition (proviso).

The presentation in [WB92] gives an explicit characterization of bound and free variables, and of substitution, which makes encoding it very straightforward. Also included are rules (axioms) for the basic expressions which occur in Z (rules concerning set membership, cartesian tuple equality, etc.), and rules which permit the definitions introduced in a specification (in schemas, generic definitions etc.) to be incorporated as antecedents.

2.2 2OBJ: A Metalogical Theorem Prover

In order to support an encoding of $\mathcal{W}$ a suitably general theorem-proving assistant is needed. Chapter 1 has discussed the use of *logical frameworks* for this purpose. The tool chosen here was 2OBJ: whilst still being developed, it had the advantage of being produced locally and providing a moderately good user interface.

2OBJ should not be confused with OBJ3 (version 2, [GW88]) upon which it is built. OBJ3 may be viewed as a *term rewriting system*. Programs in OBJ3 consist of *sort* (datatype) declarations and *equations* which are generally used as left-to-right rewrite rules. The system is able to 'reduce' terms, using all of the rewrite rules exhaustively, or to apply individual rewrites to selected terms.[3]

2OBJ consists of a number of OBJ3 modules (the *2OBJ System*), together with an X-windows based user interface to OBJ3 (the *2OBJ Tool*, `X2obj`). The tool provides windows for easy interaction with the underlying OBJ3 system; button presses being converted into OBJ3 input commands. The output from OBJ3 is redirected into a number of windows, so that, for example, proofs under construction can be represented as trees in a manner which corresponds to a pencil-and-paper proof.

2OBJ imposes very few assumptions about the logic being encoded. The OBJ3 modules construct an abstract datatype of `Proofs`, together with operations for constructing such proofs from `Goals` and `Rules`. Such `Rules` may be combined to form `Tactics`. The user must supply OBJ3 modules which define a term algebra for the object language (see below). These are then linked to the 2OBJ system by identifying one form of term (in our case, the *sequent*) with the OBJ3 sort `Goal`. The inference rules of the system are then described as objects of sort `Rule`. The user describes the behaviour of these rules by giving equations for the built-in operator[4]

[2] Readers unfamiliar with this style will be surprised to see sequents such as $\Phi \vdash$. This is valid if and only if the predicates in Φ are contradictory (hence, $\Phi \vdash$ is equivalent to $\Phi \vdash$ false).

[3] 2OBJ extends these options, by permitting rules to be de-selected from the reduction system, and by permitting sets of named rules to be applied exhaustively.

[4] The keyword `op` introduces an operator definition. In this instance, this is a degenerate case of OBJ3's arbitrary *mixfix* syntax. The simple juxtaposition of a `Rule` and a `Goal` forms an application of this operator.

```
op _ _ :  Rule Goal -> GoalList .
```

Thus rules are viewed as functions from `Goals` to lists of `Goals`.

The encoding is comprised of a few modules containing such equations. The user of JigsaW creates a module containing definitions from a Z specification (see, for example, those in Section 2.8 below), importing these modules too. After this module has been loaded into 2OBJ, the user may specify a goal term (using the 2OBJ tool), and the system uses the user-supplied rules, together with built-in rules for manipulating proof trees, to construct a proof tree. This construction is entirely user-driven; to apply a rule to a particular node in the tree, the user simply has to click on that node and specify the rule/tactic to be applied.

The theory underlying 2OBJ is presented in [GSHH92]. It was intended that the implementation of 2OBJ should be shown to conform to its specification in a formal categorical proof theory. This framework would make possible a proof that the encoding is faithful to W—and thus that proofs produced using JigsaW are indeed sound (that is, as sound as W). A methodology for undertaking this proof has not been forthcoming.

2.3 Syntax

The first step in producing an encoding of a logic in 2OBJ, then, is to provide an OBJ3 module defining a term algebra[5] for the logic under consideration. Z has a rich concrete syntax, described using a context-free grammar in the draft Z standard [BN+92]. This can be translated into OBJ3 in a fairly systematic manner, due to OBJ3's order-sorted algebra and arbitrary mixfix operator definitions.

Each of the *main* non-terminal symbols in the grammar becomes an OBJ3 *sort*. In principle *every* non-terminal could be an OBJ3 sort, but since many are not referred to outside the grammar, it suffices to collapse many of the productions in the grammar, using operator precedences to ensure that the same language is described. An example of a part of this encoding is given below. (The expression in square brackets describes the operator precedence; operators with lower numbers bind tighter.)

```
op |A| _ <.> _ :  SchemaText Predicate -> Predicate [ prec 40 ] .
op |E| _ <.> _ :  SchemaText Predicate -> Predicate [ prec 40 ] .
op |El| _ <.> _ :  SchemaText Predicate -> Predicate [ prec 40 ] .
op _ <=> _ :  Predicate Predicate -> Predicate [ prec 35 ] .
op _ => _ :  Predicate Predicate -> Predicate [ prec 32 ] .
op _ V _ :  Predicate Predicate -> Predicate [ prec 30 ] .
op _ ^ _ :  Predicate Predicate -> Predicate [ prec 26 ] .
```

Some of the non-terminals are given by productions with potential repetitions; these are represented using extra sorts. For example, sequence displays are defined as follows:

Expression5 = ... | Sequence | ...
Sequence = '⟨', Expression0, { ',', Expression0},'⟩'

In OBJ3, this is expressed (with all the Expression classes collapsed into one) by defining the comma as an associative operator which forms lists of expressions from

[5] That is, the constants and operator symbols for the language, together with laws describing which strings of symbols make valid terms.

shorter lists; single expressions being the simplest of those lists. Sequences are formed by surrounding such lists of expressions with angle brackets.

```
sort Expressions .
subsort Expression < Expressions .
op _ , _ :  Expressions Expressions -> Expressions [ assoc ] .

op < _ > :  Expressions -> Expression [ prec 15 ] .
```

The grammar is also careful to specify the rôle of parentheses in Z. This is slightly unfortunate, in that parentheses in OBJ3 have a built-in meaning—they are used to modify operator precedence. In most cases, Z uses them for the same purpose, and to avoid circularity in the grammar. By explicitly giving operator precedence to the symbols being defined, and collapsing together some of the non-terminals, explicit mentioning of parentheses in the grammar can be avoided. The parentheses defining tuples, though, have genuine syntactic value. Because in OBJ3 they serve only to group objects, this definition

```
op ( _ ) :  Expressions -> Expression .
```

is meaningless. Instead, we write

```
op Tuple( _ ) :  Expressions -> Expression .
```

There are few instances of this type of problem, so the result of this activity is a concrete syntax which is tolerably readable. The precise choices for concrete syntax were entirely arbitrary; chosen for readability and as a reasonable approximation to typeset Z. Good concrete syntax makes the encoding of rules easy to read and so increases confidence in the accuracy of the code.

Type-Checking

As yet, no attempt has been made to include type-checking in the encoding. This is a significant problem as, clearly, the soundness of the logic is dependent on its input being well-typed. $\mathcal{W}$ ensures that most of the inference rules preserve type-correctness. The only exception is *cut*, which introduces new predicates (*cut* is discussed at greater length in Section 2.6 below).

$$\frac{e \mid \Phi \vdash p, \Psi \qquad e \mid p, \Phi \vdash \Psi}{e \mid \Phi \vdash \Psi}\ (cut(p))$$

Therefore, the minimum requirement is that both initial goals, and sequents produced by *cut* (in backward reasoning) be type-checked. Having no mechanism available for type inference means that all generics must be fully instantiated at input time (and hence at all points of interaction with the tool)—writing $\varnothing[X]$ instead of merely $\varnothing$ and $(S, T) \in (\_ \subseteq \_)[X]$ instead of $S \subseteq T$, for example.

This is consistent with $\mathcal{W}$'s approach—the logic simply assumes that all the terms it encounters are well-typed, and that all generics are fully instantiated. In 'pen-and-paper' reasoning, such details can often be overlooked; this option is not available since the tool must work entirely formally. We could extend the logic to include another form of judgement; one indicating that a particular expression has a particular type. This would permit type-checking to be performed at the same time as proof (generally via automatic tactics, since Z's type system is decidable), allowing generic parameters

to be supplied when necessary. However, it would mean that the logic being encoded was far removed from $\mathcal{W}$. Section 4.2 considers this issue further.

The 2OBJ documentation [SH92] suggests using *sort constraints* in encoding the grammar, so that terms will be syntactically well-formed only if they are type-correct. Support for this has not been implemented, and although this is appealing, even if it is possible to express the Z type system in this way, performance of the tool is likely to render it useless (parsing is already very slow).

Another possibility is to pre-process the user's OBJ3 code using a tool similar to *f*UZZ [Spi92b], both checking for type-correctness and providing generic parameters. The *cut* rule presents a problem in this scheme (since its parameter needs to be type-checked, but is not (in general) available to be pre-processed). It would be possible to have the rule write its resulting sequents out to a file which could later be type-checked (the soundness of the proof being dependent on the success of the type-checking). This is problematic because *LIFT* (see Section 2.5) introduces nested scopes. When *cut* is used within the lifted proof, the cut term may contain variables which are in scope in the context of the *LIFT*, but not in global scope; their types may not, therefore, be readily apparent unless each application of *LIFT* also makes an entry in the file.

$\mathcal{W}$ Meta-Functions and Syntax Extensions

Since 2OBJ assumes very little about the logic being encoded, it is necessary to define the sequent explicitly[6] (identifying it with the sort `Goal` mentioned above), and notions of free variables (ϕ), alphabet of declarations(α), and substitution. These are carefully defined in the presentation of $\mathcal{W}$ [WB92] and/or the semantics [BN+92], using sets of equations.

Substitution, for example (accomplished using explicit bindings) is specified with expressions like

$$\begin{aligned} b.(\forall d \mid p \bullet q) &\equiv \forall b.d \mid (\alpha d \lhd b).p \bullet (\alpha d \lhd b).q \\ &\quad \text{provided } \alpha d \cap \phi_e(\alpha d \lhd b) = \varnothing \\ b.(\neg p) &\equiv \neg\, b.p \\ b.(p \wedge q) &\equiv b.p \wedge b.q \\ b.(p \vee q) &\equiv b.p \vee b.q \end{aligned}$$

These equations translate directly into OBJ3 (the keyword `eq` introduces an equation, `ceq` introduces a conditional equation):

```
ceq (b .  (|A| d | p <.> q)) = (|A| (b .  d) |
        ((alpha(d) \dsub b) .  p)<.> ((alpha(d) \dsub b) .  q))
    if ((alpha(d) inter phie(alpha(d) \dsub b)) == *nil*) .
eq (b .  (~ p)) = ~ (b .  p) .
eq (b .  (p ^ q)) = (b .  p) ^ (b .  q) .
eq (b .  (p V q)) = (b .  p) V (b .  q) .
```

As immediate substitution is not always required (for example, the presentation of Leibniz's rule requires that there be a predicate present of the form $(\!| x \rightsquigarrow t |\!).p$) these

[6] Note that whereas in $\mathcal{W}$ missing declarations and predicates are denoted by white space, pattern-matching in the encoding is greatly aided by inclusion of symbols for empty declarations and empty lists of predicates. As a result, the empty sequent ($\vdash$) is denoted by `% | * |- *`.

rules are presented using 2OBJ's ability to 'turn off' rewrites; they are used only when the rule *subst* is selected. Some rewrite rules must not be applied exhaustively (as *subst* does)—they are applied when necessary via a *subst* tactic (see Section 2.13).

The equations for α and ϕ are presented in a similar manner. Since they are features common to most logics, 2OBJ provides fast built-in operators for implementing them. These operators do not appear to be sufficient to describe Z's binding constructs, and so they are encoded directly. Section 4.6 considers this matter further.

2.4 General Rules

$\mathcal{W}$ is based on a classical sequent calculus, and thus includes a full set of well-known inference rules. For example, the classical rules for *or*-introduction on the right and the left could be written in $\mathcal{W}$ as

$$\frac{d \mid \Phi \vdash \Psi, p, q}{d \mid \Phi \vdash \Psi, p \lor q}\ (\vdash\lor) \qquad \frac{d \mid \Phi, p \vdash \Psi \qquad d \mid \Phi, q \vdash \Psi}{d \mid \Phi, p \lor q \vdash \Psi}\ (\lor\vdash)$$

However, in the presentation of $\mathcal{W}$ [WB92], rules are presented in the following simplified form, together with a theorem on *rule-lifting* (this theorem is reproduced in Section 2.5 below):

$$\frac{\vdash p, q}{\vdash p \lor q}\ (\vdash\lor) \qquad \frac{p \vdash \qquad q \vdash}{p \lor q \vdash}\ (\lor\vdash)$$

It is convenient to present the encoding in a similar way, with simple rules together with a meta-rule for rule-lifting (also discussed below). Thus two rules above are implemented by `|-or` and `or|-` (these are defined as constant operators of sort `Rule`; recall that a rule juxtaposed with a goal forms an instance of the rule application operator, so these equations are between `GoalLists`):

```
op |-or :  -> Rule .
eq |-or (% | * |- p V q) = (% | * |- p , q) .

op or|- :  -> Rule .
eq or|- (% | p V q |- *) = (% | p |- *) , (% | q |- *) .
```

The *assumption* rule is distinctive in that it has no premiss:

$$\frac{}{d \mid p \vdash p}\ (\textit{assumption}) \ ,$$

and so the 2OBJ rule generates an empty list of subgoals:

```
op assumption :  -> Rule .
eq assumption (d | p |- p ) = [] .
```

This implementation leads to a most uncluttered encoding, which is easily seen to be correct, and can also easily be verified correct (that is, faithful to $\mathcal{W}$).

2.5 Lifting

The theorem on *rule-lifting* serves both to simplify the presentation of $\mathcal{W}$ (by making it easier to read) and to help structure the proof that $\mathcal{W}$ is sound with respect to Z's semantics. It factors-out elements which would otherwise be common to each rule.

Theorem 2.5.1 **(Rule-lifting)**

If the inference rule $\dfrac{e;\ d' \mid \Psi' \vdash \Phi'}{e;\ d \mid \Psi \vdash \Phi}$ *is sound,*

then the rule $\dfrac{f;\ e;\ d' \mid p, \Psi' \vdash q, \Phi'}{f;\ e;\ d \mid p, \Psi \vdash q, \Phi}$ *is also sound,*

providing that $(\alpha d \cup \alpha d') \cap (\phi p \cup \phi q) = \varnothing$.

The theorem could readily be generalized to cover rules (proofs) with more than one premiss.

For similar reasons, it is useful to provide a meta-rule in the encoding (justified by this theorem) which takes a rule, R, and some collection of terms from the current goal, and returns a new goal which is the result of applying R to the selected terms, leaving the other terms unchanged (read $\uparrow$ ($\downarrow$) as selecting (excluding) predicates/declarations indicated by position number, hence $(p,q,r,s)\uparrow(1\ 3) \equiv (p,r)$ and $(p,q,r,s)\downarrow(1\ 3) \equiv (q,s)$):

$$\frac{d';\ d\downarrow i \mid \Phi';\ \Phi\downarrow j \vdash \Psi';\ \Psi\downarrow k}{d \mid \Phi \vdash \Psi}\ (LIFT(i,j,k,R))$$

whenever $\dfrac{d' \mid \Phi' \vdash \Psi'}{d\uparrow i \mid \Phi\uparrow j \vdash \Psi\uparrow k}\ (R)$

provided $((\alpha(d\uparrow i) \cup \alpha(d')) \setminus (\alpha(d\uparrow i) \cap \alpha(d'))) \cap (\Phi\downarrow j \cup \Psi\downarrow i) = \varnothing$.[7]

In some languages, this description would almost serve to define rule-lifting. This, however, is merely a specification of some rather ugly OBJ3 code, which is not reproduced here. Proving that this specification of the rule is sound, and proving that the implementation of rule-lifting satisfies it, is one of the major outstanding questions regarding the demonstration of soundness for JigsaW (see Section 4.7).

The reason why this rule is needed may not be immediately clear. Systems such as LCF [Pau87] have no comparable construction. The problem lies in the fact that 2OBJ is a *logical framework* whereas LCF implements a particular logic. The rule-lifting takes place at a very low level in the latter; in the former it must be defined by the user (the author of the encoding). If the logic under consideration were more unusual (linear logic, for example) the forms of lifting which would produce faithful encodings would be much more restricted; this is why lifting cannot easily be built-in to 2OBJ.

Possible Variations

In Andrew Stevens' encoding of first order predicate calculus in 2OBJ (the example encoding in [SH92]), each of the primitive rules is expressed in its full form, with lifting 'built-in'. For example:

```
op ore : NzInt -> Rule .
ceq ore (N) (H |- X) =
        ( H ; hyp(N,H) : 1 |- X ), ( H ; hyp(N,H) : 2 |- X )
    if matches( Z v Y, hyp(N, H) ) .
```

which simply makes this inference:

$$\frac{H;\ Z \vdash X \qquad H;\ Y \vdash Z}{H \vdash X}\ (ore(N))\ \text{whenever } H\uparrow N = Z \vee Y\ .$$

This makes the rule hard to read—and verify—and tends to make tactics hard to write. Moreover, the side condition present in the rule-lifting theorem would need to be duplicated in all the rules which modify the declaration part of the sequent.

By contrast, if it were seen as desirable to have a rule such as the above in Jigsa*W*, it could readily be defined (as a tactic) using *LIFT* (a '0' is used as an argument to LIFT to denote parts of the sequent from which nothing is to be lifted):

```
op OR|- :  NzNat -> Tactic .
eq OR|- (n) Seq = LIFT(0,n,0,or|-) .
```

The use of lifting within tactics leads to some interesting results; see Section 2.6 below.

This form of rule lifting is tied closely to counting the positions of predicates (and declarations) in lists. Another approach might be to make use of associative/commutative matching, writing patterns which would match lists (sets) of predicates containing one to which the rule would apply. A problem here would be that in the case of multiple matches, one would need a means of indicating to which predicate the rule is to be applied.

Again, if a rule of this sort—one which matches any applicable part of the goal—is needed, it can be written using *LIFT*, with an auxiliary function `find` which finds a match (using the 2OBJ built-in `matches`) in a list of predicates and returns its position number:

```
op |-OR : -> Tactic .
eq |-OR (d | PHI |- PSI) = LIFT(0,0,find(p V q,PSI),|-or) .
```

Thus the high-level *LIFT* meta-rule appears to be a very general formulation, both making derived rules and tactics very easy to write and making any proofs about the encoding easy to structure, since it corresponds well with the original presentation. Similar notions of lifting are present in Isabelle [Pau89] (where lifting over assumptions and over quantification (declaration parts) are treated separately) and in the Edinburgh LF [HHP91, Chapter 4] presentation of first-order logic. Those presentations are in the context of natural deduction; this formulation with a sequent calculus would appear to be slightly novel.

The greatest problem with this approach is that there is much potential for inefficiency. Having the selection of predicates from goals as a high-level operation (on a par with tactic interpretation) rather than as a fast built-in, hidden from the user, leaves the user free to write very inefficient tactics which frequently pull sequents apart and then put them back together again. However, the judicious combination of *LIFT* with tactics can lead to very efficient tactics where each rule is directly applicable to the goal at hand, with no need for searching or selection (see below, example on page 19).

Whilst lifting aids the construction of tactics, it can be rather difficult to use interactively: counting predicates as they are printed on the screen is very error-prone. A good user interface would permit clicking on the predicates to which a rule is to be applied. The rule could be wrapped in a *LIFT* whenever the form of the selected goal demanded it.

2.6 Tactics

Tactics are programs which perform proofs. A tactic captures the essence of a formal proof, in some sense, and thus storing the tactic enables the proof to be repeated at a

later date, maybe in differing circumstances. This is an important software engineering consideration since a system specification may be expected to change from time to time, and it should be possible to check that theorems are still provable without unnecessary additional effort. The proof performed by a tactic may be very general, in which case the tactic is available as a derived inference rule.

The fact that the tactic is built from primitive rules means that the proof will succeed only if the side-conditions on those rules are met, even though they are not (usually) mentioned explicitly in the tactic. By virtue of being built solely from sound inference rules, tactics may make large inference steps without any need for additional proofs of soundness beyond those necessary for the primitive rules. In making reasoning steps, the user need not be aware which are the result of rules, and which are implemented by tactics. Thus, for example, although the definitions from the Z mathematical toolkit are provided, the user will not generally need to be concerned with the details of the rules for applying generic definitions; instead a toolkit tactic will bring the relevant definition into the antecedent, apply it to the selected term, and remove any unwanted definitions. An example of such a tactic is given below.

2OBJ presents the user with a rich tactic language, and much of the remainder of this chapter (and the following one) will be spent considering the construction of suitable tactics for proof in Z. Such tactics—like most programs—often become sufficiently large that their effects are not clear. Part II of this thesis is given over to finding succinct, formal descriptions of the actions of tactics.

Tactic Language

In 2OBJ the tactic language is an extension of the language of rules. All rules can be considered as tactics, and tactics can be combined using the following LCF-style *tacticals* (t_1 and t_2 are tactics, *ts* is a list of tactics):

t_1 *THEN* t_2 (sequential composition) applies t_1 to the goal then applies t_2 to the resulting subgoal(s).

t_1 *THENL* *ts* (parallel composition) applies t_1 to the goal, then applies the tactics in *ts* *zip*-wise to the resulting goals. Thus, for example, if t_1 produces three subgoals, *ts* must consist of three tactics; *ts*[1] is applied to the first subgoal, *ts*[2] to the second, etc.

t_1 *ELSE* t_2 applies t_1 to the goal; if it succeeds, the tactic terminates, otherwise t_2 is applied *to the initial goal.*

As the tactics are written in OBJ3 they may be arbitrarily complex. Tactics may call other tactics, in a functional programming style, and recursion is available, making the language very powerful. Thus these tacticals may be used to construct various other familiar programming language features. The 2OBJ documentation [SH92] suggests tactics for iteration (*REPEAT*) and exhaustive application of rules (*EXHAUST*). Both are recursive, *REPEAT* applying the supplied tactic and then calling itself with an index which is the predecessor of the original index; and *EXHAUST* using *ELSE* to continue execution until application of the supplied tactic fails.

```
op REPEAT : Nat ProofTactic -> Tactic .
eq REPEAT( N, PT ) Seq = if N == 0 then idtac
                  else PT THEN REPEAT(p(N) , PT) fi .
```

```
op EXHAUST : ProofTactic -> Tactic .
eq EXHAUST( PT ) Seq = (PT THEN EXHAUST(PT)) ELSE idtac .
```

`idtac` is an identity tactic; it always succeeds, leaving the goal to which it is applied unchanged. `Seq` is a place-holder denoting any sequent—see below for tactics defined as applying to more specific goals. The presence of a goal term on the left-hand side permits tactics to behave differently on different goals, and allows patterns matched as being in the present goal to be parameters to the right-hand side (see, for example, the definition of `BINDINGMEM` below). That the goal term does not appear on the right-hand side is one of the more peculiar features of 2OBJ's tactic interpreter.

Often-used forms

Tactics are made more general by being parametrised. *CUT* takes a parameter which gives the predicate to be cut into the goal. The 'raw' *cut* rule in $\mathcal{W}$ is applicable only to a goal consisting of an empty sequent, an unlikely goal:

$$\frac{p \vdash \qquad \vdash p}{\vdash} \ (cut(p)) \ .$$

However, writing

```
op CUT : Predicate -> Tactic .
eq CUT (p) Seq = LIFT(0,0,0,cut(d)) .
```

yields the more useful rule

$$\frac{e \mid \Phi \vdash p, \Psi \qquad e \mid p, \Phi \vdash \Psi}{e \mid \Phi \vdash \Psi} \ (CUT(p)) \ .$$

Often, *cut* is used to introduce a lemma (or theorem, or axiom) for which a proof (tactic) already exists. *CUT* can be used to construct a tactic which introduces the predicate p to a goal's hypothesis list, proving $\vdash p$ using the provided tactic, *PT*.

```
op CUTLEM : Predicate ProofTactic -> Tactic .
eq CUTLEM (p, PT) Seq = CUT(p) THENL(LIFT(0,0,1,PT), idtac) .
```

Many axioms (see below) are expressed in the form $\vdash p \Leftrightarrow q$, from which it is a simple matter to prove, for example, the validity of the following inference rule

$$\frac{\vdash q}{\vdash p} \ .$$

The transformation from an equivalence to an inference rule is accomplished by the tactic `|-EQULIFT=>`. As the goal must match one side of the equivalence (the left-hand side, in this case), this tactic uses pattern matching on the goal to ensure that it is only applied where appropriate. 2OBJ's tactics are defined using an equation involving a goal (in the above, any goal `Seq` is satisfactory) so that the tactic's action can be conditional on the form of the goal.

`|-EQULIFT=>` uses `CUTLEM` (above) to introduce $p \Leftrightarrow q$ to the hypothesis list, and then splits it apart using $\Leftrightarrow\vdash$ and $\Rightarrow\vdash$. The latter produces two subgoals: the first is the required sequent (reduced to $\vdash q$ by *thin*) and the second is of the form $p, \ldots \vdash p$; which is discharged by *assumption*.

```
op |-EQULIFT=> :  Predicate ProofTactic -> Tactic .
eq |-EQULIFT=> (p <=> q, PT) (% | * |- p) =
    ( CUTLEM (p <=> q, PT) THEN LIFT(0,1,0,equ|-)
      THEN LIFT(0,2,0,imp|-)
      THENL (THIN(0,1,2), LIFT(0,1,1,assumption) ) ) .
```

The goal-term is not necessary in this case; it simply avoids the inefficiency of applying the tactic when it is certain to fail. It also serves to ensure that the position numbers supplied to *LIFT* are correct (i.e. that there are no spurious predicates present which might become involved in the proof); the tactic will, in general, need to be *lift*-ed before use. The tactic could be more concisely written, with the user merely supplying q, and the tactic forming the predicate $p \Leftrightarrow q$ to be supplied to *CUTLEM*. This is of little consequence, as |-EQULIFT=> will generally be used within other tactics, as illustrated below.

Tactic Transformation

With such a rich tactic language, there will be many tactic forms which will be functionally equivalent. It has already been noted that lifting can be used with tactics to create efficient new tactics. *CUTLEM*, for example, can be more efficiently expressed with the *THENL* within the scope of the *LIFT*:

```
eq CUTLEM (p, PT) Seq = LIFT(0,0,0,(cut(p) THENL(PT, idtac))) .
```

This uses only one instance of *LIFT* instead of the two above. A more concrete example is this derivation involving the axiom of extension (expressed here as an inference. See page 20 for a description of the rule and tactic.)

$$\dfrac{\dfrac{d \mid \Phi \vdash \forall x : t \bullet x \in u, \Psi \qquad d \mid \Phi \vdash \forall x : u \bullet x \in t, \Psi}{d \mid \Phi \vdash \forall x : t \bullet x \in u \wedge \forall x : u \bullet x \in t, \Psi}\ (\vdash\wedge)}{d \mid \Phi \vdash t = u, \Psi}\ (\vdash \mathit{EXTENSION}(x))$$

which could be programmed as

```
LIFT(0,0,1,|-EXTENSION(x)) THEN LIFT(0,0,1,|-and)
```

but is better written as

```
LIFT(0,0,1,|-EXTENSION(x) THEN |-and) .
```

It seems that rule lifting frequently distributes through *THEN*. Similarly some tactics expressed using pattern matching to take different actions depending on the form of the goal could also be written using *ELSE*. Some ways of expressing tactics will be much more efficient than others. Chapters 5 and 6 explore tactic equivalences which could be used to transform tactics into their most efficient form. It may even be worthwhile to have OBJ3 undertake such a transformation before applying the tactic.

2.7 Expressions

In order to reason about Z specifications, *W* provides a number of axioms which describe how sets and functions and the predicate calculus are related. There is also a theorem which permits axioms to be expressed as premiss-free inference rules, so

that, for example, the axiom concerning binding membership becomes a premiss-free inference rule.

$$\vdash b \in S \Leftrightarrow b.S \qquad \text{becomes} \qquad \frac{}{\vdash b \in S \Leftrightarrow b.S}\ (\mathit{bindingMem}) \ .$$

Such rules are readily implemented as rules which produce no new subgoals, but to apply them in this form would be tedious in the extreme. Fortunately, it is easy to incorporate them in a tactic which makes the rule very usable:

$$\frac{\vdash b.S}{\vdash b \in S}\ (\vdash \mathit{BINDINGMEM}) \ .$$

```
op |-BINDINGMEM : -> Tactic .
eq |-BINDINGMEM (% | * |- b \in S) =
    |-EQULIFT=>(b \in S <=> b .  S, bindingMem] .
```

This scheme also allows parameters to be provided to the inference rules. For example, the axiom of extension quantifies over a variable, with certain freeness conditions:

$$\vdash t = u \Leftrightarrow \forall x : t \bullet x \in u \wedge \forall x : u \bullet x \in t \quad \text{provided } x \notin (\phi_e u \cup \phi_e t) \ .$$

The tactic |-EXTENSION implements this axiom as an inference rule, allowing the user to choose the bound variable (its freeness being assured by the rule `extension`).

```
op extension : -> Rule .
eq extension (% | * |- (t = u) <=> ((|A| (x :  t) <.> (x \in u)) ^
                    (|A| (x :  u) <.> (x \in t)))) = []
    if (((x) inter (phie(u) union phie(t))) == *nil*) .
op |-EXTENSION : Word -> Tactic .
eq |-EXTENSION(x) (% | * |- t = u) =
    |-EQULIFT=>(t = u <=> |A| x :  t <.> x \in t ^
                          |A| x :  u <.> x \in t, extension) .
```

Section 2.7 illustrates the use of this tactic. Moreover, when |-EXTENSION is applied it will invariably be followed by $\vdash\wedge$ and $\vdash\forall$, so it may be bundled into a tactic which does precisely this, and automatically chooses a fresh bound variable:

```
op |-EXT-TAC : -> Tactic .
eq |-EXT-TAC (% | * |- t = u) =
    |-EXTENSION(new(x,x)) THEN |-and THEN |-all .
```

So this tactic makes a (relatively) large reasoning step:

$$\frac{x_1 : t \vdash x \in u \qquad x_1 : u \vdash x \in t}{\vdash t = u}\ (\text{|-EXT-TAC}) \ .$$

Rules which are expressed using ellipses present the greatest difficulty. Their presentation in $\mathcal{W}$ is essentially informal. Before they can be encoded, they must be formalized. So

$$\vdash (t_1, \ldots, t_n) = (u_1, \ldots, u_n) \Leftrightarrow t_1 = u_1 \wedge \ldots \wedge t_n = u_n$$

is more precisely expressed as

$$\begin{array}{ll} \vdash ts = us \Leftrightarrow \xi(ts, us) & \\ \quad \text{where} & \xi(t, u) \equiv (t = u) \\ \quad \text{and} & \xi((t, ts), (u, us)) \equiv (t = u) \wedge \xi(ts, us) \ . \end{array}$$

Once the axiom is expressed in this form, an OBJ3 implementation becomes natural:

```
op cartProdEqu :  -> Rule .
op mkeqconj :  Expressions Expressions -> Predicate .
eq mkeqconj (t,u) = (t = u) .
eq mkeqconj ((t,ts),(u,us)) = (t = u) ^ mkeqconj(ts,us) .

cq cartProdEqu (% | * |- (Tuple(ts) = Tuple(us)) <=> p) = []
    if (p == mkeqconj(ts,us)) .
```

and, again, a tactic makes it usable:

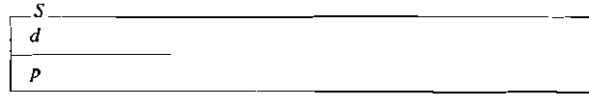
```
op |-CARTPRODEQ : -> Tactic .
eq |-CARTPRODEQ (% | * |- Tuple(ts) = Tuple(us)) =
    |-EQULIFT=>((Tuple(ts) = Tuple(us)) <=>
                mkeqconj(ts,us) , cartProdEqu) .
```

2.8 Declarations

$\mathcal{W}$ includes rules for making use of the large structuring constructs in a Z specification: schemas, axiomatic/generic definitions, etc. These inference rules are implemented as 2OBJ `Rules` with schemas etc. as parameters.

Schemas

The schema



permits this inference to be made:

$$\frac{S = [d \mid p] \vdash}{\vdash} \quad .$$

Which is expressed in 2OBJ as

```
op schdef :  SchemaDef -> Rule .
eq schdef (SCH S IS d ST p END) (% | * |- *) =
                          (% | S = [ d | p ] |- *) .
```

Clearly the user does not want to type out the schema definition each time it is used, so we encode the definitions from a given Z specification in an OBJ3 module. By defining an auxiliary operator

```
op _ -def :  SchemaName -> SchemaDef .
```

we enable the user to make definitions like

```
eq S -def = SCH S IS x :  T ST x \in U END .
```

and refer to `S -def` in invoking the rule. In fact, this is made still easier by a tactic:

```
op SCHDEF : SchemaName -> Tactic .
eq SCHDEF (S) Seq = LIFT(0,0,0,schdef(S -def)) .
```

This encoding also makes alphabet calculations very straightforward: when the `alpha` function encounters a schema name `S` as part of a declaration, it tries to expand `S -def`.

The primitive inference rules are not particularly useful, in that the predicates introduced to the sequent will invariably be used with an application of Leibniz's rule. So, for schemas, for example, we have a tactic:

```
op APPLY-SCHDEF : SchemaName -> Tactic .
eq APPLY-SCHDEF (S) (% | * |- p) =
    ( SCHDEF(S) THEN
      |-LEIBNIZ THEN
      THIN(0,1,0) THEN
      subst THEN
      EXHAUST(|-and) ) .
```

This brings the schema equality into the assumptions, uses it to rewrite the right-hand side, removes the definition again, applies subst (as `|-LEIBNIZ` introduces a binding, rather than actually rewriting the right-hand side) and then exhaustively applies $\vdash\wedge$ to split the goal into its constituent parts. For example, given

$$S == [x : \mathbb{N} \mid x \leq 6] \ ,$$

we infer

$$\frac{\vdash x \in \mathbb{N} \qquad \vdash x \leq 6}{\vdash S} \ \texttt{(APPLY-SCHDEF(S))}$$

The tactic `AUTO-SCHDEF` uses `APPLY-SCHDEF`, choosing the schema name from the form of the goal.

```
op AUTO-SCHDEF : -> Tactic .
eq AUTO-SCHDEF (% | * |- S) = APPLY-SCHDEF(S) .
eq AUTO-SCHDEF (% | * |- b . S) = APPLY-SCHDEF(S) .
```

Since schemas are often nested, application of `EXHAUST(AUTO-SCHDEF)` is a common paradigm in tactics where schema definitions need to be expanded, for example in the tactic for initial state theorem proofs.

Generic Definitions

In a similar way, whilst the generic definitions which comprise the Z mathematical toolkit are available to the user of JigsaW, they will generally be used via tactics which hide the instances of `GenDef` (the equivalent for generic definitions of `schdef` for schemas). For example, dom has the following definition

```
let domdef = ( GEN [ X , Y ] BAR
        (dom :  (PP(X \cross Y) --> PP(X)))
    ST
        ([A] R : PP(X \cross Y) <.> dom R =
            { (x :  X) ; (y :  Y) | ((x |-> y) \in R) <.> x })
    END ) .
```

and the tactic below makes, for example, the inferences shown in Figure 2.1.

```
op |-DOM-TAC(| _ |, _) :  Expressions Expression -> Tactic .
eq |-DOM-TAC(|X,Y|, S) (% | * |- p) =
(   GENDEF(domdef,{X,Y}) THEN
    CUT(<| R  > S |> \in [ R : PP(X \cross Y) ]}
    THENL
    (
        ( THIN(0,1 2,2) THEN |-BINDINGMEM THEN subst THEN
            |-POWERSET(x) THEN |-all )
    ,
        ( LIFT(0,1 3,0,all|-) THEN THIN(0,1 2 4,0) THEN subst THEN
            LIFT(0,1,1,|-LEIBNIZ) THEN THIN(0,1,0) )
    )
) .
```

The aim is to simplify a goal of the form $\vdash z \in \mathrm{dom}[X, Y]S$. Application of *GenDef* introduces the signature and predicate from the definition of dom as antecedents. The *cut* rule introduces a binding which can be used to specialize the universal quantifier from the definition of dom (i.e. it identifies R in the definition with S in the goal). This produces two subgoals. The first asserts that the chosen binding belongs to the correct (schema) type. This is simplified, using *bindingMem* etc., to an assertion that S is a subset of $X \times Y$. The second subgoal is rewritten using $\forall \vdash$, *thin* and *subst* so that the antecedent contains a definition of $\mathrm{dom}[X, Y]S$ using a set comprehension. Leibniz's rule is used to instantiate that definition, and then the rule of *thin* is used to remove the definition from the antecedent.

Abbreviations

One further class of definitions found in Z is the abbreviation definitions:

$$\varnothing[X] == \{x : X \mid false\} \ .$$

$\mathcal{W}$ has, so far, given no rule for dealing with such definitions, but in order to accomplish useful proofs, a way is needed of dealing with them (for example, the definition of $\varnothing$ is essential in many initial state theorem proofs). The abbreviation is intended as shorthand for the generic definition

$$\begin{array}{|l}
\hline
[X] \\
\varnothing : \mathbb{P} X \\
\hline
\varnothing = \{x : X \mid false\} \\
\hline
\end{array}$$

but to make this translation, type inference is needed—which we do not have. The encoding used, for the time being, keeps to the spirit of the abbreviation; it is implemented as a direct OBJ3 rewrite:

```
[axiom emptydef] eq \empty [ t ] =
                    (<| X  > t |> .  { x :  X | False }) .
```

The binding is used to prevent any problems of variable capture: it allows substitution to remain in the $\mathcal{W}$ scheme, rather than conforming to OBJ3's ideas of rewriting. The labels '`axiom`' and '`emptydef`' serve respectively to prevent the rule from being used as a general rewrite, and to identify the rule when the user wishes to apply it. The rule is applied using `|-apply`, which uses the 2OBJ operation `namedred` to invoke a named rewrite rule in the consequent. (An analogous `apply|-` also exists.)

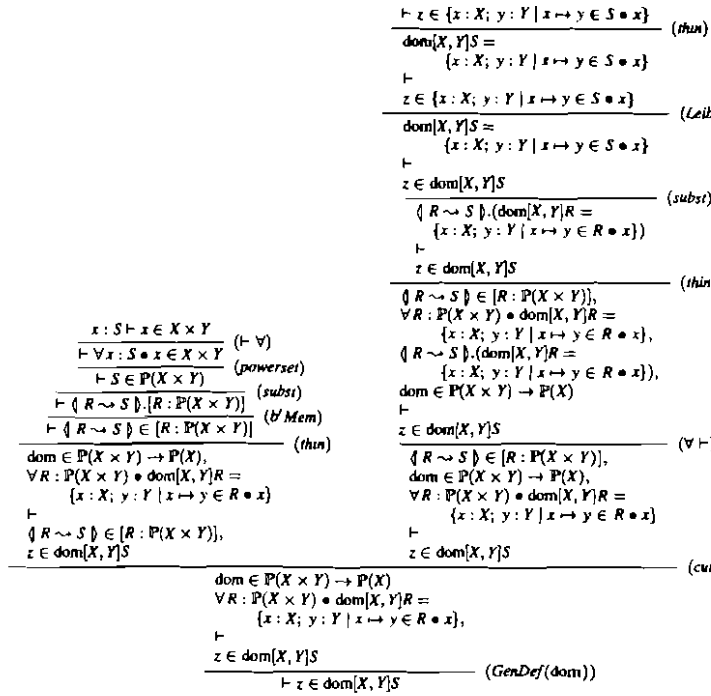


Figure 2.1: Application of | -DOM-TAC

A common use of this definition is to discharge goals of the form $x \in \varnothing[X] \vdash$ by reducing the left-hand side to *false*. This is accomplished by EMPTY-TAC|-.

```
op EMPTY-TAC|- : -> Tactic .
eq EMPTY-TAC|- (% | t \in \empty [ u ] |- *) =
    ( apply|-('emptydef) THEN subst
      THEN SETCOMP|- THEN
      exist|- THEN
      LIFT(0,1,0,and|-) THEN
      LIFT(0,1,0,false|-) ) .
```

This tactic exhibits a subtle problem: the definition of $\varnothing$ mentions a variable, x. Following the application of $(\exists\vdash)$, x is present in the declaration part of the schema, so to satisfy the side condition on *LIFT*, x must not appear free in the goal sequent, otherwise the tactic fails. Since the tactic does not produce any new subgoals, this problem could be avoided by beginning the tactic with a $TRY(chHypVars(| x \rightsquigarrow x_1 |))$. For a tactic which returns a new goal, however, such a renaming might be disconcerting for the user. $\mathcal{W}$ allows for α-conversion under the quantifier, but this has not been implemented.

2.9 Decision Procedures

In using a logic like $\mathcal{W}$, it quickly becomes apparent that at many (indeed, most) points in the proof the choice of which rule to apply is entirely determined by the form of the goal. No creativity is required, and so a simple tactic can be constructed to enable this choice to be made automatically.

An obvious target for this form of automated reasoning is the propositional calculus; for any given proposition at most one rule is appropriate. This rule is selected by PROP-TAC:

```
eq PROP-TAC (% | * |- p ^ q) = |-and .
eq PROP-TAC (% | * |- p V q) = |-or .
eq PROP-TAC (% | * |- ~ p) = |-not .
eq PROP-TAC (% | * |- p => q) = |-imp .
    :
eq PROP-TAC (% | p ^ q |- *) = and|- .
eq PROP-TAC (% | p V q |- *) = or|- .
eq PROP-TAC (% | ~ p |- *) = not|- .
eq PROP-TAC (% | p => q |- *) = imp|- .
    :
eq PROP-TAC DEFAULT Seq = failtac .
```

DEFAULT is a special tactic keyword which enables a tactic to be defined as having a particular behaviour in the event that none of the supplied patterns matches the current goal.

In general, we will wish to apply PROP-TAC exhaustively across the sequent. The EXHAUST tactic previously presented is not sufficient for this task, as it applies tactics to the sequent *as a whole*, whereas PROP-TAC, like many of the primitive rules, applies only to sequents containing single predicates. As a result, we define MPROP, which uses LIFT to apply PROP-TAC to each term in the sequent separately, and to do so recursively.

```
op MPROP : -> Tactic .

op MPROP|- :  ProofTactic -> Tactic .
eq MPROP|- (d | PSI |- PHI) =
           LIFT(0,count-preds(PSI),0,TRY(PROP-TAC THEN MPROP)) .

op |-MPROP : ProofTactic -> Tactic .
eq |-MPROP (d | PSI |- PHI) =
           LIFT(0,0,count-preds(PHI),TRY(PROP-TAC THEN MPROP)) .

eq MPROP (d | PSI |- PHI) = REPEAT(count-preds(PSI),MPROP|-) THEN
                REPEAT(count-preds(PHI),|-MPROP) .
```

|-MPROP works by applying PROP-TAC to the *last* predicate in consequent. If this succeeds, MPROP is applied recursively to the resulting subgoals. If it fails, the TRY ensures that the tactic behaves like idtac. Since this happens in the scope of a LIFT, the resulting predicates (new, or unchanged) are appended to the *front* of the list of predicates in the consequent. MPROP applies this tactic (and the one which works on the antecedent) repeatedly (once for each predicate), and the cycling described above ensures that each predicate in the original goal is considered exactly once.

When writing a recursive tactic it is clearly desirable to ensure that the tactic will necessarily terminate. This is readily established in this case by the fact that each recursive call of MPROP is preceded by a call of PROP-TAC, and each successful call of PROP-TAC reduces by one the (finite) number of propositional connectives in the sequent. This number is bounded below by zero, and so the recursion necessarily terminates. That the tactic is *correct*—i.e. that it removes all possible propositional connectives is harder to demonstrate; such considerations are part of the motivation for Part II of this thesis.

Another tactic which can be used to good effect is one which attempts to discharge the current goal via the `assumption` rule, trying each antecedent-consequent pair in turn:

```
op MASSUM : -> Tactic .
op MASSUMR : Nat -> Tactic .
op MASSUML : Nat -> Tactic .

eq MASSUM (d | PSI |- PHI) = MASSUML(count-preds(PSI)) .
eq MASSUML(N) (d | PSI |- PHI) =
    if N == 0 then failtac
    else if N == 1 then MASSUMR(count-preds(PHI))
    else (MASSUMR(count-preds(PHI))
        ELSE (LIFT(0,count-preds(PSI),0,idtac) THEN MASSUML(p(N))))
    fi fi .
eq MASSUMR(N) (d | PSI |- PHI) =
    if N == 0 then failtac
    else if N == 1 then
        LIFT(0,count-preds(PSI),count-preds(PHI),assumption)
    else (LIFT(0,count-preds(PSI),count-preds(PHI),assumption)
        ELSE (LIFT(0,0,count-preds(PHI),idtac) THEN MASSUMR(p(N))))
    fi fi .
```

As with MPROP, this tactic relies on the cycling obtained by repeatedly applying a lifted tactic to the last predicate in the antecedent/consequent. MASSUML calls MASSUMR once for each predicate in the antecedent. MASSUMR attempts to match that predicate with each of the predicates in the consequent. Execution of this tactic is bounded by the sizes of PSI and PHI in the original sequent. It either returns with a completed proof, or it returns failure. (In practice, the `failtac` is replaced by a call to 2OBJ's exception-handling mechanism, so that the point of the failure can (optionally) be determined).

A sufficient decision procedure for the propositional calculus is achieved by first removing all propositional connectives (via the rules called by PROP-TAC) and then looking for subgoals where the `assumption` rule applies. If this rule applies to *all* the subgoals, then the original sequent is a tautology, and a theorem of $\mathcal{W}$. As a result, the following tactic is a decision procedure for the propositional fragment of $\mathcal{W}$—it returns no subgoals whenever it is applied to a tautology:

```
MPROP THEN MASSUM .
```

Since the action of MPROP is invariably useful, even if the resulting subgoals are not discharged by `assumption`, we would not wish the failure of MASSUM to undo the work done by MPROP. As a result, a more useful tactic is

```
MPROP THEN TRY(MASSUM) .
```

Clearly, these tactics can readily be generalized to apply any tactic in a 'truly' exhaustive fashion. MPROP immediately generalises by allowing a tactic argument, and simply replacing all instances of PROP-TAC with that tactic.

```
op TOEACH : ProofTactic -> Tactic .

op TOEACH|- : ProofTactic -> Tactic .
eq TOEACH|- (PT) (d | PSI |- PHI) =
    LIFT(0,count-preds(PSI),0,TRY(PT THEN TOEACH(PT))) .

op |-TOEACH : ProofTactic -> Tactic .
eq |-TOEACH (PT) (d | PSI |- PHI) =
    LIFT(0,0,count-preds(PHI),TRY(PT THEN TOEACH(PT))) .

eq TOEACH (PT) (d | PSI |- PHI) =
        REPEAT(count-preds(PSI),TOEACH|-(PT)) THEN
        REPEAT(count-preds(PHI),|-TOEACH(PT)) .
```

Likewise, MASSUM can be parametrised to use any rule/tactic which applies to a pair of predicates.

```
op |-ANY-PAIR : ProofTactic -> Tactic .
op |-PAIR-REPR : Nat ProofTactic -> Tactic .
op |-PAIR-REPL : Nat ProofTactic -> Tactic .

eq |-ANY-PAIR (PT) (d | PSI |- PHI) = |-PAIR-REPL(count-preds(PSI),PT) .
eq |-PAIR-REPL(N,PT) (d | PSI |- PHI) =
    if N == 0 then !  'PairRepLFail
    else if N == 1 then |-PAIR-REPR(count-preds(PHI),PT)
    else (|-PAIR-REPR(count-preds(PHI),PT)
        ELSE (LIFT(0,count-preds(PSI),0,idtac)
            THEN |-PAIR-REPL(p(N),PT)))
    fi fi .
eq |-PAIR-REPR(N,PT) (d | PSI |- PHI) =
    if N == 0 then !  'PairRepRFail
    else if N == 1 then LIFT(0,count-preds(PSI),count-preds(PHI),PT)
    else (LIFT(0,count-preds(PSI),count-preds(PHI),PT)
        ELSE (LIFT(0,0,count-preds(PHI),idtac)
            THEN |-PAIR-REPR(p(N),PT)))
    fi fi .
```

Comments about termination are again relevant here. In the case of TOEACH, the argument tactic (PT) must, like PROP-TAC, have the property that it is applicable to every goal (and if necessary has a DEFAULT clause to catch this); and it must, after some finite time, fail to apply to any goal (i.e. it must decrease some bound function).

Another tactic, having very similar properties to TOEACH, can be described. The design of this tactic owes more to that of ANY-PAIR than to that of TOEACH. The tactic searches for somewhere to apply its argument tactic (PT), terminating immediately after doing so. If no such place can be found, the tactic fails. As such, exhaustive behaviour is achieved by EXHAUST(|-TRY-EACH).

```
op |-TRY-EACH : ProofTactic -> Tactic .
op |-REP : Nat ProofTactic -> Tactic .

eq |-TRY-EACH (PT) (d | PSI |- PHI) = |-REP(count-preds(PHI),PT) .
eq |-REP(N,PT) (d | PSI |- PHI) =
    if N == 0 then failtac
    else if N == 1 then LIFT(0,0,count-preds(PHI),PT)
```

```
    else (LIFT(0,0,count-preds(PHI),PT)
        ELSE (LIFT(0,0,count-preds(PHI),idtac) THEN |-REP(p(N),PT)))
    fi fi .

op TRY-EACH|- :  ProofTactic -> Tactic .
op REP|- :  Nat ProofTactic -> Tactic .

eq TRY-EACH|- (PT) (d | PSI |- PHI) = REP|-(count-preds(PSI),PT) .
eq REP|-(N,PT) (d | PSI |- PHI) =
    if N == 0 then failtac
    else if N == 1 then LIFT(0,count-preds(PSI),0,PT)
    else (LIFT(0,count-preds(PSI),0,PT)
        ELSE (LIFT(0,count-preds(PSI),0,idtac) THEN REP|-(p(N),PT)))
    fi fi .

op TRY-EACH : ProofTactic -> Tactic .
eq TRY-EACH(PT) Seq = |-TRY-EACH(PT) ELSE TRY-EACH|-(PT) .
```

These tactics are sufficiently complex that their structure (and correctness) may not be apparent. Chapter 6 revisits these definitions, in a clearer form, suggesting that TOEACH(PT) and EXHAUST(TRY-EACH(PT)) are equivalent in their effect, with TOEACH performing a *depth-first* search, and TRY-EACH forming the basis of a *breadth-first* search.

2.10 Tactics for $\mathcal{W}$'s Expression Axioms

Just as the rules dealing with propositions are usefully collected together into the tactic PROP-TAC, most of $\mathcal{W}$'s axioms dealing with expressions can similarly be viewed as a set of mutually exclusive tactics, with the form of the goal determining which tactic is to be applied.[8] The tactic EXP-TAC deals with such situations—dealing principally with predicates of the form $t \in u$, or $t = u$.

For example, the code for dealing with binding membership of schemas looks like this:

```
eq EXP-TAC(x) (% | * |- b \in Sc) = |-BINDINGMEM THEN subst .
eq EXP-TAC(x) (% | b \in Sc |- *) = BINDINGMEM|- THEN subst .
```

Set comprehensions may appear in two different forms—with or without a term after the •. The rule setabbr converts from the short to the long form (providing the characteristic tuple of the definition). EXP-TAC must be able to deal with both forms:

```
eq EXP-TAC(x) (% | * |- t \in { St }) = |-apply('setabbr) THEN
                                        |-SETCOMP THEN
                                       TRY(|-ONEPT) .
eq EXP-TAC(x) (% | * |- t \in { St <.> u }) =|-SETCOMP THEN
                                              TRY(|-ONEPT) .
```

In the $t \in u$ case, the choice of tactic is dependent entirely on the form of u. The case of $t = u$ is rather less certain. The best case is when t and q are equal: then the rule of *reflection* applies; otherwise, an instance of the axiom of extension is needed.

```
eq EXP-TAC(x) (% | * |- t = u ) = reflection ELSE |-EXTENSION(x) .
```

[8] Compare B's *Theories*.

Sometimes equality predicates have been introduced to the antecedent in order to be used in rewriting other expressions; here Leibniz's rule is used. The form given in $\mathcal{W}$ is rather contrived, so various tactics are needed in order to accomplish such rewriting in practice.

For example, the rule of Leibniz is expressed as

$$\frac{s = t, \langle\!| x \rightsquigarrow t |\!\rangle.p \vdash}{s = t, \langle\!| x \rightsquigarrow s |\!\rangle.p \vdash} \ (Leibniz) \ ,$$

so it is reasonable to construct a tactic which performs a similar inference in the consequent:

$$\frac{s = t \vdash \langle\!| x \rightsquigarrow t |\!\rangle.p}{s = t \vdash \langle\!| x \rightsquigarrow s |\!\rangle.p} \ (Leib) \ .$$

```
op |-LEIB : -> Tactic .
eq |-LEIB (% |t = u |- <| M ~> u |> .  p) =
(   CUT(<| M ~> t |> .  p)
    THENL
    ( THIN(0,0.2)
    ,   ( LIFT(0,2 1,0,Leibniz) THEN
          ASSUMPTION(2,1) )
    )
) .
```

A more devious manœuvre must be employed if rewriting is needed, but the predicate to be rewritten has no binding attached:

$$\frac{x = t, \langle\!| x \rightsquigarrow t |\!\rangle.p \vdash}{x = t, p \vdash} \ (Leibniz*) \ .$$

```
op LEIBNIZ*|- :  -> Tactic .
eq LEIBNIZ*|- (% | x = t , p |- *) =
(   LIFT(0,2,0,TSBUS|-(<| x ~> x |> .  p)) THEN
    LIFT(0,2 1,0,Leibniz) THEN
    LIFT(0,2,0,SUBST) THEN
    LIFT(0,2 1,0,idtac)
) .
```

The crucial feature of this tactic is TSBUS, a tactic which behaves like SUBST in reverse (e.g. in this case, it takes the goal $p \vdash$ and replaces it with $\langle\!| x \rightsquigarrow x |\!\rangle.p$).

```
op TSBUS|- :  Predicate -> Tactic .
eq TSBUS|- (q) (% | p |- *) =
(   CUT(q)
    THENL
    ( GenASSUM(1,1),
    THIN(0,2,0) ) ) .
```

(GenASSUM simply applies the substitution rules to the sequent—to normalize the terms—before attempting the `assumption` rule.)

A similar approach can be employed to construct REFL|-, a tactic which reflects equalities (converting $t = u \vdash$ into $u = t \vdash$), so that the Leibniz tactics can be applied to goals in which the equality is backwards. For example,

```
LEIBNIZ*|- ELSE (REFL|- THEN LEIBNIZ*|-)
```

will behave like the *Leibniz** described above, whether the goal presented is $(x = t, p \vdash)$ or $(t = x, p \vdash)$. A number of tactics for applying Leibniz's rule in various antecedents and consequents are defined. The tactic MLEIB tries each of these in turn.

2.11 A Toolkit Tactic

One of the features of Z which makes it practical for use as a specification language is the large library of specification constructs which are described in the language reference documents [Spi92a, BN+92], and may be assumed in any specification—usually referred to as the Mathematical Toolkit. The encoding of Jigsa$\mathcal{W}$ implements a small subset of these definitions; sufficient merely to support the case studies which have been undertaken.

The encoding provides a *toolkit tactic*, TK-TAC, which describes how to rewrite a given goal to remove any reference to a function named in the toolkit—for example TK-TAC for 'dom' is almost identical to the DOM-TAC previously described. Likewise, goals featuring empty sets can be handled with a part of TK-TAC which invokes EMPTY-TAC. Within the small set of such definitions encoded in Jigsa$\mathcal{W}$, various patterns emerge, suggesting that it would not be hard to extend TK-TAC to cover the whole toolkit. For example, DOM-TAC can be converted into a tactic for dealing with $\_ \cup \_$ with very few changes, the chief of which is to change the term which is the subject of the CUT to include a binding containing two terms instead of one. Union is described in the same generic box as intersection and set difference. A suitably general tactic will use GENDEF to bring that generic definition into the antecedent of the goal, and then use ANY-PAIR to find a consequent term which can be rewritten using one of the definitions.

Part of the presentation of the toolkit in [Spi92a] is an *ad hoc* collection of general-purpose laws which have been found to be useful in reasoning about Z specifications. Our aim is that TK-TAC, together with the tactics outlined above (collected together in the next section) should be able to prove such laws, as they arise in proofs. This is not the most efficient approach to theorem-proving, but provides a demonstrably sound method of applying these laws (even in the presence of modifications to the logic) and since 2OBJ has no lemma-storing capability this is the only way to construct such a library in this frame. Moreover, the set of laws in [Spi92a] makes no pretence at being complete; a tactic able to prove a wide class of such rules may be a useful way of completing the set.

2.12 Combining Tactic Actions

A natural extension of the propositional calculus decision procedure is the tactic below, which attempts to apply exhaustively all of the propositional calculus rules, the expression-handling tactics, the toolkit definitions, and the 'easy' two predicate calculus rules.

```
op BIG-TAC : -> Tactic .
eq BIG-TAC Seq = FIRST(TK-TAC,EXP-TAC(x),PRED-TAC2,PROP-TAC) .

op NEW-TAC3 :  -> Tactic .
eq NEW-TAC3 Seq =
    EXHAUST (TRY-EACH(BIG-TAC) ELSE
             |-ANY-PAIR(assumption) ELSE
             ANY-PAIR|-(OTHER-WIDER-PRED)
             ) .
```

(Note that FIRST is a generalization of ELSE: it attempts to apply each of the tactics in turn, returning with the result of the first successful application (or failing if none is applicable).

This tactic could also include calls to AUTO-SCHDEF within its list of tactics to try. This has been omitted here as it can lead to the 'explosion' of a proof. In particular, the indiscriminate expansion of schema definitions in the antecedent leads to very large sequents, with a consequential deterioration of efficiency. Instead, the tactics for dealing with particularly common goals (see below) can be crafted to include appropriate expansions of included schemas. If the expansion of schema terms is avoided, this tactic can be reasonably efficient. The time taken by OBJ3 to try to apply the various rules/tactics listed in BIG-TAC is very small in comparison to the time taken to calculate the side-condition on rule-lifting. More discussion on tactic construction will be found in Section 4.5.

2.13 Binding Substitution

Most of the binding substitution rules given in [WB92, Tables 5–7] can be implemented directly as OBJ3 rewrites, as Section 2.3 suggests. Because delayed substitution is often useful (for example, in order to apply Leibniz's rule), these are introduced using the label [axiom] which 2OBJ takes to mean that they should not be used in 'normal' reductions, but only on request—when called by name, or when a built-in such as filtdred is used.

Certain rules cannot be applied in this way. For example, the 'leap-frog' rule:[9]

$$b.(c.p) \equiv c.(b.p) \quad \text{whenever } \alpha c \cap \phi_e b = \varnothing \wedge \alpha(b) \cap \phi_e c = \varnothing$$

must be applied sparingly if infinite loops are to be avoided. This is the most problematic of the substitution rules. Others must, though, be applied with caution.

$$b.p \equiv p \quad \text{whenever } \alpha b \cap \phi_p p = \varnothing$$

(pred-subst) is one such rule. This is because the empty binding $\emptyset$ is frequently used with substitution to explode certain definitions (schemas used as predicates, for example). For this to work, the binding must be distributed through the schema (by a rule like $b.[d \mid p] \equiv b.[d] \wedge (b.p)$), and not simply removed by the rule above.

These special cases are covered by having the subst rule look like this:

```
let excluded-rules = ( 'leap-subst ::
                       ('split-subst ::
                       ('pred-subst ::
                       'decor-subst))) .
eq subst-rule (d | PHI |- PSI) =
    (filtdred ('%, excluded-rules ,top,(d | PHI |- PSI) )) .
```

The other exceptional rules being split-subst:

$$\langle\!| x \rightsquigarrow t;\ B |\!\rangle.p \equiv \langle\!| B |\!\rangle.(\langle\!| x \rightsquigarrow t |\!\rangle.p) \qquad \alpha\langle\!| B |\!\rangle \cap \phi t = \varnothing$$

[9] This is omitted in [WB92], but is essential when nested substitutions are used, in expanding generic definitions, for example.

and `decor-subst`

$$b.(S^{\mathrm{q}}) = [b.S]^{\mathrm{q}} \quad \text{provided } \alpha b \cap \alpha(S^{\mathrm{q}}) = \varnothing$$

When rewriting involving these special rules is required, a tactic applies them in a controlled way. A generalized substitution tactic is

```
op GenSUBST* :  -> Tactic .
eq GenSUBST* (% | * |- (b .  p) ) =
(
    subst THEN EXHAUST(|-apply('pred-subst)) THEN
    subst THEN EXHAUST(|-apply('decor-subst)) THEN
    subst THEN TRY(|-apply('lsap-subst)) THEN
    subst THEN EXHAUST(|-apply('decor-subst)) THEN
    subst THEN EXHAUST(|-apply('sch-subst)) THEN
    subst THEN EXHAUST(|-apply('pred-subst)) THEN
    subst
) .
```

Discovering which combinations of rules led to rewriting problems was very much an *ad hoc* process. Ideally, the set of syntactic equivalences in the standard [BN+92] should be Church–Rosser and terminating (i.e. if used as left-to-right rewrites they should produce the same end result, regardless of the order of application, and do so in a finite time).

Schemas as Predicates

$\mathcal{W}$ uses a trick with binding substitutions in order to expand sequents of the form $\vdash [S]t$, making, for example, the inference

$$\frac{\vdash x \in \mathbb{N} \wedge x \leq 6}{\vdash [x : \mathbb{N} \mid x \leq 6]} \ .$$

The expansion comes by exploiting the syntactic equivalence $b.[d \mid p] \equiv b.[d] \wedge b.p$, and then $b.[x : t] \equiv b.(x \in t)$, etc. By choosing b to be the empty binding $\emptyset$, a general-purpose transformation can be accomplished. `MKBIND-TAC` transforms a predicate into one prefixed by an empty binding (the step $(*)$ is accomplished by the rule `pred-subst`, see above):

$$\frac{\dfrac{\vdash \emptyset.p}{\vdash \emptyset.p, p} \quad \dfrac{\overline{p \vdash p}}{\emptyset.p \vdash p}\ (*)}{\vdash p}$$

```
op MKBIND-TAC : -> Tactic .
eq MKBIND-TAC (( CUT(</> .  p)
    THENL
    ( THIN(0,0,2)
    ,
     ( LIFT(0,1,0,apply|-('pred-subst)) THEN assumption)
    )
) .
```

Then a tactic like `GenSUBST*`—but *not* beginning with `pred-subst`—can be employed to normalize the predicate.

Chapter 3

Case Study

AS A DEMONSTRATION of the way the system described in the previous chapter can be used, this chapter describes the process involved in discharging some of the usual proof obligations arising in a 'realistic' Z specification. In the pages which follow, tactics for producing suitable proofs for a case study by Woodcock [Woo92] (*A Multi-Level Security System*) are presented.

3.1 Specification

This specification comes from a problem domain in which Z has been used extensively —the design of secure systems. We describe (at a high level) the essential components of the system:

Users are individuals who may use the system, in order to access (read and/or write) data.

Subjects are active processes in the system.

Objects are data items in the system (documents, files).

Levels are the classifications which objects attract (and to which users are cleared)— restricted, classified, ... top secret.

Profiles are records of which users may access data at which level of classification.

Access modes may be read, write or execute.

The given sets of the specification, then, will be as follows:

$$[Level, User, Subject, Object]$$

A common requirement for such multi-level security systems is one of *non-interference*. This is commonly characterized [BL74] as *No read up; No write down*:

if x reads y, then x must be cleared to access data classified at least as highly as y. Conversely, if x writes y then y must be sufficiently highly classified as to cover anything x might write.

These ideas lead to the need for a security level dominance relation. We specify it here, without giving any formal properties for it, though we would expect it to be reflexive, antisymmetric and transitive.

$$\_ \geq \_ : Level \leftrightarrow Level$$

Profiles (as mentioned above) will be given by a (fixed) function from users to sets of clearance levels.

$$profile : User \rightarrow \mathbb{P}\, Level$$

Access modes are specified by a free datatype.

$$Mode ::= read \mid write \mid execute$$

The state of the system consists of functions giving levels to subjects and objects, recording which subjects are accessing which objects with which modes, and which users own which subjects. The invariant says that all accesses must be made by subjects with classifications upon objects with classifications, and that each subject owned by a user must be classified—and each classified subject must belong to a user.

S

$$sub : Subject \nrightarrow Level$$
$$obj : Object \nrightarrow Level$$
$$acc : (Subject \times Object) \leftrightarrow Mode$$
$$prin : Subject \nrightarrow User$$

$$\mathrm{dom}\, acc \subseteq (\mathrm{dom}\, sub) \times (\mathrm{dom}\, obj)$$
$$\mathrm{dom}\, prin = \mathrm{dom}\, sub$$

The security requirements for the system are given by three additional schemas, each adding an extra predicate to the state invariant. Firstly, each subject must be cleared to a level which is one of those possible for its owner.

S1

$$S$$

$$\forall s : \mathrm{dom}\, sub \bullet sub\, s \in profile(prin\, s)$$

Secondly, whenever a read is taking place, its subject must be at least as highly classified as the object being read.

S2

$$S$$

$$\forall s : Subject;\ o : Object \bullet ((s, o) \mapsto read) \in acc \Rightarrow sub\, s \geq obj\, o$$

Finally, whenever a write is taking place, the object being written must be at least as highly classified as the subject doing the writing.

S3

S

$\forall s : Subject;\ o : Object \bullet ((s, o) \mapsto write) \in acc \Rightarrow obj\, o \geq sub\, s$

The system is secure when these three conditions hold simultaneously.

$$SecS == (S1 \wedge S2) \wedge S3$$

Initially, nothing is classified, no access is taking place and no objects exist.

SecSInit

$SecS'$

$sub' = \varnothing$
$obj' = \varnothing$
$acc' = \varnothing$
$prin' = \varnothing$

In the following section, we shall demonstrate that such a system exists—i.e. that the initial state predicates satisfy the state and security invariants.

$$\vdash \exists SecSInit \bullet true$$

A typical operation on the state is to open an object for reading. The operation must be supplied with an object and a subject on which to operate.

OpenToRead0

ΔS
$s? : Subject$
$o? : Object$

$s? \in \mathrm{dom}\, sub$
$o? \in \mathrm{dom}\, obj$
$\forall s : Subject \bullet (s, o?) \notin \mathrm{dom}\, acc$
$sub\, s? \geq obj\, o?$
$sub' = sub$
$obj' = obj$
$acc' = acc \cup \{(s?, o?) \mapsto read\}$
$prin' = prin$

We shall demonstrate that the precondition of this schema is

OpenToRead0Pre

S
$s? : Subject$
$o? : Object$

$s? \in \mathrm{dom}\, sub$
$o? \in \mathrm{dom}\, obj$
$\forall s : Subject \bullet (s, o?) \notin \mathrm{dom}\, acc$
$sub\, s? \geq obj\, a?$

3.2 Initialization Theorem

It is usual, when giving the Z specification of a system's state, to describe the initial state of the system. This generally serves two purposes:

- it gives suitable initializations for the state variables when the system is refined to code; and
- the initial state usually gives explicit values to the state variables, so demonstrating that it satisfies the global predicates on the state—i.e. demonstrating that the state has a model.

Therefore, it is necessary to show that the initial state satisfies the state schema. The following theorem is sufficient to demonstrate this:

$$\vdash \exists SecSInit \bullet true$$

Recalling the form of the inference rule for $\vdash\exists$, the first step, clearly, is to cut in a binding, allowing the existential quantifier to be removed:

$$\frac{\vdash b \in [SecSInit], \exists SecSInit \bullet true \qquad \dfrac{\begin{array}{l} b \in [SecSInit] \\ \vdash \\ \exists SecSInit \bullet true, \\ b.true \end{array}}{\begin{array}{l} b \in [SecSInit] \\ \vdash \\ \exists SecSInit \bullet true \end{array}}\ (\vdash\exists)}{\vdash \exists SecSInit \bullet true}\ (cut)$$

The right-hand branch is trivial to resolve. The left-hand branch must proceed with binding membership and substitution, followed by an application of the definition of schema *SecSInit*.

$$\frac{\dfrac{\vdash b.SecS' \quad \vdash b.(sub' = \varnothing) \quad \vdash b.(obj' = \varnothing) \quad \vdash b.(acc' = \varnothing) \quad \vdash b.(prin' = \varnothing)}{\vdash b.SecSInit}\ \text{(APPLY-SCHDEF)}}{\vdash b \in [SecSInit], \exists SecSInit \bullet true}\ (thin, bindingMem, subst)$$

A general tactic which makes these steps, supplied with a binding b and a schema name SN is

```
CUT(b \in | SN |)
THENL
( ( THIN(0,0,2) THEN
    |-BINDINGMEM THEN
    subst THEN
    APPLY-SCHDEF(SN) )
, ( |-exist THEN
    LIFT(0,0,2,subst THEN |-true) )
)
```

Clearly the binding necessary b is the one which will allow most of these subgoals to be discharged by *reflection*:

$$\langle\!|\ sub' \rightsquigarrow \emptyset;\ obj' \rightsquigarrow \emptyset;\ acc' \rightsquigarrow \emptyset;\ prin' \rightsquigarrow \emptyset\ |\!\rangle\ .$$

Following application of the tactic given above, it will be necessary to continue to expand schema definitions, and to simplify propositional/predicate formulae. Thus a tactic *like* NEW-TAC3 is needed. We could, at the same time, apply rules from TK-TAC, but for reasons of efficiency—mentioned above, and explored further in Section 4.5—this tactic is not included. The tactic below combines the steps already taken with this general exhaustive behaviour, producing a general tactic for commencing initialization theorems.

```
op INIT-TAC : WordDec BindingExtn -> Tactic .
eq INIT-TAC(x,b) (% | * |- (|E| SN <.> True)) =
(
    CUT(b \in [ SN ])
    THENL
    (
        THIN(0,0,2),
        |-exist
    ) THEN
    EXHAUST(
        TRY-EACH(FIRST(PRED-TAC2,PROP-TAC,AUTO-SCHDEF,EXP-TAC(x)))
        ELSE |-EACHPAIR(assumption)
    )
) .
```

Since the schema *SecSInit* consists merely of equality predicates, together giving explicit values to all the state variables, there is no need for the user to construct the binding b; a simple OBJ3 function can do it, permitting the definition of AUTO-INIT-TAC. This is worthwhile since writing schemas like *SecSInit* is a very common Z style.[1]

```
op ax-part :  SchemaDef -> AxiomPart .
eq ax-part (SCH SN IS DP ST AP END) = AP .
eq ax-part (SN =^= [ d | p ]) = p .

op init-binding :  AxiomPart -> BindingExtn .
eq init-binding (x = t) = <| x ~> t |> .
eq init-binding ((x = t) \\ AP) =
        <| x ~> t |> bindcat init-binding(AP) .

op AUTO-INIT-TAC : WordDec -> Tactic .
eq AUTO-INIT-TAC(x) (% | * |- |E| SN <.> True) =
        INIT-TAC(x,init-binding(ax-part(SN .def))) .
```

Subgoals

Applying AUTO-INIT-TAC to our initial goal yields 24 subgoals. Many of these are duplicated, because *SecS* is defined as the conjunction of three very similar schemas, each of which includes the state schema *S*. Hence, the state schema features in three

[1] Notice that by defining the auxiliary functions seen here, nothing is added to the logic. If they contain errors, the tactic will simply fail to apply (or will produce unsolvable subgoals).

branches of the proof tree—and in each case leads to seven subgoals. The remaining three subgoals are respectively contributed by $S1$, $S2$ and $S3$. Omitting duplicates, these subgoals are:[2]

$$\vdash \varnothing \in Subject \nrightarrow Level$$
$$\vdash \varnothing \in Object \nrightarrow Level$$
$$\vdash \varnothing \in (Subject \times Object) \leftrightarrow Mode$$
$$\vdash \varnothing \in Subject \nrightarrow User$$
$$\vdash \mathrm{dom}\, \varnothing \subseteq \mathrm{dom}\, \varnothing \times \mathrm{dom}\, \varnothing$$
$$y : \mathrm{dom}\, \varnothing \vdash y \in \mathrm{dom}\, \varnothing$$
$$s : \mathrm{dom}\, \varnothing \vdash \varnothing\ s \in profile(\varnothing\ s)$$
$$s : Subject;\ o : Object \mid ((s, o) \mapsto read) \in \varnothing \vdash \varnothing\ s \geq \varnothing\ o$$
$$s : Subject;\ o : Object \mid ((s, o) \mapsto write) \in \varnothing \vdash \varnothing\ o \geq \varnothing\ s$$

It is worth noting that Woodcock arrives (by a different route) at the same list of subgoals in [Woo92, page 11]. At this point he writes 'Each of these follows from properties of the correct instantiations of $\varnothing$'. Some effort is needed to verify this in $\mathcal{W}$, but all this work can be accomplished by the general tactic described above (NEW-TAC3); i.e. each of these goals is an instance of a rule which might be presented alongside the toolkit definitions.

For example, first goal is solved by rewriting with the definitions of $\nrightarrow$ and $\varnothing$. The definition of $\nrightarrow$ is given by a set comprehension. Showing that $\varnothing[Subject \times Level]$ is a member of this set entails showing

$$\vdash \varnothing \in Subject \leftrightarrow Level$$

and

$$\vdash \forall x : Subject;\ y_1, y_2 : Level \bullet (x \mapsto y_1 \in \varnothing \land x \mapsto y_2 \in \varnothing) \Rightarrow y_1 = y_2\ .$$

The first of these subgoals is simplified by using the definition of $\leftrightarrow$ to rewrite it as $\vdash \varnothing \in \mathbb{P}(Subject \times Level)$. This can be rewritten using the axiom for powerset membership (as a rule), and predicate calculus, giving

$$x \in \varnothing \vdash x \in Subject \times Level\ .$$

The definition of $\varnothing$, and more predicate calculus, eventually gives false $\vdash \ldots$, which completes the proof (via the rule *false*$\vdash$).

The second of the subgoals can be simplified by predicate calculus, to give

$$x : Subject;\ y_1, y_2 : Level \mid x \mapsto y_1 \in \varnothing, x \mapsto y_2 \in \varnothing \vdash y_1 = y_2\ ,$$

and then either of the predicates in the antecedent can be rewritten to false, as before, completing the proof for this subgoal, too.

[2] Note that $y : \mathrm{dom}\, \varnothing \vdash y \in \mathrm{dom}\, \varnothing$ is not entirely trivial. The generic parameters of dom and $\varnothing$, which are omitted here, are different in the antecedent and the consequent. The subgoal is in fact $y : \mathrm{dom}[Subject, User]\varnothing[Subject \times User] \vdash y \in \mathrm{dom}[Subject, Level]\varnothing[Subject \times Level]$.

3.3 Precondition Theorem

Another proof which it is often instructive to produce is the proof that an operation has a particular precondition. This is conventionally produced in a constructive manner: the precondition is that schema which is obtained by hiding (existentially quantifying over) all the output and final-state variables; this schema is then simplified. In $\mathcal{W}$ such an approach is not an option, and so instead we must propose a precondition and then demonstrate that it is logically equivalent to the schema produced by hiding. That is, for schema P, having proposed precondition $Ppre$, we must show that

$$\mathrm{pre}\, P \Leftrightarrow Ppre \ .$$

However, this cannot simply be packaged into a $\mathcal{W}$ sequent as a consequent, since it has many free variables. One solution is to add a declaration part which will 'close' the sequent. This is hard to do neatly, so instead we prove two theorems:

$$\mathrm{pre}\, P \vdash Ppre$$

and

$$Ppre \vdash \mathrm{pre}\, P \ .$$

These are closed theorems provided the alphabets of $\mathrm{pre}\, P$ and $Ppre$ are identical. Type-checking is sufficient to guarantee this.

Therefore, for the *OpenToRead*0 operation, we must show

$$\mathrm{pre}\, OpenToRead0 \vdash OpenToRead0Pre$$

and

$$OpenToRead0Pre \vdash \mathrm{pre}\, OpenToRead0 \ .$$

First Goal

When the specification is written in a style which makes the precondition explicit in the operation schema—as in this case—the first of these goals is quite straightforward. The declaration part must be brought into the antecedent, and the definition of 'pre' expanded.

$$\frac{\dfrac{\mathrm{pre}\, OpenToRead0 \mid OpenToRead0 \setminus (sub', obj', acc', prin') \vdash OpenToRead0Pre}{\mathrm{pre}\, OpenToRead0 \mid \mathrm{pre}\, OpenToRead0 \vdash OpenToRead0Pre}\ (predef)}{\mathrm{pre}\, OpenToRead0 \vdash OpenToRead0Pre}\ (Declpred)$$

Application of the rule for hiding requires the presence of a predicate asserting that (in this case) for all instances of *OpenToRead*0, some schema text is satisfied, with that schema text having as its alphabet the variables being hidden:

$$\frac{\exists St \bullet S \vdash}{\forall S \bullet [St], S \setminus (x_1, \ldots, x_n) \vdash}\ (hiding \vdash)[\alpha St = \{x_1, \ldots, x_n\}]$$

A suitable schema text is simply the restriction of the declaration part of *OpenToRead*0 to its post-state and output variables. A tactic introduces this via *cut*, and uses schema expansion to demonstrate that it is true (`postdecls` extracts the relevant parts of the declaration from the named schema):

```
op PRECOND-FORALL-INTRO : SchemaName -> Tactic .
eq PRECOND-FORALL-INTRO (SN) (d | PHI |- p) =
(
    CUT (|A| SN <.> [ postdecls(SN) ])
    THENL
    ( ( THIN(0,0,2) THEN
        LIFT(0,0,1,|-all THEN DECLPRED) THEN
        LIFT(0,1,0,EXHAUST(TRY-EACH(AUTO-SCHDEF))) THEN
        LIFT(0,0,1,EXPLODE) THEN
        LIFT(0,0,1,MPROP) THEN
        MASSUM )
    ,
      idtac
    )
) .
```

Following application of this tactic, the hiding rule can be applied, and then the $\exists\vdash$ rule. After this, all that is necessary is to expand all of the schema definitions, and discharge all of the goals (or most of them, depending on the style of operation schema) via the *assumption* rule. The following tactic accomplishes all this.

```
op PRECOND-A-TAC : -> Tactic .
eq PRECOND-A-TAC ((pre Sc) | * |- T1) =
(
    DECLPRED THEN
    LIFT(0,1,0,apply|-('predef) THEN subst) THEN
    PRECOND-FORALL-INTRO(Sc) THEN
    LIFT(0,1 2,0,hiding|- THEN exist|-) THEN
    LIFT(0,1,0,EXHAUST(TRY-EACH(AUTO-SCHDEF))) THEN
    LIFT(0,0,1,AUTO-SCHDEF THEN EXPLODE THEN MPROP) THEN
    TRY(MASSUM) THEN
    LIFT(0,0,1,AUTO-SCHDEF THEN EXPLODE THEN MPROP) THEN
    TRY(MASSUM)
) .
```

The expansion of schemas in this tactic is a compromise between generality and efficiency. A more general tactic would exhaustively expand schemas and propositions, trying the *assumption* rule after each action. As such this tactic specialized to the particular instance of the precondition theorem under consideration here.

Second Goal

The second goal requires more work. However, it begins in a similar manner. This time `predef` and `hiding` must be applied in the *consequent*. The tactic `PRECOND-FORALL-INTRO` is used again, to provide a suitable predicate to satisfy the hiding rule.

After this, in order to satisfy the resulting existential quantification in the consequent, it is necessary to use *cut* to provide a binding. Since the operation is deterministic—i.e. each of the post-state variables has its value described by an equation of the form $x' = f(S)$, such a binding can be created in a similar manner to that used in the initialization theorem above. The tactic `post-CUT-TAC` does this.

```
op PRECOND-B-TAC : -> Tactic .
eq PRECOND-B-TAC (T1 | * |- (pre Sc)) =
( DECLPRED THEN
    LIFT(0,0,1,|-apply('predef) THEN subst) THEN
```

```
        LIFT(0,0,0,PRECOND-FORALL-INTRO(Sc)) THEN
        LIFT(0,1,1,|-hiding) THEN
        post-CUT-TAC THENL
        [ [ THIN(0,0,2) THEN
            LIFT(0,0,1,EXP-TAC(y) THEN SUBST*) THEN
            LIFT(0,1,0,EXHAUST(TRY-EACH(AUTO-SCHDEF))) THEN
            LIFT(0,0,1,MPROP) THEN
            TRY(MASSUM) ]
        , [ LIFT(0,1,1,|-exist THEN THIN(0,1,1)) THEN
            LIFT(0,1,0,AUTO-SCHDEF) THEN
            LIFT(0,0,1,AUTO-SCHDEF) THEN
            TRY(LIFT(0,0,1,reflection)) THEN
            TRY(MASSUM) ]
        ]
    ] .
```

The result of applying this fully-general tactic to the goal $Ppre \vdash \mathit{pre} P$ is three subgoals

$$
\begin{array}{l}
\mathit{OpenToRead0Pre} \mid s? \in \mathit{Subject}, o? \in \mathit{Object}, \\
\quad \mathit{acc} \in (\mathit{Subject} \times \mathit{Object}) \leftrightarrow \mathit{Mode} \\
\vdash \mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\} \in (\mathit{Subject} \times \mathit{Object}) \leftrightarrow \mathit{Mode}
\end{array}
$$

$$
\begin{array}{l}
\mathit{OpenToRead0Pre} \mid s? \in \mathrm{dom}\, \mathit{sub}, o? \in \mathrm{dom}\, \mathit{obj}, \\
\quad (\forall s : \mathit{Subject} \bullet (s, o?) \notin \mathrm{dom}\, \mathit{acc}), \\
\quad \mathit{sub}\ s? \geq \mathit{obj}\ o?, S, s? \in \mathit{Subject}, o? \in \mathit{Object} \\
\vdash (\!| \mathit{sub}' \rightsquigarrow \mathit{sub};\ \mathit{obj}' \rightsquigarrow \mathit{obj}; \\
\qquad \mathit{acc}' \rightsquigarrow \mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\};\ \mathit{prin}' \rightsquigarrow \mathit{prin} |\!).\Delta S
\end{array}
$$

$$
\begin{array}{l}
s : \mathit{Subject};\ \mathit{OpenToRead0Pre} \mid \forall s : \mathit{Subject} \bullet (s, o?) \notin \mathrm{dom}\, \mathit{acc} \\
\vdash (s, o?) \notin \mathrm{dom}\, \mathit{acc}
\end{array}
$$

The third of these subgoals is the easiest to satisfy: it suffices to provide a binding (the identity) to specialize the universal quantification so that the *assumption* rule can be applied:

```
CUT( <| s ~> s |> \in [ s :  Subject ])
THENL [ (LIFT(0,0,1,EXP-TAC(x) THEN subst) THEN MASSUM)
      , (WIDER-PRED-TAC THEN MASSUM) ]
```

The first subgoal is proved by unpacking the consequent using the toolkit definition of $\leftrightarrow$, and then the axiom (inference rule) for powerset. This gives a goal of the form $y \in \mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\} \vdash y \in (\mathit{Subject} \times \mathit{Object}) \times \mathit{Mode}$ This is solved by invoking the toolkit definition of $\cup$, which entails showing that the arguments to $\cup$ belong to its declared type (generic parameters are suppressed here). Thus it becomes necessary to show $\{(s?, o?) \mapsto \mathit{read}\} \in \mathbb{P}((\mathit{Subject} \times \mathit{Object}) \times \mathit{Mode})$. All this might be accomplished by NEW-TAC3, but the number of definitions involved makes it inappropriate, for efficiency reasons[3], and so a hand-crafted tactic is used: [4]

[3] No doubt NEW-TAC3 could be improved upon, but the speed of the tool precludes experimentation with heuristics for finding proofs efficiently.

[4] This entails a small cheat. *mode* is introduced by an axdef, rather than a free datatype definition (the two are equivalent—see [WB92]). The rule for the latter has not been implemented.

```
LIFT(0,0,1,TK-TAC THEN EXP-TAC(y) THEN |-all THEN DECLPRED)
THEN LIFT(0,1,0,TK-TAC)
THENL ( (LIFT(0,0,1,EXP-TAC(z)) THEN
         MPRED THENL
         ( (LIFT(0,1,0,apply|-('<->def)) THEN subst THEN MASSUM),
           idtac)),
         (LIFT(0,1,0,EXP-TAC(z)) THEN MASSUM) ) THEN
LIFT(0,0,1,EXP-TAC(z) THEN |-all THEN
         DECLPRED THEN LIFT(0,1,0,EXP-TAC(y))) THEN
LIFT(0,1,0,TK-TAC THEN subst THEN TUPLEMEM|-
         THEN and|- THEN LIFT(0,1,0,TUPLEMEM|- THEN and|-)) THEN
LIFT(0,0,1,EXP-TAC(x) THEN |-and) THENL
( (LIFT(0,0,1,EXP-TAC(x) THEN |-and)
  THENL ( (LIFT(0,1,1,|-LEIB-TAC2(x)) THEN MASSUM),
          (LIFT(0,2,1,|-LEIB-TAC2(x)) THEN MASSUM) )) ,
  (LIFT(0,3,1,|-LEIB-TAC2(x)) THEN AXDEF(defmode) THEN MASSUM) )
```

Clearly this tactic has no generality whatsoever. It is, however, the sort of tactic which is worthwhile retaining as a 'recipe' of how to prove this goal—so that if the proof has to be repeated (when the specification changes) it can (with luck) be done so automatically. A more well-structured tactic would, of course, be easier to re-use.

Finally, the second goal must be simplified using schema expansion:

```
LIFT(0,0,1,AUTO-SCHDEF) THEN
TRY(MASSUM) THEN
LIFT(0,5,0,AUTO-SCHDEF) THEN
LIFT(0,0,1,AUTO-SCHDEF) THEN
TRY(MASSUM)
```

This leads to two subgoals. The first of these is the first subgoal above. The second is

$$\begin{array}{l} \mathit{OpenToRead0Pre} \\ \quad \vdash \mathrm{dom}(\mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\}) \subseteq (\mathrm{dom}\, \mathit{sub} \times \mathrm{dom}\, \mathit{obj}) \end{array}$$

An application of TK-TAC to this yields unsurprisingly two new subgoals:

$$\begin{array}{l} \mathit{OpenToRead0Pre} \\ \vdash \{\!|\, S \rightsquigarrow \mathrm{dom}(\mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\});\ T \rightsquigarrow \mathrm{dom}\, \mathit{sub} \times \mathrm{dom}\, \mathit{obj} \,|\!\} \in \\ \qquad\qquad [S : \mathbb{P}(\mathit{Subject} \times \mathit{Object});\ T : \mathbb{P}(\mathit{Subject} \times \mathit{Object})] \end{array}$$

$$\begin{array}{l} \mathit{OpenToRead0Pre};\ x : \mathit{Subject} \times \mathit{Object} \mid \\ \quad x \in \mathrm{dom}(\mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\}) \\ \vdash x \in \mathrm{dom}\, \mathit{sub} \times \mathrm{dom}\, \mathit{obj} \end{array}$$

The first of these simplifies to

$$\mathit{OpenToRead0Pre} \vdash \mathrm{dom}(\mathit{acc} \cup \{(s?, o?) \mapsto \mathit{read}\}) \in \mathbb{P}(\mathit{Subject} \times \mathit{Object})$$

and

$$\mathit{OpenToRead0Pre} \vdash \mathrm{dom}\, \mathit{sub} \times \mathrm{dom}\, \mathit{obj} \in \mathbb{P}(\mathit{Subject} \times \mathit{Object})$$

The proof continues in similar vein for a while. Each definition (of '$\mapsto$', dom, $\_ \cup \_$, etc.) must be expanded (with TK-TAC), and the function arguments shown to belong to the relevant domains. NEW-TAC3 is able to accomplish many of these

steps, but with the complexity of the terms growing (as definitions are expanded, their bound variables become variables introduced in the sequent's declaration part, and the predicates involving them are multiplied), its performance becomes appalling. Certain steps need applications of Leibniz's rule—to rewrite the consequent using equations from the antecedent—and these are best guided by hand. (X2OBJ also crashes when the goals exceed 1024 characters, which has made completing the proof rather difficult.)

3.4 Conclusions

This case study demonstrates that proof of worthwhile theorems is possible in Jigsaw. The simple general-purpose tactics outlined in Sections 2.10, 2.11, and 2.12 prove useful in structuring the proof and adding reusability, but the general tactic NEW-TAC3 is limited in its usefulness due to efficiency problems. Its structure could be improved by giving it a better heuristic (rather than simply applying each rule at each predicate, breadth-first).

The proofs are protracted due to many arguments using points. We might hope that tactics could bring the level of reasoning up to the algebra of sets and functions. The need to demonstrate that functions are applied within their domains, however, tends to re-introduce such low-level detail—see Section 4.2.

Chapter 4

Discussion

THE GOAL OF THIS PIECE OF WORK was to implement a prototype theorem-proving system in which the proof obligations arising in Z specifications could be discharged. This goal has been achieved. The level of automation possible is heavily dependent on the style of specification under consideration: for the specifications written in the style of Chapter 3, automation is moderately advanced.

An initial estimate of the effort involved ('it should take two weeks') proved to be grossly optimistic; many months of effort have been expended in bringing the tool to a state where the case study outlined above could be undertaken. This chapter discusses the chief difficulties and benefits which this approach has given, and compares it with other proof tools having a similar scope.

The system which has been produced has been of considerable benefit in exploring how to reason in $\mathcal{W}$; indeed, it has revealed several infelicities in the original presentation of $\mathcal{W}$. 2OBJ seems to be well-suited to producing this sort of system. The chief benefit has been the ability to construct with relative ease a (probably) faithful encoding which *looks* rather similar to the 'pencil-and-paper' presentation of the logic. Such a presentation has the advantage of being easy to verify correct (informally, at least), and one retains some intuition about how proofs should proceed.

Nevertheless, Jigsa$\mathcal{W}$ as described in the previous chapters, can be no more than a prototype. It is somewhat incomplete (most of the rules of $\mathcal{W}$ are included, but only small parts of the Z mathematical toolkit have been encoded), it requires a large, fast machine in order to run, and it is nevertheless very slow.

4.1 Soundness

One important motivation in some of the design decisions which have been taken is that of *soundness*. There is little value in conducting proofs about formal specifications in Z using a system which is not demonstrably sound with respect to the formal semantics of Z, since it will give *no* increased confidence about the correctness of

the specification/refinement (especially if the proof is constructed by a machine, and is too complex to check by hand).

There are broadly two areas in which unsoundness may arise in a proof assistant. It is necessary to demonstrate that the logic underlying the tool is sound,[1] and to show that the encoding of that logic in the proof framework is *faithful*.[2] The approach taken here is to build a tool which is based on a deductive system which has been widely circulated, and subject to peer review, and so is accepted as sound. The verification of faithfulness must entail some lack of formality (since the accepted presentation of the logic is not within the meta-logic of the theorem-proving tool)—but, as will be seen from Table 4.1, the critical code (i.e. the basic rules and the meta-syntactic definitions of α, ϕ, etc.) is very compact and, as has been seen above, reads very much like the original presentation.

4.2 Choice of Logic

Since Z is a *typed* language, some of the *intuition* which one brings from classical untyped set theory and logic is *unsound*. For example, since a theorem of classical predicate calculus is

$$(\forall x \bullet P) \Rightarrow P[x\backslash a] \ ,$$

we might casually write down

$$(\forall x : X \bullet P) \Rightarrow P[x\backslash a] \ ,$$

but this requires as a side-condition that X be non-empty.

Moreover, the semantics of Z gives special care to the treatment of undefined expressions.[3] A logic for reasoning about Z specifications must treat undefined values in a manner which is consistent with this semantics. Also, Z's mathematical toolkit is pre-determined. Some systems of computational logic will offer large libraries of computationally efficient definitions of datatypes (sets, relations, sequences, etc), and theorems about them. We cannot arbitrarily adopt such definitions without verifying that each is sound (and ideally, complete) for the Z toolkit.

As a result, the choice of reasoning system to use in conducting proofs about Z specifications is a critical one. The introduction to [WB92] surveys the competing options. Some authors have avoided confronting this issue, using fragments of logics which give sound inferences (or indeed, unsound ones, as above) within Z without addressing difficulties such as the ones above (see, for example, [Jon91b] on [Dil90]). Most of the impetus for providing a complete logic for Z has come from those seeking to provide proof tools for Z. The chief amongst these are the embedding of Z in HOL and the Zola logic (see Section 4.4).

The $\mathcal{W}$ logic, used here, has also been constructed with machine support in mind, though it actually arose out of work on producing a new semantics for Z [GLW91,

[1] The meta-logic also needs to be shown to be sound. This is claimed for 2OBJ (it has an extensive underlying theory), but a demonstration of the actual correspondence of the tool with the theory appears to be lacking.

[2] An encoding is *faithful* for a logic if it allows only the production of proofs which are permitted in the logic. It is *adequate* if it permits the production of *all* of the proofs permitted in the logic.

[3] These arise because functions defined in Z are often partial, and because of the μ (definite description) operator, which is also partial.

BN$^+$92]. Since it is presented alongside the standard, it seems, of the options presented here, the most sensible choice as the basis for a tool.

It should be observed that $\mathcal{W}$ is a very standard treatment of first order predicate calculus. The value of it is that it collects together in a coherent, unified manner rules appropriate for expressions formed within Z's type system, a sound way of dealing with (or avoiding dealing with) undefined expressions, and axioms for sets, tuples, bindings etc.

A symmetric sequent calculus like $\mathcal{W}$ is appealing because for any given predicate it is almost always clear which rule should be applied. Few inferences require creativity or insight; those common ones which do are ($\forall\vdash$) and ($\vdash\exists$), where the user must provide expressions to specialize the $\forall$ and to witness to the $\exists$. The general lack of such a need for creativity is the foundation of the major proof-structuring tactics described above. Rewriting with equality (i.e. applying Leibniz's rule) is harder, and needs more work (exhaustive application to every predicate pair in the goal is possible, but leads to a combinatorial explosion, and so is to be avoided). The *order* in which these rules are to be applied (i.e. which predicate in the goal to expand first) is less clear (see the Section on tactics (4.5) below). This affects the efficiency of the tool, but not its ability to complete proofs.

In proving something in $\mathcal{W}$, one ends up repeating many of the steps which a type-checker might make, which is unfortunate. For example, in Chapter 2 the application of DOM-TAC to the term $\vdash z \in \mathrm{dom}[X, Y]S$ generates not only the subgoal $\vdash z \in \{ x : X; y : Y \mid x \mapsto y \in S \bullet x \}$, but also the requirement to show that S is some subset of $X \times Y$—which the type-checker may have already determined, in order for the expression $\mathrm{dom}[X, Y]S$ to have been well-formed. Moreover, the type-checker can generally deduce the types X and Y whereas here they must be supplied explicitly. (These requirements are present to avoid difficulties with undefined values.)

A logic in which type-correctness was established within the same framework as the proof (via automatic tactics) might be more straightforward to use. It would reduce some duplication of effort, it would allow the arguments of *cut* to be type-checked automatically, and would permit generic parameters to be calculated when necessary. Stephen Brien's thesis [Bri95] presents type-inference rules for Z (in the style of [Spi88] and [SS90]), with each type inference corresponding to a logical inference in $\mathcal{W}$. Using the two systems together could form the basis of a more unified tool.

This work with $\mathcal{W}$ has also suffered from the fact that it has been undergoing change during the course of the work. It was necessary to fix on one account of the logic [WB92] (mainly consistent with an early version of the Z standard [Bri92]), and not to bring Jigsa$\mathcal{W}$ up-to-date with [BN$^+$92]. Again, keeping to one coherent account of the logic is important with regard to soundness; to mix-and-match rules adds to the likelihood of producing an unsound system. However, there are some problems of internal consistency with this version of the logic (problems of soundness do not directly concern the encoding)—most chiefly in the auxiliary definitions (of ϕ, α, etc.; See Section 4.6). Of course, locating such difficulties has been valuable to the development of the description of $\mathcal{W}$. (For example, crucial side-conditions on various substitution rules were discovered by noticing that rewrites occurred when they should not have done. Also, trying to implement 'rule-inversion' [WB92, Theorem 2] pointed out a major unsoundness in the conference pre-print of that paper.)

In the discussion section of [WB92], it is commented that this style of proof may not be the most convenient. One might hope that $\mathcal{W}$ could form the basis for

a system of, say, equational reasoning (so that another calculus could be proved sound by expressing it in $\mathcal{W}$, rather than proving soundness directly from the semantics). Ideally the tactics presented for Jigsa$\mathcal{W}$ could form the basis of such an account. In practice, they are a considerable distance from being able to do so.

4.3 Choice of Implementation Technology

For the reasons given in Chapter 1, having chosen a logic which is not identical to one of those classically studied by logicians, a *logical framework* is the most obvious tool to use to produce a working proof system[4]—otherwise one will be encumbered with the difficulty of expressing one logic in terms of another, with attendant problems of soundness *and* a loss of clarity in the interface.

The choice of which logical framework to use was fairly arbitrary. 2OBJ was under development in Oxford. It seemed to be fairly stable; it had a readily comprehensible meta-language (equational logic, in the style of OBJ3); OBJ3's arbitrary *mixfix* syntax permitted the adoption of a style of concrete syntax which closely mirrored that usually used in Z; it had an attractive user-interface; and it had a comprehensive underlying theory—expressible in OBJ3—thus giving some confidence that proofs produced would be faithful to $\mathcal{W}$.

Whether this choice was appropriate is unclear. The interface to 2OBJ is even better than it was initially, now offering various graphical displays of proof trees and the ability to 'fold' unwanted internal nodes of proof trees. Pop-up windows, and the ability to exploit the 'network transparency' of X-Windows (so that the proof tool can be run remotely, on a fast machine) are invaluable in making the tool usable. Defining Z syntax in OBJ3 proved quite straightforward (see Chapter 2), and this concrete syntax remains very readable.

However, the OBJ3 parser has considerable trouble with some of the larger Z constructs (generic definitions, for example). As will be seen from Table 4.1, parsing the source files for Jigsa$\mathcal{W}$ takes quite a long time (averaging 2–3 lines per second on the fastest available machine—most of the parsing time is taken up by parsing the toolkit and the sample specification). For many months, the inability of 2OBJ to produce a saved binary state meant that each invocation of the tool carried this heavy parsing overhead. During this time, insufficient computing power was available, so this overhead was a considerable handicap. Early versions of 2OBJ supported batch processing, but as the interactive interface improved this ceased to be an option.

Moreover, the overloading of some syntactic classes (schemas may appear as schema texts, expressions and predicates, for example)—implemented via OBJ3's *order sorted algebra* leads to many situations where differing parses are possible, and determining the correct one is not always straightforward (an example appears in Section 4.6). Some of these problems are subtle—and the parser does not flag errors as often as one might expect—giving potential soundness problems. Also, in dealing with the concrete syntax, there are various miscellaneous equivalences scattered through the standard. These must be implemented as rewrite rules (e.g. set comprehensions of the form $\{ St \}$ are converted, by adding the $\bullet$ and characteristic tuple, into the more general form $\{ St \bullet u \}$), the presence of which serves further to complicate the reasoning system. Furthermore, another problem of working directly with the concrete syntax

[4] A 'direct' implementation, in a functional or logic programming language is also a possibility. See Chapter 7.

is the lack of any ability to elide detail from the interface; everything which is in the proof tree is displayed on the screen. This is most principally a problem with the generic parameters already discussed.

Interpretation of the code is also very slow. A number of factors contribute to this. Clearly, a general-purpose tool will be inherently less efficient than a specialized one. The code for Jigsa$\mathcal{W}$ is quite a large OBJ3 program, and so the number of equations to be matched at each reduction is large. Many of the $\mathcal{W}$ rules have side-conditions which must be checked by calculating free variables and alphabets of declarations. When calculating the latter for schema terms, considerable effort is needed, as all the nested schema instances must be expanded. OBJ3 provides facilities for 'memoization' which would be of value here, but the 2OBJ mechanisms make these unusable.

The 2OBJ system actually interferes with much of the OBJ3 implementation. Initially, 2OBJ was to be implemented in OBJ3, and to take advantage of its type(sort)-safeness to guarantee that only sound proof trees could be constructed. Now, however, much of 2OBJ is implemented directly using the underlying Lisp system (for reasons of efficiency), so one must place considerable trust in the implementor's code (the type-safeness was badly compromised in one or two releases). This makes any attempt to demonstrate formally that the encoding is faithful to $\mathcal{W}$ rather futile.

One lack in 2OBJ (and hence Jigsa$\mathcal{W}$) is that it cannot support schematic proofs—i.e. proofs containing meta-variables (variables denoting predicates, for example). This is because rules' side conditions are all fully evaluated as the rule is applied; and freeness conditions are generally satisfied (x does not occur free in the literal 'p', even if p denotes a predicate); so unsound inferences may follow. However, if such a proof is reduced to a tactic (this is the only way to re-use proofs in 2OBJ) then whenever the tactic is applied (to ground terms), the side-conditions will be properly checked, and only sound inferences can result.

None of these problems has prevented a useful prototype from being produced, though they have added considerably to the frustration of the author. The implementors of 2OBJ were always very willing to fix bugs, but the frequency of new releases of the system became sometimes rather hard to handle. No further development work on 2OBJ can be foreseen, so future enhancement of Jigsa$\mathcal{W}$ is unlikely to be worthwhile.

4.4 Comparison with Other Approaches

Several other proof tools for Z are available. They may be classified in a variety of ways; the principal distinction appears to be whether they implement directly a logic for Z, or seek to embed Z in some other logical system.

Encodings within other Logics

One of the most successful theorem-proving assistants available is HOL [Gor88]. There are at least two encodings of Z in HOL [BG94, Jon92]. The first paper describes the difference between these two as a difference between 'shallow' or 'deep' encoding. The former is shallow in that it 'macro expands' some Z constructs (in particular, schemas) into much simpler HOL constructs. This gives an encoding which is suitable for reasoning within specifications, but is not able to prove results about the language (the commutativity of schema conjunction, for example; such a result could be proved for arbitrary schemas in $\mathcal{W}$, though not in the present encoding in 2OBJ). ProofPower

(from ICL [Jon92]), provides a deeper embedding, with all the Z operators defined in HOL. As such, one level of potential error (in the incorrect macro expansion, taking account of variable capture) is avoided. Nevertheless, neither is ProofPower 'deep' enough to be able to prove properties like the commutativity of schema conjunction.

The benefit of this approach is that HOL is widely-used, trusted and well-supported, and it is supplied with a very large tactic library, so many complex proofs can be accomplished with relatively little effort. The chief disadvantage is that whilst the representation of Z in HOL claimed to be 'semantically faithful, in that the terms chosen to represent any given construct are not only adequate to represent the construct syntactically, but also *mean the same thing* as the relevant Z construct' [ICL93], this can be verified *only* if the semantic description of (draft) standard Z is expressed in HOL (and it is not). Gordon suggests [BG94] that one could encode the entire semantic description from [BN+92] in HOL, and thus obtain a tool capable of deriving logical rules (such as those in $\mathcal{W}$)—though such a tool seems unlikely to be tractable for reasoning about specifications [Gor94].

Another encoding approach is to encode something like $\mathcal{W}$ within *Isabelle* [Pau89]. This work is in progress in Zurich.

Directly Constructed Tools

The *Zola* tool from IST is based on a logic of its own (the *Zola logic*), which is very similar to $\mathcal{W}$.[5] It is, however, not part of some other proof assistant, but constructed solely for use with Z—indeed, a large part of the Zola tool is the syntax-directed editor and type-checker, which provides, in some measure, a CASE tool for Z specification.

Zola takes the same approach as that take here to the Z mathematical toolkit. It is provided as an on-line specification document, and it is possible to use a general-purpose tactic to solve many of the 'toolkit laws' which arise in proofs. Zola's tactics are compared with those used here (and the refinement of them, in Part II) in Section 6.5.

CADiZ [JMT91] is a similar venture, though proof support there is still in its early stages. The proof engine in this system is using rules based on those in $\mathcal{W}$.

4.5 Tactics

The tactics presented above (Sections 2.9–2.12) are naïve but effective. They accomplish a useful range of proofs, but are very slow. The general approach taken is to decompose antecedents and consequents into the simplest atomic predicates (membership of base type, equality) and then attempt to apply the *assumption* rule (and, for consequents, the rule of *reflection*). An earlier application of assumption would be more efficient.[6] Replacement of the simple assumption rule with some sort of unification would improve the chances of finding a match; in particular, would make use of the tactics based on Leibniz's rule more automatable. However, such a unification algorithm would need to be expressed in 2OBJ's tactic language, and as such would probably not be fast enough in operation to be useful.

[5] Some work showing that the Zola logic is relatively sound with respect to $\mathcal{W}$ has been undertaken.

[6] One could ally the attempt at assumption with the application of tactics in a tactic like TRY-EACH, so that after a given predicate is transformed, *assumption* can be tried without attempting to match every antecedent with every consequent, as in HASSUM.

By reason of this tendency only to simplify terms, the automatic tactics do not create as much of a combinatorial explosion as one might fear. The chief source of extra material in the goals is the inclusion of schemas and other definitions. The toolkit tactic thins out the latter quite effectively (see, for example, DOM-TAC in Figure 2.1). Predicates arising from schema expansions, on the other hand, cannot readily be thinned, as they may be needed later in the proof. A bigger problem might arise from the arbitrary/exhaustive application of MLEIB—which is why it is left under user control. With hindsight, it is clear that an improved version of CUT would be worthwhile. Some thinning could usefully be employed—i.e. in the inference on page 18, the predicates Ψ could be thinned from the left-hand branch. It is hard to make such a change at this point, as so many tactics rely on term positioning and ordering. If infinite loops are to be avoided in breadth-first searches, it is necessary that each component tactic in, say, BIG-TAC should return goals which do not contain the same top-level connectives as those in their original goal.

At each iteration, a tactic like NEW-TAC3 attempts to apply a large number of rules in a number of places within the current goal. The time taken to attempt all these matches is small in comparison to the time taken to check side-conditions where rules do apply to goals, so this seems a most reasonable approach. As the side conditions are such a performance problem, care must be taken to make each tactic which appears in the scope of a LIFT do as much work as possible—using one LIFT instead of several, whenever possible, as this reduces the number of times that the LIFT side-condition needs to be checked. In a later chapter, Section 6.3 discovers exact conditions for the combination of such LIFTs.

Some of these tactics make very careful use both of success and failure in tactic application. Often it is important to allow a tactic to fail—it can then be used in the context of an EXHAUST or TRY, as appropriate. The differing design philosophies behind various tactics tend to force them into a number of 'families'. Tactics MPROP, EXPLODE, SUBST, etc. never fail, and so are used in sequential composition (using THEN). Tactics like AUTO-SCHDEF and TK-TAC fail if they are not applicable to the current goal—so they can be combined more readily using ELSE, often within the scope of an EXHAUST. This is the striking contrast between TOEACH and TRY-EACH. The former never fails—making it hard to incorporate with other tactics in a large tactic with recursion—whereas the latter may fail, so it can be applied exhaustively (in conjunction with other tactics, as appropriate). The algebra of how such combinations work is a feature of the next part of this thesis.

The proofs constructed thus far have given some insight into the sorts of tactics which will be needed in order to make proving mundane theorems an easy task. Construction and refinement of these tactics is the obvious next step. It was once suggested that 2OBJ, in functioning as a *meta*-logical theorem-prover, could assist in proving (meta-)theorems about proofs, as well as supporting reasoning *within* a given logic. This has not become possible. Instead, the next part of this thesis explores a means of reasoning about tactics—and hence about the proofs they produce.

4.6 Auxiliary Definitions

Undoubtedly, the greatest part of the effort involved in encoding W in 2OBJ came in the definition of what Section 2.3 described as W's meta-functions—α which determines the alphabet of a declaration, ϕ which extracts the set of free variables in a term,

and the syntactic equivalences (rewrite rules) for binding substitutions.

The chief problem with these functions has been the interaction of two difficulties: (a) the account of these functions in [WB92] is incomplete (very incomplete in places), so the encoding incorporates many *ad hoc* cases,[7] and (b) the 2OBJ encoding's reliance on OBJ3's order-sorting for the syntactic classes makes errors in these expressions very difficult to debug (as well as being time-consuming).

For example, the infix operator '.' (written $_{\odot}$ in [BN+92]) for attaching a binding to a term (accomplishing a substitution) is overloaded in that it may attach a binding to a schema or a predicate. The rule

$$b.[d_1;\ d_2] \equiv b.[d_1] \wedge b.[d_2]$$

therefore has an ambiguous term on the left (it might be a schema or a predicate), but the term on the right must be a predicate (the possible ambiguity between schema conjunction and predicate conjunction is not at issue here). However, schemas form a sub-sort of the sort of predicates (a schema may be used as a predicate) and as a result the parser assigns the sort `schema` to the left-hand side. In order to make the equation sort-correct, the right-hand side therefore gains a OBJ3 *retract* (which is by default invisible), `r:Predicate>Schema`. This serves to coerce the right-hand side into being a schema, which then fails to match any predicate rules (e.g. $\vdash\wedge$). Identifying this problem was rather difficult. The OBJ3 experts concluded that it was probably a bug, and suggested the work-around of defining the two instances of $_{\odot}$ in separate modules (which allows them to be distinguished).[8]

These meta-functions also form the major efficiency boule-neck in the application of rules in Jigsaw. Many rules have side conditions, or are invoked via `LIFT`, with its side conditions, and these side conditions generally require the calculation of α and ϕ for various terms. For sequents involving many terms (or nested sequents) with binding substitutions attached, such calculations can be lengthy.

2OBJ provides a library of built-in functions which are intended to make such calculations straightforward (and fast, being implemented in LISP), and to assist in the construction of fresh variables as necessary. However, Z's syntactic conventions appear to be too rich to fit into the scheme for describing which variables are bound, and which free in expressions/predicates. In particular, one gives expressions along the lines of `eq variable(x) = true`. Since some of the variables in the encoding are not simple variables, however, but compounds (e.g. `x'`), the system was defeated (it would calculate that x was a free variable of `(x')`). Moreover, attempts to circumvent this became baroque, and remained unsuccessful:

```
eq phie (t) = freevars (t)
    if (not(matches((t1 IF t2),t))) and (not(matches(xx ^q,t))) .
eq phie (t1 IF t2) = phie(t1) ;; phie(t2) ;; Op(@ IF @) .
eq phie (xx ^ q) = xx ^q .
```

The intention of the `if`-clause in the first equation is to prevent expressions such as x' and $S \cap T$ from being decomposed too far. x' is a free variable in its own right, and the free variables of $S \cap T$ include $(\_ \cap \_)$ (which is not apparent to the built-in function `freevars`).

[7] An appendix to this Chapter lists some of these *ad hoc* inclusions and tricks.

[8] The OBJ3 grammar allows terms to be differentiated by labelling them with their module and/or sort name. In this case, no amount of sort labelling seemed sufficient to convince the parser that this was an equation between predicates, and not between schemas.

Instead, then, α and ϕ are implemented directly using 2OBJ's UNIVERSAL-FINSETS module. This provides a (relatively) efficient implementation of finite sets (with operations for union, intersection, membership, etc.). One source of difficulty in debugging this code (and also of unsoundness) was the fact that OBJ3 will not automatically flag incomplete reductions as errors. Hence, if, say, the rule for `phie (t1 IF t2)` above were omitted, the term would stay un-reduced, and have an empty intersection with some list of variables—potentially allowing a law to be applied which should not be applied.

This problem might be solved by making a derived intersection/set equality test which returns `false` if the terms to be intersected are not simple sets of variables. Coding this test in OBJ3, however, would further compound performance problems in an area which is already a bottleneck. A more efficient solution is to ensure that for every term t, αt (etc.) can be fully reduced to a set of variables (that is, show that the rules for α are Church–Rosser and terminating).

4.7 Rule-lifting

The authors of [WB92] found the theorem on rule-lifting to be useful in structuring the paper account of $\mathcal{W}$. Making rule-lifting into a meta-rule turned out to be a valuable way of structuring the encoding, also. Extending this to cover tactic-lifting has provided a most useful proof(tactic)-structuring technique.

The implementation of rule-lifting (tactic-lifting) raises some soundness concerns. Its soundness with respect to the meta-logic is guaranteed by use of the built in operator TAC, which invokes the tactic interpreter; the resulting inference gets converted into a rule, which is then applied to the current goal. The soundness of that (derived) rule is guaranteed by the sound construction of tactics from rules by the tactic interpreter. Its soundness with respect to $\mathcal{W}$ can be verified only by careful study of the (rather complex) code; confirming that it is consistent with Theorem 2.5.1.

The chief outstanding difficulty with rule-lifting is in the way that terms are selected by the index of their position in the list of predicates (or declarations). Chapter 6 presents an account of rule-lifting which is generic over schemes for selecting predicates.

4.8 Strengths and Weaknesses

The work described in this first part of this thesis has explored the possibility of making a sound theorem-prover based on $\mathcal{W}$ and 2OBJ, programmable and adaptable to a wide class of problems. In so doing, it has developed for $\mathcal{W}$ some tactics which address some of the more common concerns: the more automatable ones.

This set of tactics is effective in proving certain theorems (the initialization theorem work for the case study above transferred with no alterations to the (simpler) problem of initialization for the standard *BirthdayBook* [Spi92a] example). However it is relatively narrow in its applicability, and uses heuristics which are unnecessarily inefficient. The speed of execution has prevented significant improvements in the design of these heuristics from being investigated.

The execution speed of the tool appears to be hit most significantly by the effort involved in checking side-conditions. This is, at least in part, due to the elaborate

Code for Z syntax	570	lines
Code for $\mathcal{W}$ rewrites, side conditions etc	681	lines
Code for account of logic proper	815	lines
Code for basic tactic library	1 598	lines
Code for toolkit (very incomplete)	675	lines
Code for demo8.obj (Security System)	348	lines
Total	4 687	lines
Compiled image for Jigsa$\mathcal{W}$+ Tooklit + Demo 8	15 079 928	bytes
On SparcStation10 with 32Mb:		
Time to build compiled image	30	mins
Time to prove init-thm in case study:	$\gg$ 20	hours
Time to prove pre-thm-a	1	hour
Time to prove pre-thm-b	$\gg$ 1.5	hours
On SparcStation2 with 32Mb:		
Time to prove init-thm for BirthdayBook	40	mins

Table 4.1: Some Statistics

notions of variable occurrence and scope in Z. Finding a logical framework with sufficiently general notions of variable binding may be a difficult task.

In order to give some idea of the scale of the work described here, and to give some substance to the comments about the speed of the implementation, some rough statistics relating to Jigsa$\mathcal{W}$ are presented in Table 4.1.

4.A Rules and Tricks Added

The $\mathcal{W}$-rewrites module uses a number of syntactic equivalences which are missing from the published accounts of $\mathcal{W}$ [WB92] (and/or the Z draft standard [BN+92]).

$$b.(x\{t_1, t_2\}) \equiv (b.x)[b.t_1, b.t_2]$$

$$b.(s, t) \equiv (b.s, b.t)$$
$$b.\{t\} \equiv \{b.t\}$$

$$b.(t\ IG\ u) \equiv b.t\ IG\ b.u \quad \text{provided } \alpha b \cap (\_\ IG\ \_) = \varnothing$$

$$(\!|\ (\_\ IR\ \_) \rightsquigarrow t\ |\!).(t_1\ IR\ t_2) \equiv (t_1, t_2) \in t$$
$$(\!|\ (\_\ IF\ \_) \rightsquigarrow t\ |\!).(t_1\ IF\ t_2) \equiv t(t_1, t_2)$$

$$(\!|\ x \rightsquigarrow t;\ B\ |\!).s \equiv (\!|\ B\ |\!).((\!|\ x \rightsquigarrow t\ |\!).s) \qquad \alpha(\!|\ B\ |\!) \cap \phi t = \varnothing$$

$$(\!|\ SN \rightsquigarrow S\ |\!).SN \equiv S$$
$$[(\!|\ SN \rightsquigarrow S\ |\!).SN] \equiv S$$

$$b.(c.p) \equiv c.(b.p) \quad \text{whenever } \alpha c \cap \phi_e b = \varnothing \wedge \alpha(b) \cap \phi_e c = \varnothing$$

$$b.p \equiv p \quad \text{whenever } \alpha b \cap \phi_p p = \varnothing$$

$$[S]^q \equiv [S[\lambda]] \quad \text{where } \lambda = (\!|\ x_1 \rightsquigarrow x_1^q \ldots x_n \rightsquigarrow x_n^q\ |\!), \text{ with } \alpha S = \{x_1, \ldots, x_n\}$$

$$(\!|\ SN \rightsquigarrow S\ |\!).(SN[\lambda]) \equiv S[\lambda]$$

$$\alpha(\!|\ x \rightsquigarrow t\ |\!) \equiv \{x\}$$
$$\alpha(\!|\ x \rightsquigarrow t;\ B\ |\!) \equiv \{x\} \cup \alpha(\!|\ B\ |\!)$$
$$\alpha(\!|\ SN \rightsquigarrow t\ |\!) \equiv \{SN\}$$
etc.

$$\alpha(b.d) \equiv \alpha(d) \quad \text{provided } \alpha b \cap \alpha d = \varnothing$$

The trick for calculating αSN has already been mentioned:

$$\alpha SN \equiv \alpha(mkdecls(decl\text{-}part(SN\ \text{-}def)))\ ,$$

the auxiliary functions (`mkdecls`) fetching and expanding the schema definition as necessary. Decorated schemas present little additional challenge: we get the alphabet of the undecorated schema, and then decorate each member of the set.

$$\alpha(S^q) \equiv mapdecor(\alpha S, {}^q)$$

Finally, the alphabet of a schema under renaming/binding substitution is calculated by fetching the declaration part of the schema and applying the renaming to it:

$$\alpha(SN[B]) \equiv \alpha((expdecls(SN))[B])$$
$$\alpha(b.SN) \equiv \alpha(b.(expdecls(SN)))$$

For $\alpha((b.S)[B])$, etc., we need to invoke *subst* within the alpha somehow—and therefore this must not arise in the scope of a side-condition (otherwise an infinite loop will result).

The inference rule *chHypVars* certainly needs some side conditions—those attached to alpha-conversion suffice.

Part II

Tactic Language

Chapter 5

A Tactic Language

ONE OF THE DIFFICULTIES which arose in the construction of the tactics in the previous chapters was that they quickly became rather complex. The tactics in Sections 2.9–2.12, in particular, are quite hard to understand and verify. This chapter attempts to remedy this by presenting *Angel*—a very general language for expressing tactic programs, making very few assumptions about the form of the expressions (goals) in the target logic, and about the rules which act upon them, transforming one expression into another.

It is hoped that by describing a tactic language in this way it will be possible to demonstrate functional correctness of tactics, by providing a semantics for tactics and using it to produce algebraic laws for tactic equivalence. One application of such laws is to provide correctness-preserving transformations, enabling, for example, tactic efficiency to be improved. Although the language was originally intended to support goal-directed (backward) proof in a natural way, it has been found to be much more widely applicable as a language in which general expression transformations can be described.

It is important to distinguish between *soundness* and *functional correctness* in a theorem-proving system. The former is guaranteed by ensuring that the proof *rules* are sound, that the encoding is *faithful*, and that rules (and the goals to which they apply) can be combined to make *proofs* only in a sound way. This soundness of combination is often accomplished via a *safe datatype*—that of *proof* in LCF; that of *tactic* in this chapter. Such a tactic will be *functionally correct* if it accomplishes the proof its author intended—that is, if it satisfies its specification. If the design of the tactic is amiss, then either an unexpected proof will be created (*one which will still be sound*), or (more likely) the proof will fail.

A crucial feature of the language is its 'angelic' nondeterminism. When a tactic presents a choice of possible next steps, the step(s) which will succeed (if any) will be chosen. This will generally be implemented using backtracking.[1] Our semantics

[1] Notice that the issues of success/failure and termination are orthogonal. The language is angelic with respect to the first but (naturally) not to the second. The construction of terminating functions is a well-

admits the following law, which is quite distinctive ('|' is often written ORELSE, and ';' as THEN):

$$(a \mid b)\,;c = (a\,;c) \mid (b\,;c)\ .$$

This means that if c fails after the a branch has been taken, the b branch will be attempted (followed again by c). It is more customary for such choices to be made once only (i.e. to fail if c fails, despite there being an untried choice further up the tree). This might be described as a contrast between 'shallow' and 'deep' backtracking.

Section 5.1 gives an overview of the constructs in the tactic language, and is followed by a section giving some examples of the application of these tactic constructs. Section 5.3 gives a formal treatment of the fundamental constructs in the language, and some of the algebraic laws arising from these definitions are presented in Section 5.4, together with a proof that these laws form a complete set with respect to the semantics given in Section 5.3. The theory is extended to cover recursively defined tactics in Section 5.8, and Section 5.6 describes some extensions to the language which are particularly useful for reasoning about tactics. Various derived tacticals are described in Section 5.9, and an example shows how properties of those definitions can be proved, demonstrating that the tacticals behave as expected. Section 5.10 further extends the tactic language to include 'structural combinators', which allow tactics to exploit structural properties of the expressions to which they are applied. Section 5.11 describes a method for adding pattern-matching to the language, and Section 5.12 uses the ideas from the previous two sections to make the language appropriate for proofs that bifurcate; that is, where goals arise in parallel composition.

Sections 5.1–5.10 form a paper which is expected to appear (subject to re-review) as joint paper with Paul Gardiner and Jim Woodcock, in the journal *Formal Aspects of Computing*. I am grateful to the other named authors for the ideas presented in Sections 5.1–5.3, which have enabled me to develop the rest of the chapter.

5.1 Tactic Language

In this section we give an informal description of the tactic language, explaining the operation of each language construct. The precise meaning is presented in Sections 5.3, 5.8 and 5.10. Occasionally, this will be referred-to as a *meta*-language, to distinguish it from the *object* language in which the basic rules (see below) are expressed. Expressions in the meta-language are described here, expressions in the object language will often be called 'goals'. They may be predicates, sequents, programs, or algebraic expressions (etc.), depending on the system under consideration.

The set of basic rules may be considered to be a subset of the set of tactics. It is helpful to mark the use of these atomic tactics, and so their use in tactics is signaled by the use of the keyword **rule**:

rule *rulename* .

There are two possible outcomes when applying an atomic tactic to a goal (expression): if the rule matches the expression (i.e. the expression is in the domain of the rule) then the rule is applied, producing a new expression; if the rule does not match then the

understood problem in both functional and imperative programming languages, and we envisage similar techniques being used here.

application fails. A formal treatment of failure permits reasoning about tactics whose definitions critically depend upon the failure of a rule to apply, as well as on successful application.

Two special atomic tactics exhibit these two behaviours. The first always succeeds, leaving its expression unchanged; the second always fails:

skip

fail .

Functions which operate on tactics, returning new tactics as a result, are conventionally called *tacticals*[GMW79]. Fundamentally, tactics can be combined in two ways: in alternation or in sequence. The sequential composition of two tactics is written:

$$t_1 \; ; \; t_2 \; .$$

The behaviour of this tactic is first to apply t_1, thus producing a new expression (goal), and then to apply t_2 to that expression (goal). If either t_1 or t_2 fails then so does the whole composition.

The alternation of two tactics t_1 and t_2 is written:

$$t_1 \mid t_2 \; .$$

When applied to an expression, the above tactic may succeed by applying t_1 or by applying t_2 (t_1 is tried first). The tactic fails iff both t_1 and t_2 fail. As discussed in the introduction (see below for a formal definition), this combinator gives rise to a form of angelic choice: it will choose whichever tactic allows success in future tactic applications (i.e. those later in a sequential composition), preference going to the tactic on the left.

Since this form of alternation may lead to problems of inefficient (wasteful) searches, and (in the recursive case) problems of non-termination, a *cut* operator, in the style of logic programming, is provided:

$$!t \; .$$

This tactic behaves exactly like t, but locally restricts the action of alternation. It returns the first successful tactic application; if a subsequent tactic application fails, then the whole tactic fails; alternatives within t are not re-explored.

Recursive tactics are written

$$(\mu X \bullet tac(X)) \; ,$$

where X is a variable and $tac(X)$ is a tactic in which the variable X may occur as though it were itself a tactic. The tactic $(\mu X \bullet tac(X))$ behaves as $tac(X)$, but with each occurrence of X behaving as though it were $(\mu X \bullet tac(X))$.

Inclusion of recursion in the language introduces an extra possibility for the eventual outcome of a tactic. As well as succeeding (producing a new expression) or failing to apply, it may fail to terminate and run indefinitely. Whilst such a tactic will not in general be useful when *writing* tactic programs, it is helpful to be able to reason about it. We follow Dijkstra [DS90] and call the non-terminating program

abort .

Depending on the form of the expressions to which the rules and tactics apply, it is often appropriate to incorporate *structural combinators*[2] in the tactic language. These permit the controlled application of tactics to particular sub-expressions. Whether such applications are appropriate is dependent upon the form of the rewriting system being used—monotonicity is required (so that rewriting a sub-expression produces a valid transformation of the whole expression). For example, in a term rewriting system, we might want to define tactics which operate on expressions of the form $p \wedge q$, $p \vee q$, etc., applying one tactic to that part of the goal corresponding to p, and another to that corresponding to q. The structural combinators $\boxed{\wedge}$ and $\boxed{\vee}$ accomplish this:

$$t_1 \boxed{\wedge} t_2$$
$$t_1 \boxed{\vee} t_2 \ .$$

So $(t_1 \boxed{\wedge} t_2)(g_1 \wedge g_2) = (t_1\ g_1 \wedge t_2\ g_2)$. In general, we would require one such combinator for each operator in the object language.

Similarly, a valuable construct in goal-directed proof is a parallel combinator, which applies lists of tactics to lists of goals—in this case, a pair of tactics to a pair of goals; applied to (g_1, g_2), the following tactic would apply t_1 to g_1 and t_2 to g_2:

$$t_1 \parallel t_2 \ .$$

π permits the definition of tactics which are dependent upon the goal to which they are applied (see Section 5.11 for more motivation of the definition of this tactical).

$$(\pi\, v_1, \ldots, v_n \bullet g \longrightarrow t) \ .$$

This tactic binds the (meta-)variables $v_1, \ldots, v_n$ within the scope of g and t. If goal g matches exactly the goal presented to the tactic (the variables $v_1, \ldots, v_n$ being angelically chosen, if possible, to make this be the case), the whole tactic behaves like tactic t. If not, the tactic fails

In the remainder of this chapter, we will use a range of meta-syntax for describing tactics. A simple equality will be used to introduce a named tactic—this may be read simply as a macro expansion (with parameters instantiated where necessary, and circular (recursive) tactic references replaced by suitable μ-expressions.

To avoid over-use of parentheses, we adopt the following order of precedence for operator binding: function application (including *succs* and *fails*—see below) binds closest of all; cut binds next closest; with the binary operators next (sequential composition binding most strongly, followed by structural combinators, and alternation binding the least strongly). μ and π (and con—see Section 5.11) bind their variables as far to the right as possible—i.e. they are the weakest of all.

5.2 Examples

Alpha-Conversion

Problems relating to the capture of free variables often arise in mechanized theorem-proving. These are often dealt with by the use of procedures outside the tactic/rule

[2] Michael Goldsmith has suggested the more picturesque name of *geographical* combinators.

language. This section demonstrates an approach which might be adopted in our tactic language.

Consider a system for rewriting predicate calculus expressions. The basic rewrite rules will be expressed as meta-equalities (using $\equiv$), rewriting going from left to right. A rule such as $\exists\wedge$ is limited in its usefulness by the side condition attached to it:[3]

$$\exists\wedge \;==\; \big((\exists x \bullet P) \wedge Q\big) \equiv \big(\exists x \bullet (P \wedge Q)\big)$$
$$\text{where } x \text{ is not free in } Q \ .$$

This will generally be approached by the application of a rule which performs an alpha-conversion on goals of the form $\big(\exists x \bullet P(x)\big)$:

$$\exists(y) \;\approx\!=\; \big(\exists x \bullet P(x)\big) \equiv \big(\exists y \bullet P(y)\big)$$
$$\text{where } y \text{ is not free in } P \ .$$

The difficulty in applying $\exists(y)$ comes in the choice of the variable y. Our approach is to define the following tactic which makes successive choices for possible values of y, chosen from some set of variable names $\{y_1, \ldots, y_n\}$:[4]

$$t{\sim}\exists\alpha \;==\; \big|_{y\in\{y_1,\ldots,y_n\}} \textbf{rule}\ \exists(y) \ .$$

Now we can write a tactic which generalizes the rule $\exists\wedge$:

$$t{\sim}\exists\wedge' \;\approx\!=\; (t{\sim}\exists\alpha \boxed{\wedge} \textbf{skip})\ ;\ \textbf{rule}\ \exists\wedge \ .$$

This, then, is a tactic which, when presented with a goal of the form $\big((\exists x \bullet P) \wedge Q\big)$ will search for an alpha-conversion which will permit the expression to be rewritten as $\big(\exists y \bullet (P \wedge Q)\big)$. As the tactic stands, if it were sequentially composed with a tactic which might fail (e.g. due to the chosen bound variable introducing a later conflict), it would backtrack, and produce further alpha-conversions as necessary. This behaviour may be undesirable (e.g. the applicability of the later tactic might not be improved by further alpha-conversions); in this case we could use the tactic

$$!\,(t{\sim}\exists\wedge') \ ,$$

which would proceed with the first alpha-conversion to permit $\exists\wedge$ to be applied, and permit no later backtracking (but fail instead).

Searching

Structural combinators may be combined with recursion to produce powerful tactics which search for points of applicability. For example, if presented with an expression which was in the form $(p_1 \wedge (p_2 \wedge (\ldots \wedge p_n)\ldots))$, the following tactic will find the first p_i which is a disjunction, and apply tactic t to its first disjunct. If none of the p_i is a disjunction, the tactic fails.

$$\mathit{first\_or}(t) \;\approx\!=\; \mu X \bullet \big((t \boxed{\vee} \textbf{skip}) \boxed{\wedge} \textbf{skip} \mid \textbf{skip} \boxed{\wedge} X\big)$$

In fact, the *cut* version of this tactic behaves as described above; without a *cut* it will backtrack as necessary, and apply t to each left-hand disjunct in turn.

[3] Here x, P, and Q are meta-variables; they will be bound respectively to the appropriate quantified object variable and to the predicates present when the rule is applied to a goal.

[4] Note that $\big|_{y\in\{y_1,\ldots,y_n\}} t(y)$ is a shorthand for $t(y_1) \mid \ldots \mid t(y_n)$. See also, Section 5.8.

Commutative/Associative Rewriting

When applying tactics to goals containing operators which are commutative and/or associative, it is helpful to be able to try to apply the tactic to various commutative/associative instances under tactic control. If a rule fails to apply, we would like to backtrack and apply a transformation to the goal—producing a different commutative and/or associative instance—and then try to apply the rule again.[5]

For example, in the account of $\mathcal{W}$ above, (a more general version of) the following rule appears

$$\frac{\vdash t.1 = u_1 \wedge t.2 = u_2}{\vdash t = (u_1, u_2)} \uparrow\downarrow \ (\mathit{cartProdMem})$$

In applying this rule (in the forward sense), it would be most desirable to be able to match (automatically) $t.2 = u_2 \wedge t.1 = u_1$. Using a suitable instance of *comm* (below, with $\boxed{\wedge}$ for $\boxed{\oplus}$), a more general inference would be accomplished by the tactic

rule *cartProdMem* | **rule** *comm* ; **rule** *cartProdMem* .

For a rule with more instances of $\wedge$, a more general (recursive) version of *comm* is needed.

For a binary operator $\oplus$, and a rewrite rule

$$\mathit{comm} \ == \ a \oplus b \equiv b \oplus a$$

we may write a tactic which generates as alternatives all the commutative instances of its goal:

$$\begin{aligned} \mathit{comms} = \mu X \bullet \ & (\textbf{rule}\ \mathit{comm} \mid \textbf{skip}); \\ & \big((X \mid \textbf{skip}) \ \boxed{\oplus}\ \textbf{skip}\big); \\ & \big(\textbf{skip}\ \boxed{\oplus}\ (X \mid \textbf{skip})\big) \ . \end{aligned}$$

Likewise, if the rule *assoc* expresses the associativity of $\oplus$

$$\mathit{assoc} \ == \ a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$

(we shall write *cossa* for the application of the same rule as a rewrite from right to left) then *assocs* is a tactic which generates all possible alternative associations of $\oplus$:

$$\begin{aligned} \mathit{norm} \ &= \ \big(\mathit{exhaust}(\textbf{rule}\ \mathit{cossa}) \ ; \ (\mathit{norm}\ \boxed{\oplus}\ \mathit{norm})\big) \mid \textbf{skip} \\ \mathit{assocs} \ &= \ !\mathit{norm} \ ; \ \mathit{exhaust}(\textbf{rule}\ \mathit{assoc}) \ ; \ \big((\mathit{assacs}\ \boxed{\oplus}\ \mathit{assocs}) \mid \textbf{skip}\big) \ . \end{aligned}$$

(*exhaust* is defined in Section 5.9—it applies its argument as many times as possible.) In this definition, *norm* is a tactic which 'normalizes' an expression—associating all of its $\oplus$s to the left. Ways of improving these tactics will be discussed in Section 6.1.

One might have expected that to produce a tactic which attempts all associative/commutative instances in turn, it would suffice to compose *assocs* and *comms* sequentially. This turns out to be too naïve (such a composition would need to be iterated n times if there are n instances of $\oplus$ in the goal, and that iterated tactic produces *many* duplicates); an efficient combined tactic is yet to be discovered.

[5] The tactics which follow do not guarantee to produce each instance only once; merely that all possible instances will be produced. See also, Section 6.1.

5.3 Semantic Model

In this section we give a denotational semantics for the fundamental constructs of the tactic language; that part of the language given by

$$
\begin{array}{rcl}
T & ::= & \mathbf{rule}\, R \\
& | & \mathbf{skip} \\
& | & \mathbf{fail} \\
& | & T\,;T \\
& | & T \mid T \\
& | & !\,T \qquad ,
\end{array}
$$

where R is some basic set of rules which transform one expression into another.

Our semantics makes no assumptions about the form or structure of the expressions to which our tactics apply; we shall simply describe expressions as being members (g, g_i, etc) of a set G.

The behaviour of alternation requires the model to allow that application of a tactic may produce several possible outcomes, with the order in which the outcomes arise being of some importance. We will, therefore, use lists to describe the possible outcomes of a tactic application.

A suitable account of the theory of lists will be found in [Bir88].[6] An appendix gives some of the more important definitions. For a set A, $\mathrm{seq}\, A$ is the set of all finite lists whose elements are drawn from A. $\frown$ is the list concatenation operator, and $\frown/$ represents distributed concatenation (sometimes called *flatten* or *concat*). *head'* is the function which takes a list, and returns a list consisting of the first element of the argument list; or returns the empty list, if the argument list was empty. For a total function $f : A \to B$, $f* : \mathrm{seq}\, A \to \mathrm{seq}\, B$ is the function that operates on lists by applying f to each of their elements. Conversely, $^\circ$ applies a list of functions to a single argument, producing a list as the result (so $\langle f, g, h\rangle^\circ x = \langle f\, x, g\, x, h\, x\rangle$).

Tactics, then, will be total functions from G to $\mathrm{seq}\, G$. The list of expressions returned by a tactic is the list of all possible outcomes of the tactic application, arising from all possible paths through any alternation contained within the tactic. Failure will be denoted by the empty list; when a single new expression (subgoal) is returned (as in the case where a basic rule is applied), this will be denoted by a singleton; and where there are several alternative subgoals (see alternation, below), a list of alternatives is returned.

Basic rules are the simplest tactics. When applied within their domains, they produce singleton results—with no alternatives. When applied outside their domains, they fail, and return no results at all.[7]

$$
\begin{array}{l}
g \in \mathrm{dom}\, r \;\Rightarrow\; \mathbf{rule}\, r\, g = \langle r\, g\rangle \\
g \notin \mathrm{dom}\, r \;\Rightarrow\; \mathbf{rule}\, r\, g = \langle\,\rangle
\end{array}
$$

The atomic tactics **skip** and **fail** have very simple definitions.

$$
\begin{array}{rcl}
\mathbf{skip}\, g & = & \langle g\rangle \\
\mathbf{fail}\, g & = & \langle\,\rangle
\end{array}
$$

[6] Whilst the account here makes use of Bird's theory of lists, the *notation* will be more Z-like.

[7] In the term **rule** r, the r denotes a rule *name*, whereas in $\mathrm{dom}\, r$ and $r\, g$, r denotes a function from goals to goals. Semantic brackets might be used to avoid ambiguity, but they are not generally necessary.

The fundamental tacticals described above can also be given simple definitions in this list notation. Alternation concatenates the alternatives presented by its component tactics;[8] sequential composition applies the second tactic to all the possible outcomes of the first. Cut counteracts the action of alternation by restricting the tactic application to the first successful outcome.

$$
\begin{aligned}
t_1 \mid t_2 &= \frown/ \circ \langle t_1, t_2 \rangle^\circ \\
t_1 \,;\, t_2 &= \frown/ \circ t_2^* \circ t_1 \\
!\, t &= head' \circ t
\end{aligned}
$$

5.4 Simple Laws

Our aim is to produce a calculus for reasoning about tactics written in this language. As such, this section gives a small set of laws which are proven sound with respect to the semantics in Section 5.3, and shown to be complete.

Laws

skip is a unit of sequential composition; **fail** is a unit of alternation and a zero of sequential composition.

Law 5.1 (a) $\mathbf{skip} \,;\, t = t$ (b) $t = t \,;\, \mathbf{skip}$

Law 5.2 (a) $t \mid \mathbf{fail} = t$ (b) $t = \mathbf{fail} \mid t$

Law 5.3 (a*) $t \,;\, \mathbf{fail} = \mathbf{fail}$ (b) $\mathbf{fail} = \mathbf{fail} \,;\, t$

Both sorts of composition are associative, and sequential composition distributes over alternation *on the right only*.

Law 5.4 $t_1 \mid (t_2 \mid t_3) = (t_1 \mid t_2) \mid t_3$

Law 5.5 $t_1 \,;\, (t_2 \,;\, t_3) = (t_1 \,;\, t_2) \,;\, t_3$

Law 5.6 $(t_1 \mid t_2) \,;\, t_3 = (t_1 \,;\, t_3) \mid (t_2 \,;\, t_3)$

The distributive law on the left succeeds only for *sequential* tactics, due to the ordering of alternatives: in $t_1 \,;\, (t_2 \mid t_3)$, the alternatives for t_2 and t_3 (arising from different alternatives for t_1) are interleaved; whereas in $(t_1 \,;\, t_2) \mid (t_1 \,;\, t_3)$, all of the alternatives for t_2 precede those for t_3.

Definition 5.4.1 **(Sequential Tactics)** *A tactic is* sequential *if it is* **skip**, **fail**, **rule** *r for some basic rule r, or it has the form $t_1 \,;\, t_2$, where t_1 is in one of these forms, and t_2 is a sequential tactic.*

Law 5.7 $t_1 \,;\, (t_2 \mid t_3) = (t_1 \,;\, t_2) \mid (t_1 \,;\, t_3)$ for t_1 any *sequential* tactic

[8] Readers unfamiliar with this *point-free* style may prefer to see $(t_1 \mid t_2)g = t_1\, g \frown t_2\, g$, etc.

Soundness

Theorem 5.4.2 *All of these laws are sound with respect to the semantics given in Section 5.3.*

Proof: The proofs of these laws depend on simple properties of lists and functional composition. They are all quite similar. Only a small selection is presented here. The lemmas and properties referred to below are discussed in [Bir88, Bir86].
Law 5.1(a): let g be any expression, then

$$
\begin{array}{ll}
(\mathbf{skip}\ ;\ t)g & \\
= (\frown/ \circ t* \circ \mathbf{skip})g & \text{Definition of ;} \\
= \frown/(t*(\mathbf{skip}g)) & \text{Functional composition} \\
= \frown/(t*\langle g\rangle) & \text{Definition of } \mathbf{skip} \\
= \frown/\langle t\ g\rangle & \text{Definition of } * \\
= t\ g & \text{Property of } \frown/
\end{array}
$$

Law 5.5:

$$
\begin{array}{ll}
(t_1\ ;\ t_2)\ ;\ t_3 & \\
= \frown/ \circ t_3* \circ (t_1\ ;\ t_2) & \text{Definition of ;} \\
= \frown/ \circ t_3* \circ (\frown/ \circ t_2* \circ t_1) & \text{Definition of ;} \\
= \frown/ \circ t_3* \circ \frown/ \circ t_2* \circ t_1 & \text{Property of } \circ \\
= \frown/ \circ \frown/ \circ t_3** \circ t_2* \circ t_1 & \text{Lemma } f* \circ \frown/ = \frown/ \circ f** \\
= \frown/ \circ \frown/ * \circ t_3** \circ t_2* \circ t_1 & \text{Lemma } \frown/ \circ \frown/ = \frown/ \circ \frown/ * \\
= \frown/ \circ (\frown/ \circ t_3* \circ t_2)* \circ t_1 & \text{Property of } \circ \text{ and } * \\
= \frown/ \circ (t_2\ ;\ t_3)* \circ t_1 & \text{Definition of ;} \\
= t_1\ ;\ (t_2\ ;\ t_3) & \text{Definition of ;}
\end{array}
$$

□

Completeness

In this section we prove that the laws listed above are complete for the tactic language presented in Section 5.3 *without the* cut *operator*. In this context, the set of laws can be said to be complete when tactics which are observationally equivalent (i.e. they behave identically on all goals) are provably so (using the laws). The value of such a result is that *all* transformations of tactics (which are sound for *every* set of primitive rules) may be undertaken using the laws given above, without reference to the semantic model.

The ideas are similar to those used in proving the completeness of a proof system for a process algebra, e.g. [Bro83]. The completeness is relative to the rule system over which the tactics are applied, since two rules may be equivalent without there being any tactic equivalence between them. As such, we regard tactics as being syntactic objects with the *names* of the rules they invoke as *free variables*. Demonstrating completeness involves defining a notion of tactic equivalence ($t_1 \equiv t_2$, see Definition 5.4.5) which is independent of the chosen instantiation of rule names for rules.[9]

[9] In order to be fully rigorous, we should distinguish between three different sorts of tactic equality:

The methodology of the proof is to define a normal form for tactics, to show that every tactic can be transformed into a unique normal form using the laws above, and then to show that if two tactics are equivalent then they have the same normal form.

In the following, we shall use the notation

$$\Big|_{i:I} t_i$$

to denote the alternate composition $t_{i_1} \mid \ldots \mid t_{i_n}$, with (for uniqueness of representation) the alternation combinators associated to the right. The index sequence I will be finite; if it is a singleton, then the alternation will be vacuous, and consist merely of one instance of the tactic t_i; if it is empty, the expression will denote **fail**. Similar comments apply to generalized sequential composition:

$$\mathop{;}_{i:I} t_i \quad ,$$

except that an empty sequential composition will denote **skip**.

Definition 5.4.3 **(Cut-free Normal Form)** *Say that a tactic is in* cut-free normal form *if it is of the form*

$$\Big|_{i:I} \Big(\mathop{;}_{j:J_i} \textbf{rule}\, r_j \Big)$$

where the r_j are names of primitive rules.

Lemma 5.4.4 *Any tactic expressed using basic rules, alternation, sequential composition,* **skip** *and* **fail**, *can be transformed into a unique tactic in cut-free normal form using the laws above.*

Proof: Proof is by structural induction over the possible forms of goals. The base cases are **skip**, **fail** and **rule** r, all of which are immediately in normal form.

The first inductive case is $t_1 \mid t_2$. Using the inductive hypothesis, we may assume that t_1 and t_2 are in normal form. If either is **fail**, it can be eliminated by Law 5.2; otherwise the tactic is already in normal form, or can be placed in normal form by the alternation associative law, 5.4.

The second inductive case is $t_1 ; t_2$. Again, by induction, t_1 and t_2 may be assumed to be in normal form. If either is **skip** or **fail**, Laws 5.1 and 5.3 can be used to put the tactic into normal form. Otherwise, Law 5.6 can be used to distribute t_2 onto the alternation components of t_1, and then (the components thus distributed being sequential) Law 5.7 can be used to distribute the sequential components of t_2 onto those of t_1. Finally, the associative laws 5.4 and 5.5 can be used to put the resulting tactic into cut-free normal form. □

In order to demonstrate that the rules used for putting tactics into normal form are *complete*, it is sufficient to show that for each tactic there is exactly one tactic in normal form to which it is semantically equivalent. As mentioned above, for this purpose, we may regard tactics as expressions with *rule names* as free variables. A ground instance of a tactic will have all those rule names bound to actual functions.

syntactic equality of tactics, the equality proved using the laws above, and the equivalence mentioned above—which would accurately be defined as $(\forall \rho \bullet \forall g \bullet [\![t_1]\!] \rho\ g = [\![t_2]\!] \rho\ g)$ (with ρ describing a mapping from rule names to partial functions from goals to goals). We shall flag use of the first by the phrase 'are identical' or 'has the form', the second by '$=$', and the third by '$\equiv$'.

Definition 5.4.5 **(Tactic Equivalence)** *Say that tactics t_1 and t_2 are* equivalent *($t_1 \equiv t_2$) if for all mappings of rule names to rules, for all goals g, $t_1\ g = t_2\ g$.*

Lemma 5.4.6 *If two tactics in normal form are equivalent, then they are identical.*

Proof: This is proved by demonstrating that there can be no point at which the tactics differ. Let the rules which comprise the tactics be $r_1, r_2, \ldots$, and choose an instantiation in which the the expressions (goals) are indexed by sequences of rule numbers, so that $r_j\ g_{\langle\rangle} = g_{\langle j\rangle}$ and $r_j\ g_s = g_{s^\frown\langle j\rangle}$.

In this way, the name of a given expression records the history of rules applied to it: $(\textbf{rule}\ r_{j_0}\ ;\ \textbf{rule}\ r_{j_1}\ ;\ \textbf{rule}\ r_{j_2} \mid \textbf{rule}\ r_{j_3})g_{\langle\rangle} = \langle g_{\langle j_0,j_1,j_2\rangle}, g_{\langle j_3\rangle}\rangle$, etc. A $g_{\langle\rangle}$ can occur in the result *only* if an alternation branch in the tactic is **skip**; the result goal list will be empty only if the tactic is **fail** (in this rule instantiation, no application of primitive rules to goals fails).

Using this rule instantiation, a tactic applied to $g_{\langle\rangle}$ will produce an account of that tactic's normal form (being the alternation of the sequential composition of the rules in the respective goals in the result list), and so the result is immediate. □

Theorem 5.4.7 *Two tactics are equivalent under all rule instantiations iff they have the same normal form.*

Corollary 5.4.8 *The rules given above are complete for the cut-free finite (non-recursive) tactic language.*

Of course, the value of a system without *cut* is questionable: the normal form proof demonstrates that all alternation can be distributed to the outermost level, where most applications of the language will make use only of the first outcome in the sequence of results. Nevertheless, tactics which are not in normal form may make use of the apparent angelic nondeterminism to present reasoning steps in a comprehensible manner. *Cut* gives much more scope for structuring tactics, however, and so the following sections give some laws for *cut*, and then go to some lengths to demonstrate that the larger set of laws is also complete.

5.5 Laws involving Cut

This section presents some laws showing how *cut* interacts with the other tactic combinators. These laws are proven sound in a similar way to those above; only one proof is shown here.

Atomic rules and tactics are unchanged by applications of cut.

Law 5.8 $!\textbf{skip} = \textbf{skip}$

Law 5.9 $!\textbf{fail} = \textbf{fail}$

Law 5.10 $!(\textbf{rule}\ r) = \textbf{rule}\ r$

A cut tactic enables sequential composition to distribute over alternation on the left.

Law 5.11 $!t_1\ ;\ (t_2 \mid t_3) = (!t_1\ ;\ t_2) \mid (!t_1\ ;\ t_3)$

Proof: Let g be any expression, and let $head'(t_1\ g) = \langle h \rangle$ (if $t_1\ g = \langle\,\rangle$, the proof is trivial), then:

$$
\begin{array}{ll}
(!t_1\ ;\ (t_2 \mid t_3))g & \\
= (\frown/ \circ (t_2 \mid t_3)* \ \circ !t_1)g & \text{Definition of ;} \\
= \frown/((t_2 \mid t_3)* \ (head'(t_1\ g))) & \text{Functional composition, and definition of !} \\
= \frown/((t_2 \mid t_3)* \ \langle h \rangle) & \text{Assumption} \\
= \frown/\langle t_2\ h \frown t_3\ h \rangle & \text{Definition of *, and |} \\
= t_2\ h \frown t_3\ h & \text{Property of } \frown/ \\
= (\frown/(t_2* \ \langle h \rangle)) \frown (\frown/(t_3* \ \langle h \rangle)) & \text{Property of } \frown/ \text{ and } * \\
= (\frown/(t_2* \ (head'(t_1\ g)))) \frown (\frown/(t_3* \ (head'(t_1\ g)))) & \text{Assumption} \\
= (\frown/ \circ t_2* \ \circ !t_1)g \frown (\frown/ \circ t_3* \ \circ !t_1)g & \text{Functional composition} \\
= (!t_1\ ;\ t_2)g \frown (!t_1\ ;\ t_3)g & \text{Definition of ;} \\
= ((!t_1\ ;\ t_2) \mid (!t_1\ ;\ t_3))g & \text{Definition of |}
\end{array}
$$

□

Law 5.7 is a special case of this law, thanks to Lemma 5.6.2, below.
Cut partially distributes over sequential composition and over alternation.

Law 5.12 $\quad !t_1\ ;\ !t_2 = !(!t_1\ ;\ !t_2)$

Law 5.13 $\quad !(t_1\ ;\ t_2) = !(t_1\ ;\ !t_2)$

Law 5.14 **(a)** $!(t_1 \mid t_2) = !(!t_1 \mid t_2)$ $\qquad$ **(b)** $!(t_1 \mid t_2) = !(t_1 \mid !t_2)$

Cut also produces two adsorption rules:

Law 5.15 $\quad !(t_1 \mid t_1\ ;\ t_2) = !t_1$

Law 5.16 $\quad !(t_1 \mid t_2 \mid t_1) = !(t_1 \mid t_2)$

Special cases of these are as follows; **skip** becomes a left-zero for alternation, and alternation becomes idempotent:

Law 5.17 $\quad !(\mathbf{skip} \mid t) = \mathbf{skip}$

Law 5.18 $\quad !(t \mid t) = !t$

5.6 Tactic Assertions

In reasoning about tactics, it is helpful to have a formal way to describe the success or failure of a tactic. The tactic *succs* t fails whenever t fails, and behaves like **skip** whenever t succeeds. Conversely, *fails* t behaves like **skip** if t fails, and fails if t succeeds. These tacticals are useful reasoning tools, and they have shown themselves to be useful in writing real tactics, too (see, for example, Section 6.1).

$$
\begin{array}{l}
t\ g = \langle\,\rangle \Rightarrow \\
\quad (fails\ t\ g = \mathbf{skip}\ g \ \wedge \ succs\ t\ g = \mathbf{fail}\ g) \\
t\ g = \langle h_1, \ldots h_n \rangle \Rightarrow \\
\quad (fails\ t\ g = \mathbf{fail}\ g \ \wedge \ succs\ t\ g = \mathbf{skip}\ g)
\end{array}
$$

These tactics play a similar rôle to that of *assertions* in other languages. However, as *tactics* (rather than *predicates*) they are more readily usable in reasoning, and can be manipulated directly using the algebraic laws applicable to all tactics.

Laws

The interaction of *fails* and *succs* with each other and with the other basic tactics gives rise to a large set of laws. There is a certain duality between *fails* and *succs* which means that by giving primitive laws for *fails*, corresponding laws for *succs* can be proven. Those which follow may be taken as primitive; a list of laws derived from these is presented in Section 5.13.

Law 5.19 $succs\ t\ ;t = t$

Law 5.20 * $fails\ t\ ;t = \mathbf{fail}$

Law 5.21 $fails\ t = fails\ !t = !fails\ t$

Law 5.22 $fails(succs\ t) = fails\ t$

Law 5.23 * $fails\ t_1\ ;fails\ t_2 = fails\ t_2\ ;fails\ t_1$

Law 5.24 $!(t_1 \mid t_2) = !t_1 \mid (fails\ t_1\ ;!t_2)$

Law 5.25 $!(t_1\ ;t_2) = !(t_1\ ;succs\ t_2)\ ;!t_2$

Law 5.26 $succs(t \mid u) = !(succs\ t \mid succs\ u)$

Law 5.27 $fails(t \mid u) = fails\ t\ ;fails\ u$

Law 5.28 $succs\ s\ ;succs(s\ ;t) = succs(s\ ;t)$

Law 5.29 $fails\ s = fails\ s\ ;fails(s\ ;t)$

Law 5.30 $!s\ ;fails\ t = fails(!s\ ;t)\ ;!s$

Law 5.31 $fails(fails\ s\ ;t) = succs\ s \mid fails\ s\ ;fails\ t$

Law 5.32 $fails(s\ ;fails\ t) = fails\ s \mid succs(s\ ;t)$

Derived Laws

A number of valuable laws can be proved from those above. These include:

Law 5.33 $!!t = !t$

Law 5.34 * $fails\ s\ ;succs(s\ ;t) = \mathbf{fail}$

Law 5.35 $fails(t\ ;d) = fails(t\ ;succs\ d)$

Law 5.36 $!s\ ;succs\ t = succs(!s\ ;t)\ ;!s$

Law 5.37 $fails(fails\ t) = succs\ t$

Law 5.38 * $\mathit{fails}\ t_1\ ;\ \mathit{succs}\ t_2 = \mathit{succs}\ t_2\ ;\ \mathit{fails}\ t_1$

Law 5.39 * $\mathit{succs}\ t_1\ ;\ \mathit{succs}\ t_2 = \mathit{succs}\ t_2\ ;\ \mathit{succs}\ t_1$

Law 5.40 $\mathit{succs}(\mathit{succs}\ t) = \mathit{succs}\ t$

Law 5.41 $\mathit{succs}(\mathit{fails}\ t) = \mathit{fails}\ t$

Law 5.42 $\mathit{succs}(t\ ;\ d) = \mathit{succs}(t\ ;\ \mathit{succs}\ d)$

Law 5.43 $\mathit{succs}\ t = \mathit{succs}\ !t = !\mathit{succs}\ t$

Law 5.44 $\mathit{succs}\ \mathbf{skip} = \mathbf{skip}$

Law 5.45 $\mathit{succs}\ \mathbf{fail} = \mathbf{fail}$

Law 5.46 $\mathit{fails}\ \mathbf{skip} = \mathbf{fail}$

Law 5.47 $\mathit{fails}\ \mathbf{fail} = \mathbf{skip}$

Law 5.48 $\mathit{fails}\ t\ ;\ \mathit{fails}\ t = \mathit{fails}\ t$

Law 5.49 $\mathit{succs}\ t\ ;\ \mathit{succs}\ t = \mathit{succs}\ t$

Law 5.50 * $\mathit{succs}\ t\ ;\ \mathit{fails}\ t = \mathit{fails}\ t\ ;\ \mathit{succs}\ t = \mathbf{fail}$

Law 5.51 * $\mathit{fails}\ t \mid \mathit{succs}\ t = \mathit{succs}\ t \mid \mathit{fails}\ t = \mathbf{skip}$

Law 5.52 $\mathit{succs}\ (t \mid u) = \mathit{succs}\ t \mid \mathit{fails}\ t\ ;\ \mathit{succs}\ u$

Law 5.53 $\mathit{succs}\ (\mathit{fails}\ s\ ;\ t) = \mathit{fails}\ s\ ;\ \mathit{succs}\ t$

Law 5.54 $\mathit{succs}\ (\mathit{succs}\ s\ ;\ t) = \mathit{succs}\ s\ ;\ \mathit{succs}\ t$

Law 5.55 $\mathit{succs}\ (s\ ;\ \mathit{fails}\ t) = \mathit{succs}\ s\ ;\ \mathit{fails}(s\ ;\ t)$

Law 5.56 $\mathit{succs}\ (s\ ;\ \mathit{succs}\ t) = \mathit{succs}\ s\ ;\ \mathit{succs}(s\ ;\ t)$

Law 5.57 $\mathit{fails}\ (\mathit{succs}\ s\ ;\ t) = \mathit{fails}\ s \mid \mathit{succs}\ s\ ;\ \mathit{fails}\ t$

Law 5.58 $\mathit{fails}\ (s\ ;\ \mathit{succs}\ t) = \mathit{fails}\ s \mid \mathit{succs}\ s\ ;\ \mathit{fails}(s\ ;\ t)$

Sequential Tactics

For this larger language, we may extend the earlier definition of *sequential* tactics:

Definition 5.6.1 **(Sequential Tactics)** *A tactic is* sequential *if it is* **skip**, **fail**, **rule** *r for some basic rule r,* $!t$, *fails t or succs t, for some tactic t, or it has the form* $t_1\ ;\ t_2$, *where* t_1 *is in one of these forms, and* t_2 *is a sequential tactic.*

Lemma 5.6.2 *Whenever a tactic t is sequential, we may show that* $t = !t$, *using the laws above.*

Proof: by structural induction, using Definition 5.6.1 above.

Case: **skip** = !**skip** by Law 5.8.

Case: **fail** = !**fail** by Law 5.9.

Case: **rule** r = !(**rule** r) by Law 5.10.

Case: $!t$

$$\begin{aligned} & !t \\ &= !(t \mid \mathbf{fail}) && \text{Law 5.2} \\ &= !(!t \mid \mathbf{fail}) && \text{Law 5.14} \\ &= !(!t) && \text{Law 5.2} \end{aligned}$$

Case: *fails* t = !*fails* t, by Law 5.21

Case: *succs* t = !*succs* t, by Law 5.43

Case: $t_1 \,;\, t_2$

$$\begin{aligned} & t_1 \,;\, t_2 \\ &= !t_1 \,;\, !t_2 && \text{Inductive Hypothesis} \\ &= !(!t_1 \,;\, !t_2) && \text{Law 5.12} \\ &= !(t_1 \,;\, t_2) && \text{Inductive Hypothesis} \end{aligned}$$

□

5.7 Full Completeness

Having defined *succs* and *fails*, it is now possible to prove a completeness result for the whole finite (non-recursive) language, including the cut operator:

$$\begin{aligned} T ::= \; & \mathbf{rule}\, R \\ \mid \; & \mathbf{skip} \\ \mid \; & \mathbf{fail} \\ \mid \; & T \,;\, T \\ \mid \; & T \mid T \\ \mid \; & !\, T \\ \mid \; & \mathit{succs}\, T \\ \mid \; & \mathit{fails}\, T \end{aligned} \quad ,$$

where R is some basic set of rules which transform one expression into another. The proof proceeds in much the same way as that in Section 5.4, but the conversion to normal form is now a two-stage process.

Definition 5.7.1 **(Pre-Normal Form)** *A tactic is in* pre-normal form *if it has the form*

$$\bigmid_{j:J} g_j \,;\, \left(\underset{i:I_j}{;} \mathbf{rule}\, r_i \right) \quad ,$$

where the g_j *are* guards *of the form succs* $(;_k\, r_k)$ *or fails* $(;_k\, r_k)$*, or (possibly empty) sequential compositions of such guards, and the* r_i *are instances of basic rules.*

Lemma 5.7.2 *Any tactic written in the language above can be put into pre-normal form using the laws given above.*

Proof: The proof proceeds like that for cut-free normal form, above, by structural induction. The base cases are, again, trivial; the case for $t_1 \mid t_2$ is as before (i.e. using Law 5.2 where necessary).[10]

In the sequential composition case, $t_1 \; ; \; t_2$, we may assume (using the inductive hypothesis) that the sequentially composed tactics are in pre-normal form. If either is **skip** or **fail**, Laws 5.1 and 5.3 can be used to put the composition into pre-normal form. Otherwise, Law 5.6 can be used to distribute t_2 onto the alternation components of t_1, and then, since the components of t_1 are sequential, they are equivalent to their cut forms (Lemma 5.6.2), and so Law 5.11 can be used to distribute the separate components of t_2 onto those of t_1.

This procedure will place the tactic in the form

$$\Big|_{i:I} (g_{1_i} \; ; \; t_{1_i} \; ; \; g_{2_i} \; ; \; t_{2_i})$$

Since the t_{1_i} are sequences of rules, we have that $!t_{1_i} = t_{1_i}$ (by applying Lemma 5.6.2), and so Laws 5.30 and 5.36 can be used to assemble the guard components at the beginning of each alternation branch (and the *cuts* can be removed, by applying the same lemma again). The resulting tactic will be in pre-normal form.

Additional inductive cases are needed: The tactic $!t$ is normalised via use of Law 5.24. Since t is in pre-normal form (by the inductive hypothesis), repeated use of this law will distribute the *cuts* onto the sequential components, from where they can be removed (Lemma 5.6.2). This distribution may result in the creation of nested instances of *succs* and *fails*. Laws 5.31, 5.32, and 5.53–5.58, can be used to remove those nested instances. Finally, any alternations introduced by these laws may be moved to the outermost level using the distributive laws (5.6 and 5.11, applying Lemma 5.6.2). The resulting tactic will be in pre-normal form.

For the case *fails* t, the tactic t may be assumed to be in pre-normal form, and so if it is an alternation, Law 5.27 can be used to distribute the *fails* through the alternation. Laws 5.31 and 5.32 (and laws derived from them, 5.53–5.58) can be used to remove nested instances of *succs* and *fails*. Doing so may introduce alternations, as above, and so distributive laws can be used to transform the resulting tactics into pre-normal form.

Similar arguments apply to *succs* t; normalization begins with Law 5.52, and then proceeds like that for *fails*. □

Pre-normal form is insufficient for proving completeness since it does not guarantee uniqueness; the following tactics are all in pre-normal form (provided s and t are sequences of rules), and all equivalent:

$$s \; ; \; t$$
$$\mathit{succs}\; s \; ; \; s \; ; \; t$$
$$\mathit{succs}\; (s \; ; \; t) \; ; \; s \; ; \; t$$
$$\mathit{succs}\; s \; ; \; \mathit{succs}\; (s \; ; \; t) \; ; \; s \; ; \; t$$

etc. More crucially, where alternation and *fails* are involved, it becomes impossible simply to 'complete' the guard (as in the last of the examples above). Some alternation branches would be mutually exclusive and could therefore be reordered; others would not, and their order would be important.

[10] Use of the associative laws (5.4 and 5.5) will be assumed, where necessary, throughout the following proof.

Instead, then, a tactic in normal form will be one which is factored into a number of tactics v_j in *cut-free* normal form, each one guarded (by a guard g_j) in such a way that the success of the guard is sufficient to guarantee the success of every alternative branch of the tactic it guards, with the guards being mutually exclusive. Each guard must be 'maximal' in that it must contain a *succs* or *fails* for each sequence of rule applications that might arise in any of the v_j.

The definition of general normal form, and the lemmas which follow, will be relative to some set of *rule sequences*. Rule sequences will be sequences of the form **rule** r_1 ; **rule** r_2 ; **rule** r_3, etc. A set of such sequences T will be *prefix closed* if for any t in T, T also contains all initial subsequences of t (i.e. for a set containing the rule sequence above to be prefix closed, it would also need to contain **rule** r_1 ; **rule** r_2 and the atomic tactic **rule** r_1).

For a given tactic t, in order for the normal form to be well-defined, it must be calculated relative to some *sufficiently large* set of rule sequences. 'Sufficiently large' means that it must at least contain the minimal set of rule sequences determined by consideration of the tactic t in pre-normal form (using notation from Definition 5.7.1): that set must contain all the instances of rule sequences $(;_{i:I_i}$ **rule** $r_i)$, and all the rule sequences occurring (preceded by *succs* or *fails*) in the guards g_i, and must be prefix-closed.

Definition 5.7.3 **(General Normal Form)** *A tactic is in* general normal form *relative to a set of rule sequences T, if it has the form*

$$\Big|_{j:J} g_j ; v_j \quad ,$$

where the v_j are tactics in cut-free normal form (and are not **fail***), and the g_j are guards as above, with certain provisos:*

a. *(consistency) for each guard g_j, if for some rule sequence t, g_j contains succs t, it must not contain fails s, for s any prefix of t (or t itself);*

b. *(maximality) for each j, for all t in T, either succs t or fails t must be present in g_j;*

c. *(sufficiency) for each j, the success of g_j must be sufficient to guarantee that of all the alternate clauses in v_j, i.e., if $v_j = \Big|_k v_{j_k}$, then for all k, succs v_{j_k} must be present in g_j;*

d. *(mutual exclusivity) the guards are mutually exclusive; that is, for i and j, with $i \neq j$, there must be some rule sequence t for which g_i contains succs t and g_j contains fails t (or vice versa).*

If the conditions in the definition above are met, the following properties may be proved using the laws given previously:

$$g_i \neq \textbf{fail}$$
$$succs\ t\ ;\ g_j = g_j \qquad or \qquad fails\ t\ ;\ g_j = g_j$$
$$succs\ g_j = succs\ g_j\ ;\ succs\ v_{j_k}$$
$$g_i\ ;\ g_j = \textbf{fail} \qquad (i \neq j)$$

Due to Laws 5.23, 5.39, and 5.38, we may observe that the components of the guards in a tactic in normal form can be arbitrarily re-ordered. Moreover, this property, together with the mutual exclusion property, means that the outermost alternation can be re-ordered, too: an arbitrary pair of adjacent alternation branches of a tactic in general normal form can be considered, due to the foregoing remarks, to have the form $(succs\ t\ ;\ s_1\ |\ fails\ t\ ;\ s_2)$;

$$
\begin{aligned}
& succs\ t\ ;\ s_1\ |\ fails\ t\ ;\ s_2 \\
&= (fails\ t\ |\ succs\ t)\ ;\ (succs\ t\ ;\ s_1\ |\ fails\ t\ ;\ s_2) && \text{Laws 5.51 and 5.1} \\
&= fails\ t\ ;\ succs\ t\ ;\ s_1 && \text{Laws 5.6 and 5.11, applying Lemma 5.6.2} \\
&\quad |fails\ t\ ;\ fails\ t\ ;\ s_2 \\
&\quad |succs\ t\ ;\ succs\ t\ ;\ s_1 \\
&\quad |succs\ t\ ;\ fails\ t\ ;\ s_2 \\
&= fails\ t\ ;\ s_2\ |\ succs\ t\ ;\ s_1 && \text{Laws 5.50, 5.2, 5.48, and 5.49}
\end{aligned}
$$

Since arbitrary branches may have their order swapped, the ordering of the tactic as a whole may be changed arbitrarily. This means that the normal forms achieved below will be unique only modulo these two forms of reordering.

Lemma 5.7.4 *Given a sufficiently large prefix-closed set of rule sequences, T, any tactic in pre-normal form can be put into a unique normal form relative to T (unique modulo reordering of the guards), using laws drawn from the set given above.*

Proof: Consider the tactic formed from all possible guards for rule sequences in T:

$$\mathop{;}_{r:T} (succs\ t\ |\ fails\ t)\ .$$

This is equivalent to **skip**. The distributive laws (5.6 and 5.11, applying Lemma 5.6.2) can be used to 'multiply out' this expression into the form

$$\mathop{|}_{i:I} g_i \quad ,$$

(this still being equal to **skip**). From I we may remove all those i for which g_i is equivalent to **fail**, via Law 5.34 and the commutative laws (5.23, 5.39, and 5.38), to give I'. These guards then have the mutual exclusion property mentioned in the definition of general normal form. They have the consistency property, since those which are equivalent to **fail** have been removed. They also have the maximality property, in that for each t in T, for each i, either g_i contains *fails* t or it contains *succs* t.

If we sequentially compose this tactic with the tactic in pre-normal form:

$$\left(\mathop{|}_{i:I'} g_i\right)\ ;\ \left(\mathop{|}_{j:J} h_j\ ;\ \left(\mathop{;}_{k:K_j} r_k\right)\right)$$

and again use the distributive laws, we obtain

$$\mathop{|}_{i:I'}\ \mathop{|}_{j:J} g_i\ ;\ h_j\ ;\ \left(\mathop{;}_{k:K_j} r_k\right)\ ,$$

this tactic remaining equivalent to the original tactic in pre-normal form.

By the maximality property (and the sufficient size of T), for each h_j, g_i pair, we have either that each component of h_j is present in g_i, and so by the commutativity laws (as above) and Laws 5.48 and 5.49, $g_i ; h_j = g_i$, or that g_j has a *succs* t for which g_i has a *fails* t (or vice versa), and so by the commutativity laws and Law 5.34, we have $g_i ; h_j =$ **fail**.

Therefore, the tactic above may be rewritten without the h_js, and with some if the js omitted (so J_i' replaces J):

$$\left|_{i:I'} \left|_{j:J_i'} g_i ; \left(\,;_{k:K_j} r_k\right)\right.\right. .$$

Now, by the commutative laws for *succs* and *fails* (as above), and Laws 5.20 and 5.2, those inner alternation branches whose guards which contain an instance of $fails(;_k r_k)$ can also be omitted (so J_i' becomes J_i''). In the event that J_i'' becomes empty following these changes (thus the alternation over j simply denotes **fail**), Law 5.2 can be used to omit this i from I'.

Finally, the (left) distributive law can be used to transform the tactic into the required normal form:

$$\left|_{i:I'} \left(g_i ; \left|_{j:J_i''} \left(\,;_{k:K_j} r_k\right)\right.\right)\right. .$$

This is in normal form, since the g_i remain maximal, consistent, and mutually exclusive. Their sufficiency arises as a result of the maximality and the omission (in the last step, above) of clauses which must fail. □

Lemma 5.7.5 *Two tactics in general normal form (relative to some sufficiently large prefix-closed set of rule sequences, T) are equivalent iff they are identical modulo reordering of the guards.*

Proof: The proof proceeds like that of the corresponding lemma above (5.4.6). The 'if' part is guaranteed by the soundness of the laws which make reordering possible; for the 'only if' part, we produce a rule/goal model in which the semantic behaviour of a tactic can be used to reconstruct its normal form.

Let the rules referred-to in the tactics be $r_1, \ldots, r_n$, and let the goals be decorated with pairs: a prefix-closed set of rule sequences, and a single rule sequence. The former will be those sequences of rules (drawn from T) which can succeed when applied to the indicated goal; the latter a trace of rules which have already been successfully applied.

That is, $g_{s,t} \in \mathrm{dom}\, r_i \Leftrightarrow \exists xs : s \mid head\ xs = r_i$. In this case, $r_i\, g_{s,t} = g_{s',t^\frown\langle r_i\rangle}$, where $s' = \{\, xs \mid r_i ^\frown xs \in s \,\}$; otherwise the rule application fails.[11]

By considering the behaviour of a tactic applied to various goals, it is possible to determine the guard/body[12] pairs of its normal form. The application of a tactic t to a goal $g_{s,\langle\,\rangle}$, will either result in the empty sequence as output, or a sequence of the form $\langle g_{s_1,t_1}, \ldots, g_{s_n,t_n}\rangle$. The former case will arise either because s fails to be prefix closed (corresponding to lacking the 'consistency' property of the normal form), or because it corresponds to one of those guards which was omitted above because it was

[11] This is why the definition of normal form requires that if the guards are successfully executed, the rules must not fail.

[12] i.e. cut-free normal form

guarding a tactic equivalent to **fail**. If a non-empty sequence is produced, the body part of an alternation in the tactic can be reconstructed as the alternation of $t_1 \ldots t_n$, and the guard for that branch from *succs* applied to each member of s and *fails* applied to each member of the compliment of s in T.

By considering all such initial goals $g_{s,\langle\rangle}$ (with s as any prefix-closed subset of T) for which the outcome is not the empty sequence (i.e. failure), it is possible to reconstruct the whole tactic in normal form.

The maximality is guaranteed by ensuring that every member of T is present in each guard—with either *succs* or *fails*—and the mutual exclusivity by the fact that the sets of sequences with *succs* are different in each guard. The sufficiency and consistency of the guard is ensured by the fact that it gives rise to an outcome which is not failure.

In this way, a tactic in normal form is completely characterized (modulo reordering) by the set of goals on which it succeeds and the outcomes when it does so, and the result follows immediately. □

Theorem 5.7.6 *Two tactics are equivalent under all rule instantiations iff they have the same general normal form, modulo reordering of the guards.*

Corollary 5.7.7 *The set of laws is complete for the language described above.*

5.8 Semantic Model incorporating Recursion

In this section we extend the semantic model to cover recursive tactics. To do this, we shall need a model which uses (potentially) infinite lists, in place of the finite lists used above.

Infinite Lists

The style of infinite lists which we shall use is that found in many treatments of functional programming with lazy evaluation. A suitable model for such lists will be found in [Mar93b]; the definitions remain consistent with those in Appendix A.1.

The datatype of infinite lists differs from that used above by the incorporation of *partial* and *infinite* lists. The set of lists over a set A, (denoted $\mathrm{seq}A$, as above) is augmented by the addition of an extra element, $\bot_A$. A partial order is then defined over this set of lists. $\bot_A$ is the least-defined element in the set. It is a partial list, as is any list which ends with $\bot_A$. One list is less than another (denoted by $s_1 \sqsubseteq_\infty s_2$) whenever they are equal, or the first is a partial list which forms an initial subsequence of the second. Formally, an infinite list is a *limit* of a suitable (i.e. directed) set of partial lists (the limit of the set of lists S is denoted $\bigsqcup_\infty S$).

We shall retain the same notation for list concatenation as was used on finite lists, and so the tactic definitions previously given will still apply. However, note that whenever s_1 is a partial or infinite list, then $s_1 \frown s_2 = s_1$ (for any s_2). Also, $\frown/$ is (necessarily) ill-behaved on certain infinite lists. A pathological (but important) case is

$$\frown/\langle(),(),\ldots\rangle = \bot_A \quad .$$

As tactics are defined as functions from goals to lists of goals, it is a standard construction to extend the ordering on lists of goals to be an ordering on tactics. This

is accomplished in a pointwise manner. A suitable treatment can be found in [LS87]. The relevant definitions are as follows:

$$t_1 \sqsubseteq_T t_2 \Leftrightarrow (\forall g : G \bullet t_1\ g \sqsubseteq_\infty t_2\ g)$$

$$(\bigsqcup_T s)g = \bigsqcup_\infty \{\, t : s \bullet t\ g \,\} \ .$$

Semantics

The atomic tactic **abort** is simply defined; it maps any goal to the undefined list:

$$\mathbf{abort}\ g = \bot_G \ .$$

The recursion operator is defined as a least fixed point. For f a function from tactics to tactics, we have that

$$(\mu X \bullet f(X)) = \bigsqcup_T \{\, i : \mathrm{N} \bullet f^i(\mathbf{abort}) \,\} \ .$$

Again, this is a standard construction, covered, for example in [LS87]. It requires that the tacticals used to define f are *continuous*—and those defined above can be shown to have this property. It is worth noting that this least fixed point is $\bot$ only in the case that for all i, we have that $f^i(\mathbf{abort}) = \mathbf{abort}$, i.e. exactly when $f(\mathbf{abort}) = \mathbf{abort}$.

Laws

The laws given in Section 5.4 also hold in the presence of recursive tactics, with the exception of those marked with (*). **abort** is catastrophic; sequential and alternate composition are strict in their left-hand arguments.

Law 5.59 $\mathbf{abort}\ ; t = \mathbf{abort}$

Law 5.60 $\mathbf{abort} \mid t = \mathbf{abort}$

Law 5.61 $!\,\mathbf{abort} = \mathbf{abort}$

The composition operators are not strict in their right-hand arguments (indeed, if t is not **abort** and does not always succeed, we have $\mathbf{abort} \sqsubset_T t$; **abort**, etc.), since the tactic which precedes **abort** may mask its action (i.e. in the sequential case, t may fail, and **fail** ; **abort** = **fail**).

Since the tacticals presented above (and below) are continuous with respect to the $\sqsubseteq_T$, we may use Park's Theorem [Par69] to deduce properties of fixed points.

Theorem 5.8.1 **(Park)** *For any tactic Q, and continuous function from tactics to tactics, F:*

$$\frac{F(Q) \sqsubseteq_T Q}{(\mu X \bullet F(X)) \sqsubseteq_T Q} \ .$$

In order to demonstrate equality of recursively-defined tactics, it suffices to show refinement in each direction separately. This theorem allows that refinement to be demonstrated by showing that each tactic satisfies the other's recursive equation.

For tactics which are known to terminate, and satisfy the same recursive equation, however, a simpler proof of equality is possible. In this model, a tactic will be said to terminate when applied to a given goal if it produces a finite (non-partial, non-infinite) list of subgoals.

Theorem 5.8.2 *Let X and Y be arbitrary tactics such that $Y = (\mu Z \bullet F(Z))$ and $X = F(X)$. (For some continuous F). If Y is known to terminate on all inputs then $X = Y$.*

Proof: In order to show that $X = Y$, it is sufficient, by extension, to show that, for an arbitrary goal g, $X\ g = Y\ g$. Park's theorem gives us that $Y \sqsubseteq_T X$; therefore $Y\ g \sqsubseteq_\infty X\ g$. Now,

$$\begin{aligned} & Y\ g \\ &= \bigsqcup_T \{\ i : \mathbf{N} \bullet f^i(\mathbf{abort})\ \} g && \text{Definition of } \mu \\ &= \bigsqcup_\infty \{\ i : \mathbf{N} \bullet f^i(\mathbf{abort})\ g\} && \text{Definition of } \bigsqcup_T \end{aligned}$$

Since we know that $Y\ g$ terminates, we know that there is some j such that $\bigsqcup_\infty \{\ i : \mathbf{N} \bullet f^i(\mathbf{abort})\ g\} = f^j(\mathbf{abort})\ g$, *and* that this is a finite (non-partial) list. Recalling that $\sqsubseteq_\infty$ is a strict inequality iff the expression on the left-hand side is partial, we deduce that $Y\ g = X\ g$. □

Infinite Alternation

One further extension to the tactic language which has proved useful in establishing properties of recursive tactics (see Lemma 5.9.1 below) is a generalization of the alternation operator of Section 5.3. We define an infinite alternation, such that

Law 5.62 $$\bigdoublebar_{i=n}^{\infty} f(i) = f(n) \Big| \Big(\bigdoublebar_{i=n+1}^{\infty} f(i) \Big)$$

holds, by using a vector of recursive tactics (that is, by taking a fixpoint in the function space $\mathbf{N} \to T$).

$$\bigdoublebar_{i=0}^{\infty} f(i) = \mu X \bullet F(X_0)$$
$$\text{where } F(X_i) = f(i) \mid F(X_{i+1})$$

Laws

Infinite alternation has many of the same properties as ordinary (finite) alternation:

Law 5.63 $$\bigdoublebar_{i=0}^{\infty} (f(i)\ ;\ d) = \Big(\bigdoublebar_{i=0}^{\infty} f(i) \Big)\ ;\ d$$

Law 5.64 $$!\Big(\bigdoublebar_{i=0}^{\infty} f(i) \Big) = !\Big(\bigdoublebar_{i=0}^{\infty} !f(i) \Big)$$

Law 5.65 $$\bigdoublebar_{i=0}^{\infty} \mathbf{fail} = \mathbf{abort}$$

Law 5.66 $$t\ ;\ \bigdoublebar_{i=0}^{\infty} f(i) = \bigdoublebar_{i=0}^{\infty} t\ ;f(i) \text{ provided } t = !t$$

Interaction with *fails* and *succs*

For tactics which are not known to terminate, *succs* and *fails* are of limited usefulness. For tactics which certainly do not terminate,

$$t\,g = \bot_G \Rightarrow \\ (\mathit{fails}\ t\,g = \mathbf{abort}\,g \ \wedge\ \mathit{succs}\ t\,g = \mathbf{abort}\,g)\ .$$

In other cases, typically we might be able to say

$$\mathbf{abort} \sqsubseteq_T \mathit{succs}\ t \sqsubseteq_T \mathbf{skip} \qquad \text{or} \qquad \mathbf{abort} \sqsubseteq_T \mathit{succs}\ t \sqsubseteq_T \mathbf{fail}\ ,$$

and similarly for *fails*.

In each case, the left-hand inequalities will be strict if the tactic t is known to produce *some* output (in which case, *succs* t will satisfy the left-hand equation, and not the right-hand one); if the tactic is known to terminate, the right-hand inequality will be an equality.

The combinators *succs* and *fails* are strict:

Law 5.67 $\quad \mathit{fails}\ \mathbf{abort} = \mathbf{abort}$

Law 5.68 $\quad \mathit{succs}\ \mathbf{abort} = \mathbf{abort}$

The following refinement result is of interest; by symmetry, the refinements in the rule and the proviso may be replaced by equalities.

Law 5.69 $\quad t_1 \mid t_2 \sqsubseteq_T t_2 \mid t_1 \qquad$ provided $\mathit{succs}\ t_1 \sqsubseteq_T \mathit{fails}\ t_2$

5.9 Derived Tacticals

In the many treatments of tactics and tacticals (see Section 8.2), a few derived tacticals recur frequently. These have shown themselves to be particularly valuable for describing tactic programs: operators for iteration (*repeat*), robust application (*try*) and exhaustive application (*exhaust*). We define these here, and describe some of their properties.

Definitions

Most tactic languages include a definition of a *repeat* tactical (sometimes called *iterate*). *repeat*(n, t) will run tactic t the number of times specified by n. We denote this more succinctly with a notation suggestive of iteration.

$$\begin{aligned} t^0 &= \mathbf{skip} \\ t^{n+1} &= t\,;t^n \end{aligned}$$

The limiting case (repeat t indefinitely) is not very useful:

$$\begin{aligned} t^\omega &= t\,;t^\omega \\ &= \mu X \bullet t\,;X\ . \end{aligned}$$

This tactic is either **abort** or **fail**, depending on whether t always succeeds, or eventually fails. Instead, a tactic which applies t as many times as possible, terminating (with

success) when t fails to apply, is defined. The tactical which does this is called *exhaust* (some authors call it *repeat*).

$$exhaust\ t = (\mu\, Y \bullet (t\ ;\ Y \mid \mathbf{skip}))$$

A property of *exhaust* is $exhaust\ t = t\ ;\ exhaust\ t \mid \mathbf{skip}$, and this will generally be used as its definition. This tactic is able to backtrack, both in the number of iterations, and in the evaluation of t. Sometimes this gives more freedom than is useful, and so $!(exhaust\ t)$ will often be used—Law 5.82 gives a useful alternative formulation of this tactic.

It is often useful to be able to try to apply a tactic, but to succeed whether or not the tactic applies. This is accomplished by the derived tactical *try*.

$$try\ t = !(t \mid \mathbf{skip})$$

Laws about Iteration

Iteration has some obvious properties. These can be proved by induction using the laws given above.

Law 5.70 $t^n\ ;\ t^m = t^{n+m}$

Law 5.71 $succs\ t^n\ ;\ t^{n+1} = t^{n+1}$

Law 5.72 * $fails\ t^n\ ;\ t^{n+1} = \mathbf{fail}$

Law 5.73 $fails\ t^{n+1}\ ;\ t^n = t^n\ ;\ fails\ t$

Law 5.74 $fails\ t^n = fails\ t^n\ ;\ fails\ t^{n+1}$

Law 5.75 $succs\ t^n = succs\ t^{n-1}\ ;\ succs\ t^n$

Law 5.76 $fails\ t^n = fails\ t^{n-1} \mid (succs\ t^{n-1}\ ;\ fails\ t^n)$

Law 5.77 $fails\ t^n = \mathbf{skip} \Rightarrow fails\ t^{n+1} = \mathbf{skip}$

Law 5.78 $!t = t \ \Rightarrow\ !t^n = t^n$

Laws about *try*

There are few obvious useful laws about *try*.

Law 5.79 $!(try\ t) = try\ t$

Law 5.80 $exhaust\ t\ ;\ try\ t = exhaust\ t$

Law 5.81 $try(try\ t) = try\ t$

Laws about *exhaust*

The following laws about *exhaust* help to characterize its behaviour. Those laws marked with (†) apply only when *exhaust t* must terminate, i.e. when t is not **abort** and does not always succeed—otherwise they are refinements rather than equalities. The proofs make reference to some derived laws which are listed in Section 5.13; laws which can be proved using the primitive laws given in Sections 5.4 and 5.6).

Law 5.82 † $\quad !(\mathit{exhaust}\ t) = (\mu X \bullet !(t\ ;\ X \mid \mathbf{skip}))$

Proof:

$$
\begin{array}{ll}
& !(\mathit{exhaust}\ t) \\
= & !(t\ ;\ (\mathit{exhaust}\ t) \mid \mathbf{skip}) & \text{Property of } \mathit{exhaust} \\
= & !(!(t\ ;\ (\mathit{exhaust}\ t)) \mid \mathbf{skip}) & \text{Law 5.14} \\
= & !(!(t\ ;\ !(\mathit{exhaust}\ t)) \mid \mathbf{skip}) & \text{Law 5.13} \\
= & !(t\ ;\ !(\mathit{exhaust}\ t) \mid \mathbf{skip}) & \text{Law 5.14}
\end{array}
$$

Hence, by Theorem 5.8.2, $!(\mathit{exhaust}\ t) = (\mu X \bullet !(t\ ;\ X \mid \mathbf{skip}))$. □

Law 5.83 $\quad \mathit{succs}(\mathit{exhaust}\ t) \sqsubseteq_T \mathbf{skip}$

Proof:

$$
\begin{array}{ll}
& \mathit{succs}(\mathit{exhaust}\ t) \\
= & \mathit{succs}(t\ ;\ \mathit{exhaust}\ t \mid \mathbf{skip}) & \text{Definition of } \mathit{exhaust} \\
= & !(\mathit{succs}(t\ ;\ \mathit{exhaust}\ t) \mid (\mathit{succs}\mathbf{skip})) & \text{Law 5.26} \\
= & !(\mathit{succs}(t\ ;\ \mathit{exhaust}\ t) \mid \mathbf{skip}) & \text{Law 5.44} \\
= & (\mathit{succs}(t\ ;\ \mathit{exhaust}\ t) \mid \mathit{fails}(t\ ;\ \mathit{exhaust}\ t)) & \text{Laws 5.24, 5.1, and 5.22} \\
\sqsubseteq_T & \mathbf{skip} & \text{Law 5.113}
\end{array}
$$

□

Like Law 5.113, this law can be strengthened to equality in the case that t;*exhaust t* is guaranteed to terminate.

Similar arguments give that $\bigsqcap_{i=0}^{\infty}(t^i\ ;\ \mathit{fails}\ t) \sqsubseteq_T \mathbf{skip}$; a fact which will be used below.

Proof of the remaining laws (see below) is made possible by the following lemma.

Lemma 5.9.1 *For any sequential tactic t,*

$$!(\mathit{exhaust}\ t) = \bigsqcap_{i=0}^{\infty} (t^i\ ;\ \mathit{fails}\ t)\ .$$

Proof: It suffices to show refinement in each direction separately.
(a) $!(\mathit{exhaust}\ t) \sqsubseteq_T \bigsqcap_{i=0}^{\infty}(t^i\ ;\ \mathit{fails}\ t)$

Refinement in this direction is demonstrated by showing that the infinite choice satisfies the recursive equation for $!(\mathit{exhaust}\ t)$ (Law 5.82), i.e. that $!(t\ ;\ \bigsqcap_{i=0}^{\infty}(t^i\ ;\ \mathit{fails}\ t) \mid$

skip) $\sqsubseteq_T \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.$.

$$
\begin{array}{lll}
& !(t ; \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right. \mid \mathbf{skip}) & \\
= & !\big(\mathit{fails}\big(t ; \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) ; \mathbf{skip} \mid t ; \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) & \text{Law 5.121} \\
= & !\big(\mathit{fails}\big(t ; \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) \mid t ; \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) & \text{Law 5.1} \\
= & !\big(\mathit{fails}\big(t ; \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) \mid \left|_{i=0}^{\infty}(t^{i+1} ; \mathit{fails}\ t)\right.\big) & \text{Law 5.66} \\
= & !\big(\mathit{fails}(t ; \mathit{succs}(\ \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.)) \mid \left|_{i=1}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) & \text{Law 5.35} \\
\sqsubseteq_T & !\big(\mathit{fails}\ t ; \mathbf{skip} \mid \left|_{i=1}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) & \mathit{succs}(\ \left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right.) \sqsubseteq_T \mathbf{skip} \\
= & !\big(\mathit{fails}\ t \mid \left|_{i=1}^{\infty}(t^i ; \mathit{fails}\ t)\right.\big) & \text{Law 5.1} \\
= & !\mathit{fails}\ t \mid \mathit{fails}(\mathit{fails}\ t) ; !\left|_{i=1}^{\infty}(t^i ; \mathit{fails}\ t)\right. & \text{Law 5.24} \\
= & \mathit{fails}\ t \mid \mathit{succs}\ t ; !\left|_{i=1}^{\infty}(t^i ; \mathit{fails}\ t)\right. & \text{Laws 5.37, and 5.21} \\
= & \mathit{fails}\ t \mid \left|_{i=1}^{\infty}(\mathit{succs}\ t ; t^i ; \mathit{fails}\ t)\right. & \text{Property of } \left|, \text{ and Law 5.66}\right. \\
= & \mathit{fails}\ t \mid \left|_{i=1}^{\infty}(t^i ; \mathit{fails}\ t)\right. & \text{Laws 5.19, and 5.70} \\
= & \left|_{i=0}^{\infty} t^i ; \mathit{fails}\ t\right. & \text{Law 5.62}
\end{array}
$$

(b) $\left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right. \sqsubseteq_T !(\mathit{exhaust}\ t)$

In order to prove this result, we make use of the recursive definition of infinite alternation. Writing $H(k) = t^k ; !(\mathit{exhaust}\ t)$, we show that $H(k) \sqsubseteq_T \big(F(k) \mid H(k+1)\big)$, where $F(k) = t^k ; \mathit{fails}\ t$; i.e. that H satisfies the equation for infinite alternation. Since $!(\mathit{exhaust}\ t) = H(0)$, we have that $\left|_{i=0}^{\infty}(t^i ; \mathit{fails}\ t)\right. \sqsubseteq_T !(\mathit{exhaust}\ t)$.

$$
\begin{array}{lll}
& H(k) & \\
= & t^k ; !(\mathit{exhaust}\ t) & \text{Definition of H} \\
= & t^k ; !(t ; \mathit{exhaust}\ t \mid \mathbf{skip}) & \text{Definition of exhaust} \\
= & t^k ; !(\mathit{fails}(t ; \mathit{exhaust}\ t) ; \mathbf{skip} \mid t ; \mathit{exhaust}\ t) & \text{Law 5.121} \\
= & t^k ; !(\mathit{fails}(t ; \mathit{exhaust}\ t) \mid t ; \mathit{exhaust}\ t) & \text{Law 5.1} \\
= & t^k ; !\big(\mathit{fails}\big(t ; \mathit{succs}(\mathit{exhaust}\ t)\big) \mid t ; \mathit{exhaust}\ t\big) & \text{Law 5.35} \\
\sqsubseteq_T & t^k ; !(\mathit{fails}(t ; \mathbf{skip}) \mid t ; \mathit{exhaust}\ t) & \\
& & \text{Law 5.83; } \mathit{exhaust}\ t \text{ not guaranteed to terminate} \\
= & t^k ; !(\mathit{fails}\ t \mid t ; \mathit{exhaust}\ t) & \text{Law 5.1} \\
= & t^k ; \big(!\mathit{fails}\ t \mid \mathit{fails}(\mathit{fails}\ t) ; !(t ; \mathit{exhaust}\ t)\big) & \text{Law 5.24} \\
= & t^k ; \big(\mathit{fails}\ t \mid \mathit{succs}\ t ; !(t ; \mathit{exhaust}\ t)\big) & \text{Law 5.21} \\
= & t^k ; \big(\mathit{fails}\ t \mid \mathit{succs}\ t ; !(!t ; !\mathit{exhaust}\ t)\big) & \text{Assumption, and Law 5.13} \\
= & t^k ; \big(\mathit{fails}\ t \mid \mathit{succs}\ t ; !t ; !\mathit{exhaust}\ t\big) & \text{Law 5.12} \\
= & t^k ; \big(\mathit{fails}\ t \mid \mathit{succs}\ t ; t ; !(\mathit{exhaust}\ t)\big) & \text{Assumption} \\
= & t^k ; \big(\mathit{fails}\ t \mid t ; !(\mathit{exhaust}\ t)\big) & \text{Law 5.19} \\
= & !t^k ; \big(\mathit{fails}\ t \mid t ; !(\mathit{exhaust}\ t)\big) & \text{Assumption, and Law 5.78} \\
= & !t^k ; \mathit{fails}\ t \mid !t^k ; t ; !(\mathit{exhaust}\ t) & \text{Law 5.11} \\
= & t^k ; \mathit{fails}\ t \mid t^k ; t ; !(\mathit{exhaust}\ t) & \text{Assumption, and Law 5.78}
\end{array}
$$

$$= t^k ; \mathit{fails}\ t \mid t^{k+1} ; !(\mathit{exhaust}\ t) \qquad \text{Law 5.70}$$
$$= F(k) \mid H(k+1) \qquad \text{Definitions of F and H}$$

□

This is a valuable result since it reduces a recursive expression into something resembling our earlier normal form for tactics *together with an assertion*. The assertion is essential for the correctness of the latter tactic. This may lead to a completeness result for the tactic language with recursion. See Section 8.3.

Law 5.84 † $!(\mathit{exhaust}\ t) ; t = \mathbf{fail}$

(This holds under the previously-stated assumptions; hence $\mid_{i=0}^{\infty} t^i \neq \mathbf{abort}$), and so Law 5.3 is applicable below; otherwise, the result becomes a refinement.)

Proof:

$$!(\mathit{exhaust}\ t) ; t$$
$$= (\mid_{i=0}^{\infty} t^i ; \mathit{fails}\ t) ; t \qquad \text{Lemma 5.9.1}$$
$$= (\mid_{i=0}^{\infty} t^i) ; \mathit{fails}\ t ; t \qquad \text{Law 5.63}$$
$$= (\mid_{i=0}^{\infty} t^i) ; \mathbf{fail} \qquad \text{Law 5.20}$$
$$= \mathbf{fail} \qquad \text{Law 5.3}$$

□

Law 5.85 $!(\mathit{exhaust}\ t) ; !(\mathit{exhaust}\ t) = !(\mathit{exhaust}\ t)$

Proof:

$$!(\mathit{exhaust}\ t) ; !(\mathit{exhaust}\ t)$$
$$= !(\mathit{exhaust}\ t) ; !(t ; \mathit{exhaust}\ t \mid \mathbf{skip}) \qquad \text{Definition of } \mathit{exhaust}\ t$$
$$= !(!(\mathit{exhaust}\ t) ; (t ; \mathit{exhaust}\ t \mid \mathbf{skip})) \qquad \text{Law 5.13}$$
$$= !(!(\mathit{exhaust}\ t) ; t ; \mathit{exhaust}\ t \mid !(\mathit{exhaust}\ t)) \qquad \text{Law 5.11}$$
$$= !(\mathbf{fail} ; \mathit{exhaust}\ t \mid !(\mathit{exhaust}\ t)) \qquad \text{Law 5.84}$$
$$= !(!(\mathit{exhaust}\ t)) \qquad \text{Law 5.2}$$
$$= !(\mathit{exhaust}\ t) \qquad \text{Law 5.33}$$

□

Law 5.86 $\mathit{fails}\ t ; \mathit{exhaust}\ t = \mathit{fails}\ t$

Law 5.87 $!(\mathit{exhaust}\ t) ; \mathit{fails}\ t = !(\mathit{exhaust}\ t)$

Law 5.88 $\mathit{try}(\mathit{exhaust}\ t) = !(\mathit{exhaust}\ t)$

5.10 Structural Combinators

When the tactic language is used in a particular application, it will often be useful to define tactics which exploit the structure of the goals (expressions) to which they are applied. This is accomplished by the use of *structural combinators*. An example of their use was seen in Section 5.2. Any definition of such a combinator must be sound with respect to some meta-theorem for the expressions under consideration—i.e. monotonicity is required, so that application of tactics to sub-expression gives a valid rewriting of the whole expression.

Semantics

If an binary operator $\oplus$ is present in the object (expression/goal) language, it may be lifted into the tactic language as $\boxed{\oplus}$ by the following definition:

$$\begin{aligned} \mathit{cross}\, ts &= \prod \circ (\curlyvee_{\mathrm{id}} ts) \\ \mathit{break}(p_1 \oplus p_2) &= \langle p_1, p_2 \rangle \\ \mathit{combine}(p_1, p_2) &= (p_1 \oplus p_2) \\ t_1 \boxed{\oplus} t_2 &= \mathit{combine}* \circ (\mathit{cross}\langle t_1, t_2 \rangle) \circ \mathit{break} \end{aligned}$$

The definition is the composition of three auxiliary functions. The central function *cross* does the work—it takes a list of sub-expressions and *zips* them with a list of tactics ($\curlyvee_f$, pronounced '*zip with f*', is defined in Appendix A.1). This produces a list of lists of alternatives. These alternatives must be combined via a cartesian product, $\prod$ (so that the first alternative from the first tactic may be matched with the first or the second alternative from the second tactic, and so on). The auxiliary definitions *break* and *combine* simply convert a goal with an infix $\oplus$ into a list of subgoals, and vice versa.

Laws

Certain laws have rather asymmetric provisos, as a result of the definition of $\prod$. An alternative ordering of the results of the cartesian product would give different laws. If a diagonal enumeration were used the structural combinators would be better-behaved with respect to tactics producing infinitely many alternatives, but the *abides* and distributive laws would need stronger provisos.

When defined in this way, $\boxed{\oplus}$ *abides* with sequential composition:

Law 5.89 $(t_1 \boxed{\oplus} t_2) ; (t_3 \boxed{\oplus} t_4) = (t_1 ; t_3) \boxed{\oplus} (t_2 ; t_4)$
provided $t_2 = !t_2$, or $t_1 = !t_1$ and $t_3 = !t_3$

Proof:

$$\begin{aligned} & (t_1 \oplus t_2) ; (t_3 \oplus t_4) \\ &= \frown/ \circ (t_3 \oplus t_4)* \circ (t_1 \oplus t_2) && \text{Definition ;} \\ &= \frown/ \circ (\mathit{combine}* \circ \mathit{cross}\langle t_3, t_4 \rangle \circ \mathit{break})* \circ \mathit{combine}* \circ \mathit{cross}\langle t_1, t_2 \rangle \circ \mathit{break} \\ & && \text{Definition of } \oplus \\ &= \frown/ \circ \mathit{combine}** \circ (\mathit{cross}\langle t_3, t_4 \rangle)* \circ \\ & \qquad (\mathit{break} \circ \mathit{combine})* \circ \mathit{cross}\langle t_1, t_2 \rangle \circ \mathit{break} && \text{Property of } \circ \text{ and } * \\ &= \frown/ \circ \mathit{combine}** \circ (\mathit{cross}\langle t_3, t_4 \rangle)* \circ \mathit{cross}\langle t_1, t_2 \rangle \circ \mathit{break} \\ & && \text{Property of } \mathit{combine} \text{ and } \mathit{break}\text{, and first lemma below} \\ &= \mathit{combine}* \circ \frown/ \circ (\mathit{cross}\langle t_3, t_4 \rangle)* \circ \mathit{cross}\langle t_1, t_2 \rangle \circ \mathit{break} && \text{Lemmas, below} \\ &= \mathit{combine}* \circ (\mathit{cross}\langle \frown/ \circ t_3* \circ t_1, \frown/ \circ t_4* \circ t_2 \rangle)* \circ \mathit{break} && \text{Key Lemma} \\ &= \mathit{combine}* \circ (\mathit{cross}\langle t_1 ; t_3, t_2 ; t_4 \rangle)* \circ \mathit{break} && \text{Definition of ;} \\ &= (t_1 ; t_3) \oplus (t_2 ; t_4) && \text{Definition of } \oplus \end{aligned}$$

□

Lemmas (to be found in [Bir88])

$$f \circ \mathrm{id}* = f$$
$$f* \circ \frown/ = \frown/ \circ f**$$
$$f* \circ g* = (f \circ g)*$$

Key Lemma (subject to proviso):

$$\frown/ \circ (cross(t_3, t_4))* \circ cross\langle t_1, t_2\rangle = cross\langle \frown/ \circ t_3* \circ t_1, \frown/ \circ t_4* \circ t_2\rangle$$

Proof: Let $t_1\ g_1 = \langle h_{11}, \ldots, h_{1m}\rangle$ and $t_2\ g_2 = \langle h_{21}, \ldots, h_{2n}\rangle$.

$$\begin{aligned}
&(\frown/ \circ (cross(t_3, t_4))* \circ cross\langle t_1, t_2\rangle)\langle g_1, g_2\rangle \\
&= (\frown/ \circ (\textstyle\prod \circ (\Upsilon_{\mathrm{id}}\langle t_3, t_4\rangle))* \circ \prod \circ (\Upsilon_{\mathrm{id}}\langle t_1, t_2\rangle))\langle g_1, g_2\rangle && \text{Definition of } cross \\
&= (\frown/ \circ (\textstyle\prod \circ (\Upsilon_{\mathrm{id}}\langle t_3, t_4\rangle))* \circ \prod)\langle t_1\ g_1, t_2\ g_2\rangle && \text{Definition of } \Upsilon_{\mathrm{id}} \\
&= (\frown/ \circ (\textstyle\prod \circ (\Upsilon_{\mathrm{id}}\langle t_3, t_4\rangle))* \circ \prod)\langle \langle h_{11}, \ldots, h_{1m}\rangle, \langle h_{21}, \ldots, h_{2n}\rangle\rangle \\
& && \text{Assumption} \\
&= (\frown/ \circ (\textstyle\prod \circ (\Upsilon_{\mathrm{id}}\langle t_3, t_4\rangle))*) && \text{Property of } \textstyle\prod \\
&\qquad \langle \langle h_{11}, h_{21}\rangle, \ldots, \langle h_{11}, h_{2n}\rangle, \ldots, \langle h_{1m}, h_{21}\rangle, \ldots, \langle h_{1m}, h_{2n}\rangle\rangle \\
&= \frown/\langle \textstyle\prod\langle t_3\ h_{11}, t_4\ h_{21}\rangle, \ldots, \prod\langle t_3\ h_{11}, t_4\ h_{2n}\rangle, \ldots, \\
&\qquad \textstyle\prod\langle t_3\ h_{1m}, t_4\ h_{21}\rangle, \ldots, \prod\langle t_3\ h_{1m}, t_4\ h_{2n}\rangle\rangle && \text{Definition of } *, \text{ and } \Upsilon_{\mathrm{id}} \\
&= \textstyle\prod\langle \frown/\langle t_3\ h_{11}, \ldots, t_3\ h_{1m}\rangle, \frown/\langle t_4\ h_{21}, \ldots, t_4\ h_{2n}\rangle\rangle && \ddagger \\
&= (\textstyle\prod \circ (\Upsilon_{\mathrm{id}}\langle \frown/ \circ t_3* \circ t_1, \frown/ \circ t_4* \circ t_2\rangle))\langle g_1, g_2\rangle \\
& && \text{Definition of } \Upsilon_{\mathrm{id}} \text{ and } *, \text{ and assumption} \\
&= cross\langle \frown/ \circ t_3* \circ t_1, \frown/ \circ t_4* \circ t_2\rangle\ \langle g_1, g_2\rangle && \text{Definition of } cross
\end{aligned}$$

Step ($\ddagger$) is justified by the initial proviso—both $t_2 = !t_2$, and $t_1 = !t_1$ with $t_3 = !t_3$ are sufficient conditions; their disjunction appears to be a necessary condition:

$t_2 = !t_2$ implies that $n = 1$ (or $n = 0$, which is a degenerate case). In this case, the requirement becomes that $\prod\langle t_3\ h_{11}, t_4\ h_{21}\rangle \frown \ldots \frown \prod\langle t_3\ h_{1m}, t_4\ h_{21}\rangle = \prod\langle t_3\ h_{11} \frown \ldots \frown t_3\ h_{1m}, t_4\ h_{21}\rangle$, and this is a property of $\frown$ and $\prod$.

Similarly, $t_1 = !t_1$ implies that $m = 1$, but no such law exists between $\frown$ and $\prod$ in this case, and so a further constraint—that $t_3\ h_{11} = \langle k\rangle$, for some k (again ignoring degenerate cases)—is needed. Then the equation becomes $\prod\langle\langle k\rangle, t_4\ h_{21}\rangle \frown \ldots \frown \prod\langle\langle k\rangle, t_4\ h_{2n}\rangle = \prod\langle\langle k\rangle, t_4\ h_{21} \frown \ldots \frown t_4\ h_{2n}\rangle$, which follows from the definition of $\prod$. □

Also, $\boxed{\oplus}$ distributes through alternation, and *cut* distributes through $\boxed{\oplus}$:

Law 5.90 $t_1 \boxed{\oplus} (t_2 \mid t_3) = (t_1 \boxed{\oplus} t_2) \mid (t_1 \boxed{\oplus} t_3)$

Law 5.91 $(t_1 \mid t_2) \boxed{\oplus} t_3 = (t_1 \boxed{\oplus} t_3) \mid (t_2 \boxed{\oplus} t_3)$ provided $t_3 = !t_3$

Law 5.92 $!(t_1 \boxed{\oplus} t_2) = !t_1 \boxed{\oplus} !t_2$

Notice that the proofs of these laws use (depend on) the property $break \circ combine = \mathrm{id}$. Notice too that the tactic $t_1 \boxed{\oplus} t_2$ fails if either t_1 or t_2 does, and fails if presented with a goal which is not of the form $g_1 \oplus g_2$ (this is accomplished by having *break* return an empty list in this instance; see also the definitions of Υ and $\prod$).

Law 5.93 $succs(t_1 \boxed{\oplus} t_2) = succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip}) \; ; succs(t_1 \boxed{\oplus} t_2)$

Law 5.94 $succs(t_1 \boxed{\ominus} t_2) = (succs\ t_1) \boxed{\oplus} (succs\ t_2)$

Example

With this generic definition in place, the tactical $\boxed{\wedge}$ can now be defined, simply by giving appropriate instantiations for *break* and *combine*:

$$\begin{aligned} break_{\wedge}(g_1 \wedge g_2) &= (g_1, g_2) \\ combine_{\wedge}(g_1, g_2) &= (g_1 \wedge g_2) \end{aligned}$$

Lifted parallel composition (for goal-directed proof, where branching proof trees give rise to parallel compositions of goals) will be slightly harder to define. See Section 5.12 for details.

5.11 Pattern-matching

It is desirable to have a tactic take different actions depending on the form of the goal to which it is applied. For example, in the Z frame in Chapter 2, the toolkit tactic TK-TAC was used to rewrite predicates using relevant definitions from the toolkit. This *could* have been written using only alternation, but the inefficiency involved (rewriting with each definition until the correct one is found) would be great.

Moreover, it is often useful to be able to parametrise tactics with terms that arise within the goal. Recall, for example, the tactic for replacing a schema by its definition:

$$\begin{aligned} auto\text{-}schdef(\vdash S) &= apply\text{-}schdef\ S \\ auto\text{-}schdef(\vdash b.S) &= apply\text{-}schdef\ S \\ &\text{etc.} \end{aligned}$$

We have already assumed that basic rules act in this way (see, for example, the rules used in Section 5.2). Since we have ruled out tactics being arbitrary functions on goals for reasons of soundness (only basic rules can directly manipulate goals), another tactic construct is needed—one which allows access to the terms in the goal, but is nevertheless forced to apply only basic rules.[13]

π permits the definition of tactics which are dependent upon the goal to which they are applied. Unsoundness is avoided by having π return a *tactic* rather than (say) a list of alternative subgoals:

$$(\pi\, v_1, \ldots, v_n \bullet g \longrightarrow t) \ .$$

This tactic binds the (meta-)variables $v_1, \ldots, v_n$ within the scope of g and t. If goal g matches exactly the goal presented to the tactic (the variables in $v_1, \ldots, v_n$ being angelically chosen, if possible, to make this be the case), the whole tactic behaves like tactic t. If not, the tactic fails.

For example,

$$(\pi\, x \bullet (\vdash x \in S) \longrightarrow t_1(x)) \mid (\pi\, x \bullet (\vdash x = S) \longrightarrow \mathbf{skip})$$

[13] 2OBJ takes an alternative approach, where *all* tactics are parametrised in this way. As a result most 2OBJ tactics simply discard their goal parameter; they are total and constant.

is a tactic which will behave like tactic t_1 (parametrised by x) if presented with a goal of the form $(\vdash x \in S)$, will skip if presented with a goal of the form $(\vdash x = S)$, and will fail otherwise.

Semantics

The *variables* which π introduces are *meta*-variables: that is, they may range over members of any of the *term classes* (expressions, predicates, object variables) in the object language. The account which follows will assume that the members of those term classes are denumerable, and that the particular term classes to which the variables belong will be apparent (so, in the example above, x must denote an object variable, say, and S must denote an expression). The *scope* of the meta-variable introduced in this way will be made explicit where necessary, by use of parentheses.

This construction may be given a formal meaning by defining its two components separately. First, a tactic which succeeds only if it is presented with a goal matching its argument (*equals* is a tactic parametrised by some goal g):

$$\begin{aligned} \mathit{equals}\ g\ g &= \mathbf{skip}\ g \\ \mathit{equals}\ g\ h &= \mathbf{fail}\ g \text{ where } h \neq g\ . \end{aligned}$$

This is a very tigbtly defined tactic—it calls for true equality of goals. In order to make it useful, free variables for binding names to terms will be needed. We may accomplish this by introducing another construct—logical constants, in the style of [Mor90]:

$$(\mathbf{con}\ v \bullet t)$$

This tactic introduces v as a set of free variables chosen from an appropriate syntactic class (denoted *TERM* below) in t, angelically chosen so that as many choices as possible succeed:

$$(\mathbf{con}\ v \bullet t(v)) = \left(\Big|_{v \in \mathit{TERM}}\ t(v)\right)$$

Notice that $\Big|_{v \in \mathit{TERM}}$ is intended to indicate that the possible terms are enumerated in some order. If matching over several variables simultaneously is required, they must all be introduced together—so that a 'diagonal' enumeration can be used. Note, too, that **con** may bind any term/expression in the object language. For instance, $(\mathbf{con}\ p \bullet \mathit{equals}\ (\vdash p)\ ;\ s)$ is a tactic which will apply s to any sequent having just one predicate on the right-hand side—and p may be a parameter to s.

This, however, is a very weak specification of **con**'s behaviour.[14] This definition will abort if there are no suitable instantiations of v, whereas it is usually possible to determine this in a finite time (and have the tactic fail instead)—and nested instances of **con** will not, for the same reason, be handled well (they will tend to abort, since the value chosen by the outer **con** will be fixed whilst *all* of those offered by the inner **con** are tried—if no match is possible for the first-chosen outer value, an infinite loop will ensue).

[14] It is also not a plausible implementation. Whilst the variables, expressions, predicates, etc. of our object language may be theoretically denumerable, it would clearly be inappropriate simply to attempt each one in turn.

A reasonable implementation of con will allow *two* notions of scope—there is the area in which backtracking is possible (preferably 'smart' backtracking, which will cause failure if no matches can be found) and an area in which the variables introduced by con are bound. Ideally, the extent of these two scopes will be determined by the tactic programmer—in practice, the first will be limited to a single instance of *equals* (see Section 7.2). The backtracking will generally need to be limited, so that if the tactic body fails (for reasons unrelated to the pattern-matching) the whole tactic will fail.

Now we may define

$$(\pi\, v_1, \ldots, v_n \bullet g \longrightarrow t) = (\mathbf{con}\, v_1, \ldots, v_n \bullet \mathit{equals}\, g\, ;\, t) \ .$$

Morgan observes that con is not (cannot be) *code* in any imperative language. The situation is slightly better here—con introduces a simple pattern-matching problem,[15] which is solvable for a first-order language, and may be partially soluble for other languages. In the light-weight implementation introduced in Chapter 7, this is achieved by appealing to the unification algorithm in the underlying functional language, thus giving one of the implementations proposed above. Implementation of a unification algorithm to give a tactic interpreter precisely the semantics described here is also possible.

Laws on *equals*

Proof of laws about *equals* is elementary.

Law 5.95 $\mathit{equals}\, g_1\, ;\, \mathit{equals}\, g_2 = \mathbf{fail}$ if $g_1 \neq g_2$

Law 5.96 $\mathit{equals}\, g\, ;\, \mathit{equals}\, g = \mathit{equals}\, g$

Law 5.97 $!\mathit{equals}\, g = \mathit{equals}\, g$

Laws on con

con distributes whenever no variable capture is caused. The side-conditions are phrased using ϕ (by analogy with the Z semantics (see Chapter 2))—intended to extract the set of free (meta-)variables of the tactic to which it is applied. The 'weak' semantics proposed above would, of course, allow the first two laws to be proved without the side-conditions, but if backtracking is to be restricted as discussed in the prose, these provisos are needed.

Law 5.98 $(\mathbf{con}\, v \bullet t_1\, ;\, t_2) = (\mathbf{con}\, v \bullet t_1)\, ;\, t_2$ provided $v \notin \phi t_2$

Law 5.99 $(\mathbf{con}\, v \bullet t_1\, ;\, t_2) = t_1\, ;\, (\mathbf{con}\, v \bullet t_2)$ provided $v \notin \phi t_1$ and $t_1 = !t_1$

Law 5.100 $(\mathbf{con}\, v \bullet t) = t$ provided $v \notin \phi t$

con may be expected to satisfy other common quantifier rules:

Law 5.101 $(\mathbf{con}\, v \bullet t) = (\mathbf{con}\, u \bullet t[v \backslash u])$ provided $u \notin \phi t$

Law 5.102 $\mathit{equals}\, g\, ;\, (\mathbf{con}\, g' \bullet \mathit{equals}\, g'\, ;\, t) = \mathit{equals}\, g\, ;\, t[g' \backslash g]$

[15] For certain classes of problem, a con which signalled unification might be of benefit.

Laws on π

The above laws may be used to derive some properties of π, but in general it will be as easy to deal with reasoning about con and *equals*, so no derived laws will be presented here.

5.12 Parallel Composition

We shall handle proof trees that bifurcate using structural combinators, as described in Section 5.10. There is, however, an added complication, in that the application of a tactic to a proof tree branch may lead to *no subgoals*—the proof of that branch may be completed. That is to say, the type of proof nodes consists of empty nodes (), single nodes g, and compound nodes $g_1 \parallel g_2$ (g_1 and g_2, being, in turn, empty, single or compound).[16] These symbols will carry these meanings for the remainder of this section.

Tactic parallel composition $\boxed{\parallel}$ may be defined as a structural combinator over this type by making the following definitions for *break* and *combine*:

$$\begin{aligned} \mathit{break}_{\parallel}\ (g_1 \parallel g_2) &= \langle g_1, g_2 \rangle \\ \mathit{break}_{\parallel}\ g &= \langle\,\rangle \\ \mathit{break}_{\parallel}\ () &= \langle\,\rangle \\ \mathit{combine}_{\parallel}\ \langle g_1, g_2 \rangle &= g_1 \parallel g_2 \end{aligned}$$

Of course, the application of basic rules must fail if they are applied to proof nodes other than single goals (i.e. to empty or compound nodes).

Clarity will generally be improved by removing the empty nodes from parallel compositions as they arise. We add two basic laws:

$$\begin{aligned} \mathit{nullidL} &== \big(() \parallel g\big) \equiv g \\ \mathit{nullidR} &== \big(g \parallel ()\big) \equiv g\ , \end{aligned}$$

and then define a tactic parallel composition[17] to be the application of the appropriate structural combinator, followed by an attempt to apply these identity laws:

$$t_1 \parallel t_2 = (t_1 \boxed{\parallel} t_2)\ ;\ !(\textbf{rule}\ \mathit{nullidL} \mid \textbf{rule}\ \mathit{nullidR} \mid \textbf{skip})$$

Now, whilst the abides property (Law 5.89) holds for $\boxed{\parallel}$, tactic parallel composition ($\parallel$) has this property only when the *nullid* rules are not invoked, i.e. only when neither t_1 nor t_2 completes a proof branch.

Law 5.103 $(t_1 \parallel t_2)\ ;\ (t_3 \parallel t_4) = (t_1\ ;\ t_3) \parallel (t_2\ ;\ t_4)$
provided the t_i satisfy the conditions for Law 5.89 and those above

[16] In dealing with an object logic in which the proof trees may divide into more than two branches (i.e. where application of a rule to a goal may produce more than two subgoals), there may be some value in generalizing this construction so that compound nodes may contain arbitrary lists of other (potentially compound) nodes. This compound structure (rather than simply a list of simple goals) must be retained, nevertheless, if parallel composition is to be in any way similar to the other structural combinators, and if tactics are to be able to exploit the structure present in the proof tree

[17] A small measure of overloading seems appropriate here.

This definition permits the definition of a parallel closure tactical—$t^{(\|)}$ applies t to all the individual goals in a parallel composition:

$$t^{(\|)} = \left(\begin{array}{l} \pi \bullet () \longrightarrow \mathbf{skip} \\ \mid \pi\, g_1, g_2 \bullet g_1 \parallel g_2 \longrightarrow t^{(\|)} \parallel t^{(\|)} \\ \mid \pi\, g \bullet g \longrightarrow t \end{array} \right)$$

That definition is more accurately rendered:

$$t^{(\|)} = (\mu X \bullet \left(\begin{array}{l} \pi \bullet () \longrightarrow \mathbf{skip} \\ \mid \pi\, g_1, g_2 \bullet g_1 \parallel g_2 \longrightarrow X \parallel X \\ \mid \pi\, g \bullet g \longrightarrow t \end{array} \right))$$

Parallel closure can be used to define the following useful tactical, which corresponds closely to THEN in 2OBJ. It applies t_1 to its goal, and then applies t_2 to each of the resulting subgoals. The notation (a semicolon, with the dot replaced by an asterisk) is reminiscent of that used for *map*—t_2 being applied to each goal resulting from t_1:

$$t_1 \; ; \; t_2 = t_1 \; ; \; t_2^{(\|)}$$

Laws

Law 5.104 $(t^{(\|)})^{(\|)} = t^{(\|)}$

Proof:

$$\begin{array}{ll}
& (t^{(\|)})^{(\|)} \\
= & (\mu X \bullet \mathit{equals}\ () \\
& \quad \mid (\mathbf{con}\, g \bullet \mathit{equals}\ g \; ; \; t^{(\|)}) \\
& \quad \mid (\mathbf{con}\, g_1, g_2 \bullet \mathit{equals}\ (g_1 \parallel g_2) \; ; \; (X \parallel X)) \;) & \text{Definition of } {}^{(\|)} \text{ and of } \pi \\
= & (\mu X \bullet \mathit{equals}\ () \\
& \quad \mid (\mathbf{con}\, g \bullet \mathit{equals}\ g \; ; \; (\mu X \bullet \left(\begin{array}{l} \mathit{equals}\ () \\ \mid (\mathbf{con}\, g' \bullet \mathit{equals}\ g' \; ; \; t) \\ \mid (\mathbf{con}\, g_1', g_2' \bullet \\ \quad \mathit{equals}\ (g_1' \parallel g_2') \; ; \; (X \parallel X)) \end{array} \right)) \\
& \quad \mid (\mathbf{con}\, g_1, g_2 \bullet \mathit{equals}\ (g_1 \parallel g_2) \; ; \; (X \parallel X)) \;) & \text{Definition of } {}^{(\|)} \text{ and of } \pi \\
= & (\mu X \bullet \mathit{equals}\ () \\
& \quad \mid (\mathbf{con}\, g \bullet \mathit{equals}\ g \; ; \left(\begin{array}{l} \mathit{equals}\ () \\ \mid (\mathbf{con}\, g' \bullet \mathit{equals}\ g' \; ; \; t) \\ \mid (\mathbf{con}\, g_1', g_2' \bullet \mathit{equals}\ (g_1' \parallel g_2') \; ; \; (t^{(\|)} \parallel t^{(\|)})) \end{array} \right)) \\
& \quad \mid (\mathbf{con}\, g_1, g_2 \bullet \mathit{equals}\ (g_1 \parallel g_2) \; ; \; (X \parallel X)) \;) & \text{Definition of } \mu \text{ and } {}^{(\|)} \\
= & (\mu X \bullet \mathit{equals}\ () \\
& \quad \mid (\mathbf{con}\, g \bullet \left(\begin{array}{l} \mathit{equals}\ g \; ; \mathit{equals}\ () \\ \mid \mathit{equals}\ g \; ; (\mathbf{con}\, g' \bullet \mathit{equals}\ g' \; ; \; t) \\ \mid \mathit{equals}\ g \; ; (\mathbf{con}\, g_1', g_2' \bullet \mathit{equals}\ (g_1' \parallel g_2') \; ; \; (t^{(\|)} \parallel t^{(\|)})) \end{array} \right)) \\
& \quad \mid (\mathbf{con}\, g_1, g_2 \bullet \mathit{equals}\ (g_1 \parallel g_2) \; ; \; (X \parallel X)) \;) & \text{Law 5.11 and Law 5.97} \\
= & (\mu X \bullet \mathit{equals}\ ()
\end{array}$$

$$
\begin{array}{ll}
& \quad |(\mathbf{con}\, g \bullet \left(\begin{array}{l} \mathbf{fail} \\ |\, equals\ g\ ;\ t \\ |\, (\mathbf{con}\, g_1', g_2' \bullet equals\ g\ ;\ equals\ (g_1' \parallel g_2')\ ;\ (t^{(\mathbb{I})} \parallel t^{(\mathbb{I})})) \end{array}\right)) \\
& \quad |(\mathbf{con}\, g_1, g_2 \bullet equals\ (g_1 \parallel g_2)\ ;\ (X \parallel X))\ ) \qquad \text{Laws 5.95, 5.102 and 5.99} \\
= & (\mu X \bullet equals\ () \\
& \quad |(\mathbf{con}\, g \bullet (equals\ g\ ;\ t) \mid \mathbf{fail}) \\
& \quad |(\mathbf{con}\, g_1, g_2 \bullet equals\ (g_1 \parallel g_2)\ ;\ (X \parallel X))\ ) \\
& \qquad\qquad \text{Laws 5.2, 5.95, 5.100, and 5.3} \\
= & (\mu X \bullet equals\ () \\
& \quad |(\mathbf{con}\, g \bullet equals\ g\ ;\ t) \\
& \quad |(\mathbf{con}\, g_1, g_2 \bullet equals\ (g_1 \parallel g_2)\ ;\ (X \parallel X))\ ) \qquad \text{Law 5.2} \\
= & \ t^{(\mathbb{I})} \qquad \text{Definition of } {}^{(\mathbb{I})}
\end{array}
$$

□

Law 5.105 $t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})} = (t_1\ ;\ t_2^{(\mathbb{I})})^{(\mathbb{I})}$ provided that $t_1^{(\mathbb{I})}$ satisfies the proviso for the $\|$-abides law, and that t_1 and t_2 are terminating

Proof: It suffices to show that $t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}$ satisfies the recursive equation for $(t_1\ ;\ t_2^{(\mathbb{I})})^{(\mathbb{I})}$:

$$
\begin{array}{ll}
& t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})} \\
= & equals\ ()\ ;\ t_2^{(\mathbb{I})} \\
& \quad |(\mathbf{con}\, g_1, g_2 \bullet equals\ (g_1 \parallel g_2)\ ;\ (t_1^{(\mathbb{I})} \parallel t_1^{(\mathbb{I})}))\ ;\ t_2^{(\mathbb{I})} \\
& \quad |(\mathbf{con}\, g \bullet equals\ g\ ;\ t_1)\ ;\ t_2^{(\mathbb{I})} \qquad \text{Definition of } {}^{(\mathbb{I})}\text{, and Law 5.6} \\
= & equals\ ()\ ;\ t_2^{(\mathbb{I})} \\
& \quad |(\mathbf{con}\, g_1, g_2 \bullet equals\ (g_1 \parallel g_2)\ ;\ (t_1^{(\mathbb{I})} \parallel t_1^{(\mathbb{I})})\ ;\ t_2^{(\mathbb{I})}) \\
& \quad |(\mathbf{con}\, g \bullet equals\ g\ ;\ t_1\ ;\ t_2^{(\mathbb{I})}) \qquad \text{Law 5.98} \\
= & equals\ () \\
& \quad |(\mathbf{con}\, g_1, g_2 \bullet equals\ (g_1 \parallel g_2)\ ;\ ((t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}) \parallel (t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}))) \\
& \quad |(\mathbf{con}\, g \bullet equals\ g\ ;\ t_1\ ;\ t_2^{(\mathbb{I})}) \qquad \text{Lemmas}
\end{array}
$$

The lemmas used above are:

$$
\begin{array}{rcl}
equals\ ()\ ;\ t_2^{(\mathbb{I})} & = & equals\ () \\
(t_1^{(\mathbb{I})} \parallel t_1^{(\mathbb{I})})\ ;\ t_2^{(\mathbb{I})} & = & ((t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}) \parallel (t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}))
\end{array}
$$

The first one is a straightforward consequence of the laws presented earlier. The second depends on the proviso—allowing $t_2^{(\mathbb{I})}$ to be rewritten as $t_2^{(\mathbb{I})} \| t_2^{(\mathbb{I})}$, and allowing the abides law to be used.

$$
\begin{array}{ll}
& (t_1^{(\mathbb{I})} \parallel t_1^{(\mathbb{I})})\ ;\ t_2^{(\mathbb{I})} \\
= & (t_1^{(\mathbb{I})} \parallel t_1^{(\mathbb{I})})\ ;\ (t_2^{(\mathbb{I})} \parallel t_2^{(\mathbb{I})}) \qquad \text{Definition of } {}^{(\mathbb{I})} \\
= & (t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}) \parallel (t_1^{(\mathbb{I})}\ ;\ t_2^{(\mathbb{I})}) \qquad \text{Abides Law 5.103}
\end{array}
$$

□

These laws give rise to some derived laws about the generalized sequential composition (;):

Law 5.106 $t_1 \,;\, (t_2 \,;\, t_3) = (t_1 \,;\, t_2) \,;\, t_3$ provided the t_i are terminating, and provided t_2 satisfies the proviso for the $\|$-abides law

Proof:

$$
\begin{aligned}
& t_1 \,;\, (t_2 \,;\, t_3) \\
&= t_1 \,;\, (t_2 \,;\, t_3^{(\|)})^{(\|)} && \text{Definition of ;} \\
&= t_1 \,;\, (t_2^{(\|)} \,;\, t_3^{(\|)}) && \text{Law 5.105} \\
&= (t_1 \,;\, t_2^{(\|)}) \,;\, t_3^{(\|)} && \text{Law 5.5} \\
&= (t_1 \,;\, t_2) \,;\, t_3 && \text{Definition of ;}
\end{aligned}
$$

□

Law 5.107 $\mathbf{skip} \,;\, t = t^{(\|)}$

Law 5.108 $t \,;\, \mathbf{skip} = t$

Law 5.109 $\mathbf{fail} \,;\, t = \mathbf{fail} = t \,;\, \mathbf{fail}$

Law 5.110 $(t_1 \mid t_2) \,;\, t_3 = t_1 \,;\, t_3 \mid t_2 \,;\, t_3$

Proof:

$$
\begin{aligned}
& (t_1 \mid t_2) \,;\, t_3 \\
&= (t_1 \mid t_2) \,;\, t_3^{(\|)} && \text{Definition of ;} \\
&= t_1 \,;\, t_3^{(\|)} \mid t_2 \,;\, t_3^{(\|)} && \text{Law 5.6} \\
&= t_1 \,;\, t_3 \mid t_2 \,;\, t_3 && \text{Definition of ;}
\end{aligned}
$$

□

5.13 Other Derived Laws

This section is present for the sake of completeness; it lists various laws which were omitted from the earlier account, but seem useful nevertheless.

Law 5.111 $\mathit{succs}\ \mathbf{abort} = \mathbf{abort}$

These two laws may be strengthened to equalities if t is guaranteed to terminate (Laws 5.50 and 5.51):

Law 5.112 $\mathit{succs}\ t \,;\, \mathit{fails}\ t = \mathit{fails}\ t \,;\, \mathit{succs}\ t \sqsubseteq_T \mathbf{fail}$

Law 5.113 $\mathit{fails}\ t \mid \mathit{succs}\ t = \mathit{succs}\ t \mid \mathit{fails}\ t \sqsubseteq_T \mathbf{skip}$

Law 5.114 $!(succs\ t \mid t) = succs\ t$

Law 5.115 $!(t \mid succs\ t) = !t$

Law 5.116 $!t \mid fails\ t = !(t \mid \mathbf{skip}) = fails\ t \mid !t$

Law 5.117 $t_1 \mid t_2 = t_2 \mid t_1$ provided $succs\ t_1 = fails\ t_2$

Proof:

$succs\ t_1 = fails\ t_2$

$\Rightarrow succs\ t_1 \sqsubseteq_T fails\ t_2 \wedge fails\ t_2 \sqsubseteq_T succs\ t_1$ — property of $\sqsubseteq_T$

$\Rightarrow succs\ t_1 \sqsubseteq_T fails\ t_2 \wedge succs\ t_2 \sqsubseteq_T fails\ t_1$ — Laws 5.37,5.22

$\Rightarrow t_1 \mid t_2 \sqsubseteq_T t_2 \mid t_1 \wedge t_2 \mid t_1 \sqsubseteq_T t_1 \mid t_2$ — Law 5.69

$\Rightarrow t_1 \mid t_2 = t_2 \mid t_1$ — property of $\sqsubseteq_T$

□

Law 5.118 $!(t\ ;\ d \mid fails\ t) = !(t\ ;\ d \mid \mathbf{skip})$ provided $succs\ d = \mathbf{skip}$

Law 5.119 $fails(t\ ;\ d) \sqsubseteq_T fails\ t$ provided $succs\ d \sqsubseteq_T \mathbf{skip}$

Proof:

$fails(t\ ;\ d)$

$= fails(t\ ;\ succs\ d)$ — Law 5.35

$\sqsubseteq_T fails(t\ ;\ \mathbf{skip})$ — supposition

$= fails\ t$ — Law 5.1

□

Law 5.120 $!(t\ ;\ d \mid \mathbf{skip}) \sqsubseteq_T !(t\ ;\ d \mid fails\ t)$ provided $succs\ d \sqsubseteq_T \mathbf{skip}$

Proof:

$!(t\ ;\ d \mid \mathbf{skip})$

$= !(t\ ;\ d \mid fails(t\ ;\ d))$ — Law 5.116

$\sqsubseteq_T !(t\ ;\ d \mid fails\ t)$ — Law 5.119

□

Law 5.121 $!(t_1 \mid t_2) = !(fails\ t_1\ ;\ t_2 \mid t_1)$

$!(t_1 \mid t_2)$

$= !(!(t_1 \mid t_2))$ — Law 5.33

$= !(!t_1 \mid fails\ t_1\ ;\ !t_2)$ — Law 5.24

$= !(t_1 \mid fails\ t_1\ ;\ t_2)$ — Laws 5.14 and 5.13

$= !(fails\ t_1\ ;\ t_2 \mid t_1)$ — Law 5.69

Chapter 6

Applications of Tactics

As was commented at the beginning of Chapter 5, the tactic language described above seems to be very wide in its applicability. This chapter looks at a number of these applications, and aims to demonstrate how the language can be used to improve the readability and explore properties of some of the tactics previously presented, and how the algebraic laws previously given can be of value in validating and transforming these tactics.

These tactics serve to demonstrate both the power of the language and its limitations. Convincing proofs of tactic correctness can be constructed—though for all but the simplest tactics, these are something of a *tour de force*. Such proofs serve to highlight the properties of the application area which are being exploited: the proof of tactic equivalence generally fails until some property of the basic rules is assumed.

6.1 Associative/Commutative Matching

norm Improved

The tactic presented in Section 5.2 for normalizing (i.e. left-associating) associative expressions can be improved upon, and shown to be correct. The improvement became apparent to the author in the process of the correctness argument which follows.

An improved version of *norm* is

$$norm = (!exhaust(\mathbf{rule}\ cossa)\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ norm))\ |\ \mathbf{skip}\ .$$

(The difference being that a recursive instance of *norm* has been replaced by a **skip**. This leads to an efficiency improvement in execution of some 25%.)

In order to demonstrate that this tactic does indeed produce terms in 'normal form', consider the following tactic, which checks for normal form. It succeeds when its argument is in normal form, and fails otherwise:

$$\begin{aligned} isnormal &= fails\ compound\ |\ (fails\ compound)\ \boxed{\oplus}\ isnormal \\ compound &= \mathbf{skip}\ \boxed{\oplus}\ \mathbf{skip} \end{aligned}$$

(Note that $isnormal = !isnormal$.)

It is clear that *norm* always terminates: each recursive application of *norm* is within the scope of a $\boxed{\oplus}$. Any term presented to *norm* will contain only finitely many instances of $\oplus$, and so the number of iterations is bounded.[1] As a result, in order to show that *norm* always produces a goal which is in 'normal' form, it suffices to prove that $succs(!norm\ ;\ isnormal) = succs(norm)$. In order to do this, some lemmas are useful:

Lemma 6.1.1 $fails\ compound\ ;\ isnormal = fails\ compound$

Proof:

$$
\begin{array}{ll}
& fails\ compound\ ;\ isnormal \\
= & fails\ compound; \\
& \quad (fails\ compound \mid fails\ compound \boxed{\oplus} isnormal) \qquad \text{Definition of } isnormal \\
= & fails\ compound\ ; fails\ compound \\
& \quad \mid fails\ compound\ ;\ (fails\ compound \boxed{\oplus} isnormal) \qquad \text{Law 5.11} \\
= & fails\ compound \\
& \quad \mid fails\ compound\ ;\ (fails\ compound \boxed{\oplus} isnormal) \qquad \text{Law 5.48} \\
= & fails\ compound \mid \mathbf{fail} \qquad \text{Law 6.1 and definition of } compound \\
= & fails\ compound \qquad \text{Law 5.2}
\end{array}
$$

□

Lemma 6.1.2 $fails(!exhaust(\mathbf{rule}\ cossa)\ ;\ (\mathbf{skip} \boxed{\oplus} norm)) = fails\ compound$

Proof:

$$
\begin{array}{ll}
& fails(!exhaust(\mathbf{rule}\ cossa)\ ;\ (\mathbf{skip} \boxed{\oplus} norm)) \\
= & fails(!exhaust(\mathbf{rule}\ cossa)\ ;\ succs(\mathbf{skip} \boxed{\oplus} norm)) \qquad \text{Law 5.35} \\
= & fails(!exhaust(\mathbf{rule}\ cossa)\ ;\ succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip})) \\
& \qquad \text{Law 5.94, and } succs\ norm = \mathbf{skip} \\
= & fails(succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip})\ ;\ !exhaust(\mathbf{rule}\ cossa)) \qquad \text{Property of } cossa \\
= & fails(succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip})\ ;\ succs(!exhaust(\mathbf{rule}\ cossa))) \qquad \text{Law 5.35} \\
= & fails(succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip})\ ;\ \mathbf{skip}) \qquad \text{Law 5.83} \\
= & fails(skiptac \boxed{\oplus} \mathbf{skip}) \qquad \text{Laws 5.1 and 5.22} \\
= & fails\ compound \qquad \text{Definition of } compound
\end{array}
$$

The property of *cossa* used in this proof is that it does not effect the success or failure of $\mathbf{skip} \boxed{\oplus} \mathbf{skip}$, and so

$$
\begin{array}{l}
!exhaust(\mathbf{rule}\ cossa)\ ;\ succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip}) \\
\quad = succs(\mathbf{skip} \boxed{\oplus} \mathbf{skip})\ ;\ !exhaust(\mathbf{rule}\ cossa);
\end{array}
$$

□

[1] Termination also depends on the termination of $exhaust(\mathbf{rule}\ cossa)$. This is assured by a similar argument, there only being finitely many *cossa*-redexes in a finite term, and fresh redexes not being introduced by applications of *cossa*

Lemma 6.1.3 $!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ !(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\ \mathit{isnormal} =$
$!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ (!\mathit{norm}\ ;\ \mathit{isnormal}))$

Proof: Noting the law

Law 6.1 $(t_1\ \boxed{\oplus}\ t_2)\ ;\mathit{fails}(\mathbf{skip}\ \boxed{\oplus}\ \mathbf{skip}) = \mathbf{fail} = \mathit{fails}(\mathbf{skip}\ \boxed{\oplus}\ \mathbf{skip})\ ;\ (t_1\ \boxed{\oplus}\ t_2)$

we observe first that

$!(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\ \mathit{isnormal}$
$= !(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\ (\mathit{fails\ compound}\ |\ (\mathit{fails\ compound})\ \boxed{\oplus}\ \mathit{isnormal})$
Definition of *isnormal*
$= !(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\mathit{fails\ compound}|$
$\quad !(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\ (\mathit{fails\ compound}\ \boxed{\oplus}\ \mathit{isnormal})$ Law 5.11
$= \mathbf{fail}\ |\ !(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\ (\mathit{fails\ compound}\ \boxed{\oplus}\ \mathit{isnormal})$
Laws 6.1 and 5.92
$= (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm})\ ;\ (\mathit{fails\ compound}\ \boxed{\oplus}\ \mathit{isnormal})$ Laws 5.2 and 5.92
$= \mathit{fails\ compound}\ \boxed{\oplus}\ !\mathit{norm}\ ;\ \mathit{isnormal}$

Now, it is a property of the rule *cossa* that

$$\mathit{fails\ compound}\ \boxed{\oplus}\ \mathbf{skip} = \mathit{fails}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathbf{skip})$$

And so

$!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ !(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ ;\ \mathit{isnormal}$
$= !\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\mathit{fails\ compound}\ \boxed{\oplus}\ !\mathit{norm}\ ;\ \mathit{isnormal}$ Above
$= !\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathit{fails\ compound}\ \boxed{\oplus}\ \mathbf{skip});$
$\quad (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm}\ ;\ \mathit{isnormal})$ Laws 5.89 and 5.1
$= !\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\mathit{fails}(\mathbf{rule}\ \mathit{cossa});$
$\quad (\mathbf{skip}\ \boxed{\oplus}\ \mathbf{skip})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm}\ ;\ \mathit{isnormal})$ Above
$= !\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm}\ ;\ \mathit{isnormal})$ Laws 5.87 and 5.1

□

This proof demonstrates a valuable technique made possible by the abides law for structural combinators—factorizing a tactic $t_1\ \boxed{\oplus}\ t_2$ as

$$t_1\ \boxed{\oplus}\ \mathbf{skip}\ ;\ \mathbf{skip}\ \boxed{\oplus}\ t_2\ .$$

It also demonstrates the practical effect of the comment that completeness is *relative* to the rule system in use—this proof has used a property of the rule *cossa* which (inherently) cannot be proved within the tactic language.

Theorem 6.1.4 $\mathit{succs}(!\mathit{norm}\ ;\ \mathit{isnormal}) = \mathit{succs}(\mathit{norm})$

Proof: First, observe that

$$
\begin{aligned}
& \mathit{succs}(\mathit{norm}) \\
&= \mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ |\ \mathbf{skip}) \\
& \qquad\qquad \text{Definition of } \mathit{norm} \\
&= !!(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm}))\ |\ \mathbf{skip}) \\
& \qquad\qquad \text{Laws 5.26 and 5.33} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ \mathit{succs}(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})) \\
&\quad |\mathit{fails}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm}))) \qquad \text{Laws 5.24 and 5.42} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ \mathit{succs}(\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})) \\
&\quad |\mathit{fails\ compound}) \qquad \text{Lemma 6.1.2} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathit{succs\ norm})) \\
&\quad |\mathit{fails\ compound}) \qquad \text{Law 5.94}
\end{aligned}
$$

Moreover,

$$
\begin{aligned}
& \mathit{succs}(!\mathit{norm}\ ;\ \mathit{isnormal}) \\
&= \mathit{succs}(!(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathit{norm})\ |\ \mathbf{skip})\ ;\ \mathit{isnormal}) \\
& \qquad\qquad \text{Definition of } \mathit{norm} \\
&= \mathit{succs}(!(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm})\ |\ \mathbf{skip})\ ;\ \mathit{isnormal}) \\
& \qquad\qquad \text{Laws 5.14, 5.13, 5.8 and 5.92} \\
&= \mathit{succs}((!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm})\ ;\ \mathit{isnormal}) \\
&\qquad |\mathit{fails}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm}))\ ;\ \mathit{isnormal}) \\
& \qquad\qquad \text{Laws 5.24 and 5.6} \\
&= \mathit{succs}((!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm})\ ;\ \mathit{isnormal}) \\
&\qquad |\mathit{fails\ compound}\ ;\ \mathit{isnormal}) \qquad \text{Lemma 6.1.2} \\
&= \mathit{succs}((!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm})\ ;\ \mathit{isnormal}) \\
&\qquad |\mathit{fails\ compound}) \qquad \text{Lemma 6.1.1} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ !\mathit{norm})\ ;\ \mathit{isnormal}) \\
&\quad |\mathit{fails\ compound}) \qquad \text{Laws 5.26 and 5.41} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ (!\mathit{norm}\ ;\ \mathit{isnormal}))) \\
&\quad |\mathit{fails\ compound}) \qquad \text{Lemma 6.1.3} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cosso})\ ;\ \mathit{succs}(\mathbf{skip}\ \boxed{\oplus}\ (!\mathit{norm}\ ;\ \mathit{isnormal}))) \\
&\quad |\mathit{fails\ compound}) \qquad \text{Law 5.42} \\
&= !(\mathit{succs}(!\mathit{exhaust}(\mathbf{rule}\ \mathit{cossa})\ ;\ (\mathbf{skip}\ \boxed{\oplus}\ \mathit{succs}(!\mathit{norm}\ ;\ \mathit{isnormal}))) \\
&\quad |\mathit{fails\ compound}) \qquad \text{Law 5.94}
\end{aligned}
$$

Therefore, since *succs norm* and *succs*(!*norm* ; *isnormal*) satisfy the same recursive equation, we have, by Theorem 5.8.2, that they are equal. □

Wider Improvements

The whole associative-instance-generating tactic can be improved by calling *norm* only once, and using calls of *exhaust*(**rule** *cossa*) on recursive calls:

$$\begin{aligned} assocs' &= !norm\,;\, assocrec \\ assocrec &= (!exhaust(\mathbf{rule}\, cossa)\,;\, exhaust(assoc); \\ &\qquad (assocrec \boxed{\oplus} assocrec)) \mid \mathbf{skip} \end{aligned}$$

assocs' should be equal to the *assocs* presented in Section 5.2, but a proof is not attempted here.

Efficiency can be improved further by reducing the number of duplicate instances which *assocs'* produces. One way in which to do this is to introduce a tactic which guards the recursive call of *assoc* $\boxed{\oplus}$ *assoc*—allowing it to happen only if it will produce any more associative instances; i.e. only if it will be applied to $e_1 \oplus (e_2 \oplus e_3)$ or $(e_1 \oplus e_2) \oplus e_3$.

$$\begin{aligned} guard &= succs(guardexpr_1 \mid guardexpr_2) \\ guardexpr_1 &= (\mathbf{skip} \boxed{\oplus} (\mathbf{skip} \boxed{\oplus} \mathbf{skip})) \\ guardexpr_2 &= ((\mathbf{skip} \boxed{\oplus} \mathbf{skip}) \boxed{\oplus} \mathbf{skip}) \\ assocrec' &= (!exhaust(\mathbf{rule}\, cossa)\,;\, exhaust(assoc); \\ &\qquad guard\,;\, (assocrec' \boxed{\oplus} assocrec')) \mid \mathbf{skip} \end{aligned}$$

Without this guard, the number of duplicates grows very quickly with the size of the expression—as each application of *assocrec* $\boxed{\oplus}$ *assocrec* entails, at some point, a **skip** which repeats the goal expression. Because *assocrec'* differs in its list of outcomes from *assocrec*, the two are *not* equivalent as tactics.[2]

6.2 A Tactic Proof of Lemma 5.4.4

Some of the proofs in Chapter 5 are very algorithmic in nature. The obvious way to represent these formally is to convert them into tactics.[3] This section presents some tactics which might be used to give a demonstration of the correctness of Lemma 5.4.4 (for the cut-free language, but using the basic rules involving *cut* to accomplish right-distribution for sequential tactics).

The Laws 5.1–5.18, will be the basic rules of the instance of the tactic language used here. They will be referred-to by (hopefully obvious) names, rather than numbers, for ease of reading.

[2] One can imagine that a 'unique instances' operator—used like *cut*, but producing a list of alternatives which were all different, might be a useful addition to the tactic language. We might then expect to be able to prove *uniq assocrec* = *uniq assocrec'*. The value of such an operator would depend on the relative efficiencies of suitably filtering the list of outcomes, and of applying the subsequent tactics more often than would otherwise be necessary.

[3] Edsger Dijkstra has advocated using computational techniques for proving a range of mathematical theorems. See, for example [Dij94].

Into Normal Form

The following tactic converts an arbitrary tactic into (cut-free) normal form.

$$
\begin{array}{l}
\mathit{norm} = \mu X \bullet \\
\qquad \pi \bullet \mathbf{skip} \longrightarrow \mathbf{skip} \\
\quad \mid \ \pi \bullet \mathbf{fail} \longrightarrow \mathbf{skip} \\
\quad \mid \ \pi\, r \bullet \mathbf{rule}\, r \longrightarrow \mathbf{skip} \\
\quad \mid \ \pi\, t_1, t_2 \bullet t_1 \mid t_2 \longrightarrow (X \boxed{\mid} X) ; ! \left(\begin{array}{l} \mathbf{rule}\, \mathit{failida} \\ \mid \mathbf{rule}\, \mathit{failidb} \\ \mid \mathbf{skip} \end{array} \right) \\
\quad \mid \ \pi\, t_1, t_2 \bullet t_1 \,;\, t_2 \longrightarrow (X \boxed{;} X) ; ! \left(\begin{array}{l} \mathbf{rule}\, \mathit{failzeroa} \\ \mid \mathbf{rule}\, \mathit{failzerob} \\ \mid \mathbf{rule}\, \mathit{skipida} \\ \mid \mathbf{rule}\, \mathit{skipidb} \\ \mid \mathbf{rule}\, \mathit{ldistr} \,;\, X \\ \mid \mathit{rdistrtac} \,;\, X \\ \mid \mathbf{skip} \end{array} \right)
\end{array}
$$

All of the components of this tactic are in the set of basic laws, except the tactic *rdistrtac*, which applies the left distributive law (5.11) if possible (that is, if the term on the right-hand side of the sequential composition can be replaced by the cut version of itself; i.e. if it is sequential).

$$
\begin{array}{l}
\mathit{rdistrtac} = \ (\mathit{makecut} \boxed{;} \mathbf{skip}); \\
\qquad\qquad \mathbf{rule}\, \mathit{rdistr}; \\
\qquad\qquad (\mathit{unmakecut} \boxed{;} \mathbf{skip}) \ .
\end{array}
$$

Tactics *makecut* and *unmakecut* add cuts to those tactics which may be cut without changing their meaning—the *sequential* tactics. This corresponds to the proof in Lemma 5.6.2.

$$
\begin{array}{rcl}
\mathit{makecut} & = & \pi \bullet \mathbf{skip} \longrightarrow \mathbf{rule}\, \mathit{cutskip} \\
& & \mid \pi \bullet \mathbf{fail} \longrightarrow \mathbf{rule}\, \mathit{cutfail} \\
& & \mid \pi\, r \bullet \mathbf{rule}\, r \longrightarrow \mathbf{rule}\, \mathit{cutrule} \\
& & \mid \pi\, t_1, t_2 \bullet t_1 \,;\, t_2 \longrightarrow \ (\mathit{makecut} \boxed{;} \mathit{makecut}); \\
& & \qquad\qquad\qquad\qquad (\mathbf{rule}\, \mathit{cutseq}') \\
\mathit{unmakecut} & = & \pi \bullet !\mathbf{skip} \longrightarrow \mathbf{rule}\, \mathit{cutskip} \\
& & \mid \pi \bullet !\mathbf{fail} \longrightarrow \mathbf{rule}\, \mathit{cutfail} \\
& & \mid \pi\, r \bullet !\, \mathbf{rule}\, r \longrightarrow \mathbf{rule}\, \mathit{cutrule} \\
& & \mid \pi\, t_1, t_2 \bullet !(t_1 \,;\, t_2) \longrightarrow \mathbf{rule}\, \mathit{cutseq}'; \\
& & \qquad\qquad\qquad\qquad (\mathit{unmakecut} \boxed{;} \mathit{unmakecut}) \ .
\end{array}
$$

After the main tactic has executed, it is necessary to move the brackets so that all of the ';'s and '|'s are associated to the left. This is readily accomplished by the tactic *norm* of Section 5.2. Here we shall use instantiations of this tactic as *leftassocseq* (to associate sequential compositions to the left) and as *leftassocalt* (to do the same for alternations).

Since a tactic in normal form will consist of (potentially) many sequential compositions separated by alternations, a recursive version of *leftassocseq* is needed:

$$
\mathit{recleftassocseq} = (\mathit{recleftassocseq} \boxed{\mid} \mathit{leftassocseq}) \mid \mathit{leftassocseq} \ .
$$

The over-all normalisation is therefore

$$norm' = norm\,;\, leftassocalt\,;\, !recleftassocseq\ .$$

Checking Normal Form

Another tactic can be defined to *check* that its goal is in normal form. It fails if the tactic is not in normal form, and behaves like **skip** otherwise.

First, some tactics which recognise the atomic terms:

$$\begin{aligned} isskip &= \pi \bullet \mathbf{skip} \longrightarrow \mathbf{skip} \\ isrule &= \pi r \bullet \mathbf{rule}\, r \longrightarrow \mathbf{skip} \\ isfail &= \pi \bullet \mathbf{fail} \longrightarrow \mathbf{skip}\ . \end{aligned}$$

Then recall that a tactic is in sequential form if it is **skip**, or it is an atomic rule, or it is the sequential composition of a non-**skip** sequential tactic and an atomic rule:

$$\begin{aligned} isseq &= isskip \mid isrule \mid (isseq' \boxed{;} isrule) \\ isseq' &= isrule \mid (isseq' \boxed{;} isrule)\ . \end{aligned}$$

Finally, a tactic is in normal form whenever it is **fail**, or it is sequential, or it is the alternation of a tactic in normal form with a sequential tactic:

$$\begin{aligned} isnormal &= isfail \mid isseq \mid (isnormal' \boxed{|} isseq) \\ isnormal' &= isseq \mid (isnormal' \boxed{|} isseq)\ . \end{aligned}$$

Proof

The goal of this subsection is to prove that the sequential composition

$$!norm'\,;\, isnormal$$

always succeeds, and hence that *norm'* always produces a tactic in normal form.

The termination of these tactics is the first thing to prove. *norm* must terminate, since each recursion is guarded by an operation which strictly decreases the number of |s in the scope of a ;. *norm'* terminates because *norm* does, and because *lass* must terminate on finite terms (at each recursion, it is applied to a smaller portion of the initial goal). Similar comments apply to *isnormal*.

The *successful* termination of the tactic above may be approached by defining another normality-checking tactic; one which does not expect the operators to be left-associated:

$$\begin{aligned} isseqish &= isskip \mid isrule \mid (isseqish' \boxed{;} isseqish') \\ isseqish' &= isrule \mid (isseqish' \boxed{;} isseqish') \\ isnormish &= isfail \mid isseqish \mid (isnormish' \boxed{|} isnormish) \\ isnormish' &= isseqish \mid (isnormish' \boxed{|} isnormish')\ . \end{aligned}$$

Having made these definitions, we are in a position to prove that $succs(norm'\,;\, isnormal) = \mathbf{skip}$. The proof will depend on two lemmas:

Lemma 6.2.1 $succs(isnormish) = succs(leftassocalt\ ;\ !recleftassocseq\ ;\ isnormal)$

Proof: (outline). We have, by Law 5.26 and the definition of *isnormish*,

$$succs(isnormish) = !\left(\begin{array}{l} succs\ isfail \\ |succs\ isseqish \\ |succs(isnormish' \boxed{|} isnormish') \end{array}\right)$$

and, by the definition of *isnormal*, Law 5.26, and Law 5.11:

$$succs(leftassocalt\ ;\ !recleftassocseq\ ;\ isnormal) = \\ !\left(\begin{array}{l} succs(leftassocalt\ ;\ !recleftassocseq\ ;\ isfail) \\ |succs(leftassocalt\ ;\ !recleftassocseq\ ;\ isseq) \\ |succs(leftassocalt\ ;\ !recleftassocseq\ ;\ (isnormal' \boxed{|} isseq)) \end{array}\right) .$$

It suffices to show that the corresponding branches of the above terms are equal. For the first branch we must show that:

$$succs\ isfail = succs(leftassocalt\ ;\ !recleftassocseq\ ;\ isfail) \ .$$

This is clear: if the goal is **fail** then *leftassocalt* and *recleftassocseq* will both skip; if it is not, then no amount of associating will make it so.

The second branch requires:

$$succs\ isseqish = succs(leftassocalt\ ;\ !recleftassocseq\ ;\ isseq) \ .$$

Expanding the definition of *recleftassocseq*, and using Law 5.24 to remove the *cut* around it, and the distributive laws, the right-hand side can be rewritten to

$$succs((leftassocalt\ ;\ (recleftassocseq \boxed{|} leftassocseq)\ ;\ isseq) \\ |(leftassocalt\ ;\ fails(recleftassocseq \boxed{|} leftassocseq)\ ;\ leftassocseq\ ;\ isseq))$$

The first of these branches is identically *fail* (if the structural combinator succeeds, then *isseq* fails, and vice versa). Since *recleftassocseq* and *leftassocseq* never fail, $(recleftassocseq \boxed{|} leftassocseq)$ fails if and only if **skip** $\boxed{|}$ **skip** does. $fails($**skip** $\boxed{|}$ **skip**$)$ is equivalent to *isseqish*, and *leftassocalt* never fails, so the problem reduces to demonstrating

$$isseqish = succs(isseqish\ ;\ leftassocseq\ ;\ isseq)$$

which may be shown by expanding the definition of *leftassocseq*.
The third branch needs:

$$succs(isnormish' \boxed{|} isnormish') = \\ succs(leftassocalt\ ;\ !recleftassocseq\ ;\ (isnormal' \boxed{|} isseq)) \ .$$

This proceeds along similar lines, but is more complex. □

Lemma 6.2.2 $succs(!norm\ ;\ isnormish) =$ **skip**

Proof: (outline). Since *norm* always succeeds, it suffices to show that $!norm;isnormish$ succeeds whenever *norm* does. Observe that $!norm = norm$. A proof by structural induction over the possible forms of goals is appropriate.

We have, by the definition of *norm*, and Law 5.26

$$
\begin{array}{l}
succs(!norm\,;\,isnormish) \\
= !(succs(\pi \bullet \mathbf{skip} \longrightarrow isnormish) \\
\quad succs(\pi \bullet \mathbf{fail} \longrightarrow isnormish) \\
\quad succs(\pi\, r \bullet \mathbf{rule}\, r \longrightarrow isnormish) \\
\quad succs(\pi\, t_1, t_2 \bullet t_1 \mid t_2 \longrightarrow \\
\qquad\qquad (norm \,\boxed{|}\, norm)\,;\,! \left(\begin{array}{l} \mathbf{rule}\, failida \\ |\,\mathbf{rule}\, failidb \\ |\mathbf{skip} \end{array} \right) ;\, isnormish) \\
\quad succs(\pi\, t_1, t_2 \bullet t_1\,;\, t_2 \longrightarrow \\
\qquad\qquad (norm \,\boxed{;}\, norm)\,;\,! \left(\begin{array}{l} \mathbf{rule}\, failzeroa \\ |\,\mathbf{rule}\, failzerob \\ |\,\mathbf{rule}\, skipida \\ |\,\mathbf{rule}\, skipidb \\ |\,\mathbf{rule}\, ldistr\,;\,norm \\ |rdistrtac\,;\,norm \\ |\mathbf{skip} \end{array} \right) ;\, isnormish) \\
)
\end{array}
$$

The first three branches (base cases) are immediately equivalent to

$$succs(\pi \bullet \mathbf{skip} \longrightarrow \mathbf{skip})$$

etc. The fourth may be rewritten, using Law 5.24 and the distributive laws, as

$$
\begin{array}{l}
succs(\pi\, t_1, t_2 \bullet t_1 \mid t_2 \longrightarrow \\
\left(\begin{array}{l} (norm \,\boxed{|}\, norm)\,;\,\mathbf{rule}\, failida\,;\, isnormish \\ |(norm \,\boxed{|}\, norm)\,;\,\mathbf{rule}\, failidb\,;\, isnormish \\ |(norm \,\boxed{|}\, norm)\,;\, fails(\mathbf{rule}\, foilida)\,;\, fails(\mathbf{rule}\, failidb)\,;\, isnormish \end{array} \right))
\end{array}
$$

Now, we have, as a property of *failida*, that

$$
\begin{array}{l}
succs(\mathbf{rule}\, failida\,;\, isnormish) \\
= succs((isnormish \,\boxed{|}\, isnormish)\,;\,\mathbf{rule}\, failida)
\end{array}
$$

and similar properties for the other two branches. Therefore, by Law 5.42, the abides law (5.89) and induction, we have that the fourth branch succeeds exactly when that in *norm* does.

The fifth branch may be treated similarly. The instances of $succs(norm\,;isnormish)$ which arise in the distributive cases can be asserted as being equivalent to *succs norm*, again by induction. □

Theorem 6.2.3 $succs(norm'\,;\, isnormal) = \mathbf{skip}$

Proof:

$$
\begin{aligned}
& \mathit{succs}(\mathit{norm} \,;\, \mathit{isnormish}) = \mathbf{skip} && \text{Lemma 6.2.2} \\
\Rightarrow\ & \mathit{succs}(\mathit{norm} \,;\, \mathit{succsisnormish}) = \mathbf{skip} && \text{Law 5.42} \\
\Rightarrow\ & \mathit{succs}(\mathit{norm} \,;\, \mathit{succs}(\mathit{leftassocalt} \,;\, !\mathit{recleftassocseq} \,;\, \mathit{isnormal})) = \mathbf{skip} && \text{Lemma 6.2.1} \\
\Rightarrow\ & \mathit{succs}(\mathit{norm} \,;\, \mathit{leftassocalt} \,;\, !\mathit{recleftassocseq} \,;\, \mathit{isnormal}) = \mathbf{skip} && \text{Law 5.42} \\
\Rightarrow\ & \mathit{succs}(\mathit{norm}' \,;\, \mathit{isnormal}) = \mathbf{skip} && \text{Definition of } \mathit{norm}
\end{aligned}
$$

□

6.3 Lifting

One of the most interesting features of the encoding of $\mathcal{W}$ in 2OBJ in Chapter 2 was the meta-rule described as rule-lifting. Rule-lifting enables the simple inference rules to be presented without reference to the unchanging parts of a goal, thus simplifying the presentation, and collecting most of the provisos regarding free variables, into one place: the meta-rule *lift*.

In the 2OBJ frame this rule was implemented as a function from rules to rules, and extended to *tactics* by the use of a 2OBJ primitive which converted a proof (produced using a tactic) into a rule (which could then be applied within the 'real' proof tree). In the new scheme of things, however, tactics may backtrack (i.e. produce alternatives), whereas rules may not.

As a result, two different versions of tactic-lifting are presented here. The first implements tactic-lifting as a special structural combinator. This will be taken as the working definition in the laws which follow. The second approach is to implement tactic-lifting combined with *cut*—so that the approach taken in 2OBJ (above) can be used.

An unsatisfactory feature of the 2OBJ encoding is the means by which terms are *selected* for lifting. This was accomplished by presenting three sequences of numbers to the meta-rule. These denoted positions of predicates (declarations) in the respective predicate (declaration) lists to which the rule was to be applied. This tended to restrict unnecessarily the way in which lifting was used, and prevented general algebraic laws about lifting from being established.

The presentation here is generic over selection schemes. The relevant tacticals will take a selection component s, and apply it to a goal g such that $g \uparrow s$ denotes the goal formed by selecting certain components from g, and $g \downarrow s$ denotes the goal formed by excluding those components from g. A function $\updownarrow$ is a partial inverse for these, so that $(g \uparrow s) \updownarrow (g \downarrow s)$ has the same semantic value as g—though the terms may appear in a different order. In some selection schemes, $\updownarrow$ will be a true inverse, permitting *lift* to satisfy rather more laws than when it is not. (In fact, the requirement is that $(g_1 \updownarrow_s g_2) \uparrow s = g_1$ and $(g_1 \updownarrow_s g_2) \downarrow s = g_2$. Even if $\updownarrow_s$ is carefully defined, this holds only for very well-behaved tactics, and so, in general, we do not have that $\mathit{comb} \circ \mathit{brk} = \mathrm{id}$.)

Tactic-lifting is essentially similar to applying a structural combinator—some parts of the goal are selected, and have the lifted tactic applied to them; other parts are not.

and have **skip** applied to them.[4]

$$
\begin{aligned}
\mathit{lift}\ s\ t &= \mathit{comb}* \circ (\mathit{filter\_cross}\langle t, \mathbf{skip}\rangle) \circ (\mathit{brk}\ s) \\
\mathit{brk}\ s\ g &= \langle g \uparrow s, g \downarrow s\rangle \\
\mathit{comb}\langle g_1, g_2\rangle &= g_1 \updownarrow g_2 \\
\mathit{comb}\langle g_1 \parallel g_2, g_3\rangle &= \mathit{comb}\langle g_1, g_3\rangle \parallel \mathit{comb}\langle g_2, g_3\rangle \\
\mathit{comb}\langle g, ()\rangle &= () \\
\mathit{comb}\langle (), g\rangle &= ()
\end{aligned}
$$

In place of the usual *cross* in this definition, a *filtered* version is needed, as lifting is subject to a side-condition. *filter_cross* returns only those goal alternatives which satisfy the side-condition.[5]

$$
\begin{aligned}
\mathit{filter\_cross}\ \langle t_1, t_2\rangle\ \langle g_1, g_2\rangle &= ((\mathit{filt}_{\langle g_1, g_2\rangle}) \circ \mathit{cross})\langle g_1, g_2\rangle \\
\mathit{filt}_{\langle g_1, g_2\rangle} &= \frown/ \circ \mathit{ok}_{\langle g_1, g_2\rangle}* \\
\mathit{ok}_{\langle g_1, g_2\rangle}\langle h_1, h_2\rangle &= \mathbf{if} \frown/(\mathit{proviso}* h_1) = h_1 \\
&\quad\ \mathbf{then}\langle\langle h_1, h_2\rangle\rangle \\
&\quad\ \mathbf{else}\langle\ \rangle \\
\mathit{proviso}\ g &= \mathbf{if}(\alpha(\mathit{decls}\ g_1) \cup \alpha(\mathit{decls}\ g)) \cap \\
&\qquad\qquad \phi(\mathit{preds}\ g2) = \varnothing \\
&\quad\ \mathbf{then}\langle g\rangle \\
&\quad\ \mathbf{else}\langle\ \rangle
\end{aligned}
$$

An alternative approach to lifting is to define it in the same way as was done in the 2OBJ version; as a function acting upon rules.

$$
\begin{aligned}
\mathit{liftrule}\ s\ r\ g &= (\mathit{combine\_sequent}(g \downarrow s, g \uparrow s))* (r\ (g \uparrow s)) \\
\mathit{combine\_sequent}\ (g_1, g_2)\ x &= \mathbf{if}(\alpha(\mathit{decls}\ g_2) \cup \alpha(\mathit{decls}\ x)) \\
&\qquad\qquad \cap\ \phi(\mathit{preds}\ g_1) = \varnothing \\
&\quad\ \mathbf{then}(g_1 \updownarrow x) \\
&\quad\ \mathbf{else}\langle\ \rangle
\end{aligned}
$$

As previously, this definition can be used to define a version of lifting for tactics—using a function *tacrule*, which takes a tactic and returns a rule which behaves like the *cut* version of the tactic.

$$\mathit{lifttactic}\ s\ t = \mathbf{rule}(\mathit{liftrule}\ s\ (\mathit{tacrule}\ t))$$

(The precise details of the definition of *tacrule* are an artifact of the implementation details of rules, tactics and goals. Converting a tactic into a rule may seem to be a dangerous activity from soundness point of view, but since a tactic may only make sound inferences, there is no problem with regarding it as a rule.)

Laws

Law 6.2 $\quad \mathit{lift}\ s\ (t_1 \mid t_2) = \mathit{lift}\ s\ t_1 \mid \mathit{lift}\ s\ t_2$

[4] Observe that a more general form of lifting could be defined—many rules could be applied in parallel, to differing parts of the sequent, with the proviso being checked only once, at the end.

[5] *filter_cross* could be defined using the functional programming *filter*, but the definition here avoids treating predicates as functions—this definition comes from an idea in [Bir86]; it is more in keeping with a Z style of doing things.

Proof: Let (f, g) be the function which, when presented with a 2-tuple applies f to the first argument and g to the second (i.e. formally, $(f, g) = \nabla(f \circ \pi_1, g \circ \pi_2)$, where π_1 and π_2 are projection functions).

$$
\begin{aligned}
& \mathit{lift}\ s\ (t_1 \mid t_2) \\
&= \mathit{comb}* \circ \mathit{filter\_cross}\langle t_1 \mid t_2, \mathbf{skip}\rangle \circ \mathit{brk}\ s && \text{Definition of } \mathit{lift} \\
&= \mathit{comb}* \circ \mathit{filt} \circ \mathit{cross}\langle t_1 \mid t_2, \mathbf{skip}\rangle \circ \mathit{brk}\ s && \text{Definition of } \mathit{filter\_cross} \\
&= \mathit{comb}* \circ \mathit{filt} \circ \mathit{cross}\big\langle \frown\!/ \circ \langle t_1, t_2\rangle^\circ, \mathbf{skip}\big\rangle \circ \mathit{brk}\ s && \text{Definition of } \mid \\
&= \mathit{comb}* \circ \mathit{filt} \circ \frown\!/ \circ \big\langle \mathit{cross}\langle t_1, \mathbf{skip}\rangle, \mathit{cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ \circ \mathit{brk}\ s \\
& && \text{Lemma (below)} \\
&= \mathit{comb}* \circ \frown\!/ \circ \Upsilon_{\mathrm{id}}\ \langle \mathit{filt}, \mathit{filt}\rangle \circ \big\langle \mathit{cross}\langle t_1, \mathbf{skip}\rangle, \mathit{cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ \circ \mathit{brk}\ s \\
& && \text{Property of } \Upsilon_{\mathrm{id}}, \frown \text{ and } \mathit{filt} \\
&= \mathit{comb}* \circ \frown\!/ \circ \big\langle \mathit{filt} \circ \mathit{cross}\langle t_1, \mathbf{skip}\rangle, \mathit{filt} \circ \mathit{cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ \circ \mathit{brk}\ s \\
& && \text{Property of } \Upsilon_{\mathrm{id}} \text{ and } ^\circ \\
&= \mathit{comb}* \circ \frown\!/ \circ \big\langle \mathit{filter\_cross}\langle t_1, \mathbf{skip}\rangle, \mathit{filter\_cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ \circ \mathit{brk}\ s \\
& && \text{Definition of } \mathit{filter\_cross} \\
&= \frown\!/ \circ \langle \mathit{comb}*, \mathit{comb}*\rangle \circ \\
&\qquad \big\langle \mathit{filter\_cross}\langle t_1, \mathbf{skip}\rangle, \mathit{filter\_cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ \circ \mathit{brk}\ s \\
& && \text{Property of } \Upsilon_{\mathrm{id}}, \frown \text{ and } * \\
&= \frown\!/ \circ \big\langle \mathit{comb}* \circ \mathit{filter\_cross}\langle t_1, \mathbf{skip}\rangle, \\
&\qquad \mathit{comb}* \circ \mathit{filter\_cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ \circ \mathit{brk}\ s && \text{Property of } ^\circ \text{ and } \Upsilon_{\mathrm{id}} \\
&= \frown\!/ \circ \big\langle \mathit{comb}* \circ \mathit{filter\_cross}\langle t_1, \mathbf{skip}\rangle \circ \mathit{brk}\ s, \\
&\qquad \mathit{comb}* \circ \mathit{filter\_cross}\langle t_2, \mathbf{skip}\rangle \circ \mathit{brk}\ s\big\rangle^\circ && \text{Property of } ^\circ \\
&= \mathit{lift}\ s\ t_1 \mid \mathit{lift}\ s\ t_2 && \text{Definition of } \mid
\end{aligned}
$$

The Lemma referred-to above is the property

$$\frown\!/ \circ \big\langle \mathit{cross}\langle t_1, \mathbf{skip}\rangle, \mathit{cross}\langle t_2, \mathbf{skip}\rangle\big\rangle^\circ = \mathit{cross}\langle \frown\!/ \circ \langle t_1, t_2\rangle^\circ, \mathbf{skip}\rangle\ ,$$

which follows from the distributivity of $\prod$ over $\frown$ in the first argument. □

Law 6.3 $\mathit{lift}\ s\ (t_1\ ;\ t_2) = \mathit{lift}\ s\ t_1\ ;\ \mathit{lift}\ s\ t_2$
provided $\mathit{brk} \circ \mathit{comb}\ s = \mathrm{id}$ and for all g, $\mathit{decls}(t, g) = \mathit{decls}\ g$

Proof: (outline) $\mathit{lift}\ s\ t$ is a special instance of a structural combinator $t \oplus_s \mathbf{skip}$. If this behaves like any typical structural combinator, the proof is immediate:

$$
\begin{aligned}
& (t_1 \oplus_s \mathbf{skip})\ ;\ (t_2 \oplus_s \mathbf{skip}) \\
&= (t_1\ ;\ t_2) \oplus_s (\mathbf{skip}\ ;\ \mathbf{skip}) && \text{Law 5.89} \\
&= (t_1\ ;\ t_2) \oplus_s \mathbf{skip} && \text{Law 5.1}
\end{aligned}
$$

The proof of the abides property (Law 5.89) is contingent upon the property $\mathit{comb} \circ \mathit{brk} = \mathrm{id}$ and upon the 'key lemma', which, for $\oplus$,**skip** will be

$$\frown\!/ \circ \big(\mathit{filter\_cross}\langle t_2, \mathbf{skip}\rangle\big)* \circ \mathit{filter\_cross}\langle t_1, \mathbf{skip}\rangle = \mathit{filter\_cross}\langle \frown\!/ \circ t_2* \circ t_1, \mathbf{skip}\rangle$$

The former is covered by the first proviso in the statement of the law; the latter is proved in the same way as in Section 5.10, provided the instances of *filt* do not interfere—the second proviso is sufficient to guarantee this. □

Law 6.4 $\mathit{lift}\ s\ (\mathit{lift}\ s\ t) = \mathit{lift}\ s\ t$ provided $(g \uparrow s) \uparrow s = g \uparrow s$ and $(g \uparrow s) \downarrow s = (\vdash)$

Proof:

$$
\begin{array}{ll}
& \mathit{lift}\ s\ (\mathit{lift}\ s\ t)g \\
= & \big(\mathit{comb}* \circ \mathit{filter\_cross}\langle \mathit{lift}\ s\ t, \mathbf{skip}\rangle \circ \mathit{brk}\big)\ g & \text{Definition of } \mathit{lift} \\
= & \big(\mathit{comb}* \circ \mathit{filter\_cross}\langle \mathit{lift}\ s\ t, \mathbf{skip}\rangle\big)\ \langle g \uparrow s, g \downarrow s\rangle & \text{Definition of } \mathit{brk}\ s \\
= & \big(\mathit{comb}* \circ \mathit{filt} \circ \textstyle\prod\big)\langle \mathit{lift}\ s\ t\ (g \uparrow s), \langle g \downarrow s\rangle\big\rangle & \text{Definition of } \mathit{filter\_cross}, \text{ etc.} \\
= & \big(\mathit{comb}* \circ \mathit{filt} \circ \textstyle\prod\big) \\
& \quad \big\langle\big(\mathit{comb}* \circ \mathit{filter\_cross}\langle t, \mathbf{skip}\rangle\big)\langle (g \uparrow s) \uparrow s, (g \uparrow s) \downarrow s\rangle, \langle g \downarrow s\rangle\big\rangle \\
& & \text{Definition of } \mathit{lift}, \text{ etc.} \\
= & \big(\mathit{comb}* \circ \mathit{filt} \circ \textstyle\prod\big) \\
& \quad \big\langle\big(\mathit{comb}* \circ \mathit{filter\_cross}\langle t, \mathbf{skip}\rangle\big)\langle g \uparrow s, (\vdash)\rangle, \langle g \downarrow s\rangle\big\rangle & \text{Proviso} \\
= & \big(\mathit{comb}* \circ \mathit{filt} \circ \textstyle\prod\big)\big\langle t(g \uparrow s), \langle g \downarrow s\rangle\big\rangle & \text{Lemma} \\
= & \big(\mathit{comb}* \circ \mathit{filter\_cross}\langle t, \mathbf{skip}\rangle\big)\big\langle g \uparrow s, g \downarrow s\big\rangle \\
& & \text{Definition of } \mathit{filter\_cross}, \text{ etc.} \\
= & \big(\mathit{comb}* \circ \mathit{filter\_cross}\langle t, \mathbf{skip}\rangle \circ \mathit{brk}\ s\big)g & \text{Definition of } \mathit{brk}\ s \\
= & \mathit{lift}\ s\ t\ g & \text{Definition of } \mathit{lift}
\end{array}
$$

The lemma is

$$\mathit{comb}* \circ \mathit{filter\_cross}\langle t, \mathbf{skip}\rangle\langle h, (\vdash)\rangle = t\ h\ ,$$

which is proved by simple properties of *comb* (noting that $\mathbf{skip}(\vdash) = \langle(\vdash)\rangle$, that the filter condition is necessarily true in this case, and that $\mathit{comb}\langle h, (\vdash)\rangle = h \updownarrow (\vdash) = h$). □

Law 6.5 $\mathit{lift}\ s(t_1\ ;\ \mathit{lift}\ s\ t_2) = \mathit{lift}\ s\ t_1\ ;\ \mathit{lift}\ s\ t_2$
provided s satisfies the provisos for Laws 6.3 and 6.4

Proof:

$$
\begin{array}{ll}
& \mathit{lift}\ s(t_1\ ;\ \mathit{lift}\ s\ t_2) \\
= & \mathit{lift}\ s\ t_1\ ;\ \mathit{lift}\ s\ (\mathit{lift}\ s\ t_2) & \text{Law 6.3} \\
= & \mathit{lift}\ s\ t_1\ ;\ \mathit{lift}\ s\ t_2 & \text{Law 6.4}
\end{array}
$$

□

6.4 Propositional Calculus, again

We are now in a position to return to the decision procedure for the propositional calculus in JigsaW, described in Section 2.9. This section shows how to define this tactic in *Angel*—a process which immediately revealed to the author an efficiency-improving separation of concerns, detailed below.

First Approach

Recall that the fundamental propositional calculus tactic is *proptac*, which applies whichever propositional tactic is appropriate. It is defined using a distributed version of |, called *first*,[6] after a similar tactic in LCF.

$$
\begin{aligned}
\mathit{first}\ \langle\,\rangle &= \mathbf{fail} \\
\mathit{first}\ (x{:}xs) &= x \mid \mathit{first}\ xs \\
\mathit{proptac} &= \mathit{first}(\ \mathbf{rule}\ \mathit{truth}, \\
&\qquad \mathbf{rule}\ \mathit{contradiction}, \\
&\qquad \mathbf{rule}\ \mathit{negationL}, \\
&\qquad \mathbf{rule}\ \mathit{negationR}, \\
&\qquad \mathbf{rule}\ \mathit{conjunctionL}, \\
&\qquad \mathbf{rule}\ \mathit{conjunctionR}, \\
&\qquad \mathbf{rule}\ \mathit{disjunctionL}, \\
&\qquad \mathbf{rule}\ \mathit{disjunctionR}, \\
&\qquad \mathbf{rule}\ \mathit{implicationL}\ ;\ (\mathit{liftLfirst}\ (\mathbf{rule}\ \mathit{thin}) \parallel \mathbf{skip}), \\
&\qquad \mathbf{rule}\ \mathit{implicationR}, \\
&\qquad \mathbf{rule}\ \mathit{equivalenceL}, \\
&\qquad \mathbf{rule}\ \mathit{equivalenceR}\ )
\end{aligned}
$$

The addition after *implicationL* is due to the fact that in v1.0 of the Z Base Standard [BN+92], this rule leaves the implication in its first premiss. Using a lifting selection scheme like that chosen in Jigsaw, we define $\mathit{liftLfirst} = \mathit{lift}(\langle\,\rangle, \langle 0 \rangle, \langle\,\rangle)$.

Having made this definition, it simply remains to define a tactic which can apply something of this form to every predicate in a sequent.

$$
\begin{aligned}
\mathit{toeach}\ t &= \big(\pi\, d, \Psi, \Phi \bullet (d \mid \Psi \vdash \Phi) \longrightarrow (\mathit{toeachL}\ t)^{\#\Psi}\ ;\ (\mathit{toeachR}\ t)^{\#\Phi}\big) \\
\mathit{toeachL}\ t &= \mathit{liftLlast}\big(\mathit{try}(t\ ;\ \mathit{toeach}\ t)\big) \\
\mathit{toeachR}\ t &= \mathit{liftRlast}\big(\mathit{try}(t\ ;\ \mathit{toeach}\ t)\big)
\end{aligned}
$$

As before, the special lifts, (*liftLlast* and *liftRlast*) are derived from *lift*, using selectors which return the sequent containing only the rightmost predicate of the antecedent (consequent) of the original sequent.

Having made these definitions, *mprop*, which applies propositional tactics exhaustively, is simply *toeach proptac*.

An alternative approach is to write a tactic which is not itself recursive—that is, it is more like *repeat* (i.e. t^*)—but is applied exhaustively.

$$
\begin{aligned}
\mathit{tryeachR}\ t &= \big(\pi\, d, \Psi, \Phi \bullet (d \mid \Psi \vdash \Phi) \longrightarrow \mathit{srepR}(\#\Phi, t)\big) \\
\mathit{srepR}(0, t) &= \mathit{failtac} \\
\mathit{srepR}(1, t) &= \mathit{liftRlast}\ t \\
\mathit{srepR}(n+1, t) &= \mathit{liftRlast}\ t \mid \big(\mathit{liftRlast}\mathbf{skip}\ ;\ \mathit{srepR}(n, t)\big) \\
\mathit{tryeachL}\ t &= \big(\pi\, d, \Psi, \Phi \bullet (d \mid \Psi \vdash \Phi) \longrightarrow \mathit{srepL}(\#\Psi, t)\big) \\
\mathit{srepL}(0, t) &= \mathit{failtac} \\
\mathit{srepL}(1, t) &= \mathit{liftLlast}\ t \\
\mathit{srepL}(n+1, t) &= \mathit{liftLlast}\ t \mid \big(\mathit{liftLlast}\mathbf{skip}\ ;\ \mathit{srepL}(n, t)\big) \\
\mathit{mprop2} &= \mathit{exhaust}(\mathit{tryeachL}\ \mathit{proptac} \mid \mathit{tryeachR}\ \mathit{proptac})
\end{aligned}
$$

[6] The name is now inaccurate since it will apply as many of the alternatives as possible.

The relationship between *mprop* and *mprop2* is straightforward—the former is a depth-first search (each application of *proptac* is followed by a recursive instance of *toeach proptac*) whereas the latter is a breadth-first search (when an application of *proptac* succeeds, its result is returned to the main goal sequent, which is processed in a cyclic manner; so after all of the original predicates have been tried, those which were produced by applications of *proptac* become inputs to *proptac* again).

As such, both tactics may be expected to produce the same result (with all propositional connectives removed)—but with the alternative goals (less complete reductions) in differing orders. Hence we postulate that

$$!mprop = !mprop2$$

but the complexity of the terms precludes arguments even at the informal level of the previous section.

Greater Efficiency

In examining these definitions, it became clear that the full generality of *proptac* is inappropriate. Whenever *proptac* is applied, it is already apparent whether it is being applied on the left- or the right-hand side of a sequent. Hence, we may partition *proptac* into two tactics:

$$\begin{aligned}
proptacL \;=\; first\,\langle\; & \textbf{rule}\; contradiction,\\
& \textbf{rule}\; negationL,\\
& \textbf{rule}\; conjunctionL,\\
& \textbf{rule}\; disjunctionL,\\
& \textbf{rule}\; implicationL\,;\,(liftLfirst\;(\textbf{rule}\; thin)\parallel \textbf{skip}),\\
& \textbf{rule}\; equivalenceL\;\rangle\\
proptacR \;=\; first\,\langle\; & \textbf{rule}\; truth,\\
& \textbf{rule}\; negationR,\\
& \textbf{rule}\; conjunctionR,\\
& \textbf{rule}\; disjunctionR,\\
& \textbf{rule}\; implicationR,\\
& \textbf{rule}\; equivolenceR\;\rangle\;.
\end{aligned}$$

We may then define a new version of *toeach*:

$$\begin{aligned}
toeach'\;(t_L,t_R) \;&=\; \big(\pi\, d, \Psi, \Phi \bullet (d \mid \Psi \vdash \Phi) \longrightarrow\\
&\qquad (toeachL\; t_L)^{\#\Psi}\;;\;(toeachR\; t_R)^{\#\Phi}\big)\\
toeachL\;(t_L,t_R) \;&=\; liftLlast\big(try(t_L\;;\;toeach'\;(t_L,t_R))\big)\\
toeachR\;(t_L,t_R) \;&=\; liftRlast\big(try(t_R\;;\;toeach'\;(t_L,t_R))\big)
\end{aligned}$$

and write $mprop = toeach'(proptacL, proptacR)$. In a similar vein, $mprop2 = exhaust(tryeachL\; proptacL \mid tryeachR\; proptacR)$.

These new tactics are, on average, some 25% faster than those previously presented.

massum

To complete the decision procedure which is made possible using *mprop* or *mprop2*, a tactical which tries to apply the *assumption* rule is needed. This can again be defined

using a more general tactic, which applies a tactic applicable to a pair of goals:

$$\begin{aligned} \mathit{topairs}\ t &= (\pi\, d, p, \Psi, \Phi \bullet (d \mid p, \Psi \vdash \Phi) \longrightarrow \\ &\qquad \mathit{lift}\ (\langle\,\rangle, \langle 0\rangle, \mathit{all})\ (\mathit{topairs}'\ t) \mid \mathit{thinR}\ 0\ ;\ \mathit{topairs}\ t) \\ \mathit{topairs}'\ t &= (\pi\, d, \Psi, q, \Phi \bullet (d \mid \Psi \vdash q, \Phi) \longrightarrow \\ &\qquad \mathit{lift}\ (\langle\,\rangle, \langle 0\rangle, \langle 0\rangle)\ t \mid \mathit{thinL}\ 0\ ;\ \mathit{topairs}'\ t) \\ \mathit{massum} &= \mathit{topairs}(\mathbf{rule}\ \mathit{assumption}) \end{aligned}$$

Decision Procedures

A sufficient decision procedure for propositional calculus is

$$\mathit{mprop}\ ;\ \mathit{massum}\ .$$

The first outcome of this tactic is success (with no goals) if the initial goal is a tautology; it fails otherwise. In general, a *much* more efficient tactic is

$$!\mathit{mprop}\ ;\ \mathit{massum}\ ,$$

since it is only the first outcome of *mprop* which is interesting; only the first alternative is a candidate for consideration by *massum*. Since the outcome of *mprop* is interesting whether or not *massum* is applicable, it will frequently be useful to use the tactic

$$!\mathit{mprop}\ ;\ \mathit{try}(\mathit{massum})\ .$$

These tactics are dependent on properties of selection (in order to cycle through the predicates, etc.). As such they will work only with the non-invertible forms of *lift*. This points to the desirability of defining two sorts of lifting (or rather, selection)—this one, and a *liftinplace* version—which obeys more laws, but is harder to use in tactics such as these.

6.5 Towards a Library of Tacticals

This chapter, together with the preceding one, has described a collection of tactieals; some particularly suited to sequent calculi like $\mathcal{W}$, some more general in their application:

try t attempts to apply *t*; if *t* fails to apply, it succeeds, leaving the goal unchanged.

lift s t applies *t* to a selection of predicates from the goal sequent.

t^n repeats tactic *t*, *n* times, failing if any of those iterations fails.

exhaust t repeats tactic *t* as many times as possible (which may be zero) and always succeeds (provided *t* fails eventually).

$exhaust_n\ t$ was not defined above, but behaves like *exhaust*, but with an upper limit (of *n*) on the number of times *t* is applied: $exhaust_0\ t = \mathbf{skip}$ and $exhaust_{n+1}\ t = t\ ;\ exhaust_n \mid \mathbf{skip}$.

toeach t applies tactic *t* exhaustively to each predicate (in depth-first fashion) in the goal sequent; terminating with success when no further points of applicability can be found.

tryeach t attempts to apply tactic *t* to each predicate in the goal in turn, terminating with success when that application is successful—or failing if there is no predicate to which *t* may be applied. *exhaust*(*tryeach t*) performs a breadth-first exhaustive application of *t*.

topairs t behaves like *tryeach t*, except that each attempt to apply *t* involves one antecedent and one consequent.

$assoc_{op,rule}$ generates associative instances of terms in *op*, by use of *rule*.

Many of these tacticals are also present in other theorem proving systems; *exhaust*, for example, is almost universal. The names differ, of course, and some systems use different means of accomplishing the same effect. In *Zola* (where a logic very much like *W* is used) some of the above are present, and the effect of others is accomplished via *tactic keywords* [Bla94]. These may be defined per-tactic, but some common ones are `addr` and `jump`, which target application on particular parts of the goal—akin to rule-lifting and structural combinators (the account in [Bla94] also highlights difficulties regarding the ordering of changing/unchanging predicates—similar to problems which arise in the discussion of inverses in $\uparrow$,$\downarrow$ and $\updownarrow$ (Section 6.3).

Chapter 7

Implementation using *Gofer*

SINCE THE TACTIC LANGUAGE described above is given a semantics using lazy lists, it lends itself to a simple implementation in a lazy funetional programming language. This chapter describes a lightweight approach to the creation of such an implementation, and outlines briefly a new prototype theorem-prover for Z, written in the *Gofer* language [Jon91a]—a dialect of *Haskell*. Initial results have been encouraging, with execution speed some two orders of magnitude better than that achieved in 2OBJ.

The first section of this tactic describes how the tactic language of Chapter 5 is implemented in *Gofer*. The following sections show how the encoding in 2OBJ may be re-cast in this scheme, and present a small case studv. This time, the deductive system used is not $\mathcal{W}$, but that of LittleZ [BHW94].

7.1 LittleZ

LittleZ is a sub-language of Z which contains the familiar constructs of set theory together with a typing structure following the style used in the typed lambda calculus. Stephen Brien's thesis [Bri95] describes LittleZ in some detail, and proposes a reasoning system for the language. That system is used here.

In contrast to $\mathcal{W}$, judgements in this logic are sequents with a *single* predicate as the consequent, and a list of LittleZ *paragraphs* (given sets ($[X]$), declarations ($x : X$), definitions ($x := e$), and predicates) as the antecedent. These paragraphs are separated by ‡, so a typical sequent might have the form

$$[X] \ddagger x, y : X \ddagger z := f\, x \ddagger \vdash p(z) \ .$$

The material in the antecedent should be understood as a Z specification, with each new paragraph in the scope of the previous ones.

7.2 Basic Tactic Interpreter

The core of the implementation is a *Gofer* program which implements directly the definitions of Section 5.3:

```
type TACTIC = GOAL  -> [GOAL]

skiptac g = [g]
failtac g = []

thentac t1 t2 = concat.(map t2).t1
elsetac t1 t2 = concat.(toevery [t1,t2])
cuttac t = head'.t
```

The implementation of **rule** is slightly more complicated, for two reasons. First, a means is needed for checking the side-condition $g \in \mathrm{dom}\, r$. This is accomplished by having the basic rules return members of a structured datatype—either `Fails`, for application outside of the rule's domain, or `Succs xs` where `xs` is a (possibly empty) list of subgoals:

```
data TAGGED = Succs [SEQUENT]
            | Fails
type RULE = SEQUENT -> TAGGED
```

rule can then be defined:

```
rule :: RULE -> TACTIC
rule r (Singl s) | r s /= Fails = [goalify t]
                 | otherwise    = []
                                  where Succs t = r s
```

Second, the relationship between SEQUENT and GOAL (suggested by the presence of `Singl s` and `goalify t` above) is that *rules* return lists of SEQUENTs, whereas *tactics* operate on GOALs—a compound structure, like that outlined in Section 5.12. The function `goalify` converts the former to the latter.[1]

```
data GOAL = Singl SEQUENT
          | Parcomp GOAL GOAL
          | Nogoal

goalify [x] = Singl x
goalify (x:y:ys) = Parcomp (Singl x) (goalify (y:ys))
goalify [] = Nogoal
```

Since such structured goals are present, we implement a structural combinator for applying tactics in parallel to parallel compositions of goals.

```
partac t1 t2 = (map combine_par).(cross [t1,t2]).(break_par)

combine_par [Nogoal,g2] = g2
combine_par [g1,Nogoal] = g1
combine_par [g1,g2] = Parcomp g1 g2

break_par (Parcomp g1 g2) = [g1,g2]
cross ts = cp.(zipwithapply ts)
```

[1] To need a conversion function is a little unfortunate. Of course, the two representations are isomorphic—but the presentation of the rules is more readable if they return lists. Nevertheless, conciseness would be improved by doing away with the lists of sequents. Conversely, if this were a description of a generic frame (rather than a lightweight, adaptable implementation) `goalify` would serve to make specific the embedding of the object logic (sequents) in the generic framework (GOALs).

Finally, there is the implementation of π to consider. Whilst a fully-general unification procedure could be written, it often suffices to make do with the pattern-matching in the underlying language. We define

```
pitac t g = (t g) g
```

and then implement $t_1 = (\pi\, \vec{v} \bullet g \longrightarrow t)$ as

```
t1 = pitac s
     where s g = t
           s _ = failtac
```

(where g is some suitable pattern with the variables of $\vec{v}$ free). This construction *does not support* backtracking (angelic choice) over the bound variable assignments—i.e. it is akin to $!(\pi\, \vec{v} \bullet g \longrightarrow t)$, except that backtracking *within* t is still permitted. In general, this is not a problem, as the most general solution is found by the interpreter.[2]

In order to make the basic combinators readily usable, we define infix operators for them, giving precedences as outlined in Section 5.1. Unfortunately, the symbols '`;`', '`||`' and '`|`' are reserved in *Gofer*.

```
(.;.)  = thentac
(.||.) = partac
(.|.)  = elsetac

infixl 5 .;.
infixl 4 .||.
infixl 3 .|.
```

This set of definitions may be used as a starting-point for various tactic-based systems. The particular care over the implementation of **rule** is determined by the calculus which follows; the rest is quite generic.

7.3 Syntax

In order to make use of the tactic definitions, as with encoding $\mathcal{W}$ in 2OBJ, the first step is to describe the syntax of LittleZ as a number of *Gofer* datatypes. This time, however, we encode the *abstract* syntax, and must give explicit constructors for each production. This reduces readability (and introduces a need for a separate parser), but does give greater assurance than was possible in the encoding in 2OBJ that the terms subsequently used do indeed correspond to the expected productions in the grammar.

Predicates, for example, inhabit the datatype PRED:

[2] For arbitrary instances of con, this solution would not be suitable, but for π (where the action of con is determined by an instance of *equals*), this is sufficient.

$$
\begin{array}{rl}
Pred ::= & Expr \in Expr \\
| & Expr = Expr \\
| & true \\
| & false \\
| & \neg\, Pred \\
| & Pred \wedge Pred \\
| & Pred \vee Pred \\
| & Pred \Rightarrow Pred \\
| & Pred \Leftrightarrow Pred \\
| & \forall Name : Expr \bullet Pred \\
| & \exists Name : Expr \bullet Pred \\
| & (\!| Name := Expr |\!) Pred
\end{array}
$$

```
data PRED = In EXPR EXPR
| Equ EXPR EXPR
| ZTrue
| ZFalse
| Not PRED
| And PRED PRED
| Or PRED PRED
| Imp PRED PRED
| Iff PRED PRED
| Forall NAME EXPR PRED
| Exists NAME EXPR PRED
| Substp SUBST PRED
```

The meta-syntactic functions can then be defined over this datatype in a very straightforward manner. `phip` returns a *set* of values—and a datatype of sets is readily implemented in *Gofer*. Incompletenesses in this account will lead to exceptions being raised in execution, not to silent unexpected truth of side-conditions, as in 2OBJ.

```
phip (In e s) = phie e `union` phie s
phip (Equ e v) = phie e `union` phie v
phip ZTrue = emptyset
phip ZFalse = emptyset
phip (Not p) = phip p
phip (Or p q) = phip p `union` phip q
phip (And p q) = phip p `union` phip q
phip (Imp p q) = phip p `union` phip q
phip (Iff p q) = phip p `union` phip q
phip (Forall x e p) = phie e `union` (phip p `diff` (singleton x))
phip (Exists x e p) = phie e `union` (phip p `diff` (singleton x))
```

7.4 Basic Rules

The judgements of this logic are sequents which have a list of Z paragraphs on the left-hand side (antecedent), and a single Z predicate on the right (consequent).

```
type SEQUENT = ([PAR],PRED)
```

An inference rule is a partial function from one of these sequents to a list of subgoal sequents. It is implemented as a *total* function onto a disjoint union—if the (true) rule is applied within its domain, the result is placed in the part of the union tagged with `Succs`; if not, the result is `Fails`—see above.

Most of the inference rules concern the consequent predicate; some are defined in terms of the *last* antecedent paragraph. As very few rules refer to any of the other antecedents, and as *Gofer* offers pattern-matching on *cons*-lists, the list of antecedents will be stored in *reverse order*.

The basic propositional calculus rules, then, will be

$$\frac{\Sigma \vdash P \quad \Sigma \vdash Q}{\Sigma \vdash P \wedge Q}$$

```
andI (ps,p `And` q) = Succs [ (ps,p) , (ps,q) ]
andI _              = Fails
```

$$\frac{\Sigma \vdash P \wedge Q}{\Sigma \vdash P}$$

```
andEr q (ps,p)      = Succs [ (ps,p `And` q) ]
```

$$\frac{\Sigma \vdash P \wedge Q}{\Sigma \vdash Q}$$

```
andEl p (ps,q)      = Succs [ (ps,p `And` q) ]
```

$$\frac{\Sigma \vdash P}{\Sigma \vdash P \vee Q}$$

```
orIr (ps,p 'Or' q)  = Succs [ (ps,p) ]
orIr _              = Fails
```

$$\frac{\Sigma \vdash P}{\Sigma \vdash P \vee Q}$$

```
orIl (ps,p 'Or' q)  = Succs [ (pa,q) ]
orIl _              = Fails
```

$$\frac{\Sigma \vdash P \vee Q \quad \Sigma \ddagger P \vdash R \quad \Sigma \ddagger Q \vdash R}{\Sigma \vdash R}$$

```
orE p q (ps,r) = Succs [ (ps,p 'Or' q),((Predpar p):ps,r),
                                        ((Predpar q):ps,r) ]
```

$$\frac{\Sigma \ddagger P \vdash Q}{\Sigma \vdash P \Rightarrow Q}$$

```
impI (ps,p 'Imp' q) = Succs [ ((Predpar p):ps,q) ]
impI _              = Fails
```

$$\frac{\Sigma \vdash P \quad \Sigma \vdash P \Rightarrow Q}{\Sigma \vdash Q}$$

```
impE p (ps,q)       = Succs [ (ps,p),(ps,p 'Imp' q) ]
```

$$\frac{\Sigma \vdash \mathit{false}}{\Sigma \vdash P}$$

```
falseE (ps,p)       = Succs [ (ps,ZFalse) ]
```

$$\frac{\Sigma \ddagger \neg P \vdash \mathit{false}}{\Sigma \vdash P}$$

```
notE (ps,p)         = Succs [ ((Predpar (Not p)):ps,ZFalse) ]
```

Extending these to predicates is not hard:

$$\frac{\Sigma \ddagger x : S \vdash P}{\Sigma \vdash \forall x : S \bullet P}$$

```
allI (ps,Forall x s p) = Succs [ ((Decl x s):ps,p) ]
allI _                 = Fails
```

$$\frac{\Sigma \vdash \forall x : S \bullet P \quad \Sigma \vdash e \in S}{\Sigma \vdash \{ x := e \} P}$$

```
allE s (ps,Substp(x,e) p) | x 'notin' phie e
                       = Succs [ (ps,Forall x s p),(ps,e 'In' s) ]
allE _ _               = Fails
```

The reader will notice immediately that there is a certain loss of readability, as compared with the 2OBJ encoding (due to a larger number of constructor functions being present, and these being given with names, rather than symbols) but that the encoding is more-or-less immediate.

subst

In the 2OBJ encoding, substitution was accomplished via a special rule which invoked certain OBJ3 rewrite rules not normally applied in the rule-application rewriting. This implementation uses an inference rule which applies a meta-function `subst` to the current goals. The function `subst` is overloaded via *Gofer*'s *type classes*, and propagates substitutions appropriately within the goal.

```
instance Subst PRED where
  subst(b 'Substp' (e 'In' s)) =
      (subst(b 'Subste' e))'In'(subst(b 'Subste' s))
  subst(b 'Substp' (e 'Equ' u)) =
      (subst(b 'Subste' e))'Equ'(subst(b 'Subste' u))
  subst(b 'Substp' ZTrue) = ZTrue
  subst(b 'Substp' ZFalse) = ZFalse
  subst(b 'Substp' (Not p)) = Not (subst(b 'Substp' p))
  subst(b 'Substp' (p 'And' q)) =
      (subst(b 'Substp' p))'And'(subst(b 'Substp' q))
       :
  subst((x,v) 'Substp' (Forall y s p))
        | x /= y && y 'notin' phie v =
      Forall y (subst((x,v) 'Subste' s)) (subst((x,v) 'Substp' p))
  subst((x,v) 'Substp' (Exists y s p))
```

```
        | x /= y && y `notin` phie v =
      Exiats y (subst((x,v) `Subste` s)) (subst((x,v) `Substp` p))
subst(b `Substp` (Inop n e1 e2)) =
          Inop n (subst(b `Subste` e1)) (subst(b `Subste` e2))
subst p = p
```

Notice the catch-all clause at the end—unrecognised expressions (those *not* of the form $\langle\!| x := e |\!\rangle p$) are unchanged by *subst*.

Substitution is then invoked by a rule:

```
apply_subst (ps,q) = Succs [(map subst ps, subst q)]
```

Making the system behave like $\mathcal{W}$

Proofs in this new calculus seem to be rather more contrived than those in $\mathcal{W}$. It is useful to construct tactics which accomplish (approximately) the same inferences as the basic rules in $\mathcal{W}$.

Where $\mathcal{W}$ had one rule for disjunction in the consequent, this calculus has two (since the consequent is a single predicate)—one approach to taking account of this with a tactic is to put the two in alternation; whichever one gives the correct result will (ultimately) be chosen. (Of course, this might be very inefficient in practice.)

```
t_and = rule andI
t_or  = rule orIl .|. rule orIr
t_imp = rule impI
t_not = pitac t
        where t (Singl(ps,Not p)) = (rule notE) .:.
                                    (cut_tac p) .:.
                                    ( (rule notE) .:. swap_tac .:.
                                            not_t .:. assum_tac
                                      .||.
                                      swap_tac .:. (thinr_tac 1))
              t _                 = failtac
```

The last tactic simulates the 'cross-over' rule for $\vdash\neg$. It is rather more complex, entailing use of a logical *cut*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \ddagger \neg P \vdash \neg P}\,assumption}{\Sigma \ddagger \neg P \ddagger \neg \neg P \vdash false}\,not\_l}{\Sigma \ddagger \neg \neg P \ddagger \neg P \vdash false}\,swap}{\Sigma \ddagger \neg \neg P \vdash P}\,notE \qquad \dfrac{\Sigma \ddagger P \vdash false}{\Sigma \ddagger \neg \neg P \ddagger P \vdash false}\,swap, thin}{\Sigma \ddagger \neg \neg P \vdash false}\,cut\_tac(P)$$
$$\dfrac{\Sigma \ddagger \neg \neg P \vdash false}{\Sigma \vdash \neg P}\,notE$$

The tactic for $\vdash\forall$ is simply the `allI` rule. $\forall\vdash$ is more complex:

```
t_all = rule allI

all_t e = pitac t
          where t (Singl((Predpar (Forall x s p)):ps,q))
                  = cut_tac (Substp(x,e) p) .:.
                    ((rule (allE(s)) .:. (assum_tac .|. skiptac))
                     .||.
                     skiptac)
                t _ = failtac
```

$$\frac{\dfrac{\overline{\Sigma \ddagger \forall x : S \bullet P} \quad \Sigma \ddagger \forall x : S \bullet P}{\vdash \forall x : S \bullet P \quad \vdash x \in S} \qquad \Sigma \ddagger \forall x : S \bullet P \ddagger (\!| x := e |\!) P \vdash Q}{\Sigma \ddagger \forall x : S \bullet P \vdash Q}$$

7.5 A Case Study

As a simple case study in the use of this new tool, we present a proof of the first law in the Z mathematical toolkit (from [Spi92a, page 89]).

In LittleZ, we might represent this law as

$$[X] \ddagger x : X \ddagger y : X \ddagger \forall x : X \bullet (\forall y : X \bullet x \neq y \Leftrightarrow \neg (x = y))$$
$$\vdash$$
$$x \neq y \Rightarrow y \neq x \ .$$

The proof proceeds by specializing the universal quantifiers in two different ways (to gain predicates containing $(x \neq y)$ and $(y \neq x)$), then using the cross-over rules (tactics), Leibniz's rule, and the rule of reflection to complete the proof.

Some infelicities in the account of α-conversion of quantified terms[3] make the theorem above rather hard to prove, and so instead we demonstrate

$$[X] \ddagger x_1 : X \ddagger y_1 : X \ddagger \forall x : X \bullet (\forall y : X \bullet x \neq y \Leftrightarrow \neg (x = y))$$
$$\vdash$$
$$x_1 \neq y_1 \Rightarrow y_1 \neq x_1 \ ,$$

by means of the following tactic:[4]

```
tac = all_t_discharge 2 (Ident "x1") .:.
      all_t_discharge 2 (Ident "y1") .:.
      iff_t .:. thinr_tac 1 .:. t_imp .:. mp_tac .:.
      cut_tac (Not((Ident "y1")'Equ'(Ident "x1"))) .:.
      (
         (  t_not .:. swap_tac .:. not_t .:.
            (t_tsbus ("y1",Ident "y1")) .:.
            (rule (leibniz (Ident "x1"))) .:.
            ( (subst_tac .:. (rule refl))
              .||.
              assum_tac)  )
      .||.
         ( (repeat_tac 5 drop_snd) .:.
           swap_tac .:.
           (all_t_discharge 2 (Ident "y1")) .:.
           (all_t_discharge 4 (Ident "x1")) .:.
           iff_t .:. (repeat_tac 4 drop_snd) .:.
           swap_tac .:. mp_tac .:. assum_tac )
         )
```

The tactic `all_t_discharge n` used repeatedly above is derived from `all_t`. It makes use of the declaration in the nth position from the $\vdash$ to discharge the $e \in s$ condition which arises when `all_t` is used:

[3] In order to specialize the universally quantified term for y, it is necessary first to α-convert the *inner* quantification. This appears to be impossible in the present account of the logic.

[4] Which, in contrast to a comparable 2OBJ tactic, executes in less than one second.

```
all_t_discharge n e = all_t e .:.
                          ((thinr_tac n .:.assum_tac) .||. subst_tac)
```

7.6 Discussion

This chapter has described a simple implementation of a theorem-prover in *Gofer*. The same approach as the one described here has been used to implement a number of tools—an implementation of part of $\mathcal{W}$, a tool for testing the associative/commutative tactics of Section 5.2, and a tactic-based tool for reasoning about tactics (permitting the animation of the normalization tactic of Section 6.2). In each case, the soundness of the implementation is easy to verify, the execution is quick, and the user interface is poor.

The ease with which this implementation is constructed may be contrasted with the effort involved in the production of the encoding of $\mathcal{W}$ in 2OBJ. Certainly, *Gofer* provides a much more stable platform for implementation than does 2OBJ. Likewise, the deductive system used here has benefited from two more years' consideration—and from feedback regarding the usability of $\mathcal{W}$. The encoding is also made easier by being the second such piece of work that the implementor has undertaken.

Since one of the most significant performance problems with 2OBJ came in the checking of side-conditions, we must be concerned with whether the same problem is likely to arise here. Initial indications are that it will not be so serious a difficulty. This is both because the tool is much faster to begin with (but note that the complexity of the calculations rises exponentially with the depth of schema nesting), and because this logic presents fewer instances where bound/free variable calculation for the whole sequent are required. If performance problems do become a serious problem, it may be necessary to use a more advanced functional programming technique (monads [Wad93], perhaps)—to provide a means of ensuring that each schema has its free variables and alphabet calculated once only.

User Interface

The output from the tool is simple to improve—a simple pretty printer produces a readable ASCII rendering of the terms, or LaTeX mark-up. Input to the tool is, at present, only via the abstract syntax datatypes described above. As a result, the goal in the previous section was entered as

```
Sing1([Predpar neqdef,decl2,decl1],concl)
      where neqdef = (Forall "x" (Ident "X")
                        (Forall "y" (Ident "X")
                           (((Ident "x")`neq`(Ident "y"))`Iff`
                             Not((Ident "x")`Equ`(Ident "y"))))),
            decl1 = Decl "x1" (Ident "X")
            decl2 = Decl "y1" (Ident "X")
            concl = ((Ident "x1")`neq`(Ident "y1"))`Imp`
                      ((Ident "y1")`neq`(Ident "x1"))
```

Clearly, this is not practical. A parser written in *Gofer* might be used here; a parser written with standard UNIX tools *lex* and *yacc* (providing translation into the form seen above) would be much more efficient, and provide a simple front-end for the tool. Interactive use of a *Gofer* program is also possible; this would necessitate use of a parser for tactics, too.

A reasonable short-term goal, then, is a theorem-prover with a 'command-line' interface. Specifications can be written in LaTeX, or the Z standard interchange format, with goals interspersed, each provided with a tactic. The tool will read a file of such input, and output a file with each goal replaced by the result of applying its tactic to it.

Related Work

Of course, there have been *many* previous implementations of theorem-provers in functional languages. The approach taken here is, at its core, similar to many of them. The central idea is that theorems (or equivalently, proofs) are encoded as a datatype, and that the strong typing ensures that only sound theorems (proofs) can be produced.

Probably the most successful of these—and the originator of the notion of a proof as a 'safe datatype' is LCF (as Edinburgh LCF [GMW79] and later Cambridge LCF [Pau87]). That system does not have a primitive type of inference rules, but it uses a similarly small set of functions between theorems. The chief difference with the approach taken here is that in LCF tactics are higher-order functions: as well as returning lists of goals (corresponding to the *parallel* composition of goals above, not the alternation—LCF makes no allowance for alternative outcomes) they also return *proofs*, which are functions deducing one theorem from another:

```
type proof = thm list -> thm;
type tactic = goal -> (goal list * proof);
```

The `proof` component is described as a *validation*. It must be a composition of inference rules. This gives potentially a more efficient implementation than the one described above, since rule application is deferred until the end of a proof, so time-consuming checks can be taken out of the interactive part of the proof activity. The user must therefore take care that only *valid* tactics are used, otherwise the proof construction will have been in vain. On-line checking of conditions—processing power permitting—may be less likely to lead to wasted effort.

LCF is implemented in (and gave rise to) ML. By contrast, choosing a lazy language such as *Gofer*, makes straightforward implementation of backtracking possible.

In most projects, the first system built is barely usable.
It may be too slow, too big, awkward to use, or all three.
There is no alternative but to start again...

—Frederick P. Brooks, Jr.
The Mythical Man-Month: Essays on Software Engineering

Chapter 8

Conclusions

THIS THESIS HAS DESCRIBED the creation of one prototype theorem-prover for Z, and laid the groundwork for another. Both tools are demonstrably sound with respect to the published semantics of Z—and in this they are distinctive. Both have been used to demonstrate that the deductive systems which they encode are tractable for proof in Z. The development of such proofs—via tactics—has been given a formal software engineering treatment.

8.1 Proof Tools

Chapter 1 presented some criteria by which a proof tool may be judged. The first of these was soundness: the requirement that any theorem which can be proved by the tool could also be demonstrated using the (published) semantics of (draft) Standard Z. By paying special attention to this issue, a proof tool has been produced in which a user may place reasonable confidence, *and* the activity of producing and testing the tool has shown up (minor) flaws in the presentation of the deductive system on which it is based.

The encoding of $\mathcal{W}$ in 2OBJ also scores well on the user interface side, but falls down badly on efficiency—the situation is reversed for the tool described in Chapter 7. Both have made an attempt at providing a tactic library, and in both cases, that library remains incomplete and not the most efficient possible. No attempt is made adapt some of the well-known and powerful proof techniques from the more specialized world of automated theorem proving (resolution, forward chaining, unification, Knuth-Bendix completion, etc.). Chapter 4 has discussed whether the rules of $\mathcal{W}$ are appropriate for the construction of a proof tool; the answer seems to be that they are among the best available.

Comparing Chapters 2 and 7, the relative merits of two different approaches to implementation can be seen. Whilst the second looks as though it will ultimately be more useful, the value of the first (the discipline involved in producing the encoding, and the ease of use of the resulting interface) should not be overlooked.

The main rôle for such tools is in taking away some of the technical noise from a Z proof (i.e. the demonstration of internal consistency, and, potentially, the proof of correctness of a data refinement), leaving the specifier to worry about the deep issues involved (whether the specification really captures the required behaviour of the system). Ideally, one's software engineering methodology should abstract away from that noise, and leave the user dealing with the deep material. Clearly this is not the case with Z—in demonstrating (even) that a specification is self-consistent many mundane proof obligations arise. People use Z, and so they need a tool which leaves them able to devote time to the more important issues. Such an approach is akin, ideally, to other software engineering tools—the type checker, parser, etc—which find simple bugs, not profound ones. They help to spot well-formed theories (those which obey the simple syntax and typing rules), not necessarily correct ones. Of course, the proof tool *does* do slightly better than this—if a formal proof of some deeper property of the specification is required, it can be accomplished (with user guidance).

8.2 Tactic Language

The work on the tactic language attempts to meet the software engineer's programming concerns. Having presented a general tactic language having a concise semantics and a complete transformational calculus, we may define tactics which can be used, re-used, and substituted with confidence.

This may be contrasted with the tactic languages used by at least two popular commercial software engineering tools with proof components: **B** has a tactic/control language which is closely tied to the theory under consideration (changes in which may change the control structure); and *Zola* has a tactic language based on Lisp in which tactic behaviours may be modified by keywords in a variety of non-compositional ways (see Section 6.5).

The fact that the semantics is based on lists (rather than sets) gives rise to some unpleasant side-conditions on laws—two tactics which produce the same alternatives, but with differing orders for those alternatives, are regarded as distinct. Since any implementation of a tactic language will ultimately have a sequential character, it does not seem unreasonable to model this in the tactic semantics. Few would argue that in a tactic such as $!(t \mid \mathbf{skip})$, the application of t is equally preferable with the application of **skip**.

The angelic nondeterminism ('deep' backtracking) present in the tactic language permits the succinct expression of some tactics, especially those which require *searching*, and those for which various alternatives need to be tried (e.g. the *assocs* tactic). The structural combinators permit tactics to be applied to sub-expressions with ease, and in a more intuitive way than that used in Lisp (and hence *Zola*), or in OBJ3 (where sub-expressions are selected via a list of position numbers for expressions within expressions). The negative effect of using a semantics based on lists is seen most strongly in laws relating backtracking and structural combinators—transformations are possible only when most of the tactics are sequential (i.e. have the property $t = !t$).

It may be commented that the tactic language presented here promotes reusability of a less-than-desirable kind. When a tactic is rewritten/improved, the old version can be left in place—in alternation after the new—so that if the new implementation should cause some subsequent tactic to fail, the system may backtrack and use the old implementation instead. This would be poor style both because it may impair

efficiency and because it might lead to code which would be very hard to maintain.

Comparison with Other Work

The work on the tactic language follows a long history of work in the theorem-proving and functional programming communities. The fundamental tacticals introduced here originate in the work on Edinburgh LCF [GMW79]. There, they appear as THEN and ORELSE—the latter being a *cut* version of our alternation operator. The LCF treatment of tactics differs from that presented here, in that LCF's inference rules are quite distinct from the set of tactics, and not a subset, as we have presented them here. Rules are used for forwards proof; tactics are used in backwards (goal-oriented) proof search, and return 'validation functions', which are compositions of proof rules. As a result, the safe datatype is that of proofs, not (as here) that of tactics; tactics are *valid* if they are able to be validated by compositions of primitive rules. *Strongly valid* tactics are related to our functionally correct ones.[1]

Milner [Mil84] generalizes the ideas from LCF somewhat, observing how the notions present in tactic programming (goal, strategy, achievement, and failure) stand together in a far more general setting than merely in the area of machine-assisted proof.

An independent semantics for LCF's tactics is found in [Sch84]. Schmidt's goal is similar to ours—the discovery of a language which will facilitate 'formulation of high level algorithms that can be compared, analyzed, and even ported across theorem proving systems.' He does not present a formal treatment of failure, or of recursion.

2OBJ [GSHH92] builds on many of the ideas in LCF, but implements rules and tactics in the way which has been discussed above—with rules as a (clearly delineated) subsort of the sort of tactics. The ELSE operator retains the semantics of LCF's ORELSE, though this may change in later versions of the tool.

Paulson describes a simple theorem-prover in [Pau91], in which the tactics are treated in a similar way to those in his *Isabelle* system. Here, as in the semantics presented in Chapter 5, the tacticals return sequences of alternatives. There are two alternation tactics: APPEND implements our alternation operator, whereas `t1 ORELSE t2` is equivalent to $t_1 \mid (\mathit{fails}\ t_1\ ;\ t_2)$. As a result, Isabelle's REPEAT `t` permits backtracking within `t`, but not on the number of repetitions of `t` (unlike *exhaust*). Isabelle implements cut as DETERM. The semantics is given using the tacticals' ML definitions; only one algebraic law is given—the one stating that `all_tac` (i.e. **skip**) is an identity for THEN.

Felty's tactic language [Fel93] is also very similar to that presented in this thesis, complete with the backtracking suggested by the alternation tactical. Its semantics is given via its logic-programming implementation, and as such has a more relational style of approach than that seen here. Nevertheless, a cut version of `orelse` is presented, since such pruning of the proof-search is needed in interactive use of the system she describes. No algebraic laws are given.

The usefulness of lazy lists to implement backtracking has been known in functional programming circles for some time. Burge [Bur75] discusses such backtracking in the context of top-down parsing, and Wadler [Wad85] presents a whole parser toolkit in this style. The parser combinators are very similar to those given here. For example, `lit x` is a parser combinator which matches a string whose first character

[1] Strongly valid tactics are those which cannot lead 'up a blind alley'. If given an achievable (provable) goal, they return achievable subgoals.

is x. Parser combinators return a list of tuples containing the matched portion of their argument string in the first place, and the remaining string in the second:

```
lit 'a' "apple"  = [("a","pple")]
lit 'a' "banana" = []
```

These combinators may be placed in alternate and sequential composition:

```
alt (lit 'a') (lit 'b') "banana" = [("b","anana")]

seq list2 (lit 'b') (lit 'a') "banana" = [("ba","nana")]
```

The *exhaust* tactical is coded as `rep`:

```
rep (lit 'a') "aardvark" =
                    [("aa","rdvark"),("a","ardvark"),("","aardvark")]
```

Thus we would expect the theory described in Chapter 5 to be applicable to this work—and to systems based on it, such as the parser described in [FL89]. Wadler notes that an added benefit of this method of handling backtracking and failure is the avoidance of any need to consider exception handling.

Moreover this tactic language may be applied to another idea in the theorem-proving world: in LCF/HOL, *rewriting* is extensively used for *simplification* in theorem-proving. In [Pau83], Paulson describes a means of directing such conversions using combinators which closely mirror those present in the tactic language. A unified theory of such combinators/tacticals may make implementation easier, and certainly makes for a more straightforward conceptual framework.

8.3 Further Work

The work with 2OBJ can be considered as being at an end. Some serious bugs remain in 2OBJ, and no further development work on that project is being undertaken. As a result, there seems to be little value in making the much-needed improvements in the implementation of $\mathcal{W}$ in 2OBJ (even if it were clear how to improve the implementation). Nor is there much point in making improvements to the tactic library—but ideas from there may usefully be carried forward into the *Gofer* implementation—see below.

The theory behind the tactic language could bear a little tidying. Ideally, one would obtain a completeness result for the language including recursion (and, ideally, structural combinators and pattern-matching, too). Perhaps the most promising approach to proving that recursive tactics can be put into a normal form is the tactic-based approach of Section 6.2. A complete tactic-based theorem prover for reasoning about tactics would be an interesting curiosity, and may be useful for conducting meta-proofs about other tactic-based systems.

A valuable contribution to the treatment of recursion would be to improve the rather vague treatment of termination. A formal treatment of sufficient conditions for termination would be useful, or some other condition sufficient to guarantee that a recursive equation describes a unique least fixed point (akin to the notion of *guarded* recursion in CSP [Hoa85, page 28]).

To produce a usable proof tool for Z—at least insofar as providing a means of animating the logic presented in the standard is concerned—it seems sensible to build on the implementation in Chapter 7, in the way outlined there—providing, at least, a

Z proof tool with a command-line interface. Some of the work there is in construction of the parser, and the output routines. Most of the work, however, is in expanding the coverage from LittleZ to the whole language (Stephen Brien's thesis [Bri95] explains how to do this), and providing a tactic library targeted upon proving Z specification properties—complete with a formal description of how the tactics behave.

The breadth of applicability of the tactic language could also be explored further. The language might be used to express a range of algorithms used in theorem-proving and term-manipulation systems, and laws like those presented here used to transform and/or validate them. Ideas for animating other logics in *Gofer*, in the style used above, could lead to a toolkit for producing lightweight theorem proving tools—a lightweight logical framework.

8.4 Finally

This chapter ends with a quote from Edsger Dijkstra. He was asked for his opinion on the use of proof tools; he replied by asking why we should let computers take away our fun. I hope that this thesis has showed that computers can take away the mundane bits of program verification—leaving us to have fun with the rest—*and* we can have fun directing the theorem provers too.

A reusable theorem is one which can be proved over and over again.

I have no problem with people using automated theorem provers.
Personally I wouldn't like to do so:
why delegate to a machine that which is so much fun to do for yourself?

—Edsger W. Dijkstra
(paraphrased, comments at Marktoberdorf Summer School, 1994)

Bibliography

[Abr91] J.-R. Abrial. A formal introduction to mathematical reasoning. Technical report, BP Research International, 1991.

[BG94] J. P. Bowen and M. J. C. Gordon. Z and HOL. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

[BGH+92] R. J. Boulton, A. D. Gordon, J. R. Harrison, J. M. J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Cicuit Design: Theory, Practice and Experience: Proceedings of the IBIP TC10/WG 10.2 International Conference*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[BHW94] S. M. Brien, W. T. Harwood, and J. C. P. Woodcock. Logic and description in Z-like languages, April 1994. Submitted to *FACS*.

[Bir86] R. S. Bird. An introduction to the theory of lists. Technical Monograph PRG-56, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK, 1986.

[Bir88] R. S. Bird. Lectures on constructive functional programming. Technical Monograph PRG-69, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK, 1988.

[BL74] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical report, MITRE Corporation, Bedford, MA, 1974.

[Bla94] K. Blackburn. Example material for a Balzac reference manual. Ref: ISS/HAT/CSC3/111, February 1994.

[BN+92] S. M. Brien, J. E. Nicholls, et al. Z base standard. ZIP Project Technical Report ZIP/PRG/92/121, SRC Document: 132, Version 1.0, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK, November 1992.

[Bri92] Z Base Standard, March 1992. Version 0.5.

[Bri95] S. M. Brien. *A Model and Logic for Generically Typed Set Theory (Z)*. D.Phil. thesis, University of Oxford, 1995.

[Bro75] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[Bro83] S. D. Brookes. *A Model for Communicating Sequential Processes*. D.Phil. thesis, University of Oxford, January 1983.

[BS79] R. S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, Inc., 1979.

[Bur75] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[Dij94] E. W. Dijkstra. The argument about the arithmetic mean and the geometric mean, heuristics inculded. In M. Broy, editor, *Deductive Program Design*, NATO ASI Series. Springer-Verlag, 1994. Marktoberdorf International Summer School, 1994, to appear.

[Dil90] A. Diller. *Z: An Introductian to Formal Methods*. Wiley, Chichester, UK, 1990.

[DMLP79] R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social process and proofs of theorems and programs. *Communications of the ACM*, 22:271–280, May 1979.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[Fel93] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11:43–81, 1993.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[For92] Formal Systems (Europe), Ltd. *Failures divergence refinement, User Manual and Tutorial*, 1992.

[GLW91] P. H. B. Gardiner, P. J. Lupton, and J. C. P. Woodcock. A simpler semantics for Z. In Nicholls [Nic91], pages 3–11.

[GMW79] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[Gor88] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[Gor94] M. J. C. Gordon. Private communication, 1994.

[GSHH92] J. Goguen, A. Stevens, H. Hilberdink, and K. Hobley. 2OBJ: A Metalogical Theorem Prover based on Equational Logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, Prentice-Hall, 1992.

[GW88] J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988.

[HHP91] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Report series, LFCS, Department of Computer Science, University of Edinburgh, 1991.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[HS85] C. A. R. Hoare and J. C. Shepherdson, editors. *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.

[ICL93] ICL Ltd. *Tutorial Notes on Proof in Z*, 1993. Tutorial material at *FME'93: Industrial-Strength Formal Methods*.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer Verlag, 1991.

[JMT91] D. Jordan, J. A. McDermid, and I. Toyn. CADiZ – computer aided design in Z. In Nicholls [Nic91], pages 93–104.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, second edition, 1990.

[Jon91a] M. P. Jones. *An Introduction to Gofer*, 1991.

[Jon91b] R. B. Jones. Book review of [Dil90]. *Science of Computer Programming*, 16(3):286–288, 1991.

[Jon92] R. B. Jones. ICL ProofPower. *BCS FACS FACTS*, Series III, 1(1):10–13, Winter 1992.

[KB95] I. Kraan and P. Baumann. Implementing Z in Isabelle. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *LNCS*, pages 355–373. Springer-Verlag, 1995.

[LS87] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley-Teubner Series in Computer Science, second edition, 1987.

[Mar93a] A. Martin. Encoding $\mathcal{W}$: A Logic for Z in 2OBJ. In Woodcock and Larsen [WL93], pages 462–481.

[Mar93b] A. Martin. Infinite lists in Z. Draft paper, 1993.

[Mil84] R. Milner. The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society, London. Series A*, 312:411–422, 1984. Also in [HS85].

[Mor90] C. C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990.

[Nic91] J. E. Nicholls, editor. *Z User Workshop, Oxford 1990*, Workshops in Computing. Springer-Verlag, 1991.

[ORSvH93] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Some lessons learned. In Woodcock and Larsen [WL93], pages 482–500.

[Par69] D. Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5:59–78, 1969.

[Pau83] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

[Pau87] L. C. Paulson. *Logic and Computation—Interactive Proof with Cambridge LCF*. CUP, 1987.

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989. Also University of Cambridge Computer Laboratory Technical Report No. 130.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. CUP, 1991.

[Sch84] D. A. Schmidt. A programming notation for tactical reasoning. In R. E. Shostak IV, editor, *7th International Conference on Automated Deduction*. Springer-Verlag, LNCS Volume 170, 1984.

[SH92] A. Stevens and K. Hobley. *Mechanized Theorem Proving with 2OBJ: A Tutorial Introduction*, 1992.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.

[Spi92a] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, second edition, 1992.

[Spi92b] J. Spivey. *The fuzz Manual*. Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 2nd edition, 1992.

[SS90] J. M. Spivey and B. A. Sufrin. Type inference in Z. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z—Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 426–451. Springer-Verlag, 1990.

[Wad85] P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 113–128. Springer-Verlag, September 1985.

[Wad93] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series F, pages 233–264. Springer-Verlag, 1993. Marktoberdorf International Summer School, 1992.

[WB92] J. C. P. Woodcock and S. M. Brien. *W*: A Logic for Z. In *Proceedings 6th Z User Meeting*. Springer-Verlag, 1992.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z*. Prentice-Hall, 1996.

[WL93] J. C. P. Woodcock and P. G. Larsen, editors. *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Woo92] J. C. P. Woodcock. Case Study: A Multilevel Security System, 1992.

Appendix A

On Lists

A.1 Definitions

The treatment of lists in this thesis is broadly derived from that in [Bir88]. The definitions which follow are consistent with those in that monograph (though there, Bird uses $\frown$-lists rather than the *cons*-lists described here). This being a thesis based on Z, the notation used is more Z-like than Bird's; in particular, we use $\langle\ \rangle$ for list brackets, instead of $[\ ]$. Z's lists are strictly finite, but the definitions which follow also suffice for infinite lists (see [Mar93b]).

A list over a set X is either the empty list, or of the form $x : xs$, where x is a member of X, and xs is a list. The list $x_1 : x_2 : \ldots : x_n : \langle\ \rangle$ is generally written as $\langle x_1, x_2, \ldots x_n \rangle$. The partial lists of Section 5.8 are modelled as lists of the form $x_1 : x_2 : \ldots : x_n : \bot_X$. Infinite lists are limits of partial lists.

The *map* operator $*$ is defined thus:

$$\begin{aligned} f* \ \langle\ \rangle &= \langle\ \rangle \\ f* \ (x : xs) &= (f\ x) : f* \ xs \ . \end{aligned}$$

The dual notion is $^\circ$ ('all applied to'):

$$\begin{aligned} \langle\ \rangle^\circ x &= \langle\ \rangle \\ (f : fs)^\circ x &= (f\ x) : (fs^\circ x) \end{aligned}$$

List concatenation is similarly defined:

$$\begin{aligned} \langle\ \rangle \frown ys &= ys \\ (x : xs) \frown ys &= x : (xs \frown ys) \ . \end{aligned}$$

Distributed concatenation is a special case of a more general operator, usually called 'reduce', and written '/':

$$\begin{aligned} \oplus/\langle x \rangle &= x \\ \oplus/(x_1 : (x_2 : xs)) &= x_1 \oplus (\oplus/(x_2 : xs)) \end{aligned}$$

The *head'* function used here is a totalized version of the more common *head* function:

$$\begin{aligned} head'\langle\,\rangle &= \langle\,\rangle \\ head'(x : xs) &= \langle x\rangle \ . \end{aligned}$$

Function composition is the common (mathematical) backward composition, and ∇ defines a pairwise composition.

$$\begin{aligned} (f \circ g)(x) &= f\big(g(x)\big) \\ (f \nabla g)x &= (f\,x, g\,x) \ . \end{aligned}$$

The operators needed in Section 5.10 are slightly more complicated, though still standard list processing functions from [Bir88].[1] Firstly *zip with* $\oplus$:

$$\begin{aligned} \langle\,\rangle \curlyvee_{\oplus} \langle\,\rangle &= \langle\,\rangle \\ (t : ts) \curlyvee_{\oplus} (g : gs) &= \mathbf{if} \#ts = \#gs \\ &\qquad \mathbf{then} \langle t \oplus g\rangle \frown (ts \curlyvee_{\oplus} gs) \\ &\qquad \mathbf{else} \langle\,\rangle \ . \end{aligned}$$

Note that # calculates the length of a list. This operator is used in the definition of structural combinators as $\curlyvee_{\mathrm{id}}$. The operator used to *zip with* is 'id', since the desired result is that functions in the first list be applied to arguments in the second:

$$\begin{aligned} &\mathrm{id}\, t\, g \\ &= (\mathrm{id}\, t) g \\ &= t\, g \end{aligned}$$

The cartesian product $\prod$, is defined using a list cross product:

$$\begin{aligned} xs \mathbin{\mathbf{X}}_{\oplus} \langle\,\rangle &= \langle\,\rangle \\ xs \mathbin{\mathbf{X}}_{\oplus} (y : ys) &= ((\oplus y)* \ xs) : (xs \mathbin{\mathbf{X}}_{\oplus} ys) \\ \prod &= \mathbf{X}\frown / \circ \langle id\rangle^{\circ} ** \end{aligned}$$

The behaviour of these two operators is illustrated thus:

$$\begin{aligned} \langle a, b\rangle \mathbin{\mathbf{X}}_{\oplus} \langle c, d, e\rangle &= \langle a \oplus c, b \oplus c, a \oplus d, b \oplus d, a \oplus e, b \oplus e\rangle \\ \prod \langle \langle a, b\rangle, \langle c\rangle, \langle d, e\rangle\rangle &= \langle \langle a, c, d\rangle, \langle b, c, d\rangle, \langle a, c, e\rangle, \langle b, c, e\rangle\rangle \end{aligned}$$

[1] However, in order to make the structural combinators fail correctly (when inputs are mis-matched), we totalize the definition of $\curlyvee$—making it return the empty list when its inputs are mis-matched—and choose to put $\mathbf{X}\frown / \langle\,\rangle = \langle\,\rangle$, for similar reasons (conventionally, this is $\langle\langle\,\rangle\rangle$).