Roll your own Jape logic
Encoding logics for the Jape proof calculator

Richard Bornat (richard@bornat.me.uk)

November 2, 2005

Preface

Jape is a lightweight, uncommitted, transparent proof calculator. It's designed to present an excellent graphical interface and a very short and shallow 'learning curve' to all its users, whether novices learning how to make formal proofs or experts — logicians, teachers, sofware engineers, practitioners of any kind — describing inference systems. This manual is directed at people who have experimented with one or more of the inference systems distributed with Jape and now want to develop something of their own, or those who just want to understand what it is that Bernard Sufrin and I have done in our own encodings.

The chapters of this manual describe by example how to encode several interesting logics in Jape. They are intended to be read in sequence, as the earlier chapters give most description of the early stages of the encoding, and later chapters concentrate on more esoteric features.

A manual which described a task only by example would be inadequate, and I therefore include a complete description of the various internal 'languages' of Jape:

- the *term* or *formula* or *sequent* language, in which problems are stated, and which appears on-screen when a proof is displayed — described in appendix A;
- the *tactic* language, which includes the statement of the inference rules of a logic, and which allows the user to control the course of a proof — described in appendix B;
- the *paragraph* language, which is the notation used to describe a logic and its associated tactics and stuff to Jape — described in appendix A;
- the *dialogue* language, which is the notation in which the graphical interface sends commands to the main proof engine, and in which you can type as text in a graphical interface window — described in appendix C.

This manual doesn't discuss how to use Jape — that's covered in other manuals. The graphical illustrations are taken from the MacOS X implementation (version 7_d4 or later).

Maintenance only

Jape was once several experiments in one. One was an experiment in user interfaces: could we make a really nice proof calculator? Another was an experiment in abstraction: could we make it customisable? Another was educational: could we teach logic with it? And no doubt there were others which I've now forgotten.

The Jape idea isn't dead, but I think it's safe to say that Bernard and I have found other things to do with our time, and the project is stalled. It's open source, so it could conceivably get started under other hands, but I made it so damn complicated that I don't expect that will happen. Jape is now maintenance-only, though if anybody has a really good simple idea to improve it I might get itchy again.

Because of its architecture and detail internal design, some things which you might think really easy to do are in fact rather hard. So: no apologies for the following list of deficiencies, none of which are likely to be fixed.

- Jape doesn't check the proof store when you redefine a rule or theorem, and re-run all the proofs that depend on it (though it does guard against circularities in proofs).
- Jape can't handle sequents in which one side is an optional single formula.
- Jape has no treatment of definitional equality (syntactic equivalence), so you have to handle it with rules and inference steps.
- Jape has a long-standing problem in that it can't encode 'families of rules'. Bernard employed considerable ingenuity to allow you to encode finite collections of slightly different rules, but that's the nearest you'll get.
- Jape's parser generator is horribly complicated, but extremely limited.
- I never quite reached modal logic, and now I never will.
- Separation logic is right out of the question.

I'll add to this list as further deficiencies are pointed out. Mail to bugs@jape.org.uk, please.

The side effects of madness

Part way through editing this manual I suffered a severe episode of depression, and went slightly mad. The pills (Citalopram) that I took had a devastating effect on my memory, which I sincerely hope will be temporary. You may, however, find places in the text where I simply say that I can't remember something or other. Please notify me at bugs@jape.org.uk whenever you see one, so that I can take it out.

For the same reason, you may find that there are features of Jape which are used in example code but not explained in this manual. Please notify me via bugs@jape.org.uk whenever you notice anything like that.

Contents

1 Basic Principles **11**

1.1 Flexible syntax . . . 11

1.2 Inference rule matching . . . 11

1.3 Explicit provisos . . . 13

1.4 Conjectures and theorems . . . 14

1.4.1 Proving a conjecture — substitutions and provisos . . . 14

1.4.2 Applying a theorem: the rôle of structural rules . . . 15

1.5 Substitution forms and unification . . . 15

1.5.1 Invisible provisos . . . 17

1.5.2 Interpreting predicate notation . . . 17

1.6 Binding forms: unification, α-conversion and substitution . . . 18

1.7 The tactic language . . . 18

1.8 Gestures, menus and panels . . . 18

1.9 Proof display: trees, boxes and hiding . . . 19

1.10 Using Jape interactively . . . 20

2 Encoding the Sequent Calculus **21**

2.1 The inference rules of the (multiple-conclusion) Sequent Calculus . . . 21

2.2 Syntax description (`sequent_syntax.j`) . . . 23

2.3 Encoding the inference rules (`MCS_rules.j`) . . . 24

2.4 Automatic application of rules . . . 25

2.5 Automatic selection of rules . . . 25

2.6 Menus . . . 26

2.6.1 The Rules menu . . . 26

2.7 Conjectures (`sequent_problems.j`) . . . 27

2.8 Global variable settings . . . 28

2.9 A very small example . . . 29

3 Variations on the Sequent Calculus **31**

3.1 LF-style variables in quantifier rules . . . 31

3.1.1 Caveat . . . 34

3.2 The intuitionistic multiple-conclusion sequent calculus . . . 34
3.3 A multiple-conclusion sequent calculus with multiplicative rules . . . 34
3.4 Modal logic . . . 37
3.5 Single-conclusion sequent calculus (the intuitionistic fragment) . . . 37
3.5.1 Inference rules . . . 37
3.5.2 LF-style variables . . . 38
3.5.3 Syntax . . . 38
3.5.4 Menus and panels . . . 38
3.5.5 Global variable settings . . . 38

4 Display modes: box and tree **39**
4.1 Hiding IDENTITY steps . . . 39
4.2 The tree is still there . . . 41
4.3 That's not all . . . 43

5 Encoding natural deduction **45**
5.1 Inference rules . . . 45
5.2 Syntax (`ItL_syntax.j`) . . . 48
5.3 The rules in Japeish (`ItL_rules.j`) . . . 49
5.4 Variable settings (`ItL.jt`) . . . 49
5.5 Forward reasoning with `cut` . . . 50
5.6 Other menu entries . . . 53
5.7 Automatic rule selection (`ItL_hits.j`) . . . 54
5.8 The Conjectures panel (`ItL_problems.j`) . . . 54
5.9 Other natural deduction encodings . . . 55

6 Equational reasoning about functional programs **57**
6.1 Syntax (`equality_rules.j`, `functions_rules.j`) . . . 57
6.2 The rewrite rule and user definition of substitutions . . . 59
6.3 Hiding parts of a proof: the LAYOUT tactical . . . 61
6.4 Dealing with associative operators: token selection, Find and Flatten . . . 62
6.5 Induction . . . 65
6.6 Controlling collections of rules . . . 66
6.7 Searching collections of rules and theorems . . . 68
6.8 Transitive reasoning: `trfunctions.jt` . . . 71

7 Encoding axiomatic set theory **77**
7.1 The natural deduction encoding . . . 77
7.2 Syntax of set operations . . . 80
7.3 The axiomatic presentation of naive set theory . . . 81

7.4 The non-axiomatic rules . . . 84

8 Encoding the Hindley-Milner type-assignment algorithm 87

8.1 Syntax . . . 87
8.2 Rules . . . 89
8.3 Reading the context and specialising a type scheme . . . 90
8.4 The generalisation step . . . 91
8.5 Automatic search . . . 92
8.6 An example . . . 93
8.7 Could Jape treat other type-theoretic logics? . . . 93

9 Encoding BAN logic 95

9.1 Syntax . . . 95
9.2 Rules . . . 96
9.3 Putting rules into menus . . . 97
9.4 Dealing with tuples . . . 98
9.5 Conjectures with long assumption lists . . . 99

10 Encoding Natural Deduction more faithfully 101

10.1 Rules . . . 101
10.2 CUTIN: full-featured box-and-line steps . . . 104
10.2.1 CUTIN and backward steps . . . 105
10.3 Formula selection . . . 107
10.3.1 Forward steps with CUTIN . . . 108
10.4 Multiple hypothesis selections . . . 108
10.5 Only complete steps allowed . . . 110
10.6 Error checking and reporting . . . 112
10.6.1 Enhanced alerts . . . 112
10.6.2 Careful step-checking . . . 112
10.7 Patching internal alerts . . . 114

11 Disproof with Kripke trees 115

12 Encoding Hoare logic 117

12.1 Syntax (HOARE_SYNTAX.J) . . . 117
12.1.1 Juxtaposition and Hoare triples . . . 119
12.1.2 PUSHSYNTAX . . . 119
12.2 Rules . . . 119
12.2.1 Variable assignment, definedness and simplifiesto . . . 120
12.2.2 Choice . . . 123
12.2.3 Array-element assignment . . . 123

12.2.4 While . . . 123
12.3 An arithmetic sledgehammer . . . 124
12.4 N-ary intro and elim steps . . . 124
12.5 N-ary intro and elim steps . . . 125
12.6 And that's it . . . 126

A The paragraph and formula languages 129
A.1 Directives . . . 129
A.2 Rules, tactics and conjectures . . . 135
A.2.1 The meaning of a RULE directive . . . 135
A.2.2 Parameters in RULE directives . . . 136
A.2.3 Instantiating a rule, including interpretation of predicate notation . . . 136
A.2.4 The meaning of provisos in RULE and THEOREM directives . . . 137
A.2.5 The meaning of conjectures (stated in THEOREM directives) . . . 137
A.2.6 Proof by resolution . . . 137
A.2.7 The meaning of STRUCTURERULE directives . . . 137
A.2.8 Substitution matching . . . 138
A.2.9 The tactic store . . . 140
A.3 Forcing semantics for Kripke trees . . . 140
A.4 Flexible syntax . . . 140
A.4.1 Symbols . . . 140
A.4.2 Juxtaposition may need care . . . 141
A.4.3 Identifier classes . . . 141
A.4.4 Syntactic hierarchy . . . 142
A.4.5 Bracketed formulae . . . 143
A.4.6 Substitution and juxtaposition forms . . . 144
A.4.7 Operator syntax . . . 144
A.4.8 Binding structure . . . 144
A.4.9 Sequent structure . . . 145
A.4.10 Future work . . . 145

B The tactic language 147
B.1 Tactic verbs . . . 147
B.2 The 'current problem sequent', the 'goal' and the 'target' . . . 150
B.3 Guarded and binding tactics . . . 151

C The command language, environment variables and the default environment 153
C.1 The command language . . . 153
C.1.1 Commands you might want to use . . . 153
C.2 Variables and the default environment . . . 155

C.2.1 Useful variables 155
C.2.2 Adding your own variables 157
C.2.3 Debugging variables 157

Chapter 1

Basic Principles

Jape works by applying inference rules to sequents in proof trees. Its fundamental mechanism is unification, laced with a pragmatic treatment of explicit substitution forms. We decided on unification rather than one-way pattern-matching because it allows us to use Jape as a Prolog-style calculator, solving question-problems such as

$$\lambda x.\lambda y.(x\ y) : \_T$$

which would be completely intractable, or pointless, in a one-way-matching engine.

Tactics in Jape organise the application of other tactics. The simplest tactic is an inference rule.

On top of its basic proof mechanism Jape provides you with the opportunity to control the graphical user interface by programming its response to the basic gestures of pointing and clicking, and by defining what is included in the menus and panels shown to the user.

1.1 Flexible syntax

Jape has a built-in collection of syntactic forms which you can customise and to which you can add the particular details which are appropriate to your particular logic. It recognises numbers, strings, identifiers, unknowns, bracketed formulae, tuples, substitutions, juxtapositions, and formulae made by using user-defined prefix, postfix and infix operators, with user-defined priorities and associativity. In addition you can invent various new kinds of brackets and punctuation.

Identifiers — names like A, x or F in conjectures, theorems and rules — rarely stand for themselves. For the most part they stand for some arbitrary formula, variable or predicate which can appear in an instance of the conjecture, theorem or rule in which they are used. When you define the syntax of identifiers in your logic you say which are schematic identifiers and which are constants. At the same time you can define the syntactic category of the identifiers you use.

The flexible syntax mechanism is illustrated in every chapter, and detailed in appendix A.

1.2 Inference rule matching

Consider an example rule of the single-conclusion sequent calculus:

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ {\scriptstyle \vee\vdash}$$

Its rendition in Japeish (see the file SCS_rules.j) is a straightforward linearisation.

```
    RULE "∨⊢" FROM Γ, A ⊢ C AND Γ, B ⊢ C INFER Γ, A∨B ⊢ C
```

Jape's interpretation of the rule is as a description of a node in a proof tree by pattern: the consequent at that node has a collection of formulae on its left-hand side, one of which matches $A \vee B$ for some pair of formulae A and B and the rest of which are taken as Γ, and a single right-hand side formula which matches C. The node has two antecedents, each of which contains a sequent with the same right-hand side formula C. The left-hand antecedent will have a sequent whose left-hand side is Γ together with the formula A; the right-hand antecedent's left-hand side is Γ together with B.

Jape makes proofs by unifying (i.e. matching) consequents of rules to tips of the tree, and replacing the tip with the corresponding node. With the exception of RESOLVE and CUTIN steps, described later, that's *all* it does. Bernard and I, like many before us, fixed on that simple mechanism because we thought we had a chance of getting it right. It means that Jape only works backwards — from consequent to antecedents — even when it seems not to be.

The single rule above contains lots of different kinds of symbol. There are the schematic identifiers Γ, A, B and C: there is the connective ∨; there are the punctuation marks ⊢ and comma; there's the (quoted) rule name ∨⊢; and there are the reserved words of Japeish RULE, FROM, AND and INFER. Not all the logical identifiers are schematic: ∨ is in a sense an identifier, but it plays a fixed syntactic rôle in the logic and in the rule. Apart from the connectives, other identifiers might be non-schematic: there could be constant identifiers true and false, for example. As logic describer you have control over the matter, which you exercise by organising identifiers into syntactic categories. This not only allows you to distinguish between schematic and other identifiers, but it also allows you to distinguish, for example, between names like A which might be taken to stand for some arbitrary formula, and names like x which you might wish to stand only for variables.

Parameters of rules

In simple cases the fact that Jape uses unification rather than one-way pattern matching doesn't have a visible effect on the course of a proof. But if a rule doesn't have the subformula property — if there are names in its antecedents that don't appear in its consequent, as for example in

$$\frac{\Gamma, A \vdash B \wedge \neg B}{\Gamma \vdash \neg A}\ \neg - I$$

— then *unknowns* may appear in the proof in place of the schematic identifier B.[1]

In these and other circumstances it can be useful to allow the user to provide an argument formula which modifies the instantiation step. You do that by writing the rule definition with a parameter. The rule above is written in Japeish as

```
    RULE "¬-I"(B) IS FROM Γ,A ⊢ B∧¬B INFER Γ ⊢ ¬A
```

Given a problem sequent $E, F \rightarrow \neg E \vdash \neg F$, Jape first instantiates the rule by replacing all the schematics with corresponding unknowns; then it unifies $\_\Gamma$ with $E, F \rightarrow \neg E$ and $\_A$ with F; thus it generates the antecedent $E, F \rightarrow \neg E, F \vdash \_B \wedge \neg\_B$. If it is given the argument formula E to unify with $\_B$, it will generate the antecedent $E, F \rightarrow \neg E, F \vdash E \wedge \neg E$. In many cases an argument supplied to the application of a rule can prevent a startling proliferation of unknowns in a proof.

[1] This is a *strength* of Jape, not a weakness: we don't require the user to decide prematurely on the identity of those unknowns. As a logic designer, however, you can decide not to let your users see very many unknowns: see chapter 10, for example

The parameter of a rule may in some circumstances be decorated with the word OBJECT. That indicates that in the absence of a user-supplied argument, the instantiation step is to generate a freshly-minted identifier in its place rather than a fresh unknown. Frequently this is because the rule expresses a generalisation step in the logic and it is natural for Jape to mint a fresh name. For example, the rule

$$\frac{\Gamma \vdash \exists x.A \quad \Gamma, A[x \backslash c] \vdash C}{\Gamma \vdash C} \ (\textsc{fresh}\ c,\ c\ \textsc{notin}\ \exists x.A)\ \exists\text{-E}$$

is written as

```
RULE "∃-E"(OBJECT c) WHERE FRESH c AND c NOTIN ∃x.A IS
  FROM Γ ⊢ ∃x.A AND Γ,A[x\c] ⊢ C INFER Γ⊢C
```

OBJECT parameters can be used in other circumstances — in particular, see the discussion of substitution unification below.

1.3 Explicit provisos

Lots of rules have side conditions: the ∃-E rule above, for example, has two. There are traditional ways of dealing with these in proof machinery. Jape takes the simplistic route and includes explicit, visible *provisos* in the proof. Jape's provisos at present are NOTIN and UNIFIESWITH, plus three macro-relatives of NOTIN: FRESH, HYPFRESH and CONCFRESH.

Provisos are either satisfied or violated, and they constrain the application of rules. If an attempted application would violate a proviso, whether one contained in the rule itself or one left over from an earlier stage of the proof, then the attempt fails. If it is impossible to determine the status of a proviso, because it contains unknowns and/or substitution forms, then it is stored, displayed as part of the proof, and carried forward in the expectation that its status will become clearer.

The proviso x NOTIN E is satisfied if x doesn't appear free in E, and violated if it does — or rather, it is satisfied if x *cannot* appear free in E, no matter what future unifications may happen and no matter how the schematic identifiers of the conjecture being proved are instantiated. The proviso x NOTIN y, for example, is not trivially satisfied if x and y are schematic parameters of a conjecture being proved, even though visibly y is not the same variable as x.

NOTIN provisos are either included in the statement of a rule or generated from FRESH, HYPFRESH or CONCFRESH provisos: FRESH x generates a proviso x NOTIN E for every left- and right-hand side formula E of the sequent matching the rule; HYPFRESH x generates NOTIN provisos only for the left-hand side formulae and CONCFRESH for the right-hand side formulae.

The proviso $E1$ UNIFIESWITH $E2$ is internally generated. It allows Jape to defer difficult unifications where it can't find a most-general unifier. This can arise because of difficulties in unifying substitution forms, or when using multiplicative (context-splitting) rules. It plays a rôle in Jape's drag-and-drop mechanism for resolving context splits: see below.

1.4 Conjectures and theorems

We allow the user to state a conjecture using the THEOREM directive.[2] A proved conjecture becomes a theorem and can then be applied as a derived rule. If the state variable `applyconjectures` is set to true then unproved conjectures can be applied as well.

The THEOREM directive gives the name of a conjecture and its sequent, and it may also include provisos which will be enforced both during the proof of the conjecture and whenever the theorem is applied. It is possible to define a theorem without giving a name, in which case the sequent itself is used as the name. The THEOREMS directive allows you to state a collection of conjectures, and will give each its own sequent as a name. For example, the SCS.jt file defines a conjecture called *contradiction*

```
THEOREM contradiction IS A, ¬A ⊢ B
```

and the sequent_problems.j file includes a large collection of conjectures named by their sequent, some of which are as follows:

```
THEOREMS PropositionalProblems ARE
  P→(Q→R) ⊢ (P→Q)→(P→R)
AND  P→(Q→R), Q ⊢ P→R
  ....
AND  WHERE x NOTIN P INFER P ∧ ¬P, ∀x.P→Q, ∀x.  ¬P→Q ⊢ ∀x.Q
AND  R ∧ ¬R, ∀x.R→S, ∀x.  ¬R→S ⊢ ∀x.S
....
AND  ∃y.P ⊢ ∀y.P
END
```

The conjectures in this illustration are called “P→(Q→R) ⊢ (P→Q)→(P→R)”, “P→(Q→R), Q ⊢ P→R”, “P∧¬P, ∀x.P→Q, ∀x.¬P→Q ⊢ ∀x.Q” (the provisos aren’t part of the name), “R∧ R, ∀x.R→S, ∀x.¬R→S ⊢ ∀x.S” and “∃y.P ⊢ ∀y.P”.

1.4.1 Proving a conjecture — substitutions and provisos

A proof of a conjecture begins with a tree which consists of the base sequent of the conjecture, together with any provisos which were included in the statement of the conjecture[3]. The proof is then developed by application of rules and tactics.

One important feature of the proof is Jape’s treatment of the identifiers and unknowns that appear in the base sequent of the conjecture, and its treatment of other identifiers that may be introduced during the proof process. Jape’s theorems are theorem schemata, not particular theorem formulae; they may therefore be instantiated in the same way as a rule, replacing identifiers in the theorem sequent by unknowns or arbitrary formulae. Identifiers in the conjecture’s sequent can’t, therefore, be treated as standing for themselves during the proof.

In practice this means that substitution forms involving those identifiers may not be simplifiable: if, for example, identifiers A and x appears in the conjectured sequent then $A[x\backslash E]$ can’t be replaced by A unless

[2] Bernard and I had rows over this notation. I wanted to emphasise that it’s a conjecture till it’s proved; he wanted to emphasise that it is theorems, after all, that you are trying to prove. I still think I was righter than he was.

[3] Jape sometimes adds invisible provisos which it deduces from the binding structure of the formulae in the conjecture: see below, and also appendix C.

it is certain that there will never be an instance of the theorem in which the formula which instantiates A has a free occurrence of the variable which instantiates x. But if x is a name introduced during the proof — for example, by application of a rule which has an OBJECT parameter — and if there aren't any unknowns remaining which would allow x to be smuggled into the sequent finally proved, then we reason that whatever argument formula instantiates A, we could choose x within the proof to be distinct from all the names in A, and therefore $A[x\backslash E]$ can be replaced by A. In other circumstances the assurance that x can't occur free in A can come from a NOTIN proviso, or from meta-theoretical reasoning about the relationships of names in the conjectured sequent.

Provisos that are introduced during proof of a conjecture, by application of rules or other theorems or conjectures, and which aren't evidently satisfied or violated are retained as part of the theorem and checked whenever the theorem is applied.

The effect of our care with substitutions and provisos is that the proof tree which establishes the validity of a conjecture stands for all the proof trees of all the instances of that conjecture, and Jape is justified in using such a conjecture as a derived rule.

1.4.2 Applying a theorem: the rôle of structural rules

A theorem is, in principle, a rule with no antecedents. So Jape can instantiate it as a rule and match it to a problem sequent just as a rule is matched. There are, however, a couple of interesting points.

The first is that in many cases a theorem won't have enough left-hand or right-hand side formulae to completely match a problem sequent. The theorem "P→(Q→R) ⊢ (P→Q)→(P→R)", for example, matches only sequents with exactly one formula on the left of the turnstile and one on the right. Often a logic will include so-called 'weakening' rules which enable you to delete a formula from the left- or the right-hand side or both. If you include such rules and declare their rôle to Jape, it will allow you to apply a theorem even though it does not completely match a problem sequent.

The second difficulty is that sometimes a theorem matches on the right-hand side, but not on the left. In such a case it is often convenient to prove it 'by resolution': that is, to generate an antecedent for each of the left-hand side formulae and to set about proving them. That step is justified if the logic contains a 'cut' rule which enables you to move formulae from left- to right-hand side and a 'weaken' rule as well. Jape will make a resolution step for you if you declare the appropriate structural rules in your logic, declare their rôles, and also set the `tryresolution` variable (see appendix C) or use one of the APPLYORRESOLVE or RESOLVE tacticals (see appendix B).

1.5 Substitution forms and unification

Jape uses explicit substitution forms — $A[x\backslash c]$, $B[x,y,z\backslash E,F,G]$ — where some logics use predicate notation — $P(c)$, $Q(E,F,G)$. Substitutions are more powerful than predicates because they are more general; for the same reason they are trickier to handle. Jape's internal mechanisms are based on substitution forms, but there is also a mechanism which allows you to write rules and theorems in terms of predicate formulae — see 'interpreting predicates' below.

Explicit substitution forms are semantically scandalous, a notorious trap for novices, and an expert will ask "what does a substitution form in a rule or theorem *mean*?". It's difficult to give a simple answer. It is never necessary to include special rules to treat substitution forms — their treatment is a fundamental mechanism of Jape, and Jape tries to eliminate substitution forms from the proof whereever and whenever they appear. Therefore we can say that Jape treats a substitution form as equivalent to the result of carrying out the subsitution. But in some situations it can be persuaded to treat a substitution form as a structural pattern and will unify one unreduced substitution form with another, even though such unifications don't give the most general answer.

A substitution form is introduced into a proof, and if possible immediately eliminated, whenever the antecedent of a rule contains one. Consider, for example, the problem sequent $x > y \vdash \exists z.z > y$. If we apply the rule

$$\frac{\Gamma \vdash A[x \backslash E], \Delta}{\Gamma \vdash \exists x.A, \Delta} \vdash \exists$$

then we generate a single antecedent $x > y \vdash (z > y)[z \backslash \_E]$, which immediately simplifies to $x > y \vdash \_E > y$ provided that we know that z and y are necessarily distinct,[4] or to $x > y \vdash \_E > y[z \backslash \_E]$ otherwise.

Even more interesting is what happens when a rule contains an explicit substitution form such as $A[x \backslash E]$ in its *consequent*. When the rule is applied Jape must unify that substitution form — or rather, its instantiated form which in general will be $\_A[\_x \backslash \_E]$ — with some formula B in the problem sequent. That sort of unification is notoriously difficult, and Jape uses a number of ad-hoc strategies to help.

1. It simplifies substitution forms whenever possible, in order to avoid the problem.
2. It defers the unification of an irreducible substitution form for as long as possible, so that the results of other unifications can be used to simplify it.
3. If the user provides an argument formula *F* in place of parameter E, the instantiated form will be $\_A[\_x \backslash F]$; when Jape can no longer avoid unifying that form with B it will search for all instances of *F* inside B and try to construct a substitution form $B'[\_x \backslash F]$ which simplifies to B in presence of the proviso $\_x$ NOTIN B; if successful it will unify $\_A$ with B' in a context that records the proviso. The process is far more effective if the parameter x is decorated with the word OBJECT, so that the instantiated form becomes $\_A[z \backslash F]$ where z is a fresh variable; the formula $B'[z \backslash F]$ is easier to construct and to simplify, and the proviso z NOTIN B is easier to check[5].
4. If the user text-selects instances of a sub-formula *F* of B, then the logic encoding can employ the WITHSUBSTSEL tactical — see chapter 6 and appendix B — to calculate B' by replacing just those instances of *F* by a fresh unknown $\_y$; $B'[\_y \backslash F]$ necessarily simplifies to B given the proviso $\_y$ NOTIN B; then Jape will match the substitutions structurally, unifying $\_A$ with B', $\_x$ with $\_y$ and $\_E$ with *F* in a context which records the proviso. If the parameter x is decorated with the word OBJECT then $\_x$ is replaced by z and the proviso becomes z NOTIN B, which is once again easier to check.
5. If all else fails, Jape can generate a proviso $\_A[\_x \backslash \_E]$ UNIFIESWITH B, and await developments.

If Jape has to unify two substitution forms which have identical variable lists then it matches them structurally. For example, it can unify $A[x, y \backslash A1, A2]$ with $B[x, y \backslash B1, B2]$ by unifying A with B, *A1* with *B1*, *A2*

[4] They might not be if, for example, they both appear in the base sequent of the conjecture being proved. They may be if, for example, one or the other has been generated during the development of the proof, or if there is an explicit proviso which makes it clear that they are distinct.

[5] In fact, because z is a fresh variable, the proviso is usually obviously satisfied. But a proviso is necesssary to constrain the future course of the proof if there are unknowns in B. In those and in some other circumstances Jape may also produce UNIFIESWITH provisos to cater with the process of abstraction in the nasty bits of B.

with *B2*. This happens rarely and because it doesn't generate the most general unifier it might not be the best thing to do, but pragmatically it seems to work rather well almost every time it is used.

If Jape has to unify substitution forms with different variable lists then it extends one or the other: for example, if it has to unify $A[x\backslash A1]$ with $B[x,y\backslash B1,B2]$ it will try to construct A' such that $A[y\backslash B2]$ simplifies to A, and then unify $A[x,y\backslash A1,B2]$ with $B[x,y\backslash B1,B2]$. In certain circumstances it will even do a bit of α-conversion — but enough! this explanation is sufficiently complicated already.

The message is that Jape's unification of substitution forms is usefully pragmatic. It does not always generate a most-general unifier but it can, in practice, often generate just the unifier that the user is looking for, especially when the encoding uses the LETSUBSTSEL/WITHSUBSTSEL mechanism (see chapter 6 and appendix B) to allow the user to describe the unification to Jape. It most often breaks down when it has to deal with substitutions using variables which also appear in the base sequent of the theorem being proved. That breakdown is, I think, inevitable, (and investigations stopped some time ago).

1.5.1 Invisible provisos

Consider the conjectures $\lambda x.\lambda y.x : T1 \rightarrow T2 \rightarrow T1$ and $\forall x.\exists y.P(x) = P(y)$. Clearly, in each case, any instance of the conjecture would have to use two distinct variables. Nothing else would give the right binding structure: $\lambda z.\lambda z.z : \mathrm{int} \rightarrow \mathrm{real} \rightarrow \mathrm{int}$ isn't an instance of the first conjecture, nor $\forall z.\exists z.Q(z) = Q(z)$ of the second[6]. But Jape's mechanisms of rule and theorem instantiation don't automatically ensure this: instead, there has to be a proviso such as x NOTIN y in each case. Such provisos are fussy, have to do with the internal mechanisms of Jape, and are difficult to explain to Jape's users. Therefore Jape generates them automatically, from an analysis of the binding structure of every rule and conjecture, and then makes them invisible. You can see the invisible provisos in a proof by setting the `showallprovisos` variable to true.

1.5.2 Interpreting predicate notation

Some of our users prefer predicate notation to substitution, and in certain ways it concisely conveys more information. In the formula $\forall x.P(x)$ it is implicit that the predicate formula P doesn't contain any instances of x; in the corresponding formula $\forall x.P[v\backslash x]$ no such inference can be drawn, and the statement of a theorem which contained such a formula would require a proviso x NOTIN P to say as much as the predicate version. Fussy provisos get substitution notation a bad name, so we have implemented a mechanism which interprets predicate notation, translating it into substitution notation and inserting invisible provisos.[7] When you apply a rule which contains $\forall x.P(x)$, for example, Jape translates it into $\forall x.P[v\backslash x]$, automatically inserting x NOTIN P. When you begin a proof which contains $\forall x.P(x)$, Jape doesn't translate it, but it does insert the same proviso, making it invisible.

If you set the variable `interpretpredicates` to true, Jape treats every juxtaposition as if it were a predicate application. If `interpretpredicates` is false (the default), Jape only interprets those juxtapositions in which the first formula is an ABSTRACTION parameter name. See chapters 5 and 6 for examples.

In one respect Jape's interpretation of predicate notation is pragmatically helpful rather than careful. Consider, for example, the sequent $\exists x.\forall y.P(x,y) \vdash \forall y.\exists x.P(x,y)$. Jape translates this to $\exists x.\forall y.P[u,v\backslash x,y] \vdash \forall y.\exists x.P[u,v\backslash x,y]$, and automatically includes provisos x NOTIN P and y NOTIN P, as it should. Jape also

[6] Strictly speaking, this second *might* be an instance of the the conjecture, if there are no instances of z in Q. These are deep waters...

[7] Those invisible provisos constrain the proof. If they were too constraining, the worst that could happen is that you miss some proofs.

includes x NOTIN y, which isn't essential in order to preserve the binding structure, because it is not required that either x or y must appear free in a predicate $P(x, y)$. The effect is that certain instances of the theorem are excluded. In practice it seems that our users prefer it this way.

1.6 Binding forms: unification, α-conversion and substitution

Suppose that $\forall var.formula$ has been defined to be a binding form: then Jape will proceed as follows:

- it will unify $\forall x.A$ with $\forall x.B$ by unifying A with B;
- it will unify $\forall x.A$ with $\forall\_y.B$ by unifying x with $\_y$ and A with B
- it will unify $\forall x.A$ with $\forall y.B$ by unifying $\forall z.A[x\backslash z]$ with $\forall z.B[y\backslash z]$, where z is a fresh variable, together with the provisos z NOTIN A and z NOTIN B.

Jape respects binding forms when carrying out substitutions. Thus, for example, if $\forall var.formula$ has been defined to be a binding form and x and y are guaranteed distinct then $(\forall x.A)[x\backslash E]$ always simplifies to $\forall x.A$ and, provided that x doesn't appear free in *F*, $(\forall x.A)[y\backslash F]$ will simplify to $\forall x.(A[y\backslash F])$; in other circumstances it will simplify to by α-conversion to $\forall z.(A[x, y\backslash z, F])$ together with the proviso z NOTIN A, where z is a fresh variable.

1.7 The tactic language

Although Jape's basic operation is the application of rules to tips of a proof tree, that is by no means the whole story. You will often find it necessary to organise the application of rules by writing programs in the tactic language.

Bernard originally invented the tactic language[8] to enable us to express programmed actions like proof searches. It's become a workhorse for many more tasks than that.

The simplest tactics are inference rules. You can apply tactics sequentially (SEQ), or try one after another (ALT, WHEN), you can call tactics with arguments, you can repeat tactics (DO); there is a notion of the 'current goal' sequent in the tree which is used when tactics are applied in sequence. It is possible, under very severe constraints, to transform formulae within the goal sequent (FIND, FLATTEN, WITHSUBSTSEL).

Most of the language has to do with the interpretation of gestures and selection of an appropriate response.

Appendix B gives a complete list of all the verbs of the tactic language. The chapters of this manual give examples of their use.

1.8 Gestures, menus and panels

The user can make certain 'gestures' at the Jape graphical interface. The way in which the gestures are made — which buttons and keys are pressed and how the mouse is moved — depends on the operating system. In general a 'click' is a left-button mouse click, a 'subformula click' is a middle-button click (or a click with the alt/option key held down).

[8] Over my dead body, several times. This is one battle I'm glad I lost.

- A user can *select* a formula in a sequent by clicking it. If a rule is then applied, Jape requires by default that the selected formula is a principal formula in the rule. Thus, for example, if you select the hypothesis $E \vee F$ in the sequent $G \vee H, E \vee F, G \rightarrow E \vdash G \wedge E$ and then apply the rule

  $$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} \vee\vdash$$

  you ensure that $A \vee B$ in the rule matches $E \vee F$ in the sequent, Γ matches $G \vee H,\ G \rightarrow E$ and, of course, C matches $G \wedge E$. (A tactic can test for formula selection, discover the formula selected, and modify its behaviour accordingly.)

- A user can double-click (‘hit’) on a formula, causing the application of a tactic chosen by the logic description.

- A user can double-click on the ‘reason’ or ‘justification’ of a proof step. If there is hidden detail behind that step then it will be revealed, or if it has been revealed by an earlier double-click, it will be hidden again.

- A user can drag a formula. If there is a UNIFIESWITH proviso, generated as a result of context-splitting in a multiplicative rule, some of the other formulae mentioned in that proviso — unknown segment variables like _Γ or _Δ1 — will highlight as the formula is dragged across them.

- A user can *subformula-select* part or all of a formula in a sequent. If a rule is applied, the subformula is provided as the first argument. If a tactic is applied, it can test for subformula selection and modify its behaviour accordingly.

- A user can select an entry in a menu, and Jape will carry out the corresponding command. Most entries correspond to the command `apply T` for some tactic `T`, but a menu can contain any of the commands listed in appendix C. A good deal of your user-interface design activity will go into deciding what goes in which menus, fixing on labels for each entry and choosing just the right commands.

- A user can press a button in a panel, with or without first choosing an entry from the list of entries in the same panel. Many panels list conjectures, and their buttons allow users to prove the chosen conjecture, apply it as a theorem and so on. Other panels may be like menus. The designer controls what is in the entries and what is on the buttons, and whether or not a particular button sends just a command, or a command modified by the selected entry.

- A user can scroll the proof horizontally and/or vertically.

And that’s it. Jape uses a very impoverished vocabulary of gesture: we have chosen to make it so, in an attempt to make Jape as straightforward to use as any other application in a modern GUI environments.

1.9 Proof display: trees, boxes and hiding

The Gentzen tree is the basic proof structure on which Jape works. Behind the scenes, whatever is on the screen, is a Gentzen tree. Tactics can be used to hide selected antecedents of a proof step and alter the ‘reason’ or ‘justification’ displayed with the step; the hidden detail can be revealed to a user who double-clicks appropriate parts of the proof.

Gentzen trees are notoriously wasteful of display space, and Fitch boxes famously less so. Jape can display a proof in an approximation to Fitch box style. The display is a transcription — not a translation — of the tree, and it can be applied to any kind of logic, not simply natural deduction. This is how it’s done:

- the assumptions — left-hand side formulae — of the base sequent are written on the first line and the conclusion(s) — right-hand side formula(e) — on the last line;
- if a line is the conclusion of a proof step then the lines representing the trees of its antecedents are written out before it, working left to right through the antecedents;
- the justification of a line which is the conclusion of a proof step references the assumption line(s) to describe any left-hand side principal formulae, as well as the lines which contain the conclusions of its antecedents;
- if a line is the conclusion of a tip then a line of dots is written before it;
- if an antecedent introduces any hypotheses then its lines are written in a box, whose first line is those hypotheses and whose last line is the right-hand side formula(e) of the antecedent.

That makes a fairly compact description, in which hypotheses are written only once but conclusions may be written more often, especially when a left-hand side rule is used. It is made still more compact by hiding applications of IDENTITY (aka axiom, hypothesis) rules, and it is made to support some forms of forward reasoning (see chapters 5 and 10) by hiding, under the right circumstances, applications of a CUT rule.

If you select a conclusion formula in a box display, the effect is just as if you had selected the corresponding conclusion formula in the underlying Gentzen tree. If you select a hypothesis formula the effect can't be so simple, because a hypothesis formula is written only once even though it may occur in many sequents: Jape finds the set of sequents that you could be pointing to and disambiguates the choice using any conclusion selection that you might have made.

It doesn't make sense to use box display with a multiple-conclusion calculus for various reasons, and Jape's gesturing mechanisms therefore haven't been adapted to this use.

Our box display isn't a proper Fitch box display because you can't necessarily use the proof which ends on line j when making a proof step on a subsequent line k, even though the box structure would allow it. The reason is that line j may be part of the proof of some cousin of k, not part of the proof of k — that is, parts of the proof which are sequentially related in the box display aren't necessarily hierarchically related in the underlying Gentzen tree. There is a tactic-language solution (see CUTIN in chapter 10), but without that Jape provide some assistance by making the underlying tree structure more evident when the user selects an assumption or a conclusion: once you make a selection it will 'grey out' lines in the box display which are irrelevant because they are not hierarchically related in the underlying tree.

1.10 Using Jape interactively

Jape starts up 'empty', with no theory loaded.

You can load a new theory into Jape by using File:Open New Theory. At any time you can add additional bits of Japeish to the brew, by using File:Open. Jape works, like LISP or ML, by maintaining a store of definitions, and it is always possible to add to those definitions. The effects may be strange, especially if you try to add a new theory without getting rid of the old one first!

Chapter 2

Encoding the Sequent Calculus

Jape is, at bottom, a backwards-reasoning proof editor working on a tree of sequents. It is therefore no surprise that it is exceptionally straightforward to encode the sequent calculus in Jape. We describe in this chapter the encoding of the multiple-conclusion sequent calculus in the file `examples/sequent_calculus/MCS.jt` and the files it references.

The rules we use, and the examples, were borrowed from Roy Dyckhoff's excellent MacLogic (an inspiring precursor of Jape). This encoding draws heavily on the single-conclusion sequent calculus encoding (see `examples/SCS.jt`), developed jointly by Bernard Sufrin and myself in the early days of Jape, so 'we' is Bernard and me. The encoding is quite old, and the segmentation into files is somewhat arbitrary.

2.1 The inference rules of the (multiple-conclusion) Sequent Calculus

We have encoded a standard version of the sequent calculus. By making our left- and right-hand sides *bags* (aka multisets) of formulae we have avoided the need for exchange rules; by allowing the axiom rule to ignore unnecessary hypotheses and conclusions we have avoided the need to use weakening rules in almost every case and/or to describe context-splitting rules. See chapter 3 for an alternative treatment of quantifiers and variables and for Jape's treatment of context-splitting (multiplicative) rules.

The rules, given in tables 2.1, 2.2, 2.3 and 2.4, are additive — that is, don't split their left- or right-hand side contexts working backwards. Multiplicative (context-splitting) rules are harder to use in a backwards reasoning tool, because either the tool must force the user to decide how to split the context before the rule is applied or else it must provide machinery to allow the decision to be deferred (Jape chooses deferral: see chapter 3). In this encoding, the fact that the *axiom* rule ignores unnecessary hypothesis and conclusion formulae makes context-splitting on either side unnecessary.

Because Jape interprets predicate notation as shorthand for substitution, the *actual* quantifier rules use substitution. Table 2.5 shows the rules which Jape employs, translating the originals on input. The difference need hardly detain us: there are no additional provisos, and no substitution-matching is required. In this logic at least, it's easy to believe that Jape manipulates predicate notation directly.

Table 2.1: The `axiom` rule

$$\frac{}{\Gamma, A \vdash A, \Delta}\ \text{axiom}$$

Table 2.2: Introduction to the right of the turnstile (⊢... rules)

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \vdash\wedge \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta}\ \vdash\rightarrow \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vdash\vee \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \vdash\neg$$

$$\frac{\Gamma \vdash A \rightarrow B, \Delta \quad \Gamma \vdash B \rightarrow A, \Delta}{\Gamma \vdash A \equiv B, \Delta}\ \vdash\equiv \qquad \frac{\Gamma \vdash A(m), \Delta}{\Gamma \vdash \forall x.A(x), \Delta}\ \text{(fresh } m\text{)} \vdash\forall \qquad \frac{\Gamma \vdash A(B), \Delta}{\Gamma \vdash \exists x.A(x), \Delta}\ \vdash\exists$$

Table 2.3: Introduction to the left of the turnstile (...⊢ rules)

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge\vdash \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta}\ \rightarrow\vdash \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee\vdash \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg\vdash$$

$$\frac{\Gamma, A \equiv B \vdash C}{\Gamma, A \rightarrow B, B \rightarrow A \vdash C}\ \equiv\vdash \qquad \frac{\Gamma, A(B) \vdash \Delta}{\Gamma, \forall x.A(x) \vdash \Delta}\ \forall\vdash \qquad \frac{\Gamma, A(m) \vdash \Delta}{\Gamma, \exists x.A(x) \vdash \Delta}\ \text{(fresh } m\text{)} \exists\vdash$$

Table 2.4: Structural rules

$$\frac{\Gamma \vdash B, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ \vdash \text{weaken} \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ \vdash \text{contract} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{weaken} \vdash$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{contract} \vdash$$

Table 2.5: Modified quantifier rules

$$\frac{\Gamma \vdash A[x \backslash m], \Delta}{\Gamma \vdash \forall x.A, \Delta}\ \text{(fresh } m\text{)} \vdash\forall \qquad \frac{\Gamma \vdash A[x \backslash B], \Delta}{\Gamma \vdash \exists x.A, \Delta}\ \vdash\exists \qquad \frac{\Gamma, A[x \backslash B] \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}\ \forall\vdash$$

$$\frac{\Gamma, A[x \backslash m] \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta}\ \text{(fresh } m\text{)} \exists\vdash$$

2.2 Syntax description (`sequent_syntax.j`)

We begin with a couple of variable initialisations:[1]

```
INITIALISE interpretpredicates true
INITIALISE displaystyle tree
```

This ensures that juxtapositions are interpreted as predicate application, and that we see the proofs as trees.

We use names starting with A, B, C, D, P, Q, R and S in rules and conjectures to stand for any formula; we use names starting with u, v, w, x, y, z. m or n to stand for any variable. Names starting with Γ or Δ stand for bags (multisets) of formulae[2].

```
CLASS BAG Γ Δ
CLASS FORMULA A B C D P Q R S
CLASS VARIABLE u v w x y z m n
```

These directives also cover unknowns: an unknown which starts $\_A$ will unify with any formula, but one which starts $\_z$ will only unify with a variable or a similar unknown.

Quantifier formulae are, like if-then-else in most programming languages, bracketed without a final closing bracket:

```
LEFTFIX 20 ∀ .
LEFTFIX 20 ∃ .
```

The connectives are defined as operators with particular *binding power* and *binding direction*:

```
INFIX 100L ≡
INFIX 110R →
INFIX 150L ∧
INFIX 160L ∨
PREFIX 200 ¬
```

The relative binding power of juxtaposition and substitutions are defined:

```
JUXTFIX 300
SUBSTFIX 400
```

Working from the bottom, this defines substitution forms as the most binding, then juxtaposition. Next comes ¬ defined as a prefix operator, then the binary connectives (all but → are defined to be left-associative, while → is right-associative). Finally two special bracketed forms are defined, with the lowest syntactic priority. These definitions allow us to write:

- $\neg prim$, where $prim$ is an atomic formula, a substitution or a juxtaposition (see appendix A);
- $f1 \lor f2$, $f1 \land f2$ or $f1 \to f2$ with the interpretation that $\lor$ 'operators' have priority over $\land$, and both have priority over $\to$;

[1] The syntactic form we use is that of an assignment to a variable. The `interpretpredicates` variable can only be altered when the store of rules and variables is empty, so in practice it behaves as a parameter.

[2] Note that now there are no commas in these lists of identifier prefixes: I have eliminated use of comma as a separator in the paragraph language. See appendix A for more details

- $\forall$ *f1* . *f2* and $\exists$ *f1* . *f2*.

Note that the leftfix patterns don't constrain you to write $\forall$*variable*.*formula*: that comes later.

Because there is no closing bracket, formulae constructed with LEFTFIX bracketing are liable to have a visually ambiguous interpretation, so Jape demands that LEFTFIX-brackets aren't used like ordinary brackets: that is, you can't write things like $f1 \wedge \forall x.f2 \wedge f3$: you have to write instead either $f1 \wedge (\forall x.f2 \wedge f3)$ or $f1 \wedge (\forall x.f2) \wedge f3$.

The binding structures are given by pattern:

```
BIND x SCOPE P IN ∃x .  P
BIND x SCOPE P IN ∀x .  P
```

Any formula which matches one of these patterns is recognised as a binding formula — any variable in place of *x*, any formula, including another binding formula, in place of P. Near matches aren't allowed, so the constraint to write *quantifier variable*. *formula* is enforced.

Note that this defines only single-variable bindings. Jape has no means at present of defining families of binding structures, except by exhaustively listing them — for example, you might give BIND directives which describe the structure of $\forall$*x*,y.E, $\forall$*x*,*y*,$z.E$ and so on as we do in later chapters. But then you would find that Jape has no means of defining families of inference rules which work across the different kinds of bindings you can define, and you would have to separately define the rules — one for $\forall$*x*.E, another for $\forall$*x*,y.E, another for $\forall$*x*,*y*,$z.E$ and so on.

2.3 Encoding the inference rules (`MCS_rules.j`)

Our sequents have bags of formulae on either side:

```
SEQUENT IS BAG ⊢ BAG
```

Jape is designed to make the encoding of inference rules as transparent and straightforward as possible. In principle all you have to do is to linearise the normal description of a rule, giving its name, its provisos, its antecedents and its consequent. Writing {... } for optional inclusion and {... }* for repeated optional inclusion, the syntax of a RULE directive is

RULE	{ *name* }	— *rule name*
	{ (*parameter* {, *parameter* }*) }	— *parameters*
	{ WHERE *proviso* {AND *proviso*}* }	— *provisos*
	{ IS }	
	{ FROM *sequent* {AND *sequent*}* }	— *antecedents*
	INFER *sequent*	— *consequent*

Nearly everything is optional, but you have to put in enough reserved words to make it clear where each section begins and ends. If you leave the name out, the name is taken to be the consequent itself. Where the name of a rule isn't an identifier — if it is $\vdash\wedge$, for example — it is necessary to enclose it in quotation marks. The parameters are each an identifier or the word OBJECT followed by an identifier. Parameters in a rule definition control the process of instantiation and the treatment of argument formulae provided via text-selection and/or tactics.

The rules are:

```
RULE axiom(A) INFER Γ,A ⊢ A,Δ
RULE "⊢∧" FROM Γ ⊢ A,Δ AND Γ ⊢ B,Δ INFER Γ ⊢ A∧B,Δ
RULE "∧⊢" FROM Γ,A,B ⊢ Δ INFER Γ,A∧B ⊢ Δ
RULE "⊢∨" FROM Γ ⊢ A,B,Δ INFER Γ ⊢ A∨B,Δ
RULE "∨⊢" FROM Γ,A ⊢ Δ AND Γ,B ⊢ Δ INFER Γ,A∨B ⊢ Δ
RULE "⊢¬" FROM Γ,A ⊢ Δ INFER Γ ⊢ ¬A,Δ
RULE "¬⊢" FROM Γ ⊢ A,Δ INFER Γ,¬A ⊢ Δ
RULE "⊢→" FROM Γ,A ⊢ B,Δ INFER Γ ⊢ A→B,Δ
RULE "→⊢" FROM Γ ⊢ A,Δ AND Γ,B ⊢ Δ INFER Γ,A→B ⊢ Δ
RULE "⊢≡" FROM Γ ⊢ A→B,Δ AND Γ ⊢ B→A,Δ INFER Γ ⊢ A≡B,Δ
RULE "≡⊢" FROM Γ, A→B, B→A ⊢ Δ INFER Γ,A≡B ⊢ Δ
RULE "⊢∀"(OBJECT m) WHERE FRESH m
  FROM Γ ⊢ A(m),Δ INFER Γ ⊢ ∀x.A(x),Δ
RULE "∀⊢"(B) FROM Γ, A(B) ⊢ Δ INFER Γ,∀x.A(x) ⊢ Δ
RULE "⊢∃"(B) FROM Γ ⊢ A(B),Δ INFER Γ ⊢ ∃x.A(x),Δ
RULE "∃⊢"(OBJECT m) WHERE FRESH m
  FROM Γ,A(m) ⊢ Δ INFER Γ, ∃x.A(x) ⊢ Δ
RULE cut(A) FROM Γ ⊢ A,Δ AND Γ,A ⊢ Δ INFER Γ ⊢ Δ
RULE "weaken⊢"(A) FROM Γ ⊢ Δ INFER Γ,A ⊢ Δ
RULE "⊢weaken"(A) FROM Γ ⊢ Δ INFER Γ ⊢ A,Δ
RULE "contract⊢"(A) FROM Γ, A, A ⊢ Δ INFER Γ, A ⊢ Δ
RULE "⊢contract"(A) FROM Γ ⊢ A,A,Δ INFER Γ ⊢ A,Δ
```

The structural rules are given their proper rôles:

```
STRUCTURERULE CUT cut
STRUCTURERULE LEFTWEAKEN "weaken⊢"
STRUCTURERULE RIGHTWEAKEN "⊢weaken"
```

2.4 Automatic application of rules

It is possible to require Jape to try a tactic at the end of each proof step — that is, after producing the effects demanded by the user. You can make it apply the tactic in one of two ways: the AUTOMATCH directive requires that the tactic works without introducing or eliminating any unknowns from the proof tree, and without introducing or eliminating any provisos; the AUTOUNIFY directive doesn't have any of those constraints. With either directive, a rule within the tactic is not applied if there is more than one distinct possible result.

In the sequent calculus it is reasonable to apply *axiom* whenever possible, but because it would always be applicable whenever a conclusion or a hypothesis was a single unknown, it's prudent to restrict ourselves to applications which succeed by identical match, and we therefore include

```
AUTOMATCH axiom
```

2.5 Automatic selection of rules

When the user double-clicks on, or 'hits', a formula, the logic designer can provide that a tactic is automatically applied. The choice of tactic is made by pattern-matching and depends on whether it is a hypothesis or a conclusion that is hit. If there isn't an applicable tactic, Jape puts up an error alert.

Description of a 'hit' and what to do about it is given by one of the directives

CONCHIT *pattern* IS *tactic*
HYPHIT *pattern* IS *tactic*

The pattern is matched — by one-way matching, not unification — to the formulae which have been selected and hit. It can be as follows:

- *hypothesis* <entails> *conclusion,* in which case the user must select (click) one of the two and hit (double-click) the other;
- *hypothesis* <entails> — only in HYPHIT — in which case the user must hit a hypothesis without selecting a conclusion;
- *conclusion* or <entails> *conclusion* — only in CONCHIT — in which case the user must hit a conclusion without selecting a hypothesis.

In the sequent calculus we can automatically invoke a tactic when any formula is hit. First, we can invoke a right rule when any conclusion is hit, provided that the user hasn't confused the issue by selecting a hypothesis as well:

```
CONCHIT ⊢ B∧C IS "⊢∧"
CONCHIT ⊢ B∨C IS "⊢∨"
CONCHIT ⊢ B→C IS "⊢→"
CONCHIT ⊢ ¬B IS "⊢¬"
CONCHIT ⊢ B≡C IS "⊢≡"
CONCHIT ⊢ ∀x.B IS "⊢∀"
CONCHIT ⊢ ∃x.B IS "⊢∃"
```

We can automatically invoke *axiom* if the user hits a hypothesis having selected an identical conclusion:

```
HYPHIT A ⊢ A IS axiom
```

We can automatically invoke a left rule if the user hits a hypothesis without having selected a conclusion:

```
HYPHIT A→B ⊢ IS "→⊢"
HYPHIT A∨B ⊢ IS "∨⊢"
HYPHIT A∧B ⊢ IS "∧⊢"
HYPHIT ¬A ⊢ IS "¬⊢"
HYPHIT A≡B ⊢ IS "≡⊢"
HYPHIT ∀x.A ⊢ IS "∀⊢"
HYPHIT ∃x.A ⊢ IS "∃⊢"
```

2.6 Menus

Jape automatically provides some system menus, whose content is not described here. All other menus and panels are produced under the control of the encoder.

2.6.1 The Rules menu

To describe a menu you give its title and its contents. Each entry in the menu has a label — which the user sees — and a Jape tactic — which is transmitted to the Jape engine when the entry is selected. A rule name is the simplest form of Jape tactic, and in this logic that is all that we need:

```
MENU Rules IS
  ENTRY axiom
  SEPARATOR
  ENTRY "∧⊢"
  ENTRY "∨⊢"
  ENTRY "→⊢"
  ENTRY "¬⊢"
  ENTRY "≡⊢"
  ENTRY "∀⊢"
  ENTRY "∃⊢"
  SEPARATOR
  ENTRY "⊢∧"
  ENTRY "⊢∨"
  ENTRY "⊢→"
  ENTRY "⊢¬"
  ENTRY "⊢≡"
  ENTRY "⊢∀"
  ENTRY "⊢∃"
  SEPARATOR
  ENTRY cut
  ENTRY "weaken⊢"
  ENTRY "⊢weaken"
  ENTRY "contract⊢"
  ENTRY "⊢contract"
END
```

This produces a menu in which every label is the name of a rule, and every command a tactic of the same name. Jape allows us to save effort by defining the rules within the menu description. If we had written

```
MENU Rules IS
  RULE axiom  INFER A ⊢ A
  SEPARATOR
  RULE "⊢∧"  FROM ⊢ A AND ⊢ B  INFER ⊢ A∧B
  ...
END
```

then it would have produced exactly the same menu.

2.7 Conjectures (sequent_problems.j)

The primary object of using Jape is to prove theorems. You can state conjectures in text commands composed from the keyboard (after pressing the New... button on a conjectures panel), but it is more normal to state them in a logic-encoding file.

Panels in Jape have lists of entries and buttons. A CONJECTUREPANEL automatically includes buttons labelled New..., Prove and Show Proof, and has a default Apply button if the user defines no buttons at all. In the case of the sequent calculus we can use a straightforward CONJECTUREPANEL with a default Apply button to hold all the problems which we want to display to the user.

A conjecture can be stated in a THEOREM directive which gives its name, its parameter identifiers and provisos if any, and the sequent which is the theorem itself. The effect is to put a conjecture with that name into the 'tactic store', from which it can be retrieved in order to prove it, to apply it during a proof, or to review its proof.

Two conjectures are stated in this way in `MCS.jt`. We state the panel into which we want to put the conjectures, and give the conjectures themselves.

```
CONJECTUREPANEL "Conjectures"
  THEOREM modusponens IS A, A→B ⊢ B
  THEOREM contradiction IS A, ¬A ⊢
END
```

These directives are cumulative: the conjectures are added to the panel unless there are already entries with the same name, in which case these replace the previous versions.

Because this logic includes a cut rule and both a left and a right weakening rule, the contradiction theorem can be applied, once proved, to any sequent which has a formula and its negation in its hypotheses. It could even be applied automatically via AUTOMATCH, though we haven't done that in this encoding.

Interpretation of parameter identifiers and provisos in a THEOREM directive is the same as for inference rules.

The rest of the conjectures are given in `sequent_problems.j` without names, so that they appear in the panel under names generated from their sequent:

```
CONJECTUREPANEL "Conjectures"
  THEOREM INFER P→(Q→R) ⊢ (P→Q)→(P→R)
  THEOREM INFER P→(Q→R), Q ⊢ P→R
  THEOREM INFER R→S ⊢ (P→R) → (P→S)
  THEOREM INFER P→(P→Q) ⊢ P→Q
  THEOREM INFER P ⊢ Q→(P∧Q)
  ...

  THEOREM INFER ∀x.¬Q(x), P→(∀x.Q(x)) ⊢ ¬P
  THEOREM WHERE x NOTIN P INFER
    P∨¬P, ∀x.P→Q(x), ∀x.¬P→Q(x) ⊢ ∀x.Q(x)
  THEOREM INFER R∨¬R, ∀x.R→S(x), ∀x.¬R→S(x) ⊢ ∀x.S(x)
  THEOREM INFER ∀x.P(x)→Q(x), ∀x.Q(x)→R(x) ⊢ ∀x.P(x)→R(x)
  THEOREM INFER
    ∀x.P(x)→R(x), ∀x.Q(x)→ ¬R(x) ⊢ ∀x.(P(x)→¬Q(x))∧(Q(x)→¬P(x))
  ...

  THEOREM INFER ¬¬P ⊢ P
  THEOREM INFER P ⊢ ¬¬P
  ...
END
```

The first section adds a number of propositional theorems to the tactic store, each under the name of its sequent. The second section adds theorems which include quantified formulae, some of which need individual provisos (we included two versions of some theorems, just to show how the necessary provisos are generated during the proof if you don't add them beforehand).

2.8 Global variable settings

Jape has a number of variables which control parts of its operation — for a complete list see appendix C. In our encoding of the sequent calculus we have decided not to allow conjectures to be applied as theorems, not to allow theorems to be applied 'resolution' style, generating antecedents if all their hypotheses don't match, and to display our proofs as Gentzen trees. The initialisations are:

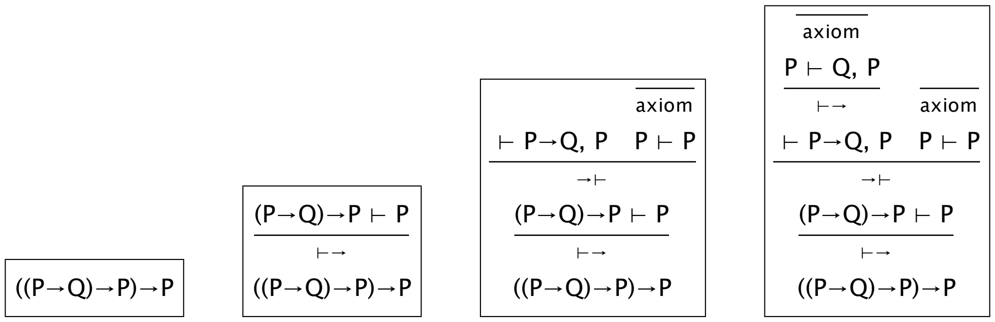


Figure 2.1: Proof of Peirce's law by double-clicking in the multiple-conclusion sequent calculus

```
INITIALISE applyconjectures false
INITIALISE tryresolution false
INITIALISE displaystyle tree
```

2.9 A very small example

Figure 2.1 shows the progress of a proof of Pierce's law in this encoding (which, you can see, is resoundingly classical).

Chapter 3

Variations on the Sequent Calculus

The calculus of chapter 2 is only one of very many possible variants. In this chapter I discuss the way in which you can encode an LF-style treatment of variables in the quantifier introduction and elimination rules, how Jape deals with non-additive rules, and two versions of the intuitionistic sequent calculus — multiple and single conclusion.

3.1 LF-style variables in quantifier rules

Jape allows redefinition of any rule, theorem or conjecture[1]. The file `examples/MSC_LF.j` redefines the quantifier rules to allow a more careful treatment of variables[2]. The new rules are shown in table 3.1.

The intention is that a variable c is '$\mathrm{inscope}$' if there is an assumption $\mathrm{var}\, c$; a formula is $\mathrm{inscope}$ if its free variables. Note that there is nothing in Jape which demands that we use these words nor this technique: it's up to the logic decoder to program it.

The file `sequent_scoping.j` defines two low priority prefix operators:

```
    PREFIX 10 var
    POSTFIX 10 inscope
```

and a structural induction to handle formulae,[3] automatically applied whenever there is an open tip:

[1] And it allows it *at any time*!! It ought to check, whenever a rule or theorem is redefined, every proof that relies upon it. It doesn't.

[2] Explanation for non-expert logicians: the effect is to make it much more careful about the treatment of possibly-empty domains of quantification. It is impossible, for example, to prove $\forall x.P(x) \vdash \exists x.P(x)$, because the proof would require that there be some m such that $P(m)$.

[3] This induction is unnecessary: in the sequent calculus we never need anything more complicated than a variable. But I wanted to see if I could do something more general, and I have left it in.

Table 3.1: LF variables in the multiple-conclusion sequent calculus

$$\frac{\Gamma, \mathrm{var}\, m \vdash A(m), \Delta}{\Gamma \vdash \forall x.A(x), \Delta}\ \text{(fresh } m\text{)} \vdash\forall \qquad \frac{\Gamma \vdash A(B), \Delta \quad \Gamma \vdash B\ \ \mathrm{inscope}}{\Gamma \vdash \exists x.A(x), \Delta}\ \vdash\exists$$

$$\frac{\Gamma, A(B) \vdash \Delta \quad \Gamma \vdash B\ \ \mathrm{inscope}}{\Gamma, \forall x.A(x) \vdash \Delta}\ \forall\vdash \qquad \frac{\Gamma, \mathrm{var}\, m, A(m) \vdash \Delta}{\Gamma, \exists x.A(x) \vdash \Delta}\ \text{(fresh } m\text{)}\ \exists\vdash$$

```
RULES "inscope" ARE   Γ, var x ⊢ x inscope
AND FROM Γ ⊢ A inscope AND Γ ⊢ B inscope INFER Γ ⊢ A→B inscope
AND FROM Γ ⊢ A inscope AND Γ ⊢ B inscope INFER Γ ⊢ A∧B inscope
AND FROM Γ ⊢ A inscope AND Γ ⊢ B inscope INFER Γ ⊢ A∨B inscope
AND FROM Γ ⊢ A inscope INFER Γ ⊢ ¬A inscope
AND FROM Γ, var x ⊢ A inscope INFER Γ ⊢ ∀x.A inscope
AND FROM Γ, var x ⊢ A inscope INFER Γ ⊢ ∃x.A inscope
END
```

Encoding of the rules is then straightforward in `MSC_LF.j`:

```
RULE "⊢∀"(OBJECT m) WHERE FRESH m
  FROM Γ, var m ⊢ A(m),Δ INFER Γ ⊢ ∀x.A(x),Δ
RULE "∀⊢"(B) FROM Γ, A(B) ⊢ Δ AND Γ ⊢ B inscope INFER Γ,∀x.A(x) ⊢ Δ
RULE "⊢∃"(B) FROM Γ ⊢ A(B),Δ AND Γ ⊢ B inscope INFER Γ ⊢ ∃x.A(x),Δ
RULE "∃⊢"(OBJECT m) WHERE FRESH m
  FROM Γ, var m, A(m) ⊢ Δ INFER Γ, ∃x.A(x) ⊢ Δ
```

I would like inscope judgements to behave like side conditions, displayed when they are a problem and hidden when they are satisfied. But they aren't provisos, because they relate a particular context and a particular formula.

In order to make these judgements behave like side conditions I use Jape's LAYOUT tactical: it allows me to run a tactic and to decide which subtrees of the resulting proof tree should be displayed and what should be written as the justification of the step. (Subtrees which contain open problem sequents are always displayed, so that nothing which might accidentally be important is hidden.) In the case of the ⊢∀ and ∃⊢ rules I would like to display the first antecedent proof (numbered 0) and hide the second (numbered 1); in either case I want to give the name of the rule as the justification of the step. The tactics are

```
TACTIC "∀⊢ with side condition hidden" IS LAYOUT "∀⊢" (0) (WITHSELECTIONS
"∀⊢")
TACTIC "⊢∃ with side condition hidden" IS LAYOUT "⊢∃" (0) (WITHSELECTIONS
"⊢∃")
```

which I put into the menu, overwriting any previous entries with the same label:

```
MENU Rules IS   ENTRY "∀⊢" IS "∀⊢ with side condition hidden"
  ENTRY "⊢∃" IS "⊢∃ with side condition hidden"
END
```

and into the list of double-click actions

```
HYPHIT ∀x.A ⊢ IS "∀⊢ with side condition hidden"
CONCHIT ⊢ ∃x.B IS "⊢∃ with side condition hidden"
```

You get all this machinery by loading `MCS.jt`, to get the multiple-conclusion sequent calculus, and then adding `MCS_LF.j`, to get the extra rules and syntax.

Under this encoding, figure 3.1 showsthe progress of a proof in which the variable rules are obeyed. One antecedent of the final step isn't shown. You can see the full tree, shown in figure 3.2, by double-clicking on the justification of that step. Clearly it is an advantage to hide the side-proof whenever possible; it makes sense to hide it when it is closed, as in this case.

Figure 3.3 shows the progress of an attempt to prove $\forall x.P(x) \vdash \exists y.P(y)$, which isn't a theorem in this logic (though it is in the logic of chapter 2. It doesn't matter what you unify with $\_B$: the side conditions won't go away, and you can't make a theorem.

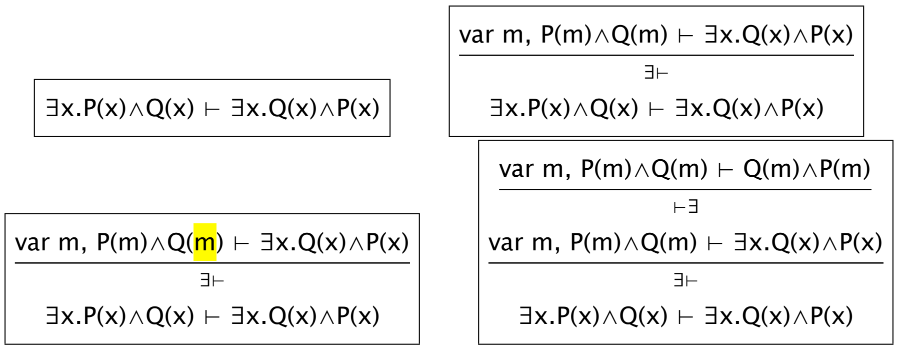

∃x.P(x)∧Q(x) ⊢ ∃x.Q(x)∧P(x)
var m, P(m)∧Q(m) ⊢ ∃x.Q(x)∧P(x)
∃⊢
∃x.P(x)∧Q(x) ⊢ ∃x.Q(x)∧P(x)
var m, P(m)∧Q(m) ⊢ ∃x.Q(x)∧P(x)
∃⊢
∃x.P(x)∧Q(x) ⊢ ∃x.Q(x)∧P(x)
var m, P(m)∧Q(m) ⊢ Q(m)∧P(m)
⊢∃
var m, P(m)∧Q(m) ⊢ ∃x.Q(x)∧P(x)
∃⊢
∃x.P(x)∧Q(x) ⊢ ∃x.Q(x)∧P(x)

Figure 3.1: Using LF variables in the multiple-conclusion sequent calculus

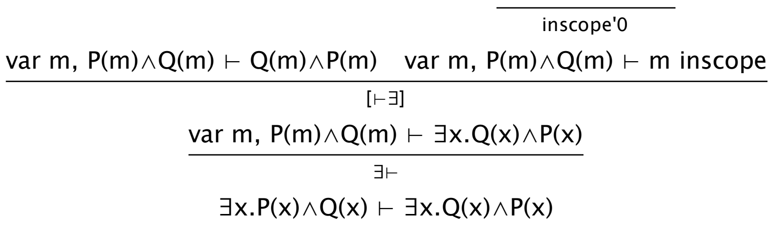

inscope'0
var m, P(m)∧Q(m) ⊢ Q(m)∧P(m)
var m, P(m)∧Q(m) ⊢ m inscope
[⊢∃]
var m, P(m)∧Q(m) ⊢ ∃x.Q(x)∧P(x)
∃⊢
∃x.P(x)∧Q(x) ⊢ ∃x.Q(x)∧P(x)

Figure 3.2: Hidden antecedents exposed

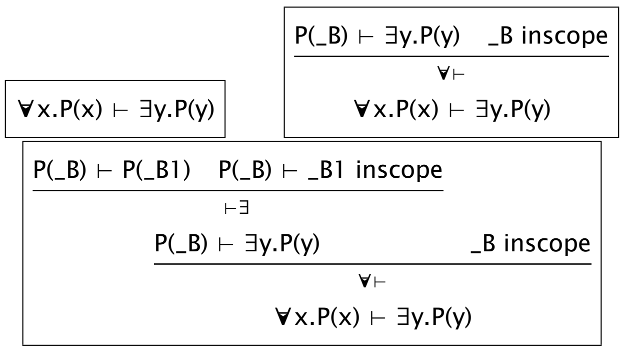


Figure 3.3: Not using LF variables in the multiple-conclusion sequent calculus

Table 3.2: Intuitionistic multiple-conclusion sequent calculus rules

$$\frac{\Gamma, A \vdash}{\Gamma \vdash \neg A, \Delta}\vdash\neg \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B, \Delta}\vdash\rightarrow \qquad \frac{\Gamma \vdash A}{\Gamma, \neg A \vdash \Delta}\neg\vdash \qquad \frac{\Gamma \vdash A \quad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta}\rightarrow\vdash$$

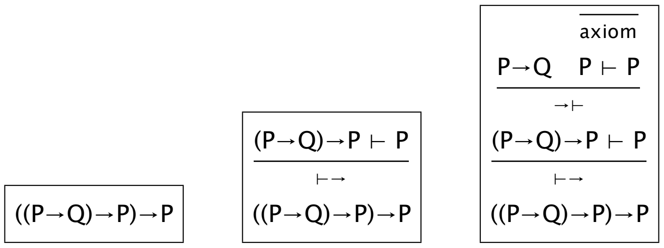


Figure 3.4: No Peirce's law in the intuitionistic multiple-conclusion sequent calculus

3.1.1 Caveat

A deficiency of Jape is that it has only one class of formula, but the contexts which will be built up in this encoding include logical formulae and extra-logical remarks like $\mathrm{var}\, c$. That would permit you, if you were actively incautious, to try to prove nonsense like $\mathrm{var}\, m \wedge \mathrm{var}\, n$. I don't know how to fix the problem.

3.2 The intuitionistic multiple-conclusion sequent calculus

The rules of the intuitionistic multiple-conclusion sequent calculus aren't simply additive, but they use little more than specialised weakening. The calculus is just that of chapter 2, with different definitions of a few rules, shown in table 3.2. These are defined in the file `IMCS.j`, which you can load after `MCS.jt` (and

```
RULE "⊢¬" FROM Γ,A ⊢ INFER Γ ⊢ ¬A,Δ
RULE "¬⊢" FROM Γ ⊢ A INFER Γ,¬A ⊢ Δ
RULE "⊢→" FROM Γ,A ⊢ B INFER Γ ⊢ A→B,Δ
RULE "→⊢" FROM Γ ⊢ A AND Γ,B ⊢ Δ INFER Γ,A→B ⊢ Δ
```

These definitions make it impossible to prove Pierce's law (see figure 3.4), as you would expect.

3.3 A multiple-conclusion sequent calculus with multiplicative rules

The logic is just the normal multiple-conclusion calculus, with all of the branching rules written in multiplicative style; for maximum effect I chose an axiom rule which doesn't ignore unmatched conclusions. See table 3.3 for the altered rules. These rules are defined in MMCS.j, ready to be loaded after MCS.jt:

Table 3.3: Multiplicative multiple-conclusion sequent calculus rules

$$\frac{}{A \vdash A}\text{axiom} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \wedge B, \Delta, \Delta'}\vdash\wedge \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'}\vee\vdash$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \rightarrow B \vdash \Delta, \Delta'}\rightarrow\vdash \qquad \frac{\Gamma \vdash B, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\text{cut}$$

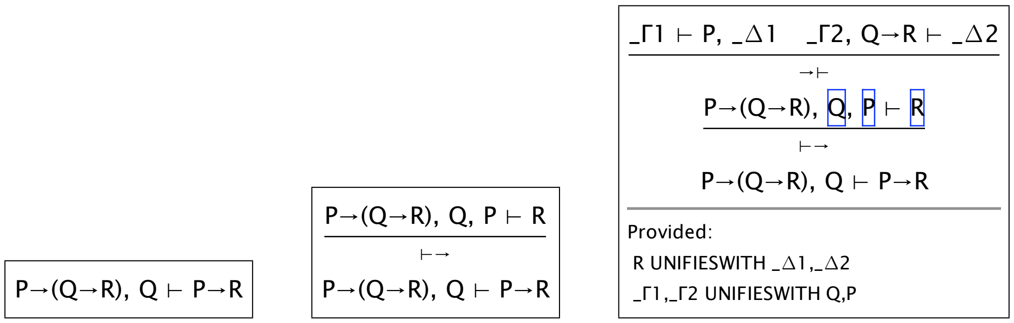


Figure 3.5: A proof in which contexts split

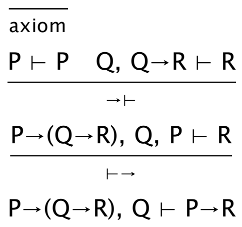


Figure 3.6: Context split resolved by axiom

```
RULE axiom(A) INFER A ⊢ A
RULE "⊢∧" FROM Γ1 ⊢ A,Δ1 AND Γ2 ⊢ B,Δ2 INFER Γ1,Γ2 ⊢ A∧B,Δ1,Δ2
RULE "∨⊢" FROM Γ1,A ⊢ Δ1 AND Γ2,B ⊢ Δ2 INFER Γ1,Γ2,A∨B ⊢ Δ1,Δ2
RULE "→⊢" FROM Γ1 ⊢ A,Δ1 AND Γ2,B ⊢ Δ2 INFER Γ1,Γ2,A→B ⊢ Δ1,Δ2
RULE "⊢≡" FROM Γ1 ⊢ A→B,Δ1 AND Γ2 ⊢ B→A,Δ2 INFER Γ1,Γ2 ⊢ A≡B,Δ1,Δ2
RULE cut(A) FROM Γ1 ⊢ A,Δ1 AND Γ2,A ⊢ Δ2 INFER Γ1,Γ2 ⊢ Δ1,Δ2
```

Since I have redefined cut, I have to redeclare its rôle to Jape:

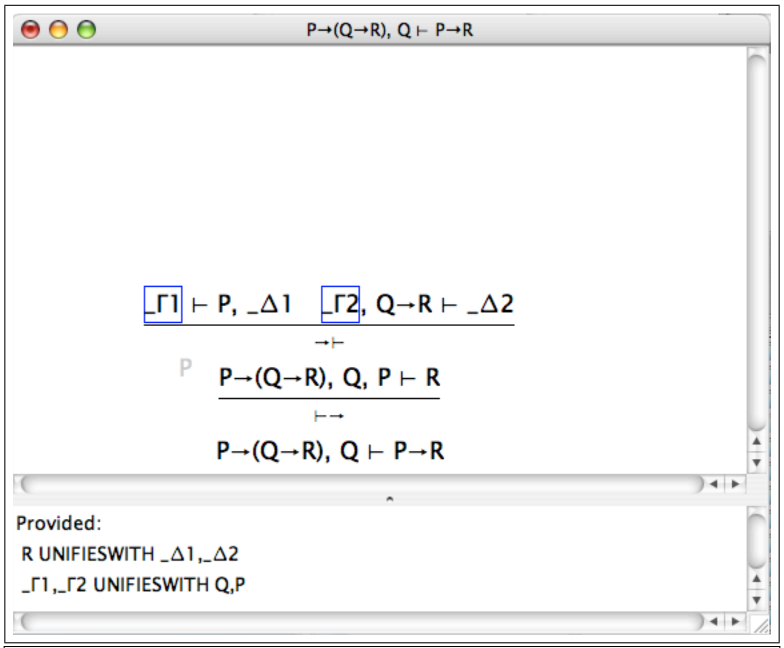
```
STRUCTURERULE CUT cut /* cos it’s different now */
```

When you use a multiplicative rule in this encoding, the left and right contexts split. Jape automatically records this fact in a UNIFIESWITH proviso, as shown in the last step of figure 3.5. In this simple example we have to decide whether to send P and Q into _Γ1 or _Γ2, R into _Δ1 or _Δ2. The axiom rule of this encoding was designed to help: select P in the left antecedent and apply axiom, to produce the result shown in figure 3.6.

Resolving context-splits with drag-and-drop

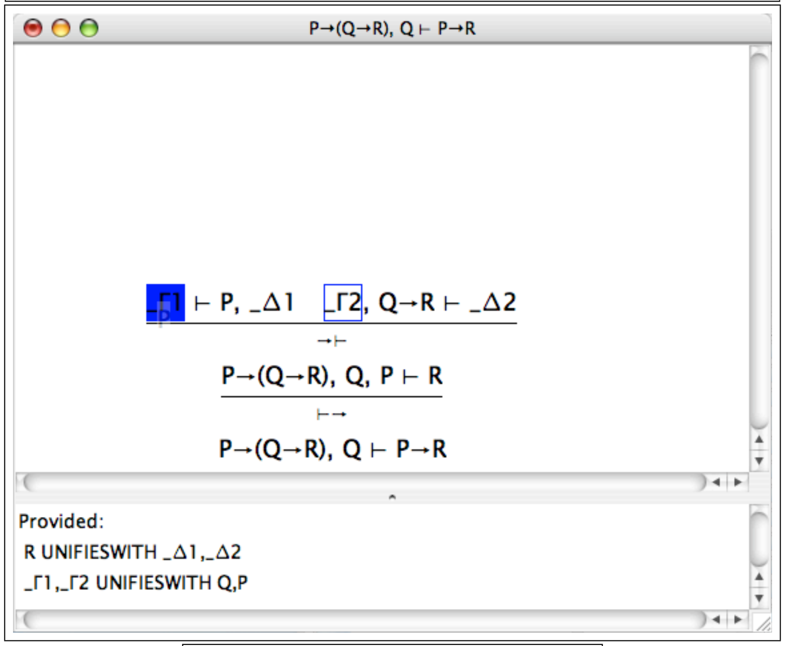
Three of the formulae in the last frame of figure 3.5 are outlined in blue. That indicates that they are candidates for dragging into destination segment-variables. If you press and drag on the outlined instance of P, for example, the outlining changes to show valid drag destinations, as shown in the first frame of figure 3.7. In the second frame we see how highlighting changes again when P is dragged over a valid destination, and the third frame shows the effect of dropping P there. Dragging Q into _Γ2 and R into _Δ2 resolves the split completely.

A similar technique can be used with rules that involve explicit weakening.


P→(Q→R), Q ⊢ P→R

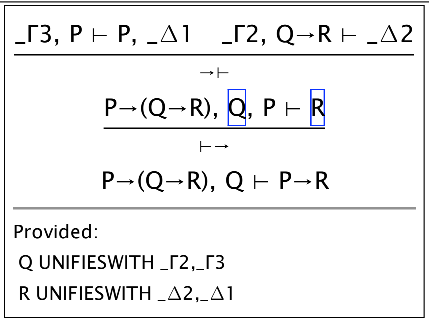
_Γ1 ⊢ P, _Δ1   _Γ2, Q→R ⊢ _Δ2
→⊢
P   P→(Q→R), Q, P ⊢ R
⊢→
P→(Q→R), Q ⊢ P→R

Provided:
R UNIFIESWITH _Δ1,_Δ2
_Γ1,_Γ2 UNIFIESWITH Q,P


P→(Q→R), Q ⊢ P→R

_Γ1 ⊢ P, _Δ1   _Γ2, Q→R ⊢ _Δ2
→⊢
P→(Q→R), Q, P ⊢ R
⊢→
P→(Q→R), Q ⊢ P→R

Provided:
R UNIFIESWITH _Δ1,_Δ2
_Γ1,_Γ2 UNIFIESWITH Q,P


_Γ3, P ⊢ P, _Δ1   _Γ2, Q→R ⊢ _Δ2
→⊢
P→(Q→R), Q, P ⊢ R
⊢→
P→(Q→R), Q ⊢ P→R

Provided:
Q UNIFIESWITH _Γ2,_Γ3
R UNIFIESWITH _Δ2,_Δ1

Figure 3.7: Resolving a context split with drag-and-drop

3.4 Modal logic

I intended to enhance Jape to deal with modal and linear logic. The drag-and-drop gesture, I imagined, would be part of the solution. Not so: not all the parts of a modal context split appear in the antecedents, so they wouldn't be in the proof window. And I never quite managed to wrestle the unifier into the shape it would need to deal with modal unifications. So it didn't happen as promised.

3.5 Single-conclusion sequent calculus (the intuitionistic fragment)

This was the first logic encoding that Jape ran, built by Bernard Sufrin and me in 1992. The encoding is in the file `examples/SCS.jt`.

3.5.1 Inference rules

Apart from the treatment of negation, these are a pretty ordinary selection. As with the multiple-conclusion calculus, we have chosen to use a hypothesis rule which ignores additional hypotheses, and we have made the left-hand side of a sequent a bag of formulae. The rules are very similar to those of chapter 2 with Δs deleted or replaced by a formula variable C. In our encoding the right-hand side of a sequent contains exactly one formula — Jape can't yet handle sequents with at most one formula on the right-hand side — and we give rules for negation — Jape can't yet handle definitional equality.

Negation is often described by defining it to be equivalent to implication of absurdity: that is, $\neg x$ is just a way of writing $x \rightarrow \bot$. Jape can't handle definitional equality, and therefore we give rules which implement that equality. The rules in `SCS_rules.j` are

```
RULE hyp(A) INFER Γ,A ⊢ A
RULE "⊢∧" FROM Γ ⊢ A AND Γ ⊢ B INFER Γ ⊢ A∧B
RULE "∧⊢" FROM Γ, A, B ⊢ C INFER Γ, A∧B ⊢ C
RULE "⊢∨(L)" FROM Γ ⊢ A INFER Γ ⊢ A∨B
RULE "⊢∨(R)" FROM Γ ⊢ B INFER Γ ⊢ A∨B
RULE "∨⊢" FROM Γ, A ⊢ C AND Γ, B ⊢ C INFER Γ, A∨B ⊢ C
RULE "⊢¬" FROM Γ ⊢ A→ ⊥ INFER Γ ⊢ ¬A
RULE "¬⊢" FROM Γ, A→ ⊥ ⊢ B INFER Γ, ¬A ⊢ B
RULE "⊢→" FROM Γ, A ⊢ B INFER Γ ⊢ A→B
RULE "→⊢" FROM Γ ⊢ A AND Γ, B ⊢ C INFER Γ, A→B ⊢ C
RULE "⊢≡" FROM Γ ⊢ A→B AND Γ ⊢ B→A INFER Γ ⊢ A≡B
RULE "≡⊢" FROM Γ, A→B, B→A ⊢ C INFER Γ, A≡B ⊢ C
RULE "⊥⊢" INFER Γ, ⊥ ⊢ A
RULE "⊢∀"(OBJECT m) WHERE FRESH m
  FROM Γ ⊢ P(m) INFER Γ ⊢ ∀x.P(x)
RULE "∀⊢"(B) FROM Γ, P(B) ⊢ C INFER Γ, ∀x.P(x) ⊢ C
RULE "⊢∃"(B) FROM Γ ⊢ P(B) INFER Γ ⊢ ∃x.P(x)
RULE "∃⊢"(OBJECT m) WHERE FRESH m
  FROM Γ, P(m) ⊢ C INFER Γ, ∃x.P(x) ⊢ C
RULE cut(A) FROM Γ ⊢ A AND Γ, A ⊢ C INFER Γ ⊢ C
RULE thin(A) FROM Γ ⊢ B INFER Γ, A ⊢ B
RULE dup(A) FROM Γ, A, A ⊢ B INFER Γ, A ⊢ B
```

3.5.2 LF-style variables

There is an encoding of an LF-style treatment of variables in the file `SCS_LF.j`. It's identical to the treatment of variables in the multiple-conclusion calculus, with the Δ symbol deleted.

3.5.3 Syntax

Formula syntax, and use of names, is exactly as in the multiple-conclusion sequent calculus (they share the `sequent_syntax.j` file).

Jape can't at present be configured to handle sequents with an optional formula on the right-hand side, but can deal with those with exactly one formula there. We therefore state

```
SEQUENT IS BAG ⊢ FORMULA
```

3.5.4 Menus and panels

The Rules menu is almost the same as that in the multiple-conclusion sequent calculus. The Conjectures panel is identical: the two encodings share the file `sequent_problems.j`.

3.5.5 Global variable settings

Just as in the case of the multiple-conclusion sequent calculus, we don't want to allow the application of conjectures as if they were proved theorems and we don't want to allow the application of theorems if their hypotheses don't match.[4] We therefore include

```
INITIALISE applyconjectures false
INITIALISE tryresolution false
```

We do, however, want to allow the user to switch display modes. In place of an INITIALISE directive for the *displaystyle* variable, we include menu entries which control it, by inserting a radio button into the Edit menu — one of the system menus of the Jape graphical interface. A radio button in a graphical interface is a control which has a number of mutually-exclusive settings. In a menu this appears as a number of entries, the selected one of which is ticked.

```
MENU "Edit"
  RADIOBUTTON displaystyle IS
      "Box display" IS box
  AND "Tree display" IS tree
  INITIALLY tree
  END
END
```

The subject of display modes is discussed in chapter 4.

[4] These are pragmatic choices, driven by our expected audience of novices learning about logic. There is, of course, nothing about the logic which forces either choice.

Chapter 4

Display modes: box and tree

Figure 4.1 shows a sample proof attempt in the intuitionistic single-conclusion sequent calculus (see `examples/SCS.jt` and chapter 3). It's a large tree, partly because the hypotheses are written out many times, once in each sequent in which they occur. The same proof attempt in box-and-line form is shown in figure 4.2: it's much narrower and, because it's essentially one-dimensional, easier to navigate. It shows exactly the same steps as the tree and is produced by a very simple algorithm:

- to list a node list its antecedent nodes, then the consequent of the sequent;
- if a node has more hypothesis formulae than its parent, show it as a box with the additional hypotheses on a line labelled 'assumptions';
- link a conclusion to its antecedents by giving their line number (or, if a box, their starting and finishing line numbers);
- precede tips (unclosed nodes) with a line of three dots.

The algorithm saves a great deal of screen space. When a problem sequent has more than a few short hypotheses the tree will often be very wide, impossible to view in one piece. The box-and-line display is still readable, and it's easier to find your way around because you only have to go up and down.

The simple algorithm is only just the beginning ...

4.1 Hiding IDENTITY steps

There is still unnecessary business in figure 4.2. Line 5, for example, is proved by `hyp`. That rule, in the SCS encoding, is

```
RULE hyp(A) INFER A ⊢ A
```

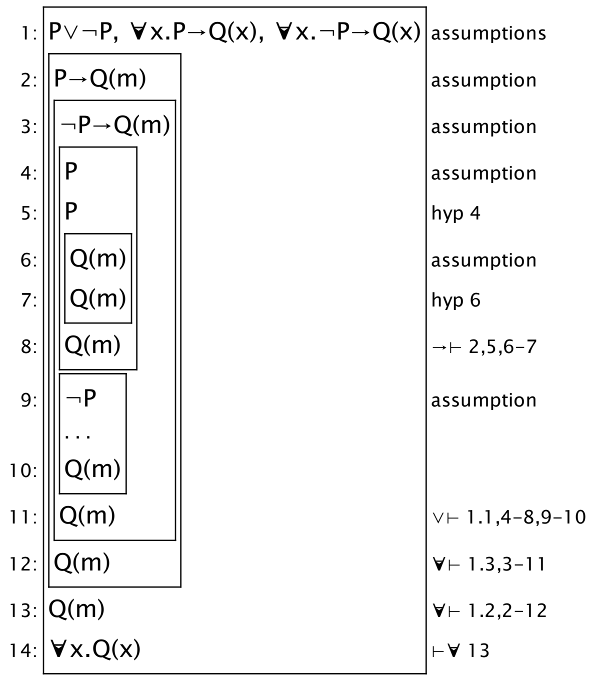
It makes sense in the tree, but in the box-and-line version it's little more than an indirection. Line 8, for example, appeals to line 5 which appeals immediately and identically to line 4. When the rôle of `hyp` is declared to Jape

```
STRUCTURERULE IDENTITY hyp
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg P\to Q(m),\ P\vdash P}\,\text{hyp} \quad \dfrac{}{\neg P\to Q(m),\ P,\ Q(m)\vdash Q(m)}\,\text{hyp}}{P\to Q(m),\ \neg P\to Q(m),\ P\vdash Q(m)}\,{\to}{\vdash} \qquad P\to Q(m),\ \neg P\to Q(m),\ \neg P\vdash Q(m)}{P\lor\neg P,\ P\to Q(m),\ \neg P\to Q(m)\vdash Q(m)}\,{\lor}{\vdash}}{P\lor\neg P,\ \forall x.\neg P\to Q(x),\ P\to Q(m)\vdash Q(m)}\,\forall{\vdash}}{P\lor\neg P,\ \forall x.P\to Q(x),\ \forall x.\neg P\to Q(x)\vdash Q(m)}\,\forall{\vdash}}{P\lor\neg P,\ \forall x.P\to Q(x),\ \forall x.\neg P\to Q(x)\vdash \forall x.Q(x)}\,{\vdash}\forall$$

Provided:
x NOTIN P

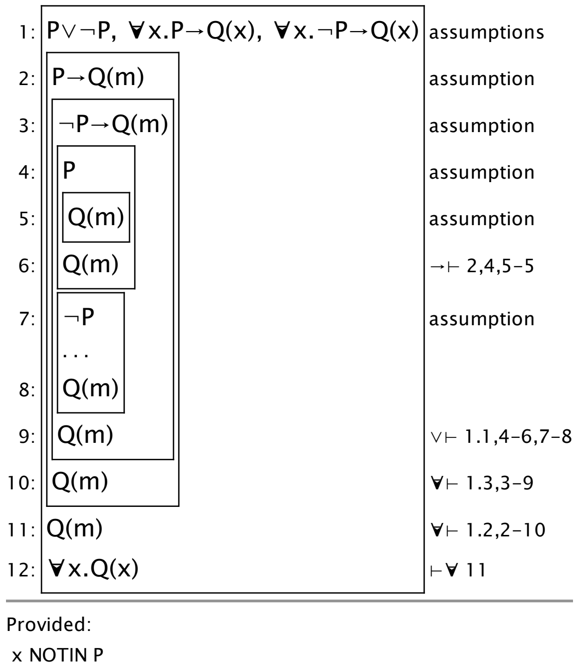
Figure 4.1: A sample proof attempt, shown as a tree



1:	$P\lor\neg P,\ \forall x.P\to Q(x),\ \forall x.\neg P\to Q(x)$	assumptions
2:	$P\to Q(m)$	assumption
3:	$\neg P\to Q(m)$	assumption
4:	P	assumption
5:	P	hyp 4
6:	$Q(m)$	assumption
7:	$Q(m)$	hyp 6
8:	$Q(m)$	$\to\vdash$ 2,5,6-7
9:	$\neg P$	assumption
	...	
10:	$Q(m)$	
11:	$Q(m)$	$\lor\vdash$ 1.1,4-8,9-10
12:	$Q(m)$	$\forall\vdash$ 1.3,3-11
13:	$Q(m)$	$\forall\vdash$ 1.2,2-12
14:	$\forall x.Q(x)$	$\vdash\forall$ 13

Provided:
x NOTIN P

Figure 4.2: A sample proof attempt, shown as box-and-line



```
 1: P∨¬P, ∀x.P→Q(x), ∀x.¬P→Q(x)   assumptions
 2: P→Q(m)                         assumption
 3: ¬P→Q(m)                        assumption
 4: P                              assumption
 5: Q(m)                           assumption
 6: Q(m)                           →⊢ 2,4,5-5
 7: ¬P                             assumption
    ...
 8: Q(m)
 9: Q(m)                           ∨⊢ 1.1,4-6,7-8
10: Q(m)                           ∀⊢ 1.3,3-9
11: Q(m)                           ∀⊢ 1.2,2-10
12: ∀x.Q(x)                        ⊢∀ 11
Provided:
 x NOTIN P
```

Figure 4.3: A sample proof attempt, shown as box-and-line, with hidden `hyp` steps

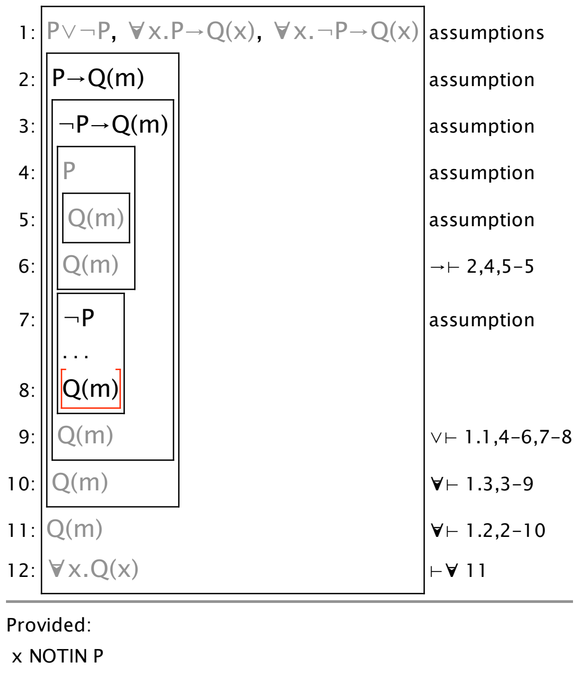
and the `hidehyp` variable is set to `true` (its default value), then the picture changes to figure 4.3. The old line 8 is now line 6 and refers directly to the assumption on line 3. The proof is more compact, and easier to read. There's still tedious repetition of the conclusion $Q(m)$ in lines 5-6 and 8-11, but that is inevitable in this calculus, where left-hand-side rules transform hypotheses and copy the conclusion. Chapter 5 shows how a Natural Deduction calculus, without left-hand rules, can avoid such repetition.

An IDENTITY step can still appear in a box-and-line proof if it is the last line of a box which has more than two lines, or the assumption line isn't a single formula which is the same as the last line.

4.2 The tree is still there

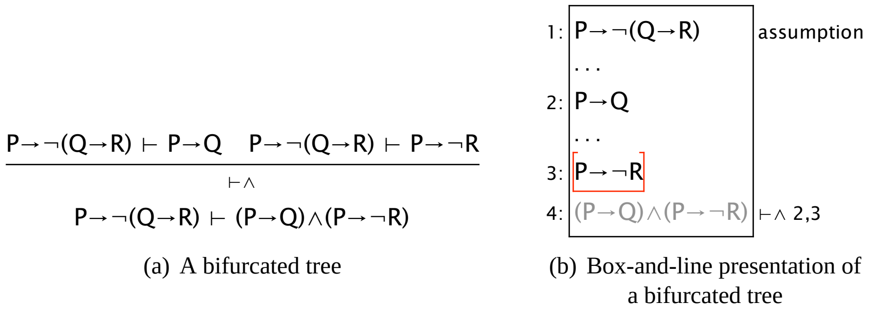
In figure 4.3, according to the rules of box-and-line proof presentation, line 8 can appeal to the assumptions on line 1, 2, 3 and 7 (it can't use line 9-12 because they are below it; it can't use lines 4-6 because they are in a box). But in the tree only $P \rightarrow Q(m)$, $\neg P \rightarrow Q(m)$ and $\neg P$ — that is, lines 2, 3 and 7 — are available. If you select the conclusion on line 8 Jape greys out all the formulae that can't be used: see figure 4.4.

The effect in figure 4.4 is produced by left-hand rules that use up hypotheses. When a rule bifurcates the proof there are other problems. In figure 4.5(b), for example, the formula on line 2 can't be selected as a hypothesis to help prove the conclusion on line 3, which breaks the normal rules of box-and-line proofs: it isn't greyed out, though, because it is possible to move the conclusion selection from line 3 to line 2. The solution to this problem is presented in chapter 10.



```
 1: P∨¬P, ∀x.P→Q(x), ∀x.¬P→Q(x)  assumptions
 2: P→Q(m)                        assumption
 3: ¬P→Q(m)                       assumption
 4: P                             assumption
 5: Q(m)                          assumption
 6: Q(m)                          →⊢ 2,4,5-5
 7: ¬P                            assumption
    ...
 8: Q(m)
 9: Q(m)                          ∨⊢ 1.1,4-6,7-8
10: Q(m)                          ∀⊢ 1.3,3-9
11: Q(m)                          ∀⊢ 1.2,2-10
12: ∀x.Q(x)                       ⊢∀ 11
Provided:
 x NOTIN P
```

Figure 4.4: In the box-and-line presentation not all accessible formulae are usable



$$\frac{P\to\neg(Q\to R)\vdash P\to Q \qquad P\to\neg(Q\to R)\vdash P\to\neg R}{P\to\neg(Q\to R)\vdash (P\to Q)\wedge(P\to\neg R)}\vdash\wedge$$

(a) A bifurcated tree

```
1: P→¬(Q→R)             assumption
   ...
2: P→Q
   ...
3: P→¬R
4: (P→Q)∧(P→¬R)         ⊢∧ 2,3
```

(b) Box-and-line presentation of a bifurcated tree

Figure 4.5: Bifurcation also causes presentation problems

4.3 That's not all

Box-and-line mode, produced by setting variable `displaystyle` to `true`, can be used as an abbreviated mode of presentation of any tree proof, if you are prepared to live with the deficiencies described above. In Natural Deduction logics, without left-hand rules, it's possible to do much more, using hidden CUT steps to produce an approximation to forward reasoning from the hypotheses (chapter 5) or, at some considerable effort, an accurate depiction (chapter 10).

Chapter 5

Encoding natural deduction

The sequent calculus encodings of chapters 2 and 3 are each straightforward encodings of a logic, with a few directives to arrange the elements of the user interface. Natural deduction challenges us to allow forward reasoning steps. This chapter describes a first attempt: it can imitate only some kinds of forward step, and some features of the background tree are still traceable in the box display. The techniques described here are used in some later chapters, and in chapter 10 a more polished attempt is described.

This encoding was used in a first-year course at QMW for several years, with increasing user satisfaction as the encoding more nearly approached the treatment used by the course lecturer. That lecturer chose the rules and, in particular, he chose to use a particular classical treatment of negation, not at all the one which I would have chosen myself, nor even the particular classical encoding which I would have preferred (see chapter 10 for my own version).

The encoding is in `examples/ItL_QMWpre2000`.

5.1 Inference rules

The rules of natural deduction are not normally stated in terms of sequents. The rules were presented to me in box-and-line form as in table 5.1 (there was also a reiteration rule, which I don't list).[1] The marginal c in the $\exists$-E and $\forall$-I rules indicates a proviso that c should not appear free outside the 'scope box' which it labels; there's a corresponding condition on $\exists$-I and $\forall$-E that c should be 'well scoped' — that is, lines i to j have to be inside a scope box for c.

These rules have well-known tree equivalents shown in table 5.2, where reiteration is replaced by a hypothesis rule, the provisos are made explicit and the $\exists$-I and $\forall$-E rules have an explicit side-condition. The provisos on $\exists$-E and $\forall$-I implement the provisos on the box rules reasonably accurately. They aren't quite the same, because there could be disjoint parts of the tree where the same name is used; in practice, because my encoding of these rules makes them introduce names new to the proof, the distinction is unnoticeable. To implement the condition on $\exists$-I and $\forall$-E I have used pseudo-predicates var and inscope; c inscope is a side condition that var c must occur in the hypotheses.

It's easy to recast these rules in sequent notation as in table 5.3. Plainly it is a difficulty, in a backwards-reasoning tool, that these are all right-hand rules. Yet if we are to be satisfy to our client's requirements, these are the rules that we must encode. In order to understand how to do that, it is necessary to understand

[1] If anybody knows how to make Latex lay out this table with adequate space before and after each row, and without picking some elements and jamming them up against the top of the box, please let me know.

Table 5.1: Natural deduction rules in box form

$\begin{array}{lll} i: & A & \\ & \dots & \\ j: & A \rightarrow B & \\ & \dots & \\ k: & B & \rightarrow -E\ i,j \end{array}$	$\begin{array}{lll} i: & A \wedge B & \\ & \dots & \\ j: & A & \wedge - E(L)\ i \end{array}$	$\begin{array}{lll} i: & A \wedge B & \\ & \dots & \\ j: & B & \wedge - E(R)\ i \end{array}$	$\begin{array}{lll} i: & \neg\neg A & \\ & \dots & \\ j: & A & \neg - E\ i \end{array}$
$\begin{array}{lll} i: & A \vee B & \\ & \dots & \\ \begin{array}{l} j: \\ \\ k: \end{array} & \boxed{\begin{array}{l} A \\ \dots \\ C \end{array}} & \text{assumption} \\ & \dots & \\ \begin{array}{l} l: \\ \\ m: \end{array} & \boxed{\begin{array}{l} B \\ \dots \\ C \end{array}} & \text{assumption} \\ & \dots & \\ n: & C & \vee - E\ i,j..k,l..m \end{array}$			
$\begin{array}{lll} \begin{array}{l} \\ i: \\ \\ j: \\ \\ \end{array} & \boxed{\begin{array}{l} \dots \\ \forall x.P(x) \\ \dots \\ P(c) \\ \dots \end{array}}^{c} & \forall - E\ i \end{array}$		$\begin{array}{lll} i: & \exists x.P(x) & \\ & \dots & \\ \begin{array}{l} j: \\ \\ k: \end{array} & \boxed{\begin{array}{l} P(c) \\ \dots \\ A \end{array}}^{c} & \text{assumption} \\ & \dots & \\ l: & A & \exists - E\ i,j..k \end{array}$	
$\begin{array}{lll} \begin{array}{l} i: \\ \\ j: \end{array} & \boxed{\begin{array}{l} A \\ \dots \\ B \end{array}} & \text{assumption} \\ & \dots & \\ k: & A \rightarrow B & \rightarrow -I\ i..j \end{array}$		$\begin{array}{lll} \begin{array}{l} i: \\ \\ j: \end{array} & \boxed{\begin{array}{l} A \\ \dots \\ B \wedge \neg B \end{array}} & \text{assumption} \\ & \dots & \\ k: & \neg A & \neg\text{-I}\ i..j \end{array}$	
$\begin{array}{lll} i: & A & \\ & \dots & \\ j: & B & \\ & \dots & \\ k: & A \wedge B & \wedge - I\ i,j \end{array}$	$\begin{array}{lll} i: & A & \\ & \dots & \\ j: & A \vee B & \vee - I(L)\ i \end{array}$	$\begin{array}{lll} i: & B & \\ & \dots & \\ j: & A \vee B & \vee - I(R)\ i \end{array}$	
$\begin{array}{lll} i: & \boxed{\begin{array}{l} \dots \\ P(c) \end{array}}^{c} & \\ & \dots & \\ j: & \forall x.P(x) & \forall - I\ i \end{array}$		$\begin{array}{lll} \begin{array}{l} \\ i: \\ \\ j: \\ \\ \end{array} & \boxed{\begin{array}{l} \dots \\ P(c) \\ \dots \\ \exists x.P(x)) \\ \dots \end{array}}^{c} & \exists - I\ i \end{array}$	

Table 5.2: Natural deduction rules in tree form

$\dfrac{}{A}\ hyp$			
$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & A \rightarrow B\end{array}}{B} \rightarrow -E$	$\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{A} \wedge - E(L)$	$\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{B} \wedge - E(R)$	$\dfrac{\begin{array}{c}\vdots \\ \neg\neg A\end{array}}{A} \neg - E$
$\dfrac{\begin{array}{ccc} & [A] & [B] \\ \vdots & \vdots & \vdots \\ A \vee B & C & C\end{array}}{C} \vee - E$		$\dfrac{\begin{array}{cc}\vdots & \\ \forall x.P(x) & c\ \text{inscope}\end{array}}{P(c)} \forall - E$	
$\dfrac{\begin{array}{cc} & [\text{var}\, c, P(c)] \\ \vdots & \vdots \\ \exists x.P(x) & A\end{array}}{A}$ (FRESH c, c NOTIN $\exists x.P(x)$) $\exists - E$			
$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \rightarrow B} \rightarrow -I$	$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B} \wedge - I$	$\dfrac{\begin{array}{c}\vdots \\ A\end{array}}{A \vee B} \vee - I(L)$	$\dfrac{\begin{array}{c}\vdots \\ B\end{array}}{A \vee B} \vee - I(R)$
$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B \wedge \neg B\end{array}}{\neg A} \neg - I$	$\dfrac{\begin{array}{c}[\text{var}\, c] \\ \vdots \\ P(c)\end{array}}{\forall x.P(x)}$ (FRESH c) $\forall - I$	$\dfrac{\begin{array}{cc}\vdots & \\ P(c) & c\ \text{inscope}\end{array}}{\exists x.P(x)} \exists - I$	

Table 5.3: Natural deduction rules in sequent form

$\dfrac{}{\Gamma, A \vdash A}\ hyp$			
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \rightarrow B}{\Gamma \vdash B} \rightarrow -E$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge - E(L)$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge - E(R)$	$\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \neg - E$
$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee - E$		$\dfrac{\Gamma \vdash \forall x.A(x) \quad \Gamma \vdash c\ \text{inscope}}{\Gamma \vdash A(c)} \forall - E$	
$\dfrac{\Gamma \vdash \exists x.A(x) \quad \Gamma, \text{var}\, c, A(c) \vdash B}{\Gamma \vdash B}$ (FRESH c, c NOTIN $\exists x.A$) $\exists - E$			
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow -I$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge - I$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee - I(L)$	$\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee - I(R)$
$\dfrac{\Gamma, A \vdash B \wedge \neg B}{\Gamma \vdash \neg A} \neg - I$	$\dfrac{\Gamma, \text{var}\, c \vdash A(c)}{\Gamma \vdash \forall x.A(x)}$ (FRESH c) $\forall$-I	$\dfrac{\Gamma \vdash A(c) \quad c\ \text{inscope}}{\Gamma \vdash \exists x.A(x)}$ $\exists$-I	

how they are intended to be used.

5.2 Syntax (`ItL_syntax.j`)

The description of the syntax differs only slightly from that used in the sequent calculus encodings of chapters 2 and 3. Quantification affects the *smallest* following formula: that is, in this encoding $\forall x.A \to B$ should parse as $(\forall x.A) \to B$.[2] The priority of quantification is therefore just less than that of negation.

```
CLASS VARIABLE x y z c d
CLASS FORMULA A B C P Q R S
CLASS BAG FORMULA Γ

PREFIX 10 var
POSTFIX 10 inscope

INFIX 100R →
INFIX 120L ∨
INFIX 140L ∧

LEFTFIX 180 ∀ .
LEFTFIX 180 ∃ .

PREFIX 200 ¬
JUXTFIX 300
SUBSTFIX 400

BIND x SCOPE P IN ∀x .  P
BIND x SCOPE P IN ∃x .  P

SEQUENT IS BAG ⊢ FORMULA
```

We set two variables (really they are parameters, because they can only be altered when the rule and theorem store is empty):

```
INITIALISE autoAdditiveLeft true
INITIALISE interpretpredicatesbtrue
```

The first of these allows us to define rules without mentioning a left context, automatically inserting a context variable Γ into every sequent in a rule definition (that is, allowing rule definition in the style of natural deduction and earlier versions of Jape). The second directs Jape to interpret every juxtaposition — everything that looks like a predicate application — as a predicate application, to translate where necessary into substitution notation and to include additional rule parameters and invisible provisos to support the translation.

[2] Our customer also wanted no punctuation between bound variable and body: I've never found a way to make Jape's parser generator allow that.

5.3 The rules in Japeish (`ItL_rules.j`)

The introduction and elimination rules are exactly those of table 5.3, all stated without repetition of the context Γ. `hyp` is AUTOMATCHed and an IDENTITY rule, just as in the sequent calculus.

```
RULE "→-E"(A) IS FROM A AND A→B INFER B
RULE "∧-E(L)"(B) IS FROM A ∧ B INFER A
RULE "∧-E(R)"(A) IS FROM A ∧ B INFER B
RULE "∨-E"(A,B) IS FROM A ∨ B AND A ⊢ C AND B ⊢ C INFER C
RULE "¬-E" IS FROM ¬¬A INFER A
RULE "∀-E"(c) IS FROM ∀x.  A(x) AND c inscope INFER A(c)
RULE "∃-E"(OBJECT c) WHERE FRESH c AND c NOTIN ∃x.A(x)
  IS FROM ∃x.A(x) AND var c, A(c) ⊢ C INFER C

RULE "→-I" IS FROM A ⊢ B INFER A→B
RULE "∧-I" IS FROM A AND B INFER A ∧ B
RULE "∨-I(L)"(B) IS FROM A INFER A ∨ B
RULE "∨-I(R)"(A) IS FROM B INFER A ∨ B
RULE "¬-I"(B) IS FROM A ⊢ B ∧ ¬B INFER ¬A
RULE "∀-I"(OBJECT c) WHERE FRESH c
  IS FROM var c ⊢ A(c) INFER ∀x .A(x)
RULE "∃-I"(c) IS FROM A(c) AND c inscope INFER ∃x.A(x)


RULE hyp(A) IS INFER A ⊢ A
AUTOMATCH hyp
STRUCTURERULE IDENTITY hyp
```

To imitate forward reasoning we need a CUT rule; to use theorems easily we need a WEAKEN rule.

```
RULE cut(B) IS FROM B AND B ⊢ C INFER C
RULE thin(A) IS FROM C INFER A ⊢ C

STRUCTURERULE CUT cut
STRUCTURERULE WEAKEN thin
```

To deal with variables we need a rule that picks up var declarations, and it's AUTOMATCHed too.

```
RULE "inscope" IS INFER var x ⊢ x inscope
AUTOMATCH "inscope"
```

(It is offensive to use the context in this way, as if var c was a logical assertion. Jape doesn't have structured contexts, more's the pity.)

5.4 Variable settings (`ItL.jt`)

We don't want novices to apply a conjecture until it is proved, and we don't want the engine to use a resolution step unless a tactic commands it:

```
INITIALISE applyconjectures false
INITIALISE tryresolution false
```

The display style should be box-and-line, and we want to control the naming of assumptions: the outermost assumption should be "premise", plural "premises" (the `innerassumption` settings look like a hangover from an earlier version of Jape, as those are the defaults anyway):

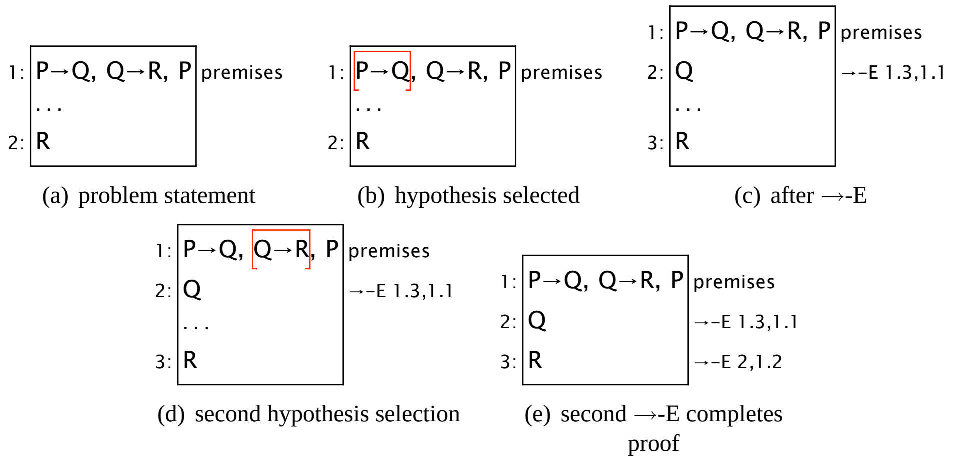


Figure 5.1: A proof using forward steps

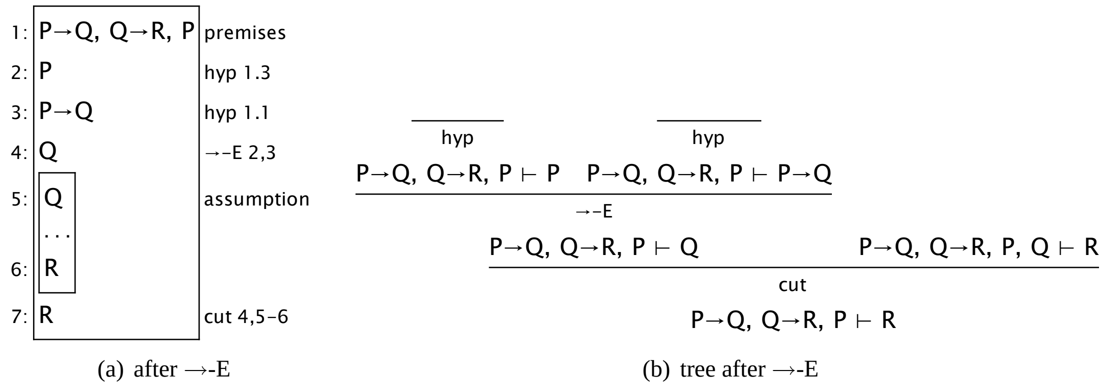


Figure 5.2: A proof using forward steps, showing `cut` and `hyp`

```
INITIALISE displaystyle box

INITIALISE outerassumptionword premise
INITIALISE outerassumptionplural premises
INITIALISE innerassumptionword assumption
INITIALISE innerassumptionplural assumptions
```

5.5 Forward reasoning with `cut`

The sort of steps that a natural-deduction reasoner might want to make is best illustrated by example. Consider the problem of proving $P \to Q, Q \to R, P \vdash R$, the second problem in the Conjectures panel defined in `ItL_problems.j`. Figure 5.1(a) shows the initial display; 5.1(b) the first selection; 5.1(c) the effect of →-E; 5.1(d) the second selection; 5.1(e) the effect of a second application of →-E.

The proof is complete, and has apparently used forward reasoning. Yet in fact it was all done with backward reasoning, and the logic has only right-hand rules — nothing that can work on hypotheses.

The magic is all done with the `hyp` and `cut` rules. If you set the `hidehyp` and `hidecut` variables to `false`, the first step goes as in figure 5.2(a): a couple of `hyp` steps move P and $P \to Q$ from left to right, making them available for the →-E step; then →-E extracts Q; then a `cut` step moves Q from right to left,

$$\boxed{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash R}}$$

(a) the problem

$$\boxed{\dfrac{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash \_B} \qquad \mathsf{P{\to}Q,\ Q{\to}R,\ P,\ \_B \vdash R}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash R}}\,\mathsf{cut}}$$

(b) after cut, with unknown $\_B$

$$\boxed{\dfrac{\dfrac{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash \_A} \qquad \mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash \_A{\to}\_B}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash \_B}}\,\mathsf{{\to}\text{-}E} \qquad \mathsf{P{\to}Q,\ Q{\to}R,\ P,\ \_B \vdash R}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash R}}\,\mathsf{cut}}$$

(c) after →-E, with unknowns $\_A$ and $\_B$

$$\boxed{\dfrac{\dfrac{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash P} \qquad \dfrac{}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash P{\to}Q}}\,\mathsf{hyp}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash Q}}\,\mathsf{{\to}\text{-}E} \qquad \mathsf{P{\to}Q,\ Q{\to}R,\ P,\ Q \vdash R}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash R}}\,\mathsf{cut}}$$

(d) after first hyp (from user selection)

$$\boxed{\dfrac{\dfrac{\dfrac{}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash P}}\,\mathsf{hyp} \qquad \dfrac{}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash P{\to}Q}}\,\mathsf{hyp}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash Q}}\,\mathsf{{\to}\text{-}E} \qquad \mathsf{P{\to}Q,\ Q{\to}R,\ P,\ Q \vdash R}}{\mathsf{P{\to}Q,\ Q{\to}R,\ P \vdash R}}\,\mathsf{cut}}$$

(e) after second hyp (by AUTOMATCH)

Figure 5.3: Stages of a sample ‘forward step’

making it available as a hypothesis to prove R. With `hyp` steps hidden lines 1 and 2 disappear. We are left with the pattern on lines 4, 5, 6 and 7: a `cut` whose left arm is proved by a rule step, and whose conclusion is the same as the right arm of the cut. Such a pattern can be expressed as a single line: add a hypothesis labelled for the rule step, as in figure 5.1(c).

In fact the proof has to be built backwards, as you can see from the tree in figure 5.2(b): `cut`, →-E, `hyp`, `hyp` . Jape uses unknowns in every step until the `hyp`s make it all concrete. The steps are shown in figure 5.3: note that the first `hyp` in figure 5.3(d) eliminates both the unknowns, so that the second `hyp` can proceed unprompted, by matching rather than unification.

The mechanism is invoked by the →-E entry in the Rules menu:

```
MENU Rules IS ENTRY "→-I"
ENTRY "∧-I"
ENTRY "∨-I(L)" IS ForwardOrBackward ForwardCut 0 "∨-I(L)"
ENTRY "∨-I(R)" IS ForwardOrBackward ForwardCut 0 "∨-I(R)"
ENTRY "¬-I"
ENTRY "∀-I"
ENTRY "∃-I" IS "∃-I with side condition hidden"
SEPARATOR
ENTRY "→-E" IS ForwardOrBackward ForwardCut 1 "→-E"
ENTRY "∧-E(L)" IS ForwardOrBackward ForwardCut 0 "∧-E(L)"
ENTRY "∧-E(R)" IS ForwardOrBackward ForwardCut 0 "∧-E(R)"
ENTRY "∨-E" IS ForwardOrBackward ForwardUncut 0 "∨-E"
ENTRY "¬-E" IS ForwardOrBackward ForwardCut 0 "¬-E"
ENTRY "∀-E" IS ForwardOrBackward ForwardCut 0 "∀-E with side condition
hidden"
ENTRY "∃-E" IS ForwardOrBackward ForwardUncut 0 "∃-E"
SEPARATOR
ENTRY hyp END
```

`ForwardOrBackward` detects whether to use a forward or a backward step:[3]

```
TACTIC ForwardOrBackward (Forward, n, rule) IS
  WHEN
    (LETHYP _P
      (ALT
        (Forward n rule)
        (WHEN
          (LETARGSEL _Q
            (Fail (rule is not applicable to assumption ′ _P ′ with argument
′ _Q ′)))
          (Fail (rule is not applicable to assumption ′ _P ′)))))
    (ALT
      (WITHSELECTIONS rule)
      (WHEN
        (LETARGSEL _P
          (Fail (rule is not applicable with argument ′ _P ′)))
        (Fail (rule is not applicable))))
```

Stripped to its bones, without the error-handling code, this tactic is

[3] For no good reason that I can recall, tactics and rules are applied “Curried” style as $t\ a_1\ a_2\ \ldots$ but defined with tuple arguments as $t\ (a_1,\ a_2,\ \ldots)$.

```
TACTIC ForwardOrBackward (Forward, n, rule) IS
  WHEN
    (LETHYP _P (Forward n rule))
    (WITHSELECTIONS rule)
```

WHEN takes the first of its arguments whose guard succeeds. LETHYP $\_P$ succeeds if the user has selected a hypothesis which matches $\_P$ — i.e. any hypothesis at all. `Forward` n *rule* does the work in that case. If LETHYP doesn't match, it's a backward step and WITHSELECTIONS *rule* does the work instead.

In the full tactic, ALT applies its arguments in turn until one succeeds, and fails if none succeed. If `Forward` fails in the forward-step arm, you get a message generated by `Fail` which varies according to whether or not you have subformula-selected $\_Q$ (the messages are somewhat archaicly constructed: later chapters will show how to make them more nicely), and similarly in the backward-step arm (with slightly different messages).

The `Forward` argument is in fact `ForwardCut`, n is 1 and *rule* is `"→-E"`. `ForwardCut` applies `cut` and then `ForwardUncut`:

```
TACTIC ForwardCut (n,rule)
  SEQ cut (ForwardUncut n rule)
```

`ForwardUncut` applies the rule and then applies `hyp` to the nth antecedent:

```
TACTIC ForwardUncut (n,rule)
  (LETGOALPATH G
    (WITHARGSEL rule)
    (GOALPATH (SUBGOAL G n))
    (WITHHYPSEL hyp)
    (GOALPATH G)
    NEXTGOAL)
```

LETGOALPATH binds G to the current position in the tree and then applies its arguments as a SEQ tactic. WITHARGSEL *rule* applies *rule* (`"→-E"` in our case), with the user's subformula selections, if any, as the first argument (see sections 1.8 and 1.2). Then it goes to the nth antecedent of the tree that *rule* generates and applies `hyp`, using the user's selected hypothesis. A rule uses the user's selections to disambiguate alternative rule matches: in this case the hypothesis selection picks out one left-hand formula for `hyp` to work on. Finally the tactic goes back to the base of the subtree it's built and moves (NEXTGOAL) to its leftmost tip, or if the tree is closed to the next available tip in the tree.

If the user doesn't select a hypothesis, WITHSELECTIONS in `ForwardOrBackward` applies *rule*, taking account of the user's conclusion and subformula-selections (if any).

Finally, as we've seen, AUTOMATCH tidies up.

5.6 Other menu entries

Two of the entries in the Rules menu give a tactic as argument to `ForwardorBackward`, not a rule. The tactics in question are:

```
TACTIC "∀-E with side condition hidden" IS
  LAYOUT "∀-E" (0) (WITHARGSEL "∀-E")
TACTIC "∃-I with side condition hidden" IS
  LAYOUT "∃-I" (0) (WITHARGSEL "∃-I")
```

The LAYOUT tactical takes a tuple describing the arguments which ought to be visible: here it's (0), which means that antecedent 1, the c inscope antecedent, is hidden. The rest of its arguments are a sequence of tactics: I have to reapply WITHARGSEL to make the rule sensitive to the user's argument selection.

As in chapter 3, there is a rule for the inscope judgement, which is tried at the end of every proof step:

```
RULE "inscope" IS INFER var x ⊢ x inscope
AUTOMATCH "inscope"
```

5.7 Automatic rule selection (`ItL_hits.j`)

There is a file ItL_hits.j, which implements double-clicking in this encoding: I didn't reference this in ItL.jt, because I didn't want to provide it by default to novices.

5.8 The Conjectures panel (`ItL_problems.j`)

Since we can apply rules either forward or backward, it would be irksome if we could only apply theorems backward. I define a tactic which can do the job. If a hypothesis has been selected it cuts and applies the theorem, requiring that the selected hypothesis be one of the principal formulae which match the theorem sequent. If no hypothesis is selected it tries in order to apply the theorem to the present problem sequent, to apply it 'by resolution' (matching only the right-hand side of the theorem sequent and the problem sequent and generating antecedents for each left-hand side theorem formula: see section 1.4.2), and finally tries to apply it forwards, one way or the other. All of the steps are made 'WITHSELECTIONS' — that is, using any hypothesis, conclusion or subformula-selection which the user may have made:

```
TACTIC TheoremForwardOrBackward(thm) IS
  WHEN
    (LETHYP _P cut (WITHSELECTIONS thm))
    (ALT
      (WITHSELECTIONS thm)
      (RESOLVE (WITHSELECTIONS thm))
      (SEQ cut (ALT (WITHSELECTIONS thm) (RESOLVE (WITHSELECTIONS thm))))
    )
```

The overall effect is to allow a prover to introduce a theorem into the proof whenever it is helpful to do so. The Conjectures panel activates this tactic from its Apply button:

```
CONJECTUREPANEL Conjectures
  THEOREM INFER P, P → Q ⊢ Q
  THEOREM INFER P → Q, Q → R, P ⊢ R
  THEOREM INFER P → (Q → R), P → Q, P ⊢ R

  ...

  THEOREM "(∀x.P(x)) → (∀x.Q(x)) ⊢ ∀x.(P(x) → Q(x)) NOT" IS ∀x.P(x) →
∀x.Q(x) ⊢ ∀x.(P(x) → Q(x))
  THEOREM "(∃x.P(x)) ∧ (∃x.Q(x)) ⊢ ∃x.(P(x) ∧ Q(x)) NOT" IS ∃x.P(x) ∧ ∃x.Q(x)
⊢ ∃x.(P(x) ∧ Q(x))

  BUTTON Apply IS apply TheoremForwardOrBackward COMMAND
END
```

(Most conjectures are named by their sequent, but the last few are specially named because a proof attempt is certain to fail.)

When the Apply button is pressed with a conjecture C selected, the effect is to send the command “apply TheoremForwardOrBackward C ” to theproof engine.

5.9 Other natural deduction encodings

Bernard’s J’n’J encoding is in `examples/jnj`, and described in “Using J’n’J in Jape”, distributed with Jape. My own encoding is in `examples/natural_deduction`, is described in my book (*An Introduction to Proof and Disproof in Formal Logic*, OUP 2005) and dissected in chapter 10 below.

Chapter 6

Equational reasoning about functional programs

Previous chapters have dealt with the encoding of logics which are variations, more or less, on the sequent calculus. This chapter describes Bernard Sufrin's treatment of a very different logic (I helped with some of the underlying mechanisms and messed up some of the details, but it's Bernard's ideas and Bernard's encoding). The problem here is to control a large number of equations used to reason about functional-programming formulae, and to present an interface which makes it look as if equational reasoning is taking place, despite the Gentzen tree in the background.

The treatment is distributed in `examples/functional_programming`.

6.1 Syntax (`equality_rules.j`, `functions_rules.j`)

The master file (`functions.jt`) defines the shape of a sequent, allows rules to be stated additively, without repetition of the context Γ, and defines the `Fail` tactic:

```
SEQUENT IS BAG ⊢ FORMULA
INITIALISE autoAdditiveLeft true
TACTIC Fail (x) IS SEQ (ALERT x) STOP
```

Jape provides juxtaposition as a primitive syntactic construction, and functional programming uses it for function application. In the same way the syntax of tupling is inherited from Jape. This encoding doesn't automatically treat every juxtaposition as predicate application, but it uses Jape's ABSTRACTION mechanism in certain key places.

The file `equality_rules.j` covers more than simple equality, since it was originally intended to be shared between different encodings. Apart from that, it is pretty straightforward. The syntactic description is:

```
CLASS VARIABLE x y
CLASS FORMULA A B C F G X Y Z
CONSTANT ⊥

SUBSTFIX 2000 { x \ A }
JUXTFIX 1000
INFIX 200L = ≥ ≤ ≠ < >
INFIX 250L + -
INFIX 260L * /
INFIX 270L ^
```

Reading from the bottom, Bernard defines some binary arithmetical operators, all left-associative, then the priority of juxtaposition and substitution. The syntax of substitution is slightly variable in Jape: you can specify the bracketing symbols and the separating symbol as well as defining whether the variables come before the names or vice-versa. The spaces between symbols and names are essential to delimit the various components of the syntactic form. Bernard chose to make *formula* { *variables* \ *formulae* } the syntax of a substitution form, because that liberates square brackets for use in their conventional rôle in functional programming as list brackets (and I think he would also have reversed the order of formulae and variables had he not already used the '/' operator for arithmetic division).

Then there is a perfectly normal definition of an identity rule:

```
RULE hyp IS A ⊢ A
IDENTITY hyp
```

There follow the basic rules of equality. Because Jape doesn't yet have any treatment of families of rules, Bernard can only give a tuple-equality rule for a fixed finite number of tuple sizes, and he restricts himself to pairs:[1]

$\dfrac{}{\Gamma \vdash X = X}$ = reflexive	$\dfrac{\Gamma \vdash X = Y \quad \Gamma \vdash Y = Z}{\Gamma \vdash X = Z}$ = transitive
$\dfrac{\Gamma \vdash X = Y}{\Gamma \vdash Y = X}$ = symmetric	$\dfrac{\Gamma \vdash X0 = X1 \quad \Gamma \vdash Y0 = Y1}{\Gamma \vdash (X0, X1) = (Y0, Y1)}$ $(,) =$

In Japeish the rules are:

```
RULE "= reflexive" IS INFER X = X
RULE "= transitive"(Y) IS FROM X = Y AND Y = Z INFER X = Z
RULE "= symmetric" IS FROM X = Y INFER Y = X
RULE "(,)=" IS FROM X0=X1 AND Y0=Y1 INFER (X0, Y0) = (X1, Y1)
```

Although most of these rules can be derived from '= reflexive' plus the rewrite rules given below, it is convenient to have them available directly (and in any case, at the time this encoding was developed, Jape couldn't prove derived rules with antecedents).

Extensionality rules are straightforward, but because each implicitly incorporates a step of generalisation, it's necessary to include FRESH provisos:

$\dfrac{\Gamma \vdash F\,(x) = G\,(x)}{\Gamma \vdash F = G}$ (FRESH x) ext	$\dfrac{\Gamma \vdash F(x, y) = G(x, y)}{\Gamma \vdash F = G}$ (FRESH x, y) ext2

The Japeish version uses OBJECT parameters so that the rules, normally used backwards, introduce new identifiers rather than unknowns:

[1] But see the treatment of BAN logic in chapter 9.

```
RULE ext (OBJECT x) WHERE FRESH x IS FROM F x = G x INFER F = G
RULE ext2(OBJECT x, OBJECT y) WHERE FRESH x, y
IS FROM F (x, y) = G (x,y) INFER F = G
```

And that, so far as this encoding is concerned, is where the simple bit ends.

6.2 The rewrite rule and user definition of substitutions

The treatment of equational reasoning is founded on substitution. You can replace occurrences of a sub-formula X within a formula A by an alternative sub-formula Y, provided only that you can prove $X = Y$. Because an equality can be used to rewrite in either direction Bernard includes two rules, whose names are arbitrarily chosen:

$\dfrac{\Gamma \vdash X = Y \quad \Gamma \vdash A\{x\backslash Y\}}{\Gamma \vdash A\{x\backslash X\}}$ rewrite	$\dfrac{\Gamma \vdash X = Y \quad \Gamma \vdash A\{x\backslash X\}}{\Gamma \vdash A\{x\backslash Y\}}$ rewritebackwards

In Japeish we give the first rule a parameter X and the second a parameter Y. Just for fun we write the rules in predicate notation using ABSTRACTION to declare the relevant parameter, and Jape translates into the equivalent of the rules written above, inserting an OBJECT declaration for x to ensure that we get a fresh variable rather than an unknown when the rule is instantiated.

```
RULE rewrite (X,ABSTRACTION AA) IS FROM X=Y AND AA(Y) INFER AA(X)
RULE rewritebackwards (Y,ABSTRACTION AA) IS FROM X=Y AND AA(X) INFER AA(Y)
```

The problem formula which matches $AA(X)$ or $AA(Y)$ will itself be an equation in all the conjectures which we shall consider, but we don't need to take account of that in the rewrite rules themselves. The parameter is called AA in Japeish rather than A for some reason that I've now forgotten but was once important in presenting proofs that use the rules: in the description which follows I'll pretend it's called A.

Matching a rewrite rule to a problem sequent would seem to require unifying a substitution $\_A\{x\backslash\_X\}$ (or $\_A\{x\backslash\_Y\}$) with the conclusion formula C of the sequent. This isn't straightforward. Jape's principal mechanism when dealing with substitution formulae in unifications is to simplify them out of existence, but this formula is almost entirely made of unknowns. In principle all that Jape ought to do is to defer the problem by constructing a proviso

```
C UNIFIESWITH _A{x\_X}
```

and await action from the user which defines $\_A$ and $\_X$, so that the substitution can be simplified and the unification can proceed. It's been programmed to be a little more pragmatic than that, though UNIFIESWITH provisos are still a backstop in a tight corner.

The original mechanism. The user provides an argument formula in place of X (by subformula selection) and Jape searches for instances of that argument in the conclusion formula, constructing a substitution $C'\{x\backslash X\}$ which is guaranteed to reduce to C; then it unifies $\_A$ and C'. The process is made easier by the fact that x is a fresh variable in the proof. The search finds every instance of the argument formula in the right-hand side of the problem sequent.

The newer mechanism. The user describes a substitution $C'\{\_v\backslash E\}$ by picking out (again, with subformula selection) instances of a formula E inside C; once again, the substitution is guaranteed to reduce to C. Then $\_A$ is unified with C', x with $\_v$ and $\_X$ with E.

The newer mechanism gives finer control, but the original mechanism can sometimes be convenient. The original mechanism is built into the innards of Jape: if you subformula-select and then apply a *rule* (not a tactic) then the selection is matched to the first argument of the rule. But in general both need tactical support, because few rules these days are applied directly.

The original mechanism is supported by LETARGSEL / WITHARGSEL, typically by

```
WHEN (LETARGSEL pat ...  (WITHARGSEL rule) ...)
```

If the user has made a single subformula selection, and that selection unifies with pat, then run the tactic sequence which follows; eventually apply $rule$ taking account of the original subformula selection.

The newer mechanism is supported by LETSUBSTSEL / WITHSUBSTSEL, typically by

```
WHEN (LETSUBSTSEL _A{_x\_B} ...  (WITHSUBSTSEL rewrite) ...)
```

LETSUBSTSEL *pattern tactic* ... is a guarded tactic whose guard succeeds if:

- the user has made one or more subformula selections;
- all those selections are of identical subformulae E within the same formula F in the current goal sequent (not necessarily a conclusion: there are LETCONCSUBSTSEL and LETHYPSUBSTSEL when you need to be definite about the right/left division within the sequent);
- it is possible to construct a substitution form $F'\{v\backslash E\}$, with fresh variable v, such that $F'\{v\backslash E\}$ reduces to F in the presence of the proviso v NOTIN F (this could fail if, for example, you've selected a subformula including a variable z inside a quantified formula with bound variable z);
- *pattern* unifies with $F'\{v\backslash E\}$, without simplifying the substitution unless it is unified with a non-substitution form.

The first three conditions make sure that the user's subformula selection can be viewed as describing a substitution. The fourth allows the tactics inside LETSUBSTSEL to see what the user is up to. If all four conditions are satisfied, the sequence *tactic* ... is executed within the context created by the unification of *pattern* with $F'\{v\backslash E\}$. Jape's unification mechanism normally simplifies substitutions whenever it comes across them, so this seems a bit daft, but there is a little more trickery here. The substitution form $F'\{v\backslash E\}$ is specially marked so that it is not simplified during the unification process unless as necessary to unify with a non-substitution form: the effect is that it will be unified by structure-matching with a substitution form in *pattern*, if any is provided. In the fragment above, for example, $\_A$ would be unified with F', $\_x$ with v, and $\_B$ with E. That is, LETSUBSTSEL lets you see the substitution formula that the user implicitly constructs with their subformula selections.

That's not enough, though, because we'd like to use exactly the same substitution in the base of the rewrite rules, unifying $\_X$ or $\_Y$ with the user's E, $\_A$ with the user's F', and x with v. That's what WITHSUBSTSEL is for: it presents the rule-matching process with a version of the target sequent in which F has been replaced by $F'\{\_v\backslash E\}$ and it marks that substitution so that it isn't always automatically simplified; there's also a proviso $\_v$ NOTIN F. The effect is what we want: a rule step that uses the particular substitution the user described.

The effect of all this machinery is that it is possible for a user to specify, simply by text-selecting them, the instances of a subformula X which are to be replaced by Y, working backwards with the rewrite rule — or Y with X, working backwards with rewritebackwards. Based on that bit of magic, a great deal becomes possible.

6.3 Hiding parts of a proof: the LAYOUT tactical

When Jape uses the rewrite rule in this logic-encoding, for the most part the left antecedent $X = Y$ is part of a function definition. These definitions — 'facts' like $\mathrm{map}\, f\ [\,] = [\,]$ — are supposed to be well-known to the user, and are therefore best kept as marginal notes in the proof. The eventual goal is to be able to show a linear equational proof like those in Bird and Wadler's *Functional Programming*, in which every step transforms a formula by equality-substitution:

$$\begin{aligned} \mathrm{rev}(\mathrm{rev}[\,]) &= \mathrm{rev}[\,] && (\mathrm{rev}[\,] = [\,]) \\ &= [\,] && (\mathrm{rev}[\,] = [\,]) \\ &= \mathrm{id}[\,] && (\mathrm{id}\, x = x) \end{aligned}$$

In this style the definitions used in each step are noted in the justification of an equality, not included as antecedents of an inference step.

One thing Jape can do is to hide some of the antecedent proof trees of a proof step, and to alter the displayed justification of that step to record some of the information which is hidden. Ths is done with the LAYOUT tactical, which is given the justification of the step, a description of the antecedents that should remain visible, and a tactic which generates the proof tree itself. One of the tactics in Bernard's encoding, for example, reads as follows:

```
TACTIC UnfoldOneSel(x) IS
  WHEN
    (LETSUBSTSEL _A (LAYOUT "Fold %h" (1) (WITHSUBSTSEL rewrite)) x)
    (LETARGSEL _A
      (Fail (The formula you selected (_A) is not a proper subformula)))
    (Fail (Please text-select an expression))
```

LETSUBSTSEL checks that the user has selected some instance or instances of a sub-formula which describe a substitution, and if so WITHSUBSTSEL applies *rewrite* to the user's selection; finally the argument tactic *x* is applied to the first antecedent of the rewrite (the $X = Y$ antecedent). The LAYOUT tactical says in its second argument that only antecedent 1 of the rewrite step — that is, the right-hand antecedent, because antecedents are numbered 0, 1, ... from left-to-right — should be shown; the first argument says that it should be shown with a text which starts with "Fold"[2] and continues (%h) with a summary of the hidden subtree. The rest of the code tries to explain what has gone wrong if the user mis-applies the tactic.[3]

Figure 6.1 shows an example proof using this encoding, after two steps. Lines 1, 2 and 3 can be read as a

[2] In function-programming parlance "fold" means using the function definition as a right-to-left rewrite rule, and "unfold" is a similar but left-to-right use. A backwards unfold is a fold when read forwards. Proofs in this encoding are made backwards but read forwards: Bernard labelled buttons and named tactics in the backward sense, but the labels on the proofs, which are inserted by LAYOUT and the Fold/Unfold with hypothesis rules, are written in the forward sense. Confusing, but necessary.

[3] The attempt to analyse errors in the application of this tactic, using LETSUBSTSEL and LETARGSEL to pick out different cases, doesn't really work. To do a proper job, the tactic would to distinguish between at least these possibilities:

- the subformulae you select aren't identical;
- they don't all come from the same formula;
- one or more of them isn't a proper subformula;
- you didn't select anything at all.

The first error message (you didn't select a proper subformula) is necessary because, unusually, this encoding uses token selection rather than subformula selection: see section 6.4.

```
   ...
1: rev(rev x)=x
2: rev(rev x)=id x  Fold id 1
3: (rev•rev)x=id x  Fold • 2
4: rev•rev=id       ext 3
```

Figure 6.1: Beginning of a proof by substitution of equals

```
1: (rev•rev)x=rev(rev x)  •
2: id x=x                 id
   ...
3: rev(rev x)=x
4: rev(rev x)=id x        [rewrite] 2,3
5: (rev•rev)x=id x        [rewrite] 1,4
6: rev•rev=id             ext 5
```

Figure 6.2: Beginning of a proof by substitution of equals, with hidden steps revealed

partly-completed equational proof, up the left-hand side and down the right:

$$\begin{array}{lll} (\mathrm{rev} \bullet \mathrm{rev})x & = \mathrm{rev}(\mathrm{rev}\, x) & (f \bullet g)\; x = f(g\; x) \\ & \ldots & \\ & = x & \\ & = \mathrm{id}\, x & \mathrm{id}\, x = x \end{array}$$

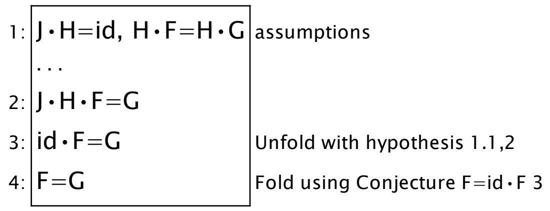
LAYOUT hides antecedents, but they can be revealed: by double-clicking on the justifications of lines 2 and line 3 the hidden steps appear, as shown in figure 6.2.

6.4 Dealing with associative operators: token selection, Find and Flatten

LETSUBSTSEL and WITHSUBSTSEL don't solve all the problems of rewriting, because of associative operators. The problem begins when a formula is input. In Jape's treatment of syntax, just as in any ordinary programming-language compiler, binary operators have relative priorities (or precedences) and an formula such as $A \times B + C$, where $\times$ has higher priority than +, is treated internally just like $(A \times B) + C$ but displayed in its unbracketed form.[4] Jape has to be told, faced with the formula $A + B + C$, whether to read it left-associatively as $(A + B) + C$ or right-associatively as $A + (B + C)$. Whichever you tell it, it will display the result unbracketed as $A + B + C$, and then inevitably some textual segment — $B + C$ in the left-associative case, $A + B$ in the right-associative — will look like a subformula but won't be.

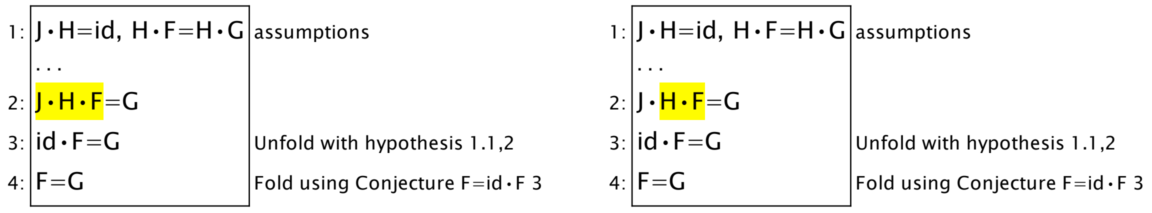
You might hope to tell Jape that the operator $+$ is neither left- nor right-associative but *associative* in the mathematical sense, so that $A + B + C$ can be read at will as either $(A + B) + C$ or $A + (B + C)$ as circumstances dictate — and then you can imagine that it ought to be possible to tell it also that $+$ is commutative, so that $A + B + C$ can be read as $(A + C) + B$ if that is what you wish. That would be pretty

[4] Jape tries to keep the user's bracketing structure. If the input is bracketed, so will be the display.



```
1: J•H=id, H•F=H•G  assumptions
   ...
2: J•H•F=G
3: id•F=G           Unfold with hypothesis 1.1,2
4: F=G              Fold using Conjecture F=id•F 3
```

Figure 6.3: A problem which involves associativity



```
1: J•H=id, H•F=H•G  assumptions
   ...
2: J•H•F=G
3: id•F=G           Unfold with hypothesis 1.1,2
4: F=G              Fold using Conjecture F=id•F 3
```

(a) subformula selection

```
1: J•H=id, H•F=H•G  assumptions
   ...
2: J•H•F=G
3: id•F=G           Unfold with hypothesis 1.1,2
4: F=G              Fold using Conjecture F=id•F 3
```

(b) token selection

Figure 6.4: Alt-press-and-drag across $H \bullet F$

difficult, and I never tried very hard to do it. Bernard's encoding tackles the problem head-on, using some specially-coded Jape internals.

Consider the function-composition operator (•). You can prove — see the Conjectures panel — that $(F \bullet G) \bullet H = F \bullet (G \bullet H)$: that is, it's associative. Given the problem in figure 6.3, you might want to extract the $H \bullet F$ subformula of line 2 and rewrite it using the equivalence in the second hypothesis. Unfortunately, • is left-associative (see `functions_rules.j`), so that line implicitly reads as $(J \bullet H) \bullet F = G$ and so $H \bullet F$ isn't a subformula. You could, however, transform line 2 into $J \bullet (H \bullet F) = G$ by substitution of equals for equals and the associative equivalence, and then perhaps complete the proof.

The problem is: how can you do this in Jape? Bernard wanted to do it automatically, without the user having to say what equivalence to call upon. (I still wish he hadn't tried, but the price of friendship is that you have to let your friends do what they want to do, and in this case I had to help him do it too.)

Begin with selection. If you try to subformula-select $H \bullet F$ with alt-press-and-drag, Jape doesn't do what you want — because what you are dragging across isn't a subformula. You first have to switch to a more simplistic form of selection, called token selection, using the Edit menu.[5] Then you can select tokens using alt-press-and-drag, and if you are careful you can select just the sequence of tokens you want, as shown in figure 6.4(b).[6]

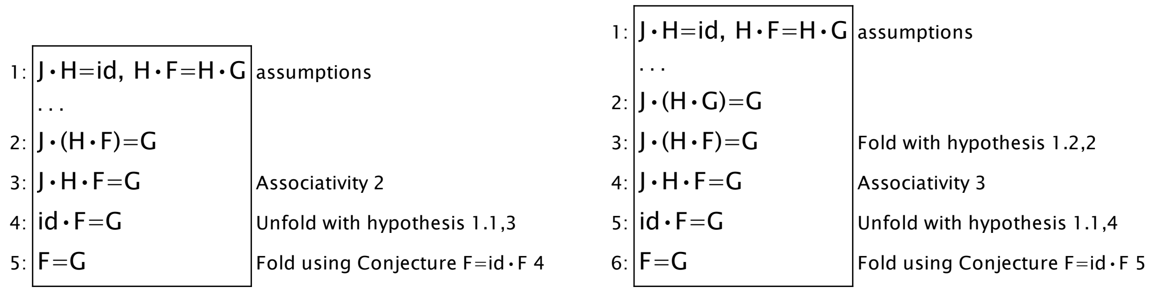
With that token-sequence selection you can use Find from the Rules menu or from either of the panels[7] to rewrite the problem so that $H \bullet F$ is a subformula: the result is figure 6.5(a), and then Unfold with hypothesis from the Rules menu gives you figure 6.5(b). Now we're in another hole: we'd like to use the $J \bullet H = id$ equivalence, but $J \bullet H$ isn't a subformula.

The solution is to use Flatten (Rules menu or panel) to transform the problem again, taking out brackets from associative formulae where possible. That gives figure 6.6, and the rest is easy. It remains to explain

[5] A token is a single word, like id or H, or a connective or a bracket.

[6] Token selection is a kind of subformula selection, one in which it's possible to make more mistakes than the normal variety. Jape's internals don't expect subformula selections to deliver proper subformulae, and deliver appropriate error messages when it doesn't.

[7] I don't think this is good GUI practice, but it's Bernard's encoding.



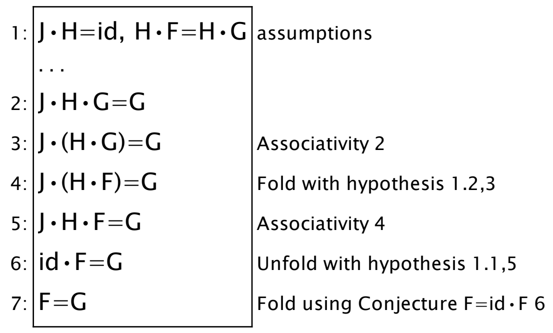
```
1: J•H=id, H•F=H•G  assumptions
   ...
2: J•(H•F)=G
3: J•H•F=G          Associativity 2
4: id•F=G           Unfold with hypothesis 1.1,3
5: F=G              Fold using Conjecture F=id•F 4
```

(a) Find

```
1: J•H=id, H•F=H•G  assumptions
   ...
2: J•(H•G)=G
3: J•(H•F)=G        Fold with hypothesis 1.2,2
4: J•H•F=G          Associativity 3
5: id•F=G           Unfold with hypothesis 1.1,4
6: F=G              Fold using Conjecture F=id•F 5
```

(b) unfold

Figure 6.5: Token selection and Find allow progress



```
1: J•H=id, H•F=H•G  assumptions
   ...
2: J•H•G=G
3: J•(H•G)=G        Associativity 2
4: J•(H•F)=G        Fold with hypothesis 1.2,3
5: J•H•F=G          Associativity 4
6: id•F=G           Unfold with hypothesis 1.1,5
7: F=G              Fold using Conjecture F=id•F 6
```

Figure 6.6: Flatten takes us out into the open again

how all this magic works.

The Find entries in menu and panels — see `functions_menus.j` — simply fire a tactic called FindSelection. That tactic is

```
TACTIC FindSelection IS
  WHEN
    (LETHYPFIND (_XOLD=_YOLD, _XNEW=_YNEW)
      (ALT
        (LAYOUT "Associativity" (2)
          (rewriteHypotheticalEquation _XOLD _XNEW _YOLD _YNEW)
          EVALUATE
          EVALUATE)
        (LETARGSEL _XSEL (Fail ("%s isn't a subterm", _XSEL)))))
    (LETCONCFIND (_XOLD=_YOLD, _XNEW=_YNEW)
      (ALT
        (LAYOUT "Associativity" (2)
          (rewriteEquation _XOLD _XNEW _YOLD _YNEW) EVALUATE EVALUATE)
        (LETARGSEL _XSEL (Fail ("%s isn't a subterm", _XSEL)))))
```

LETHYPFIND (*oldpat*, *newpat*) *tactic* ... and the similar LETCONCFIND tactical are the first part of the magic. They test that:

1. the user has made a single subformula or token selection in a hypothesis (LETHYPFIND) or conclusion (LETCONCFIND) formula respectively, dividing it into three sections (*before*, *selected* and *after*);

2. the original formula unifies with *oldpat*;

3. *selected* is a valid formula[8];

4. the text *before* (*selected*) *after* is a valid formula;

5. *before* (*selected*) *after* unifies with *newpat*.

If all these tests succeed, and if *before* (*selected*) *after* is not structurally identical (i.e. identical if you ignore unnecessary brackets) to the original, then the tactic sequence inside the tactical is run. If the tests succeed but the new formula is structurally identical to the old, the tactic sequence isn't run but the tactical still succeeds. That is, LETHYPFIND / LETCONCFIND match, and run their argument tactics, if your text selection reorganises the structure of the formula.

The FindSelection tactic calls either rewriteHypotheticalEquation or rewriteEquation: those rules are

```
RULE rewriteEquation(X, X', Y, Y', OBJECT x) IS
  FROM ASSOCEQ (X, X') AND ASSOCEQ (Y, Y') AND X'=Y' INFER X=Y
RULE rewriteHypotheticalEquation(X, X', Y, Y', OBJECT x) IS
  FROM ASSOCEQ (X, X') AND ASSOCEQ (Y, Y') AND X'=Y'⊢ P INFER X=Y ⊢ P
```

The built-in tactic EVALUATE, applied to ASSOCEQ(E, F), flattens E and F — that is, removes any internal bracketing, reducing them to a canonical left- or right-associative version as appropriate — using any relevant theorems / rules / conjectures about associativity, and succeeds if the flattened versions are identical. Each of these rules therefore replaces an equation with a provably equivalent equation. The use of layout in FindSelection hides this internal working and gives Associativity as the justification for the step.

The reverse operation is provided by FLATTEN. The menu and panel entries index the Flatten tactic (see equality_menus.j)

```
TACTIC Flatten IS
  LAYOUT "Associativity" (0)
    (WHEN  (LETARGSEL _A (FLATTEN _A))
        (LETGOAL (_X = _Y) (IF(FLATTEN(_X))) (IF(FLATTEN(_Y))))
        (LETGOAL _X (FAIL (Cannot Flatten _X)))
    )
```

This tactic gives the same justification as FindSelection; via FLATTEN it accesses the same machinery. The argument to flatten is used to determine the principal operator of the formula to be flattened; only subformulae of which that is the operator are flattened.

The effect of all this machinery is to enable the user to manipulate formulae which use associative operators without too many uses of associative rewrite laws.

6.5 Induction

Jape makes no special treatment of induction. It is handled in the same way as any other logical generalisation rule, using the FRESH proviso. Bernard encodes a form of list induction which uses concatenation

[8] Maybe we don't need this condition, but it would be very odd not to impose it.

rather than $cons$:[9]

$$\frac{\Gamma \vdash A[\,] \quad \Gamma \vdash A[x] \quad \Gamma, A(xs), A(ys) \vdash A(xs +\!\!+ ys)}{\Gamma \vdash A(B)} \ (\textsc{fresh}\ x, xs, ys)\ \text{listinduction}$$

That which usually takes two steps (by an induction principle prove $\forall zs \cdot A(zs)$, then infer $A(B)$ by specialisation) is collapsed into one. There is no need to introduce quantification into equational reasoning, and the one-step rule is perfectly convenient. Bernard encodes it directly:

```
RULE listinduction (B, OBJECT x, OBJECT xs, OBJECT ys, ABSTRACTION A)
  WHERE FRESH x, xs, ys IS
    FROM A[] AND A[x] AND A xs, A ys ⊢ A(xs++ys) INFER A(B)
```

Sometimes you will want to make a proof by induction of a proposition which is expressed in terms of some variable or other, and then you would want induction to apply to every instance of that variable. Other times you may want to be more precise in specifying just what instances of what sub-formula are to be the basis of induction, and so we require the user to specify those instances. Bernard could have allowed both mechanisms, activated by different entries in a menu, but has instead required you always to select the particular instances of a subformula which you wish to be the subject of induction. The entry in the menu which gives the user access to the list induction principle connects to a tactic which uses the letsubstsel/withsubstsel mechanism:

```
TACTIC "list induction tactic" IS
  WHEN
    (LETSUBSTSEL _A (WITHSUBSTSEL listinduction))
    (FAIL(Please select a sub-formula on which to perform induction))
```

6.6 Controlling collections of rules

In functional programs each function definition corresponds to a number of individual statements of equality. The definition of map, for example, gives three:

$$\begin{array}{llll} map & f & [\,] & = & [\,] \\ map & f & [x] & = & f\,x \\ map & f & (xs +\!\!+ ys) & = & map\ f\ xs +\!\!+ map\ f\ ys \end{array}$$

It would be tedious to be required to give a name to each individual equality, and in any case Bernard expects his users to be happy to refer to them as a collection — “use one of the map equalities”, rather than “use the map equality which applies to singletons”.

The RULES directive allows us to make and name collections of rules. If we turn all the function definitions into collections we can use them, with some instantiation of their variables, to close the left-hand antecedent of a *rewrite* rule application or to close a tip of a proof tree in the normal way. The definition of `map`, for example, goes as follows:

[9] Definining lists with concatenation rather than $cons$ has advantages, in particular the fact that it doesn’t favour either end of a list when making a reduction. It has difficulties, but it is valid. The sceptics (I am ashamed to admit that I was once one of them!) should note that you can derive this rule from the more familiar $cons$ version. As for evaluation strategies, or function definition by concatenation, that’s another story!

```
RULES map
  ARE map F [] = []
  AND map F [X]= [F X]
  AND map F (Xs++Ys) = map F Xs ++ map F Ys
END
```

This generates three rules, called `map’0`, `map’1` and `map’2`, plus a tactic `map`:

```
TACTIC map IS ALT map’0 map’1 map’2
```

In addition, for control of searching of the collections of rules, Bernard groups them into collections called ‘theories’. Part of the `List` theory, for example, as it is given in `functions_rules.j` is

```
THEORY List IS
  RULES length
  ...
  RULE  none IS none X  = [ ]
  RULE  one IS one X  = [X]
  RULE  cat IS cat = fold (++) []
  RULES rev
  ...
  RULES ++
  ...
  RULES map
  ...
  RULE filter IS filter P = cat • map (if P (one, none))
  RULES zip
  ...
  RULES fold
  ...
  RULE rev2 IS rev2 = fold rcat [] • map one
  RULE rcat IS rcat Xs Ys = Ys ++ Xs
  RULE ":" IS X:Xs = [X] ++ Xs
END
```

The effect of THEORY is to define all the rules and tactics described by its components, plus a tactic which allows search of those components. In this case the tactic is

```
TACTIC List IS ALT length none rev (++) map filter zip fold rev2 rcat (:)
```

The rule-collections — but not the theory-collections — are put into a panel of definitions. The panel is described in `functions_menus.j` as

```
TACTICPANEL "Definitions"
  TACTIC "Use any rule enabled by Searching" IS SearchTactic
  ENTRY ":"
  ENTRY "•"
  ENTRY "×"
  ENTRY "⊗"
  ...
  BUTTON "Unfold *" IS apply RepeatedlyUnfold
  PREFIXBUTTON "Unfold" IS apply UnfoldObvious
  PREFIXBUTTON "Fold" IS apply FoldObvious
  PREFIXBUTTON "Apply" IS apply
```

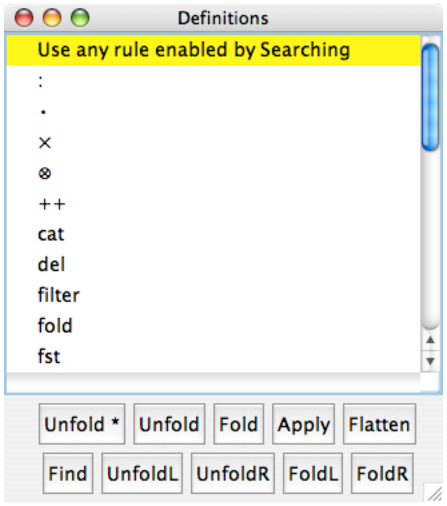


Figure 6.7: The functional programming panel of definitions

```
    BUTTON "Flatten" IS apply Flatten
    BUTTON "Find" IS apply FindSelection
  END
```

This produces the panel shown in figure 6.7. I discuss the effect of the tactics bound to the buttons and entries in what follows.

6.7 Searching collections of rules and theorems

It's quite possible, using the Unfold and Fold buttons on the Definitions panel, plus the Unfold with hypothesis and Fold with hypothesis entries in the Rules menu, to construct proofs entirely by hand — selecting the subformula to be replaced, the definition or hypothesis to be used, pressing the appropriate button or choosing the appropriate menu entry. But it's also quite easy to program Jape to do a sort of evaluation step. This involves identifying helpful equations (in the form of rules or theorems) which can be used to rewrite part of the conclusion of the problem sequent.

Jape has a number of built-in mechanisms which help with the process. The ALT tactical allows an undirected search amongst a number of possibly-applicable tactics, and I have illustrated above how the rules and theory directives automatically construct ALTs which may be useful in searching for a proof. But in equational reasoning the problem is somewhat different: we are looking for a subformula which is replaceable and a definition or hypothesis which matches it; ALT is not sufficient to do the job.

Jape's support for search in equational reasoning is at present the FOLD, UNFOLD, FOLDHYP and UNFOLDHYP tacticals. FOLD and UNFOLD tacticals take a rewrite rule and an ALT tactic, which is treated as a collection of rules. They filter the rules to consider only those whose consequents have a conclusion of the form $L\ op\ R$ for some formulae L and R and a binary operator op;[10] they search for subformulae of the conclusion of the problem sequent which match the R (FOLD) or L (UNFOLD) of one of the rules, and when they find a coincidence try to apply the rewrite rule followed by the matching filtered rule. FOLDHYP and

[10] The rules — actually rules and theorems — are doubly filtered because Jape eliminates all unproved conjectures unless the `applyconjectures` variable is set `true`.

UNFOLDHYP are similar, except that they take a pattern which allows the user to define op, and they search the list of current hypotheses rather than a collection of rules.

Although distressingly ad-hoc in my opinion (I'd rather Jape was more purely a logic engine), these techniques are fast and they work well. In this encoding the rules are such that automatic FOLDing is little use: too many equalities have right-hand sides which are similar, and searching with alt for a match rarely finds a useful one. But automatic UNFOLDing can often be fruitful: if there is a subformula which matches $map\ F\ (Xs \mathbin{++} Ys)$, for example, then unfolding with the rule $map\ F\ (Xs \mathbin{++} Ys) = map\ F\ Xs \mathbin{++} map\ F\ Ys$ is probably worthwhile.

Bernard's search mechanism, then, is based on the tactic

```
TACTIC Unfold(x) IS LAYOUT "Fold %h" (1) (UNFOLD rewrite x)
```

which is given an alt tactic `x` and which searches for (backwards) UNFOLD actions which it can carry out by rewriting with the rules within `x`.

The magic by no means stops with the unfold tactical, because Bernard also uses the collections of theories to control the search. The idea is that you should be able to 'turn on and off' the definitions and theorems in particular theories when searching. Because the variable-processing facilities of Japeish are primitive, this has to be done using radiobuttons in a special Searching menu.

Bernard grouped the equality rules into three theories: List (illustrated above), Functions and Reflect. He also grouped conjectures into collections, some of which Jape can be made to search. Here, for example, is the Listthms collection:

```
THEOREMS ListThms
ARE  rev • rev  = id
AND  rev2  = rev
AND  map F • map G  = map (F • G)
AND  map F • cat  = cat • (map (map F))
AND  none • F  = none
AND  map F • none  = none
AND  map F • one  = one • F
AND  map F • rev  = rev • map F
AND  map id  = id
AND  length•map F  = length
AND  zip • (map F ⊗ map G)  = map (F⊗G)
AND  map F • if P (G,G′)  = if P (map F • G, map F • G′)
AND  filter P  = map fst • filter snd • zip • (id ⊗ map P)
END
```

These, once proved, can be searched when automatically unfolding equalities; they can even be searched before they are proved, if applyconjectures is set to true.

The basic technique at present is a hack to get around the fact that Jape doesn't yet have any analogue of the ML *case* expression, which could be used to direct the activity of a tactic according to the value of a variable.[11] The hack is to use variables each of which is set to the name of a theory if we want to search that theory, or to the name of a tactic which is certain to fail, if we don't want to search it. The justfail tactic is

```
TACTIC JUSTFAIL IS (ALT)
```

and the Searching menu is

[11] It probably took longer to type this footnote than to implement the mechanism — but the manual must come first!

```
MENU "Searching" IS
  RADIOBUTTON dohyp IS
  "Search hypotheses" IS DoHyp
  AND "Ignore hypotheses" IS JUSTFAIL
  INITIALLY DoHyp
  END

  RADIOBUTTON list IS
  "List rules enabled" IS List
  AND  "List rules disabled" IS JUSTFAIL
  INITIALLY List
  END END
...
```

The Auto tactic is set up either to unfold or to fold — though for the reasons given above, it's never much used for folding — and is defined as

```
TACTIC Auto(foldunfold, foldunfoldhyp) IS
ALT  (dohyp foldunfoldhyp)
  (foldunfold list)
  (foldunfold listthms)
  (foldunfold function)
  (foldunfold functionthms)
  (foldunfold reflect)
  (foldunfold reflectthms)
  (FAIL (Cannot find anything to foldunfold))
```

It's called from the Unfold * button with the tactic

```
SEQ  (Auto Unfold UnfoldWithAnyHyp)
  (DO (Auto Unfold UnfoldWithAnyHyp))
```

— since DO always silently succeeds, Bernard wanted to make the button fail noisily if there was nothing at all that it could do; hence the double invocation of the tactic.

He uses the same tactic — but only singly, without repetition — if you double-click on a conclusion:

```
CONCHIT C IS Auto Unfold UnfoldWIthAnyHyp
```

The remaining parts of the jigsaw are the UnfoldWithAnyHyp tactic

```
TACTIC UnfoldWithAnyHyp IS UNFOLDHYP "Fold with hypothesis" (_A=_B)
```

and the Fold/Unfold with hypothesis pair of rules:

```
RULE "Fold with hypothesis" (X, OBJECT x) IS
   FROM X=Y ⊢ AA[x\Y] INFER X=Y ⊢ AA[x\X]
RULE "Unfold with hypothesis" (Y,OBJECT x) IS
   FROM X=Y ⊢ AA[x\X] INFER X=Y ⊢ AA[x\Y]
```

These rules are named for forward reading, so the menu entries which enable them to be used by hand have to be labelled contrariwise.

All of the other techniques that we have used have been discussed in earlier chapters, with the exception of the next, which is novel.

```
 1: rev(rev[])
 2:   = rev[]                            Unfold rev'0
 3:   = []                               Unfold rev'0
 4: rev(rev[x3])
 5:   = rev[x3]                          Unfold rev'1
 6:   = [x3]                             Unfold rev'1
    ┌────────────────────────────────┐
 7: │rev(rev xs)=xs, rev(rev ys)=ys  │   assumptions
 8: │rev(rev(xs++ys))                │
 9: │  = rev(rev ys++rev xs)         │   Unfold rev'2
10: │  = rev(rev xs)++rev(rev ys)    │   Unfold rev'2
11: │  = xs++rev(rev ys)             │   Unfold 7.1
12: │  = xs++ys                      │   Unfold 7.2
    └────────────────────────────────┘
13: (rev•rev)x
14:   = rev(rev x)                       Unfold •
15:   = x                                listinduction 1-3,4-6,7-12
16:   = id x                             Fold id
17: rev•rev=id                           ext 13-16
```

Figure 6.8: A proof of $\mathrm{rev} \bullet \mathrm{rev} = id$ exploiting transitivity and reflexitivity

6.8 Transitive reasoning: `trfunctions.jt`

There's a structural analogy between the logical cut rule and the rule of transitive equality:

$$\frac{\Gamma \vdash B \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \text{cut} \qquad\qquad \frac{\Gamma \vdash A = B \quad \Gamma \vdash B = C}{\Gamma \vdash A = C}\ {=}\ \text{transitive}$$

Each produces a formula B on the right-hand side in one antecedent, and deploys it on the left in the other. In a cut the movement is from one side of the turnstile to the other; in transitivity it's from one side of an equality to the other.

There's an analogy, too, between logical axiom (hyp in chapter 5 and this chapter) and reflexive equality:

$$\frac{}{\Gamma, C \vdash C}\ \text{hyp} \qquad\qquad \frac{}{\Gamma \vdash C = C}\ {=}\ \text{reflexive}$$

Chapter 5 shows that you can make a box-and-line version of a tree by recognising instances of hyp, and make it more powerful by recognising instances of cut. The same can be done with transitivity and reflexivity, producing proofs much more like the ones that equality-reasoners like to deal with. For example, figure **??** shows a proof that $\mathrm{rev} \bullet \mathrm{rev} = id$. Apart from details of layout, the steps on lines 1-3, 4-6, 8-12 and 13-16 look just like the equational proofs in Bird and Wadler. Much of the trickery is hiding of transitive and reflexive equality steps, and the same proof with that part of its machinery exposed is shown in figure 6.9.

Proofs like that are made by working on one side or the other of an equality. Jape needs to know that it's equality we care about. `functions.jt` already contains declarations of the rules:

1:	(rev•rev)x=rev(rev x)	Unfold •
2:	rev(rev[])=rev[]	Unfold rev'0
3:	rev[]=[]	Unfold rev'0
4:	[]=[]	= reflexive
5:	rev[]=[]	= transitive 3,4
6:	rev(rev[])=[]	= transitive 2,5
7:	rev(rev[x3])=rev[x3]	Unfold rev'1
8:	rev[x3]=[x3]	Unfold rev'1
9:	[x3]=[x3]	= reflexive
10:	rev[x3]=[x3]	= transitive 8,9
11:	rev(rev[x3])=[x3]	= transitive 7,10
12:	rev(rev xs)=xs, rev(rev ys)=ys	assumptions
13:	rev(rev(xs++ys))=rev(rev ys++rev xs)	Unfold rev'2
14:	rev(rev ys++rev xs)=rev(rev xs)++rev(rev ys)	Unfold rev'2
15:	rev(rev xs)++rev(rev ys)=xs++rev(rev ys)	Unfold 12.1
16:	xs++rev(rev ys)=xs++ys	Unfold 12.2
17:	xs++ys=xs++ys	= reflexive
18:	xs++rev(rev ys)=xs++ys	= transitive 16,17
19:	rev(rev xs)++rev(rev ys)=xs++ys	= transitive 15,18
20:	rev(rev ys++rev xs)=xs++ys	= transitive 14,19
21:	rev(rev(xs++ys))=xs++ys	= transitive 13,20
22:	rev(rev x)=x	listinduction 6,11,12–21
23:	x=id x	Fold id
24:	rev(rev x)=id x	= transitive 22,23
25:	(rev•rev)x=id x	= transitive 1,24
26:	rev•rev=id	ext 25

Figure 6.9: A proof of $\mathrm{rev} \bullet \mathrm{rev} = id$ with transitive and reflexive steps revealed

```
AUTOMATCH "= reflexive"
TRANSITIVE "= transitive"
REFLEXIVE "= reflexive"
```

and `functions_menus.j` puts a checkbox in the Edit menu:

```
CHECKBOX hidetransitivity "transformational style" INITIALLY false
```

Now `functions_transitivity.j`, invoked from `trfunctions.jt`, rubs it in and makes both the display-controlling variables true:

```
INITIALISE hidereflexivity true

MENU Transitivity IS
  CHECKBOX hidetransitivity
    "Transitive Presentation Style"
INITIALLY true
END
```

In the same file Bernard adds entries to menus and buttons to panels to allow reductions on one side or the other of an equality[12]

```
TACTICPANEL Definitions
  BUTTON "Flatten LHS" IS apply FlattenLHS
  BUTTON "Flatten RHS" IS apply FlattenRHS
  BUTTON "Unfold LHS" IS apply UnfoldL COMMAND
  BUTTON "Unfold RHS" IS apply UnfoldR COMMAND
  BUTTON "Fold LHS" IS apply FoldL COMMAND
  BUTTON "Fold RHS" IS apply FoldR COMMAND
END

MENU Unfolding IS
  ENTRY "Unfold LHS" IS (UnfoldL SearchTactic)
  ENTRY "Unfold RHS" IS (UnfoldR SearchTactic)
  ENTRY "Fold LHS" IS (FoldL SearchTactic)
  ENTRY "Fold RHS" IS (FoldR SearchTactic)
  ENTRY "Flatten LHS" IS FlattenLHS
  ENTRY "Flatten RHS" IS FlattenRHS
END
```

Most of the gesture-interpreting work is down to UnfoldR and UnfoldL. This is how UnfoldL begins:

```
TACTIC UnfoldL(rule) IS
  (WHEN
    (LETSUBSTSEL (_A{_x\_F}=_B) /* Rewrite the selection */
      "= transitive"
      (LAYOUT "Unfold %h" ()
        (rewriteLR{X,AA,x\_F,_A,_x})
        (USEHYPORRULE rule)))
```

[12] Bernard prefers to work with the variable `autoselect` set true, and strum on buttons in panels — he invented panels just so he could do that! I prefer to work by pointing at the thing I'm going to work on, so in my encodings I set `autoselect` false, and in transitivity settings I use the LETLHS / LETRHS tacticals to let Jape find out what I'm up to. Horses for courses.

— if you've made a selection on the left of the equality, run rewriteLR on it, and label the result with the name of the rule. (The use of argument-substitution in the call of rewriteLR may be an attempt to deal with the consequences if the selection is on the right of the equality, but I think WITHSUBSTSEL would be a better mechanism to use. But then, I've tinkered enough with this encoding: leave it as it is.) The rewrite tactic it uses is one of a pair from `equality_rules.j`

```
RULE rewriteLR (OBJECT x) IS FROM X=Y INFER AA{x\X}=AA{x\Y}
RULE rewriteRL (OBJECT x) IS FROM X=Y INFER AA{x\Y}=AA{x\X} /* derivable
*/
```

and `functions_transitivity.j` has

```
TACTIC USEHYPORRULE(rule) IS (WHEN (LETHYP _H (WITHHYPSEL hyp)) (ALT hyp
rule))
```

The second UnfoldL alternative is

```
(LETCONCFIND (_XOLD=_YOLD, _XNEW=_YNEW)
  (LETGOALPATH G
    "= transitive"
    (ALT
      (LAYOUT "Associativity" (2)
        (associativity _XNEW _XOLD) EVALUATE)
      (LETARGSEL _XSEL (Fail ("%s isn't a subterm", _XSEL))))))
```

— if you've made a token selection which isn't a subformula, do the ASSOCEQ thing.

The third UnfoldL alternative is

```
(LETHYP (_A=_B) /* Use selected hypothesis to rewrite everywhere */
  (LETCONC (_X=_Y)
    "= transitive"
    (ALT
      (LAYOUT "Unfold" ALL (rewriteLR{X\_A}) (WITHHYPSEL hyp))
      (Fail "Cannot use the selected hypothesis to unfold on the LHS"))))
```

— if you've selected a hypothesis, use it left-to-right to rewrite the conclusion.

The penultimate UnfoldL alternative is the tricky one. It searches the left-hand side for things to work on:

```
(LETGOAL (_A=_B) /* No selection:  a little automation */
  (LETGOALPATH G
    "= transitive"
  (ALT (UNFOLDWITH1 rule) /* Close with the rule */
    (UNFOLDWITH2 FunSearch4 rule) /* Close by rewriting inside the formula
*/
    (Fail "Nothing obvious to unfold on the LHS"))
  (GOALPATH (SUBGOAL G 1))))
```

UNFOLDWITH1 is straightforward:

```
TACTIC UNFOLDWITH1(rule1) IS (LAYOUT "Unfold %s" () rule1)
```

%s in the string is replaced by the label of the root of the tree produced by rule1.

UNFOLDWITH2 is more complicated, because of what Funsearch does. Here's the bundle of rules it uses:

```
RULE FunctionOperand IS FROM X=Y INFER F X = F Y
RULE FunctionOperator IS FROM F=G INFER F X = G X
RULE LeftPair IS FROM X=X′ INFER (X, Y)=(X′, Y)
RULE RightPair IS FROM Y=Y′ INFER (X, Y)=(X, Y′)
RULE Singleton IS FROM Y=Y′ INFER [Y]=[Y′]
```

and here's the bundle of search tactics:

```
TACTIC FunApply(rule) IS (rule)

TACTIC FunRecurse(continue, rule) IS
  (ALT (continue rule)
    (LAYOUT "%h" () FunctionOperator (continue rule))
    (LAYOUT "%h" () FunctionOperand (continue rule))
    (LAYOUT "%h" () LeftPair (continue rule))
    (LAYOUT "%h" () RightPair (continue rule))
    (LAYOUT "%h" () Singleton (continue rule)))

TACTIC FunSearch1(rule) IS (FunRecurse FunApply rule)
TACTIC FunSearch2(rule) IS (FunRecurse FunSearch1 rule)
TACTIC FunSearch3(rule) IS (FunRecurse FunSearch2 rule)
TACTIC FunSearch4(rule) IS (FunRecurse FunSearch3 rule)
```

Each of the steps hides its own action with %h, effectively transmitting its child action label down the tree. Finally, UNFOLDWITH2 has to do a little GOALPATH dance, switching from the final position (G1) to the original position (G), labelling it — LAYOUT is like an assignment to a tree node – and then switching back to G1, to make sure that the correct layout is applied to the root of the tree that it builds:

```
TACTIC UNFOLDWITH2(rule1, arg) IS
  /* override any layout set by rule1 */
  (LETGOALPATH G
    (rule1 arg)
    (LETGOALPATH G1      (GOALPATH G)
      (LAYOUT "Unfold %h" ())
      (GOALPATH G1)))
```

Finally, if nothing works, the last alternative in UnfoldL gives an error message:

```
(Fail (Cannot unfold LHS because there is no equation)))
```

And that's how it works: quite remarkably simple in the end.

Chapter 7

Encoding axiomatic set theory

The treatment of equational reasoning in the previous chapter introduced the ways in which Jape can hide parts of a proof and use substitution to achieve replacement of subformulae with rewrite rules. This chapter shows how the same techniques can be used to support the encoding of a version of naive axiomatic set theory, which uses rewriting to support equality-style reasoning in both forward and backward steps. The treatment was inspired by that of David Schmidt ("Natural Deduction Theorem Proving in Set Theory", CSR-142-83, Edinburgh); like chapter 5 this was an encoding to support a lecture course at QMW, commissioned by the lecturer.

The encoding presents four distinct things to the user: an encoding of natural deduction, as a menu of commands; an menu of rewrite actions; a menu of set-theoretic inference rules; and a panel of axioms expressed as definitions $formula \hat{=} formula$, equipped with buttons which allow those definitions to be used as left-to-right or right-to-left rewrite rules. In addition there's a menu of conjectures equipped with buttons which allow the user to exploit proved theorems as rewrite rules.

In its time this was the most ambitious use of Japeish so far to produce a slick on-screen encoding with a lot of different — but easy to use — facilities. The encoding can hardly be described as 'transparent'. The tactic programming is, indeed, at times rather subtle.

7.1 The natural deduction encoding

This is contained in the files `examples/Barwise_n_Etchemendy/BnE-Fprime.jt` and the files that it invokes; it is derived from the logic F' in "*The Language of First-Order Logic*" by Barwise and Etchemendy. It is very similar to the encoding described in chapter 5 above, with the addition of rules for a bi-implication operator, a falsity constant, equality, and a unique-existence operator:

$\dfrac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \leftrightarrow B} \leftrightarrow -I$	$\dfrac{\Gamma \vdash B \quad \Gamma \vdash A \leftrightarrow B}{\Gamma \vdash A} \leftrightarrow -E(L)$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \leftrightarrow B}{\Gamma \vdash B} \leftrightarrow -E(R)$
$\dfrac{\Gamma \vdash P \quad \Gamma \vdash \neg P}{\Gamma \vdash \bot} \bot - I$	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash P} \bot - E$	$\dfrac{}{\Gamma \vdash A = A} A = A$
$\dfrac{\Gamma \vdash A[x\backslash B] \quad \Gamma, A[x\backslash c] \vdash B = c}{\Gamma \vdash \exists! x.A}$ (FRESH c, c NOTIN B) $\exists! - E$		$\dfrac{\Gamma \vdash \exists! x.A}{\Gamma \vdash \exists x.A} \exists! - E$

plus a pair of rewrite rules for each of the bi-implication and equality operators:

$\dfrac{\Gamma \vdash A \leftrightarrow B \quad \Gamma \vdash P[v\backslash B]}{\Gamma \vdash P[v\backslash A]}$ *rewrite* $\leftrightarrow$	$\dfrac{\Gamma \vdash A \leftrightarrow B \quad \Gamma \vdash P[v\backslash A]}{\Gamma \vdash P[v\backslash B]}$ *rewrite* $\leftrightarrow$
$\dfrac{\Gamma \vdash A = B \quad \Gamma \vdash P[v\backslash B]}{\Gamma \vdash P[v\backslash A]}$ *rewrite* $=$	$\dfrac{\Gamma \vdash A = B \quad \Gamma \vdash P[v\backslash A]}{\Gamma \vdash P[v\backslash B]}$ *rewrite* $=$

These are encoded, completely straightforwardly, in the file `examples/Barwise_n_Etchemendy/BnE-Fprime_ru`
The rules are inserted into the menu as

```
MENU "System F´" IS
  ENTRY "→-I"
  ENTRY "↔-I"
  ENTRY "∧-I"
  ENTRY "∨-I(L)" IS FOB ForwardCut 0 "∨-I(L)"
  ENTRY "∨-I(R)" IS FOB ForwardCut 0 "∨-I(R)"
  ENTRY "¬-I"
  ENTRY "⊥-I"
  ENTRY "∀-I"
  ENTRY "∃-I" IS "∃-I tac"
  ENTRY "∃!-I" IS "∃!-I tac"

  SEPARATOR

  ENTRY "→-E" IS "→-E tac"
  ENTRY "↔-E(L)" IS FOB "↔-E(L) forward" 0 "↔-E(L)"
  ENTRY "↔-E(R)" IS FOB "↔-E(R) forward" 0 "↔-E(R)"
  ENTRY "∧-E(L)" IS FOB ForwardCut 0 "∧-E(L)"
  ENTRY "∧-E(R)" IS FOB ForwardCut 0 "∧-E(R)"
  ENTRY "∨-E" IS "∨-E tac"
  ENTRY "¬-E" IS FOB ForwardCut 0 "¬-E"
  ENTRY "⊥-E" IS FOB ForwardCut 0 "⊥-E"
  ENTRY "∀-E" IS "∀-E tac"
  ENTRY "∃-E" IS "∃-E tac"
  ENTRY "∃!-E(∃)" IS FOB ForwardCut 0 "∃!-E(∃)"
  ENTRY "∃!-E(∀∀)" IS FOB ForwardCut 0 "∃!-E(∀∀)"

  SEPARATOR

  ENTRY "A=A"
  ENTRY hyp IS hyp END
```

Here `fob` is essentially the tactic `ForwardOrBackward` of chapter 5, `ForwardCut` and `ForwardUncut` are also as described there, and the entries for bi-implication use the tactics

```
TACTIC "↔-E(L) forward"(Z) IS "↔-E forward" "↔-E(L)"
TACTIC "↔-E(R) forward"(Z) IS "↔-E forward" "↔-E(R)"

TACTIC "↔-E forward"(rule) IS
  WHEN (LETHYP (_A↔_B) (ForwardCut2 rule))
    (LETHYP _A (ForwardCut rule))
    (JAPE(fail(what’s this in rule forward?)))
```

Using the rewrite rules is, as we have seen in chapter 6, a little more complicated. The Substitution menu is

```
MENU "Substitution"
  ENTRY "A↔..." IS ForwardSubst "rewrite ↔ «" "rewrite ↔ »" (↔)
  ENTRY "...↔B" IS ForwardSubst "rewrite ↔ »" "rewrite ↔ «" (↔)
  ENTRY "A=..." IS ForwardSubst "rewrite = «" "rewrite = »" (=)
  ENTRY "...=B" IS ForwardSubst "rewrite = »" "rewrite = «" (=)
END
```

The ForwardSubst tactic extends the techniques of chapter 6 to allow rewriting in forward as well as backward reasoning style. We require that the user must subformula-select some subformula and also may select a hypothesis which is to be used as $A = B$ or $A \leftrightarrow B$ in the rule. The tactic is rather subtle:[1] it's given a left-to-right rewrite rule *ruleLR*, a right-to-left rewrite rule *ruleRL*, and a pattern *pat* which it uses in error alerts. Note how the menu entries alternate the use of the rewrite rules to get the correct rewriting effect when working either forward or backwards.

```
TACTIC ForwardSubst (ruleLR, ruleRL,pat) IS
  WHEN
    (LETHYPSUBSTSEL _P
      cut
      ruleRL
      (WHEN
        (LETHYP _Q
          (ALT (WITHHYPSEL hyp)
            (FAIL (the hypothesis you formula-selected wasn't a pat formula))))
        (JAPE (SUBGOAL 1)))
      (WITHSUBSTSEL hyp))
    (LETCONCSUBSTSEL _P
      (WITHSUBSTSEL ruleLR)
      (WHEN
        (LETHYP_Q
          (ALT (WITHHYPSEL hyp)
            (FAIL(the hypothesis you formula-selected wasn't a pat formula))))
        SKIP))
    (JAPE (fail(please text-select one or more instances of a sub-formula
to replace)))
```

LETHYPSUBSTSEL *pattern tactic...* succeeds when the user's subformula-selections describe a substitution in a hypothesis (left-hand side) formula; LETCONCSUBSTSEL succeeds when they describe a substitution in a conclusion (right-hand side). Working backwards with letconcsubstsel the tactic is fairly straightforward: it applies ruleLR (one of the argument rewrite rules) on the substution formula that the user has defined, and then, if the user has selected a hypothesis, tries to unify it with the conclusion of the first antecedent of the rewrite. Working forwards it makes a cut, applies ruleRL (the other rewrite rule, which will do its work in the opposite direction to ruleLR) and then either applies the user's selected hypothesis (alt...) or skips the first antecedent (jape(subgoal 1)) and then does WITHSUBSTSEL hyp, which uses the user's original subformula-selection to construct a substitution in the current problem sequent, and also does an automatic WITHHYPSEL on it, so that the hyp is bound to make use of that hypothesisfootnoteIt seems reasonable that withsubstsel should include an automatic withhyp/withconcsel, because if the newly-constructed hypothesis isn't to be used, why was it constructed? The automatic WITHHYPSEL enables us, as in this example, to

[1] You might wonder whether the complexity of this tactic programming doesn't undermine the claim that Jape is simple and easy to program. Well, yes it does. But I now realise that while encoding the rules of a logic in Jape and arranging them in menus is straightforward and transparent, the work required to hide parts of proofs or to achieve concise effects by hiding gestures is programming, and programming is always potentially intricate.

distinguish between two selected hypotheses: the one selected for application as an equality, and the one subformula-selected for rewriting.[2]

7.2 Syntax of set operations

Apart from the various operators, which have been encoded in the obvious way, the only important syntactic feature of this encoding is the treatment of *set abstractions*. Jape's parser-generator isn't very sophisticated, so I had to make some drastic simplifications.

The form of a set abstraction, in this encoding, is $\{variable \vee formula\}$, and the occurrence of the variable to the left of the bar is a binding occurrence; I also allow $\{<variable, variable> \vee formula\}$. I include, therefore, in `set_syntax.j`

```
CLASS VARIABLE u v w
CONSTANT Ø ⊥ U EQ

PREFIX 1000 Pow
PREFIX 800 ∪∪ ∩∩
POSTFIX 800 -1
INFIX 700L ∪ ∩ -
INFIX 720L •
INFIX 740L ×
INFIX 600L ⊆
INFIX 500L ∈ ∉

OUTFIX < >
OUTFIX
OUTFIX  |

BIND y SCOPE P IN  y | P
BIND x y SCOPE P IN  <x,y> | P
```

The priority numbers chosen are higher than the priority of any operator in `BnE-Fprime_syntax.j`, and otherwise have no particular significance.

Given the OUTFIX and BIND directives above, together with the standard interpretation of comma as a zero-priority associative operator, the encoding recognises the following formula-shapes:

$\{\}$ the empty set;

$\{formula\}$ a singleton set;

$\{formula, \ldots, formula\}$ a literal description of a set;

$\{variable \mid formula\}$ a set abstraction;

$\{< variable, variable >\mid formula\}$ an abstraction of a set of pairs.

Allowing set brackets with or without the vertical bar is a trick of which I was once ashamed.

[2] The encoding uses the out-of-date terminology 'text selection', because it was written before subformula selection was implemented, when all you could do was select character sequences.

7.3 The axiomatic presentation of naive set theory

We first observe, just to get it out of the way, that this encoding of set theory does not attempt to avoid Russell's paradox (see `sets_russell.j`). Schmidt's treatment was based on Gödel-Bernays set theory and had a judgement "Set A", which I didn't duplicate, principally because my client didn't want me to.

The axioms of comprehension and extension in this naive treatment are

comprehension $\forall P.\exists A.x \in A \leftrightarrow P(x)$;

extension $\forall A, B.\, A = B \leftrightarrow (\forall x.x \in A \leftrightarrow x \in B)$

Comprehension isn't first order, but it's encoded in rules which are implicitly-quantified schemata so that's ok. I couldn't find a way to incorporate comprehension as a single rule, because of the existence operator, and so I followed Schmidt and incorporated it as two rules for each of the set-abstraction forms:

$\dfrac{\Gamma \vdash P(A)}{\Gamma \vdash A \in \{y \mid P(y)\}}$	$\dfrac{\Gamma \vdash A \in \{y \mid P(y)\}}{\Gamma \vdash P(A)}$
$\dfrac{\Gamma \vdash P(A, B)}{\Gamma \vdash <A, B> \in \{<y, z> \mid P(y, z)\}}$	$\dfrac{\Gamma \vdash <A, B> \in \{<y, z> \mid P(y, z)\}}{\Gamma \vdash P(A, B)}$

The rules are encoded as a couple of ALTs

```
RULES "abstraction-I"(A, OBJECT y,OBJECT z) ARE
    FROM P(A) INFER A∈ y | P(y)
  AND FROM P(A,B) INFER <A,B>∈ <y,z> | P(y,z)
END
RULES "abstraction-E"(A, OBJECT y, OBJECT z) ARE
    FROM A∈ y | P(y)  INFER P(A)
  AND FROM <A,B>∈ <y,z> | P(y,z)  INFER P(A,B)
END
```

(`BnE-Fprime.jt` sets `interpretpredicates` true, so $P(A)$ is really shorthand for a substitution). and are incorporated into the SetOps menu in the usual way

```
ENTRY "abstraction-I" IS FSSOB ForwardCutwithSubstSel "abstraction-I"
ENTRY "abstraction-E" IS FOBSS ForwardCut "abstraction-E"
```

The FOBSS and FSSOB tactics are each a variation of the FOB tactic, requiring that the user makes a text selection when reasoning backward (FOBSS) or forward (FSSOB):

```
TACTIC FOBSS (Forward, Rule) IS
  WHEN  (LETHYP _P
      (ALT  (Forward Rule)
        (WHEN  (LETARGSEL _Q
            (JAPE(failgivingreason(Rule is not applicable to assumption ’
_P ’
                    with argument ’ _Q ’))))
          (JAPE(failgivingreason(Rule is not applicable to assumption ’
_P ’))))))
    (LETCONCSUBSTSEL _P
      (ALT  (WITHSUBSTSEL (WITHHYPSELRule))
        (LETGOAL _Q
          (JAPE(failgivingreason(Rule is not applicable to conclusion ’
_Q ’
                with substitution ’ _P ’))))))
    (ALT  (WITHSELECTIONS Rule)
      (JAPE(failgivingreason(Rule is not applicable to that conclusion))))

TACTIC FSSOB (Forward, Rule) IS
  WHEN  (LETHYPSUBSTSEL _P (Forward Rule))
    (ALT  (WITHSELECTIONS Rule)
      (WHEN  (LETARGSEL _P
          (JAPE(failgivingreason(Rule is not applicable with argument ’
_P ’))))
        (JAPE(failgivingreason(Rule is not applicable)))))

TACTIC ForwardCutwithSubstSel(Rule) IS
  SEQ  cut
    (WHEN  (LETSUBSTSEL _A Rule (WITHSUBSTSEL hyp))
        (JAPE (fail(please text-select one or more instances of a sub-formula))))
```

I incorporate extension as an axiomatic definition. I don’t include the outer quantification, as rules are implicitly-quantified schemata. It’s encoded in the Definitions panel (see below) as

```
RULE (OBJECT y) IS A=B ≜ (∀y.y∈A↔y∈B)
```

When we use this rule we will normally do so with a rewrite: replace some subformula which matches one side or other of the definition, closing the first antecedent of the rewrite with an instance of the definition. But we don’t want to see the particular instance of the axiom as part of the proof: just as in the functional programming example, it is best referred to in the justification of the rewrite step, and otherwise hidden from view.

I include the rule as part of a Definitions panel, and have two buttons on the panel which allow left-to-right and right-to-left rewriting. As with the Substitution menu, switching the rewrite rules around in the tactics associated with each button allows forward or backward rewriting, The panel definition is in `BnE-Fprime_menus.j`:

```
TACTICPANEL "Definitions" IS
  RULE INFER A≠B ≜ ¬(A=B)

  BUTTON "A≜..." IS apply ForwardSubstHiding "rewrite ≜ «" "rewrite ≜
»" COMMAND
  BUTTON "...≜B" IS apply ForwardSubstHiding "rewrite ≜ »" "rewrite ≜
«" COMMAND
END
```

The tactic ForwardSubstHiding allows the user to rewrite

- either a hypothesis or a conclusion;
- after subformula-selecting a number of instances of a subformula, just those instances;
- without subformula-selecting, the whole hypothesis or conclusion.

In fact it is only forward rewriting without text selection that is more subtle than what we have already seen.

```
TACTIC ForwardSubstHiding (ruleLR, ruleRL, thm) IS
  WHEN
    (LETHYPSUBSTSEL _P cut (LAYOUT thm (1) ruleRL thm (WITHSUBSTSEL hyp)))
    (LETCONCSUBSTSEL _P (LAYOUT thm (1) (WITHSUBSTSEL ruleLR) thm))
    /* the "second argument" of the rewrite rules has to be B */
    (LETHYP _P cut (LAYOUT thm (1) (ruleRL[B\_P]) thm (WITHHYPSEL hyp)))
    (LETGOAL _P (LAYOUT thm (1) (ruleLR _P) thm))
```

The first alternative is activated when the user has subformula-selected in a hypothesis: it cuts, uses one of the rewrite rules, closes the first antecedent with the theorem, and the second using the subformula-selection that the user made. The second alternative is activated when there is a subformula-selection in a conclusion: it uses the other rewrite rule followed by the theorem. The last alternative is activated when there is no recognisable subformula-selection and no hypothesis selection: it activates the same rewrite rule as the second alternative, but gives it the whole conclusion formula instead of the user's text selection: that is a particularly easy 'abstraction' for the substitution-unifier to resolve, and the effect is to unify the whole consequent with the left- or the right-hand side of the theorem, depending on the particular rewrite rule that is used.

The third alternative is the tricky one. It calls the same rewrite rule as the first alternative, but gives it the selected hypothesis as argument B. The rule has to be something like

```
RULE "rewrite ≜ «" (A) IS FROM A≜B AND P(B) INFER P(A)
```

But the instantiation says nothing about A, so that rule will necessarily succeed by deferred unification of $\_P'[v\backslash\_A]$ with the conclusion. Then it closes the first antecedent of the rewrite with the theorem which has to match $A \triangleq B$ and therefore tells it what $\_A$ is: but we still don't know $\_P'$. After the theorem is applied, the second antecedent of the rewrite step will be $\_P'[v\backslash B]$: using hyp we unify that with $\_P$, which is B, and the effect is just like the fourth alternative (and surely I ought to have written the fourth alternative as

```
(LETGOAL _P (LAYOUT thm (1) (ruleLR[A\_P]) thm))
```

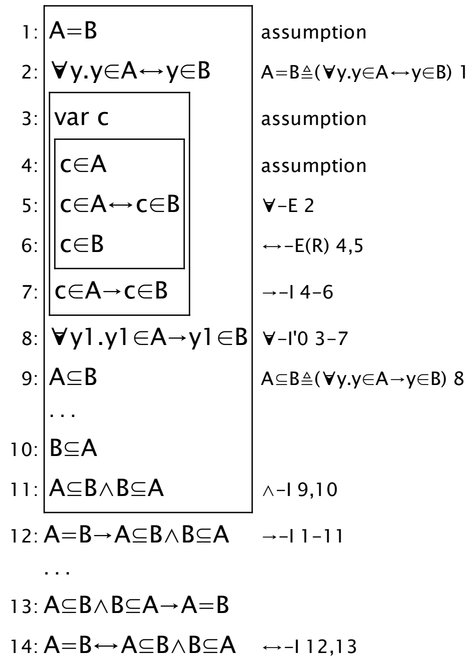
— but it was composed in the days before you could write that kind of rule application, and I didn't think to update it).

Each of the entries in the Definitions panel is intended to be used as a two-way rewrite rule, using the buttons as described above. Note that I've had to include definitions of negated operators like $\neq$ and $\notin$ in terms of negated formulae.[3]. The set additions are defined in `set_menus.j`; the OBJECT qualifiers are for convenience, so that you don't get too many unknowns in your proof:[4]

```
TACTICPANEL "Definitions" IS
  RULE IS A∉B ≜ ¬(A∈B)
  RULE IS Ø ≜ {}
  RULE (OBJECT x) IS EQ ≜ {x|x=x}
  RULE (OBJECT x) IS {A} ≜ {x|x=A}
  RULE (OBJECT x) IS {A,B} ≜ {x|x=A∨x=B}
```

[3] I always wanted to do this sort of thing by 'definitional equality', where you write $A\neg = B$ and Jape interprets it as $\neg(A = B)$ but still displays what you wrote. I never managed it.

[4] Nowadays Jape can use alpha-conversion of binding forms when unifying, so the fresh variables don't really get in the way.



```
 1: A=B                     assumption
 2: ∀y.y∈A↔y∈B              A=B≜(∀y.y∈A↔y∈B) 1
 3: var c                   assumption
 4: c∈A                     assumption
 5: c∈A↔c∈B                 ∀-E 2
 6: c∈B                     ↔-E(R) 4,5
 7: c∈A→c∈B                 →-I 4-6
 8: ∀y1.y1∈A→y1∈B           ∀-I'0 3-7
 9: A⊆B                     A⊆B≜(∀y.y∈A→y∈B) 8
    ...
10: B⊆A
11: A⊆B∧B⊆A                 ∧-I 9,10
12: A=B→A⊆B∧B⊆A             →-I 1-11
    ...
13: A⊆B∧B⊆A→A=B
14: A=B↔A⊆B∧B⊆A             ↔-I 12,13
```

Figure 7.1: Beginning of a proof using logical rules and axiomatic definitions

```
    RULE (OBJECT x) IS {A,B,C} ≜ {x|x=A∨x=B∨x=C}
    RULE (OBJECT x) IS {A,B,C,D} ≜ {x|x=A∨x=B∨x=C∨x=D}
    RULE (OBJECT y) IS A⊆B ≜ (∀y.y∈A→y∈B)
    RULE (OBJECT y) IS A=B ≜ (∀y.y∈A↔y∈B)
    RULE (OBJECT y) IS A∪B ≜ { y | y∈A∨y∈B }
    RULE (OBJECT y) IS A∩B ≜ { y | y∈A∧y∈B }
    RULE (OBJECT y) IS A-B ≜ { y | y∈A∧y∉B }
    RULE (OBJECT y) IS A⁻¹ ≜ {y | y∉A}
    RULE (OBJECT x, OBJECT y) IS ∪∪(C) ≜ { x | ∃y.  x∈y∧y∈C }
    RULE (OBJECT x, OBJECT y) IS ∩∩(C) ≜ { x | ∀y.  y∈C→x∈y }
    RULE (OBJECT x) IS Pow(A) ≜ { x | x⊆A }
    RULE (OBJECT x, OBJECT y) IS A×B ≜ { <x,y> | x∈A∧y∈B }
    RULE (OBJECT x, OBJECT y, OBJECT z) IS A•B ≜ { <x,z> | ∃y.<x,y>∈A∧<y,z>∈B
  }
  END
```

7.4 The non-axiomatic rules

A proof using the axioms will typically introduce and then eliminate logical connectives. Figure 7.1 shows the beginning of such a proof. It is clear that there will be lots of repetitive applications of ∀-E, ∀-I, →-E, →-I, and similar logical rules during this proof. It is clear that there could be introduction and elimination rules for each of the set operators. These are the ones relevant to the proof above:

$\dfrac{\Gamma, c \in A \vdash c \in B}{\Gamma \vdash A \subseteq B}$ (FRESH c) $\subseteq -I$	$\dfrac{\Gamma \vdash C \in A \quad \Gamma \vdash A \subseteq B}{\Gamma \vdash C \subseteq B}$ $\subseteq -E$	
$\dfrac{\Gamma \vdash A \subseteq B \quad \Gamma \vdash B \subseteq A}{\Gamma \vdash A = B}$ $= -I$	$\dfrac{\Gamma \vdash A = B}{\Gamma \vdash A \subseteq B}$ $= -E(L)$	$\dfrac{\Gamma \vdash A = B}{\Gamma \vdash B \subseteq A}$ $= -E(R)$

Figure 7.2 shows the proof completed using these rules, rather than the axiomatic definitions. Somewhat simpler! The rules are encoded in the obvious way in `set_menus.j`. They are described as DERIVED RULE, because they can be proved from the axioms, and in figure 7.2 as conjectured rules, because no proof of them had been provided when that proof was made.

```
 1: | A=B        |       assumption
 2: | A⊆B        |       Conjectured Rule =-E(L) 1
 3: | B⊆A        |       Conjectured Rule =-E(R) 1
 4: | A⊆B∧B⊆A    |       ∧-I 2,3
 5: A=B→A⊆B∧B⊆A          →-I 1-4
 6: | A⊆B∧B⊆A    |       assumption
 7: | A⊆B        |       ∧-E(L) 6
 8: | B⊆A        |       ∧-E(R) 6
 9: | A=B        |       Conjectured Rule =-I 7,8
10: A⊆B∧B⊆A→A=B          →-I 6-9
11: A=B↔A⊆B∧B⊆A          ↔-I 5,10
```

Figure 7.2: A proof using derived rules

Chapter 8

Encoding the Hindley-Milner type-assignment algorithm

Because Jape is based on unification, it's possible to construct a very nice version of the Hindley-Milner type-assignment algorithm, and control it in slow motion. I implemented a version of the algorithm for the lambda calculus with tuples and *let* / textitletrec bindings.

$\dfrac{C, x : T \vdash E : T'}{C \vdash \lambda x.E : T \to T'}\ \lambda - I$	$\dfrac{C \vdash F : T \to T' \quad C \vdash G : T}{C \vdash F\, G : T'}\ application - I$
$\dfrac{C \vdash E1 : T1 \quad C \vdash E2 : T2}{C \vdash (E1, E2) : (T1 \times T2)}\ tuple - I$	$\dfrac{C \vdash E : T1 \quad C \vdash T1 \prec S \quad C, x : S \vdash F : T}{C \vdash \text{let } x = E \text{ in } F \text{ end} : T}\ \text{let} -I$
$\dfrac{C, x : T1 \vdash E : T1 \quad C \vdash T1 \prec S \quad C, x : S \vdash F : T}{C \vdash \text{letrec } x = E \text{ in } F \text{ end} : T}\ \text{letrec} -I$	
$\dfrac{C(x) \mapsto S \quad S \succ T}{C \vdash x : T}\ identifier\ type$	

In each of these rules the context C is a sequence of bindings of program variables to type schemes which can be read, right to left, as a mapping from variables to type schemes. The judgement $C(x) \mapsto S$ interprets the context in just that way. The judgement $C \vdash T \prec S$ is the *generalisation step*, in which 'type variables' free in the type T but not free in the context C are used to transform type T into type scheme S. The judgement $S \succ T$ is the corresponding *specialisation step*, when the schematic variables of S are replaced by type formulae.

The difficulties of encoding the Hindley-Milner algorithm are just those of representing the schematic 'type variables', representing and interpreting the type context and implementing the generalisation and specialisation steps.

8.1 Syntax

I represent λ formulae using LEFTFIX, and I give that formula a lower priority than the colon operator, so that I don't have to use too many brackets. The type-tupling operator $\times$ is treated as an associative operator, rather like comma. I need an == operator (blechh!) because = is used in the *let / letrec* syntax. I use a double-arrow operator rather than a colon in the contexts, for no particularly good reason that I can remember. I have included additional operators • and ◃ which are used in the generalisation-step induction.

Type schemes which include schematic variables– so-called polytypes — are $\forall t.T$ or $\forall t1, t2.T$ and so on, with up to four schematic variables. Those which have no schematic variables — so-called monotypes — are #T, where T is a type formula. This is faithful to Milner's treatment in the ML description, where he describes the scheme T as a shorthand for $\forall().T$.

I have included constants *hd*, *tl* and *nil* which are useful in describing list-processing, true and false which are useful in handling booleans; there are also constant type-names *bool*, *string* and *num*.

Jape has no means of dividing names into distinct syntactic hierarchies: to do this algorithm properly I'd want to distinguish type variables t from program variables x. But I can't. To keep my head straight I make two sets of declarations. First the program names:

```
CLASS VARIABLE x y z e f g map
CLASS FORMULA E F G
CLASS CONSTANT c
CONSTANT hd tl nil
CLASS NUMBER n
CLASS STRING s
CONSTANT true false
```

and then the type names:

```
CLASS VARIABLE t
CLASS FORMULA S T /* we use T for types, S for type schemes in the rules
which follow */
CONSTANT bool string num
```

Next, operators for programs:

```
SUBSTFIX  500  { E / x }
JUXTFIX  400
INFIXC  140L  + -
INFIXC  120R  ::
INFIXC  100L  == /* we need this because we also have let f =...  */
LEFTFIX  75  λ .
INFIX  50L  =

OUTFIX [ ]
OUTFIX letrec in end
OUTFIX let in end
OUTFIX if then else fi
```

and operators for types:

```
INFIX  150T   ×
INFIX  100R   →
LEFTFIX  75  ∀.
PREFIX  75  #
INFIX  55L   • ◃
INFIX  50L   :  ⇒ ≺ ≻
```

Now bindings:

```
BIND x SCOPE E IN λ x.  E
BIND t SCOPE T IN ∀ t.  T
BIND t1 t2 SCOPE T IN ∀ t1, t2.  T
BIND t1 t2 t3 SCOPE T IN ∀ t1, t2, t3.  T
BIND t1 t2 t3 t4 SCOPE T IN ∀ t1, t2, t3, t4.  T

BIND x   SCOPE F  IN let x = E in F end
BIND x1 x2   SCOPE F  IN let x1=E1, x2=E2 in F end
BIND x1 x2 x3  SCOPE F  IN let x1=E1, x2=E2, x3=E3 in F end
BIND x1 x2 x3 x4  SCOPE F  IN let x1=E1, x2=E2, x3=E3, x4=E4 in F end
BIND x  SCOPE E F  IN letrec x = E in F end
BIND x1 x2  SCOPE E1 E2 F  IN letrec x1=E1, x2=E2 in F end
BIND x1 x2 x3  SCOPE E1 E2 E3 F  IN letrec x1=E1, x2=E2, x3=E3 in F end
BIND x1 x2 x3 x4  SCOPE E1 E2 E3 E4 F  IN letrec x1=E1, x2=E2, x3=E3, x4=E4
in F end
```

Finally, the definition of a judgement:

```
CLASS LIST C
SEQUENT IS LIST ⊢ FORMULA
```

— notice LIST and not BAG as in previous chapters.

8.2 Rules

The structural rules are very straightforwardly encoded, following the algorithm directly. Note the use of a type scheme *#T1* in the rule which deals with λ formulae.

```
RULE "F G : T"  FROM C ⊢ F: T1→T2 AND C ⊢ G : T1   INFER C ⊢ F G : T2
RULE "λx.E : T1→T2"  FROM C,x⇒#T1 ⊢ E:T2   INFER C ⊢ λx.E : T1→T2
RULE "(E,F) : T1×T2"  FROM C ⊢ E: T1 AND C ⊢ F: T2  INFER C ⊢ (E,F) : T1×T2
RULE "if E then ET else EF fi :  T"
  FROM C ⊢ E : bool AND C ⊢ ET : T AND C ⊢ EF : T  INFER C ⊢ if E then
ET else EF fi :  T
```

There are some simple rules which deal with constants:

```
RULE "n:num"  INFER C ⊢ n:num
RULE "s:string"  INFER C ⊢ s:string
RULE "true:bool"  INFER C ⊢ true:bool
RULE "false:bool"  INFER C ⊢ false:bool
```

which I apply whenever possible — in this case autounify seems to be the best mechanism:[1]

```
AUTOUNIFY "n:num" "s:string" "true:bool" "false:bool"
```

Dealing with the various forms of *let* and *letrec* formulae is a matter of tedious listing. Here are the *letrec* rules, for example:

[1] It would be dangerous if a conclusion _E:_T was ever generated, but it never is.

```
RULES letrecrules ARE
  FROM C,x⇒#T1 ⊢ E:T1 AND C ⊢ T1≺S1 AND C,x⇒S1 ⊢ F:T
    INFER C ⊢ letrec x=E in F end :  T
AND  FROM C,x1⇒#T1,x2⇒#T2 ⊢ E1 :  T1 AND C,x1⇒#T1,x2⇒#T2 ⊢ E2 :  T2
  AND C ⊢ T1≺S1 AND C ⊢ T2≺S2 AND C,x1⇒S1,x2⇒S2 ⊢ F:T
    INFER C ⊢ letrec x1=E1, x2=E2 in F end :  T
AND  FROM C,x1⇒#T1,x2⇒#T2,x3⇒#T3 ⊢ E1 :  T1 AND C,x1⇒#T1,x2⇒#T2,x3⇒#T3
⊢ E2 :  T2
  AND C,x1⇒#T1,x2⇒#T2,x3⇒#T3 ⊢ E3 :  T3 AND C ⊢ T1≺S1 AND C ⊢ T2≺S2
  AND C ⊢ T3≺S3 AND C,x1⇒S1,x2⇒S2,x3⇒S3 ⊢ F:T
    INFER C ⊢ letrec x1=E1, x2=E2, x3=E3 in F end :  T
AND  FROM C,x1⇒#T1,x2⇒#T2,x3⇒#T3,x4⇒#T4 ⊢ E1 :  T1
  AND C,x1⇒#T1,x2⇒#T2,x3⇒#T3,x4⇒#T4 ⊢ E2 :  T2
  AND C,x1⇒#T1,x2⇒#T2,x3⇒#T3,x4⇒#T4 ⊢ E3 :  T3
  AND C,x1⇒#T1,x2⇒#T2,x3⇒#T3,x4⇒#T4 ⊢ E4 :  T4
  AND C ⊢ T1≺S1 AND C ⊢ T2≺S2 AND C ⊢ T3≺S3 AND C ⊢ T4≺S4
  AND C,x1⇒S1,x2⇒S2,x3⇒S3,x4⇒S4 ⊢ F:T
    INFER C ⊢ letrec x1=E1, x2=E2, x3=E3, x4=E4 in F end :  T
END
```

8.3 Reading the context and specialising a type scheme

Things get more interesting when you consider how to handle the context-evaluation step $C(x) \mapsto S$: C maps variable name x to scheme S. The context is just a list of name↦scheme bindings, and it should be read right-to-left, so that the most recent bindings take precedence. Because program names can't appear in types in this logic, I can use a notin proviso to help to read the context in this way. Because variables and constants are different syntactic classes, I need two rules:

```
RULE "C ⊢ x⇒S" WHERE x NOTIN C′ IS INFER C,x⇒S,C′ ⊢ x⇒S
RULE "C ⊢ c⇒S" WHERE c NOTIN C′ IS INFER C,c⇒S,C′ ⊢ c⇒S
```

I declare these as identity rules so that their application is hidden in a box-and-line display of a proof:

```
IDENTITY "C ⊢ x⇒S"
IDENTITY "C ⊢ c⇒S"
```

There is a rule for each of the constant function identifiers, binding it to a polytype or a monotype as appropriate:

```
RULES constants ARE
  C ⊢ hd⇒∀tt.[tt]→tt
AND  C ⊢ tl⇒∀tt.[tt]→[tt]
AND  C ⊢ (::)⇒∀tt.tt→[tt]→[tt]
AND  C ⊢ nil⇒∀tt.[tt]
AND  C ⊢ (+)⇒#num→num→num
AND  C ⊢ (-)⇒#num→num→num
AND  C ⊢ (==)⇒∀tt.tt→tt→bool
END
```

Typing a variable or a constant is a matter of finding the type scheme and then specialising to some type. Specialisation is just a matter of substituting types for schematic variables:

```
RULES "S≻T" ARE
  INFER #T ≻ T
AND  INFER ∀tt.TT ≻ TT{T1/tt}
AND  INFER ∀tt1,tt2.TT ≻ TT{T1,T2/tt1,tt2}
AND  INFER ∀tt1,tt2,tt3.TT ≻ TT{T1,T2,T3/tt1,tt2,tt3}
AND  INFER ∀tt1,tt2,tt3,tt4.TT ≻ TT{T1,T2,T3,T4/tt1,tt2,tt3,tt4}
END
```

Then two rules put these together in just the way that the algorithm does:

```
RULE "C ⊢ x:T" IS FROM C⊢x⇒S AND S≻T INFER C⊢x:T
RULE "C ⊢ c:T" IS FROM C⊢c⇒S AND S≻T INFER C⊢c:T
```

In the menu I use a tactic which looks in three places for a type scheme and then specialises, showing none of its working when it succeeds, but trying to give some error messages when it fails:

```
TACTIC "x:T" IS   SEQ
    (ALT (LAYOUT "C(x)⇒S; S≻T" () "C ⊢ x:T" "C ⊢ x⇒S")
      (LAYOUT "C(c)⇒S; S≻T" () "C ⊢ c:T" "C ⊢ c⇒S")
      (LAYOUT "constant" () "C ⊢ c:T" constants)
      (WHEN
        (LETGOAL (_E:_T)
          (Fail (x:T can only be applied to either variables or constants:
you chose _E)))
        (LETGOAL _E (Fail (conclusion _E is not a ′ formula:type ′ judgement)))))
    "S≻T"
```

8.4 The generalisation step

I perform a structural induction on the type T in order to calculate its schematic variables. These will be unknowns, because of course I don't judiciously introduce type variables when running the algorithm (though you can if you want to): I simply introduce unknowns as necessary.

The generalisation step is run by a tactic, and all the working is normally hidden from the user. It works with a formula *type* • *scheme*$_{in}$ ◃ *scheme*$_{out}$, in which the operators • and ◃ are no more than punctuation. The starting rule is

```
RULE "T≺S" IS  FROM C ⊢ T • #T ◃ S   INFER C ⊢ T ≺ S
```

The induction works with rules which take a type apart, and two rules which are the base case. The structural rules are

```
RULE "T1→T2•..."  FROM C ⊢ T1• Sin ◃ Smid AND C ⊢ T2 • Smid ◃ Sout
    INFER C ⊢ T1→T2 • Sin ◃ Sout
RULE "T1×T2•..."  FROM C ⊢ T1• Sin ◃ Smid AND C ⊢ T2 • Smid ◃ Sout
    INFER C ⊢ T1×T2 • Sin ◃ Sout
RULE "[T]•..."  FROM C ⊢ T • Sin ◃ Sout
    INFER C ⊢ [T] • Sin ◃ Sout
```

The tactic applies these rules, we shall see, 'by matching': they aren't allowed to make any substantial unifications which alter the problem sequent to which they are applied. So if the problem sequent is *unknown* • *scheme*$_{in}$ ◃ *scheme*$_{out}$, none of the rules will be used.

The rules which deal with an unknown do so by unifying it with a freshly-minted variable name and making sure that it doesn't appear in the context or the original type:

```
RULES "new t•..." (OBJECT t1) WHERE t1 NOTIN C ARE
  C⊢ t1 • #T◁ ∀t1.T
AND  C⊢ t1 • ∀tt1.T ◁ ∀tt1,t1.T
AND  C⊢ t1 • ∀tt1,tt2.T ◁ ∀tt1,tt2,t1.T
AND  C⊢ t1 • ∀tt1,tt2,tt3.T ◁ ∀tt1,tt2,tt3,t1.T
END
```

The only formula which can possibly unify with a freshly-minted type variable is a type unknown, and these rules have a proviso that the result shouldn't be free in the context C. The effect is to replace an unknown type by a type variable, and by unification to include it in the context.

If none of these rules applies, then we must have an unknown which *does* appear in the context: that unknown must be left alone:

```
RULE "same T•..."  INFER C ⊢ T • S ◁ S
```

The whole is stitched together with a tactic which tries first the structural rules by matching, then the variable rule and finally the leave-alone rule; that tactic is used by another which starts the process, calls the induction and hides all its working:

```
TACTIC geninduct IS
  ALT  (SEQ (MATCH (ALT "T1→T2•..." "T1×T2•...")) geninduct geninduct)
    (SEQ (MATCH "[T]•...") geninduct)
    "new t•..."
    "same T•..."

TACTIC generalise IS LAYOUT "generalise" () "T≺S" geninduct
```

We also provide a 'single-step' tactic which carries out the same tasks, so that users can view the process as it evolves:

```
TACTIC genstep IS
  ALT  "T≺S"
    (MATCH "T1→T2•...")
    (MATCH "T1×T2•...")
    (MATCH "[T]•...")
    "new t•..."
    "same T•..."
```

8.5 Automatic search

In this chapter we are dealing with an encoding of an *algorithm*, not a proper logic. It's possible to get strange answers by running the steps in the wrong order. On the other hand, it's easy to write a tactic which automatically runs the algorithm. That tactic is long-winded because it has to deal, case-by-case, with the various sizes of binding structures. If only Jape could handle families of rules...

```
TACTIC Auto IS
  WHEN  (LETGOAL (_x:_T) "x:T")
    (LETGOAL (_c:_T)
      (ALT  "x:T" "n:num" "s:string" "true:bool" "false:bool"
        (JAPE (fail (_c isn't a constant from the context,
            or one of the fixed constants)))
      )
    )
    (LETGOAL (_F _G:_T) "F G : T" Auto Auto)
    (LETGOAL ((_E,_F):_T) "(E,F) : T1×T2" Auto Auto)
    (LETGOAL ((λ_x._E):_T) "λx.E : T1→T2" Auto)
    (LETGOAL (if _E then _ET else _EF fi:_T) "if E then ET else EF fi :
T" Auto Auto Auto)
    (LETGOAL (let _x=_E in _F end:_T)
        letrules Auto generalise Auto)
    (LETGOAL (let _x1=_E1, _x2=_E2 in _F end:_T)
        letrules Auto Auto generalise generalise Auto)
    ...  etc...
    (LETGOAL (letrec _x=_E in _F end:_T)
        letrecrules Auto generalise Auto)
    (LETGOAL (letrec _x1=_E1, _x2=_E2 in _F end:_T)
        letrecrules Auto Auto generalise generalise Auto)
    ...  etc...
    (LETGOAL (_E:_T) (JAPE (fail (_E is not a recognisable program formula
(Auto)))))
    (LETGOAL _E (JAPE (fail (_E is not a recognisable judgement (Auto)))))
```

There's a similar AutoStep tactic which lets the user make just one step of the algorithm.

8.6 An example

The algorithm will calculate, for example, the type of *map* and use it correctly in an application: see figure 8.1.

8.7 Could Jape treat other type-theoretic logics?

This encoding is a hack. It works for the simple case that it is applied to, but it shows some limitations of Jape. Conditions like 'C is a context' have been left out entirely, and encoding them in an interpretive tool would be pretty inefficient. Including them in rules would be tedious too. I'm aware (because Randy Pollack has several times beaten it into my head) that the 'shadowing' mechanism which allows a context to be treated as a mapping isn't generally applicable.

On a more mundane level, it points up the weaknesses of the parser-generator. I can't make a distinction between program and type names (and make it stick); I have to use an ugly operator to mark monotypes in the type environment. And the fact that it doesn't deal with 'families' of rules, so that you have to list all the different forms of $\forall$ formulae, for example, and have a rule for each form, is very tedious.

On the other hand, though the genstep mechanism is a bit messy, it does show how it's possible to run the algorithm without resorting to guessing when to use type variables and when to use unknowns. Jape really does go through the type formula, replace replaceable unknowns with type variables, generalise the result, and get the right answer.

1:	C	assumption
2:	map⇒#(t1→t2)→[t1]→[t2]	assumption
3:	f⇒#num→num	assumption
4:	f⇒#t1→t2	assumption
5:	xs⇒#[t1]	assumption
6:	(==):[t1]→[t1]→bool	constant
7:	xs:[t1]	C(x)⇒S; S≻T
8:	(==)xs:[t1]→bool	F G : T 6,7
9:	nil:[t1]	constant
10:	xs==nil:bool	F G : T 8,9
11:	nil:[t2]	constant
12:	(::):t2→[t2]→[t2]	constant
13:	f:t1→t2	C(x)⇒S; S≻T
14:	hd:[t1]→t1	constant
15:	xs:[t1]	C(x)⇒S; S≻T
16:	hd xs:t1	F G : T 14,15
17:	f(hd xs):t2	F G : T 13,16
18:	(::)(f(hd xs)):[t2]→[t2]	F G : T 12,17
19:	map:(t1→t2)→[t1]→[t2]	C(x)⇒S; S≻T
20:	f:t1→t2	C(x)⇒S; S≻T
21:	map f:[t1]→[t2]	F G : T 19,20
22:	tl:[t1]→[t1]	constant
23:	xs:[t1]	C(x)⇒S; S≻T
24:	tl xs:[t1]	F G : T 22,23
25:	map f(tl xs):[t2]	F G : T 21,24
26:	f(hd xs)::map f(tl xs):[t2]	F G : T 18,25
27:	if xs==nil then nil else f(hd xs)::map f(tl xs)fi:[t2]	if E then ET else EF fi : T 10,11,26
28:	λxs.if xs==nil then nil else f(hd xs)::map f(tl xs)fi:[t1]→[t2]	λx.E : T1→T2 5–27
29:	λf.λxs.if xs==nil then nil else f(hd xs)::map f(tl xs)fi:(t1→t2)→[t1]→[t2]	λx.E : T1→T2 4–28
30:	map⇒#(t1→t2)→[t1]→[t2]	assumption
31:	f⇒#num→num	assumption
32:	x⇒#num	assumption
33:	(+):num→num→num	constant
34:	x:num	C(x)⇒S; S≻T
35:	(+)x:num→num	F G : T 33,34
36:	x:num	C(x)⇒S; S≻T
37:	x+x:num	F G : T 35,36
38:	λx.x+x:num→num	λx.E : T1→T2 32–37
39:	(t1→t2)→[t1]→[t2]≺∀(t1,t2).(t1→t2)→[t1]→[t2]	{T≺S}
40:	num→num≺#num→num	generalise
41:	map⇒∀(t1,t2).(t1→t2)→[t1]→[t2]	assumption
42:	f⇒#num→num	assumption
43:	map:(num→num)→[num]→[num]	C(x)⇒S; S≻T
44:	f:num→num	C(x)⇒S; S≻T
45:	map f:[num]→[num]	F G : T 43,44
46:	(::):num→[num]→[num]	constant
47:	0:num	n:num
48:	(::)0:[num]→[num]	F G : T 46,47
49:	(::):num→[num]→[num]	constant
50:	1:num	n:num
51:	(::)1:[num]→[num]	F G : T 49,50
52:	(::):num→[num]→[num]	constant
53:	2:num	n:num
54:	(::)2:[num]→[num]	F G : T 52,53
55:	nil:[num]	constant
56:	2::nil:[num]	F G : T 54,55
57:	1::2::nil:[num]	F G : T 51,56
58:	0::1::2::nil:[num]	F G : T 48,57
59:	map f(0::1::2::nil):[num]	F G : T 45,58
60:	letrec map=λf.λxs.if xs==nil then nil else f(hd xs):: map f(tl xs)fi,f=λx.x+x in map f(0::1::2::nil)end:[num]	letrecrules'1 2–29,30–38,39,40,41–59

Figure 8.1: A large example

Chapter 9

Encoding BAN logic

Burroughs-Abadi-Needham (BAN) logic is a logic of authentication-protocols. It's of interest in Jape chiefly because it is a logic in which the rules don't fit into a tidy introduction / elimination structure, so it takes some ingenuity to design menus and double-clicking mechanisms to suit. Also, conjectures seem naturally to require long lists of assumptions, which makes it possible to demonstrate Jape's mechanism for folding long association lists. And its use of tuples demonstrates some new ways in which Jape can deal with families of rules.

9.1 Syntax

The syntax of the logic is very simple, although it includes a number of novel operators, most of which are in Unicode (though I never found a glyph for $|\!\sim$). I had to linearise some of the notation: for example, $A \overset{K}{\leftrightarrow} B$ (A and B share private key K) became $(A, B) \leftrightarrow K$ and $\{X\}_K$ (X encrypted by K) is now $\{X\}K$. Otherwise, I hope, I faithfully described the syntax, even if I had to guess at the syntactic hierarchy of operators.

In `examples/BAN` there are two versions of the encoding: `BAN.jt` for those who have a appropriate Unicode font available, and `BAN_ASCII.jt` for those who are prepared to put up with multi-character approximations. They are intended to be identical up to representations of the operators. The examples in this chapter are taken from the ASCII version, because even I don't have a proper Unicode font yet.

From `BAN_ASCII_syntax.jt`:

```
CLASS VARIABLE x k
CLASS FORMULA W X Y Z
CLASS CONSTANT P Q R K N T
CONSTANT A B S

SUBSTFIX 700
JUXTFIX 600
PREFIX 500 #
POSTFIX 500 $^{-1}$
INFIX 300L ⇌ ↦ ↔
INFIX 200R |∼
INFIX 150R |⇒
LEFTFIX 110 ∀ .
INFIX 100R |≡
INFIX 50L <|

OUTFIX
OUTFIX < >

BIND x SCOPE P IN ∀x .  P
SEQUENT IS BAG ⊢ FORMULA
INITIALISE autoAdditiveLeft true
```

9.2 Rules

The rules of the logic are described in [“A Logic of Authentication”, Burrows, Abadi, Needham] which is available on the Web from Martín Abadi’s home page, or in paper form as (Proceedings of the Royal Society, Series A, 426, 1871 (December 1989), 233-271). Two of the rules have a ‘from R’ side-condition which I didn’t understand so haven’t implemented. The rules are given natural-deduction style, without mentioning a context of hypotheses, but they don’t have an introduction / elimination structure so far as I can see:

$\dfrac{P \mid\equiv Q \overset{K}{\leftrightarrow} P \quad P \triangleleft \{X\}_K}{P \mid\equiv Q \mid\sim X}$	$\dfrac{P \mid\equiv \overset{K}{\mapsto} Q \quad P \triangleleft \{X\}_{K^{-1}}}{P \mid\equiv Q \mid\sim X}$	$\dfrac{P \mid\equiv Q \overset{Y}{\rightleftharpoons} P \quad P \triangleleft \langle X\rangle_Y}{P \mid\equiv Q \mid\sim X}$	
$\dfrac{P \mid\equiv \#X \quad P \mid\equiv Q \mid\sim X}{P \mid\equiv Q \mid\equiv X}$		$\dfrac{P \mid\equiv Q \mid\Rightarrow X \quad P \mid\equiv Q \mid\equiv X}{P \mid\equiv X}$	
$\dfrac{P \mid\equiv X \quad P \mid\equiv Y}{P \mid\equiv (X,Y)}$	$\dfrac{P \mid\equiv (X,Y)}{P \mid\equiv X}$	$\dfrac{P \mid\equiv Q \mid\equiv (X,Y)}{P \mid\equiv Q \mid\equiv X}$	$\dfrac{P \mid\equiv Q \mid\sim (X,Y)}{P \mid\equiv Q \mid\sim X}$
$\dfrac{P \triangleleft (X,Y)}{P \triangleleft X}$	$\dfrac{P \triangleleft \langle X\rangle_Y}{P \triangleleft X}$		
$\dfrac{P \triangleleft Q \overset{K}{\leftrightarrow} P \quad P \triangleleft \{X\}_K}{P \triangleleft X}$	$\dfrac{P \mid\equiv \overset{K}{\mapsto} P \quad P \triangleleft \{X\}_K}{P \triangleleft X}$	$\dfrac{P \mid\equiv \overset{K}{\mapsto} Q \quad P \triangleleft \{X\}_{K^{-1}}}{P \triangleleft X}$	
$\dfrac{P \mid\equiv \#X}{P \mid\equiv \#(X,Y)}$			
$\dfrac{P \mid\equiv R \overset{K}{\leftrightarrow} R'}{P \mid\equiv R' \overset{K}{\leftrightarrow} R}$	$\dfrac{P \mid\equiv Q \mid\equiv R \overset{K}{\leftrightarrow} R'}{P \mid\equiv Q \mid\equiv R' \overset{K}{\leftrightarrow} R}$	$\dfrac{P \mid\equiv R \overset{X}{\rightleftharpoons} R'}{P \mid\equiv R' \overset{X}{\rightleftharpoons} R}$	$\dfrac{P \mid\equiv Q \mid\equiv R \overset{X}{\rightleftharpoons} R'}{P \mid\equiv Q \mid\equiv R' \overset{X}{\rightleftharpoons} R}$
$\dfrac{P \mid\equiv \forall v_1, \ldots, v_n.(Q \mid\Rightarrow X)}{P \mid\equiv Q \mid\Rightarrow X\,[v_1 \ldots v_n \backslash Y_1 \ldots Y_n]}$			

The rules which don’t deal with tuples are straightforwardly encoded:

```
RULE "P|≡(Q,P)↔K, P<|{X}K ⇒ P|≡Q|~X" IS
  FROM P|≡(Q,P)↔K AND P<|{X}K INFER P|≡Q|~X
RULE "P|≡Q↦K, P<|{X}K^-1 ⇒ P|≡Q|~X" IS
  FROM P|≡Q↦K AND P<|{X}K^-1 INFER P|≡Q|~X
```

```
RULE "P|≡(P,Q)⇌Y, P<|<X>Y ⇒ P|≡Q|∼X" IS
  FROM P|≡(P,Q)⇌Y AND P<|<X>Y INFER P|≡Q|∼X
RULE "P|≡#X, P|≡Q|∼X ⇒ P|≡Q|≡X" IS
  FROM P|≡#X AND P|≡Q|∼X INFER P|≡Q|≡X
RULE "P|≡Q|⇒X, P|≡Q|≡X ⇒ P|≡X" IS
  FROM P|≡Q|⇒X AND P|≡Q|≡X INFER P|≡X

RULE "P<|<X>Y ⇒ P<|X" IS
  FROM P<|<X>Y INFER P<|X
RULE "P|≡(P,Q)↔K, P<|{X}K ⇒ P<|X" IS
  FROM P|≡(P,Q)↔K AND P<|{X}K INFER P<|X
RULE "P|≡P↦K, P<|{X}K ⇒ P<|X" IS
  FROM P|≡P↦K AND P<|{X}K INFER P<|X
RULE "P|≡Q↦ K, P<|{X}K-1 ⇒ P<|X" IS
  FROM P|≡Q↦ K AND P<|{X}K-1 INFER P<|X

RULE "P|≡(R,R′)↔K ⇒ P|≡(R′,R)↔K" IS
  FROM P|≡(R,R′)↔K INFER P|≡(R′,R)↔K
RULE "P|≡Q|≡(R,R′)↔K ⇒ P|≡Q|≡(R′,R)↔K" IS
  FROM P|≡Q|≡(R,R′)↔K INFER P|≡Q|≡(R′,R)↔K
RULE "P|≡(R,R′)⇌K ⇒ P|≡(R′,R)⇌K" IS
  FROM P|≡(R,R′)⇌K INFER P|≡(R′,R)⇌K
RULE "P|≡Q|≡(R,R′)⇌K ⇒ P|≡Q|≡(R′,R)⇌K" IS
  FROM P|≡Q|≡(R,R′)⇌K INFER P|≡Q|≡(R′,R)⇌K


RULE "P|≡∀x.X(x) ⇒ P|≡X(Y)"(Y,ABSTRACTION X) IS
  FROM P|≡∀x.X(x) INFER P|≡X(Y)
```

I had to include hyp so that the context can be used. Cut and left-weakening are, I believe, assumed by the authors:

```
RULE hyp IS INFER X ⊢ X
RULE cut(X) IS FROM X AND X ⊢ Y INFER Y
RULE weaken(X) IS FROM Y INFER X ⊢ Y

IDENTITY hyp
CUT cut
WEAKEN weaken
```

9.3 Putting rules into menus

Organising the rules into menus isn't trivial. I've included a menu for each operator and put each rule into all the menus which seem relevant to it: for example, "P|≡(Q,P)↔K, P<|XK ⇒ P|≡Q|∼X" is in the menus for ↔, ⊲ and |≡. Only hyp and the rule dealing with ∀ are in a menu labelled ‘Logic’.

I implemented forward reasoning in the style of chapter 5; for example "P|≡(Q,P)↔K, [P<|XK] ⇒ P|≡Q|∼X" is included in the ↔ menu as

```
ENTRY "P|≡(Q,P)↔K, [P<|XK] ⇒ P|≡Q|∼X" IS
  ForwardOrBackward ForwardCut 0 "P|≡(Q,P)↔K, P<|XK ⇒ P|≡Q|∼X"
```

and in the ◁ menu as

```
ENTRY "P<|XK, [P|≡(Q,P)↔K] ⇒ P|≡Q|∼X" IS
  ForwardOrBackward ForwardCut 1 "P|≡(Q,P)↔K, P<|XK ⇒ P|≡Q|∼X"
```

The square-bracketed antecedent in the menu entry is the one that *isn't* focussed upon in that step. The whole gory details are in the file `BAN_ASCII_menus.j`. I may not have included the rules in enough menus or enough times (for example, I probably ought to have `"P<|XK, [P|≡(Q,P)↔K] ⇒ P|≡Q|∼X"` in the |∼ menu twice, focussing once on each antecedent). I haven't had enough feedback to know if I got this bit of user interaction right.

9.4 Dealing with tuples

I generalised some of the BAN rules: for example, I implemented

$$\boxed{\dfrac{P \models X_1 \quad \ldots \quad P \models X_n}{P \models (X_1, \ldots, X_n)} \quad \dfrac{P \models (\ldots, X, \ldots)}{P \models X}}$$

for 2-, 3- and 4-tuples. I did it, as you might expect, by listing each version of the rule and combining them with the RULES directive. For example:

```
RULES "P|≡X, P|≡Y, ...  ⇒ P|≡(X,Y,...)" ARE
    FROM P|≡X AND P|≡Y INFER P|≡(X,Y)
  AND FROM P|≡X AND P|≡Y AND P|≡Z INFER P|≡(X,Y,Z)
  AND FROM P|≡W AND P|≡X AND P|≡Y AND P|≡Z INFER P|≡(W,X,Y,Z)
END
```

and

```
RULES "P|≡(...,X,...)  ⇒ P|≡X"(X) ARE
    FROM P|≡(X,Y) INFER P|≡X
  AND FROM P|≡(Y,X) INFER P|≡X
  AND FROM P|≡(X,Y,Z) INFER P|≡X
  AND FROM P|≡(Z,X,Y) INFER P|≡X
  AND FROM P|≡(Y,Z,X) INFER P|≡X
  AND FROM P|≡(X,Y,Z,W) INFER P|≡X
  AND FROM P|≡(W,X,Y,Z) INFER P|≡X
  AND FROM P|≡(Z,W,X,Y) INFER P|≡X
  AND FROM P|≡(Y,Z,W,X) INFER P|≡X
END
```

The second group gives an interesting forward proof problem. I would like to be able to select an item of a tuple and pick it out using one of these rules. To do so I need to be able to search the collection. Since forward proof steps are all sequences "*cut; rule; select subgoal; hyp*" I have to be sure on the second step to select the right rule. The ALT corresponding to the RULES declaration above isn't very helpful. Jape matches the conclusion of a rule against the tip of a tree, and all the rules have the same conclusion, so it's not possible to choose between them by rule-matching. I could write a special ALT, but that would be tedious. So instead (sigh!) I invented a new mechanism: WITHCONTINUATION.

WITHCONTINUATION *tactic*$_0$ *tactic*$_1$... *tactic*$_n$ sets the sequence *tactic*$_1$... *tactic*$_n$ as a continuation, and runs *tactic*$_0$. If *tactic*$_0$ is an ALT, or ends with an ALT, it will add that continuation to each of its alternatives. The effect is that an alternative won't succeed unless the continuation *tactic*$_1$... *tactic*$_n$ succeeds as well. If *tactic*$_0$ doesn't end with an ALT, then the effect is the same as SEQ *tactic*$_0$ *tactic*$_1$... *tactic*$_n$. Forward step tactics all use WITHCONTINUATION:

```
TACTIC ForwardCut (n,Rule)
  SEQ cut (ForwardUncut n Rule)

TACTIC ForwardUncut (n,Rule)
  (LETGOALPATH G
    (WITHCONTINUATION (WITHARGSEL Rule) (GOALPATH (SUBGOAL G n)) (WITHHYPSEL
hyp))
    (GOALPATH G)
    NEXTGOAL)
```

(the GOALPATH dance is to ensure that the next selected step isn't always far away from the tree that the ALT built).

Then we include in the |≡ menu, for example

```
ENTRY "P|≡(...,X,...)  ⇒ P|≡X" IS ForwardOrBackward ForwardCut 0 "P|≡(...,X,...)
⇒ P|≡X"
```

and Bob's your uncle.

9.5 Conjectures with long assumption lists

On educational grounds I thought it best to include lots of assumptions in each conjecture, because the problem for novices is to decide which assumptions are relevant and how they can be used. This makes very long conjectures. For example, one of the conjectures about the Needham-Schroeder protocol is

```
THEOREM "Needham-Schroeder:  A<|{Na,(A,B)↔Kab,#((A,B)↔Kab),{(A,B)↔Kab}Kbs}Kas
⊢ A|≡#((A,B)↔Kab)" IS
  A|≡(A,S)↔Kas, S|≡(A,S)↔Kas, B|≡(B,S)↔Kbs, S|≡(B,S)↔Kbs, S|≡(A,B)↔Kab,
  A|≡(∀k.S|⇒(A,B)↔k), B|≡(∀k.S|⇒(A,B)↔k), A|≡(∀k.S|⇒#((A,B)↔k)),
  A|≡#Na, B|≡#Nb, S|≡#((A,B)↔Kab), B|≡(∀k.#((A,B)↔k)),
  A<|{Na,(A,B)↔Kab,#((A,B)↔Kab),{(A,B)↔Kab}Kbs}Kas
  ⊢ A|≡#((A,B)↔Kab)
```

Expressed as a single line this would be several screens wide.

Jape automatically folds long assumption lists in a box-and-line proof to fit the proof window (you can turn this off, if you must, by setting `foldassumptionlines` false). The proof of this conjecture, in a moderately-sized window, is shown in figure 9.1. It can also fold long formulae, but that isn't exploited in this encoding.

```
 1: A|≡(A,S)↔Kas, S|≡(A,S)↔Kas, B|≡(B,S)↔Kbs            assumptions
 2: S|≡(B,S)↔Kbs, S|≡(A,B)↔Kab, A|≡(∀k.S|⇒(A,B)↔k)      assumptions
 3: B|≡(∀k.S|⇒(A,B)↔k), A|≡(∀k.S|⇒#((A,B)↔k)), A|≡#Na assumptions
 4: B|≡#Nb, S|≡#((A,B)↔Kab), B|≡(∀k.#((A,B)↔k))        assumptions
 5: A<|{Na,(A,B)↔Kab,#((A,B)↔Kab),{(A,B)↔Kab}Kbs}Kas    assumption
 6: A|≡(S,A)↔Kas                                         P|≡(R,R')↔K ⇒ P|≡(R',R)↔K 1.1
 7: A|≡S|~(Na,(A,B)↔Kab,#((A,B)↔Kab),{(A,B)↔Kab}Kbs)    P|≡(Q,P)↔K, P<|{X}K ⇒ P|≡Q... 6,5
 8: A|≡#(Na,(A,B)↔Kab,#((A,B)↔Kab),{(A,B)↔Kab}Kbs)      P|≡#X ⇒ P|≡#(...,X,...)'5 3.3
 9: A|≡S|≡(Na,(A,B)↔Kab,#((A,B)↔Kab),{(A,B)↔Kab}Kbs)    P|≡#X, P|≡Q|~X ⇒ P|≡Q|≡X 8,7
10: A|≡S|≡#((A,B)↔Kab)                                   P|≡Q|≡(...,X,...) ⇒ P|≡Q|≡X'7 9
11: A|≡S|⇒#((A,B)↔Kab)                                   P|≡∀x.X(x) ⇒ P|≡X(Y) 3.2
12: A|≡#((A,B)↔Kab)                                      P|≡Q|⇒X, P|≡Q|≡X ⇒ P|≡X 11,10
```

Figure 9.1: A proof with a folded assumption line

Chapter 10

Encoding Natural Deduction more faithfully

There are a lot of things wrong with the Natural Deduction encoding of chapter 5. Though it seems to allow forward and backward steps, in fact it breaks the rules of box-and-line displays. Its var/inscope mechanism isn't part of the logic it purports to imitate. Its error messages are phrased in terms of the sequent calculus, which its intended users — students taking the logic course at QMW — never saw. You could only make a single formula selection though some of the rules used in forward steps — → elimination, ¬ elimination, ∀ elimination — have more than a single antecedent and seemed naturally to need two. Worst of all, so far as novices are concerned, it allows metalogical steps involving unknowns, 'incomplete' logical steps that have to be completed by subformula selection and unification.

Some of these problems were known to me. Some were revealed as the result of a research project looking at the rôle of visualisation in learning proofs. In one harrowing episode recorded on video, an incomplete ∀ intro step in a fairly straightforward proof reduced a student to tears of incomprehension and frustration.

I fixed as many of these problems as I could, as well as I could. I'd be dishonest to claim that this encoding shows Jape's flexibility, because I had to modify its internal mechanisms quite a bit. But the mechanisms I provided are now available to all and with the exception of the Kripke-tree disproof mechanism (see next chapter), they are probably of general interest.

The encoding is in `examples/naturaldeduction`, in file `I2L.jt`.

10.1 Rules

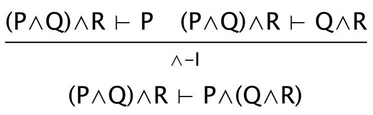
The rules of this encoding are similar to those of chapter 5, except for 'actual' in place of 'var' and 'inscope', and use of contradiction and truth symbols.

```
RULE "→ elim" IS FROM A→B AND A INFER B
RULE "∧ elim(L)" IS FROM A ∧ B INFER A
RULE "∧ elim(R)" IS FROM A ∧ B INFER B
RULE "∨ elim" IS FROM A ∨ B AND A ⊢ C AND B ⊢ C INFER C
RULE "∀ elim" IS FROM ∀x.  P(x) AND actual i INFER P(i)
RULE "∃ elim"(OBJECT i) WHERE FRESH i AND i NOTIN ∃x.P(x)
IS FROM ∃x.P(x) AND actual i, P(i) ⊢ C INFER C

RULE "→ intro" IS FROM A ⊢ B INFER A→B
RULE "∧ intro" IS FROM A AND B INFER A ∧ B
RULE "∨ intro(L)"(B) IS FROM A INFER A ∨ B
RULE "∨ intro(R)"(B) IS FROM A INFER B ∨ A
RULE "¬ intro" IS FROM A ⊢ ⊥ INFER ¬A
RULE "¬ elim"(B) IS FROM B AND ¬B INFER ⊥
RULE "∀ intro"(OBJECT i) WHERE FRESH i
IS FROM actual i ⊢ P(i) INFER ∀x .P(x)
RULE "∃ intro" IS FROM P(i) AND actual i INFER ∃x.P(x)

RULE "contra (classical)" IS FROM ¬A ⊢ ⊥ INFER A
RULE "contra (constructive)" IS FROM ⊥ INFER B
RULE "truth" IS INFER ⊤

RULE hyp(A) IS INFER A ⊢ A
```

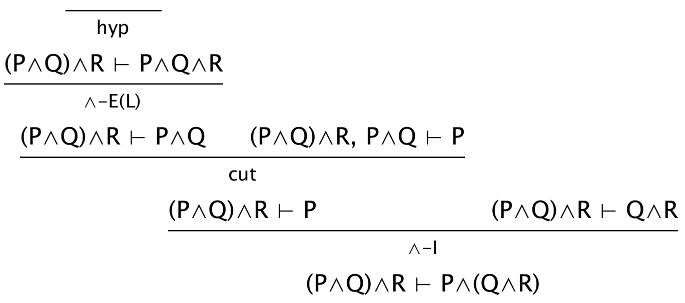
These rules are more pronounceable than those of chapter 5 — elim and intro rather than -E and -I — and they are organised into Forward and Backward menus. Although it is possible to run any rule backward, and most of them can be run forward with a bit of ingenuity, I put elim rules in the Forward menu, intro rules in Backward, and restricted cross-membership. The menu entries are labelled to give a hint of the effect of the step (and, in some cases, how to apply it).

```
MENU Forward IS
  ENTRY "∧ elim (preserving left)" ...
  ENTRY "∧ elim (preserving right)" ...
  ENTRY "→ elim" ...
  ENTRY "∨ elim (makes assumptions)" ...
  ENTRY "¬ elim" ...
  ENTRY "∀ elim (needs variable)" ...
  ENTRY "∃ elim (assumption & variable)" ...
  ENTRY "contra (constructive)" ...
  SEPARATOR
  ENTRY "∧ intro" ...
  ENTRY "∨ intro (invents right)" ...
  ENTRY "∨ intro (invents left)" ...
  SEPARATOR
  ENTRY hyp ...
END

MENU Backward IS
  ENTRY "∧ intro" ...
  ENTRY "→ intro (makes assumption)" ...
  ENTRY "∨ intro (preserving left)" ...
  ENTRY "∨ intro (preserving right)" ...
  ENTRY "¬ intro (makes assumption A)" ...
  ENTRY "∀ intro (introduces variable)" ...
  ENTRY "∃ intro (needs variable)" ...
  ENTRY "truth" ...
  SEPARATOR
  ENTRY "contra (classical; makes assumption ¬A)" ...
  ENTRY "contra (constructive)" ...
  ENTRY "¬ elim (invents formulae)" ...
  SEPARATOR
  ENTRY hyp ...
END
```

This is part of the push to make the logic more novice-accessible. But there’s far more to it than that ...

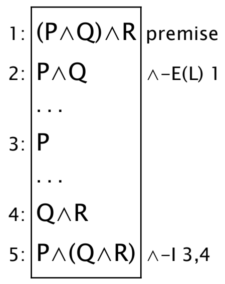


(a) ∧ intro splits tree

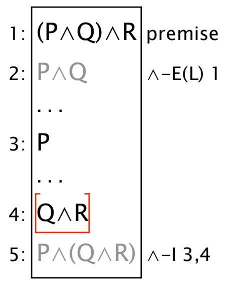


(b) expansion in one subtree

Figure 10.1: Separate subtrees have separate deductions

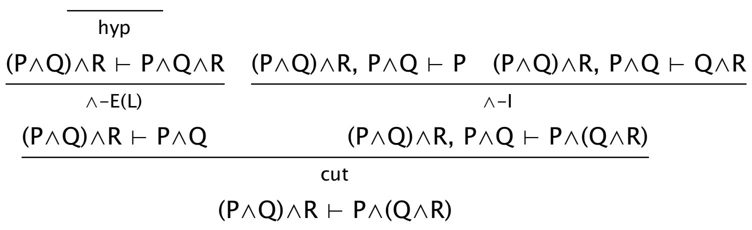


(a) faithful-looking box-and-line rendering

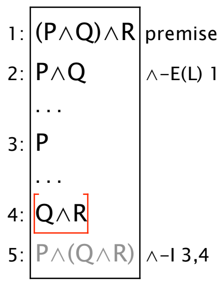


(b) is in fact a betrayal

Figure 10.2: Separate subtrees affect the box-and-line display



(a) elimination before introduction delays bifurcation



(b) and gives a more accurate box-and-line version

Figure 10.3: Separate subtrees force users to order their actions

10.2 CUTIN: full-featured box-and-line steps

The most glaring problem with the encodings that use box-and-line displays, before the advances discussed here, was that Jape broke the box-and-line rules. A simple $\wedge$ introduction in the chapter 5 encoding, for example, generates a bifurcated tree like that in figure 10.1(a). Any steps made in the left-hand subtree necessarily cannot be exploited in the right, and since all steps in this encoding take places at a tip of the tree, steps must be made in one or ther other. Suppose, for example, that the next step in the proof is an $\wedge$ elim in the left subtree: the result is the tree in figure 10.1(b). Rendered in box-and-line form this gives figure 10.2(a). The display looks fine, but clicking on line 4 in figure 10.2(b) reveals that it is not: line 1 is greyed out, inaccessible even though by box-and-line rules it can be used at line 4. That's not the only problem: although line 3 is black, it isn't available as a hypothesis selection in a proof of line 4. (Line 3 is an alternative conclusion, not greyed-out because you could select it instead of line 4. It looks just like a hypothesis, and this is very confusing to a novice who's been taught the basic rule of box-and-line proof, that lines above you and not inside a protecting box can always be called upon.)

That's not all. The tree in figure 10.3(a) is what you get if you do the $\wedge$ elimination first, then the introduction. It gives *exactly the same* box-and-line display, but this time (figure 10.3(b)) line 1 is accessible from everywhere. The order in which you make steps is important, even though it doesn't show immediately in the box-and-line display.

It's surely obvious that in a box-and-line proof you ought to be able to make forward and backward steps in whatever order you like, and that lines which look as if they could be used as hypotheses can be used as hypotheses. That's hard when the underlying structure is a proof tree. The obvious fix would have been to rework Jape to use a box-and-line structure as its underlying proof structure, but I didn't have the time to do that nor did I have any idea how to solve the difficulties that I could foresee in such a treatment. Instead I discovered how to fix the problem, somewhat laboriously, within the Jape tree. So far as forward steps are concerned, the basic idea is that cut nodes can be inserted into the tree: i.e. a tree like figure 10.1(a) can be transformed into one like figure 10.3(a) by a step of $\wedge$ elimination (see section 10.3.1).

The CUTIN tactic takes a subtree (not necessarily a tip, not necessarily the root of the current tree)

$$\begin{array}{c}\vdots\\ \Gamma \vdash C\end{array}$$

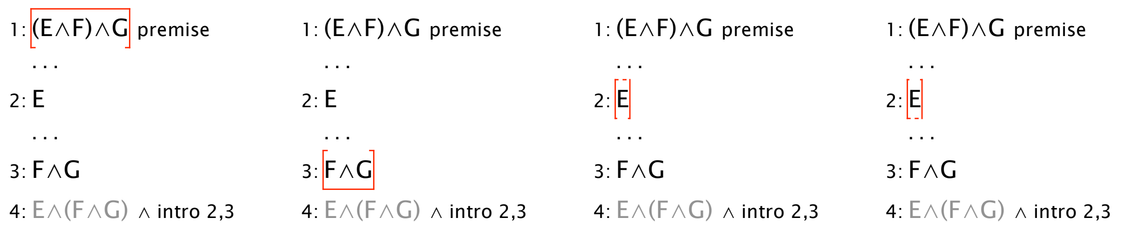
inserts an extra left-hand-side formula B and makes the result the right antecedent of a cut step, producing the new subtree

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash B\end{array}\qquad \begin{array}{c}\vdots\\ \Gamma, B \vdash C\end{array}}{\Gamma \vdash C}$$

Then it runs a tactic in the left antecedent of the cut step. Finally it returns to the position in the now-modified right antecedent at which it was originally applied.

Inserting a cut node in this way means putting an extra left-hand-side formula in every sequent of the modified subtree, and that means, in some cases, generating extra NOTIN provisos if the tree contains steps that used FRESH. Jape can only do it if the logic has a cut rule that is the right shape, and if the rules are stated additively without mentioning contexts explicitly — that is, `autoadditiveleft` is set true — so that extra left-hand formulae can be inserted freely.

Since the intention is to overcome the effects of bifurcating rules like $\wedge$ intro, it's easiest to begin by discussing how CUTIN can be used with backward rules.



(a) line 1 is always a hypothesis (b) line 3 is always a conclusion (c) line 2 can be a hypothesis (d) line 2 can also be a conclusion

Figure 10.4: Visible components in an accurate box-and-line display

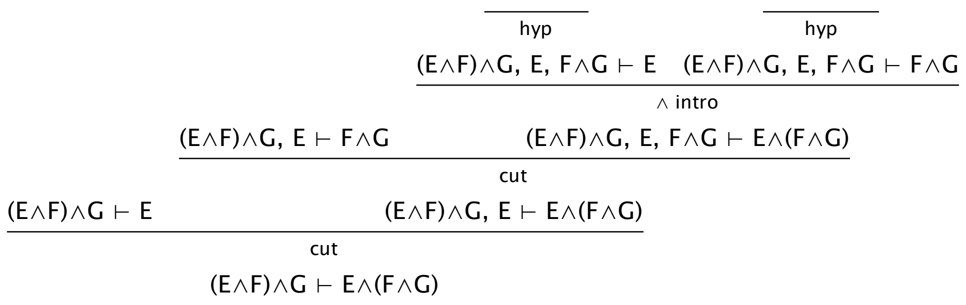


Figure 10.5: ∧ intro generates two cut nodes

10.2.1 CUTIN and backward steps

Figure 10.4 shows the various selections that can be made after a backwards ∧ intro step in the encoding of this chapter. Lines 1 and 3 have a fixed interpretation as hypothesis and conclusion respectively. Line 2, intermediate between the two, needs proof but, according to the rules of box-and-line proofs, can also be called upon if necessary in the proof of line 3. In this encoding it can: different ways of clicking on the line select it as hypothesis or conclusion (see section 10.3).

The magic is done with a couple of cuts, as shown in figure 10.5: E and $F \wedge G$ are inserted as hypotheses, and the actual ∧ intro proof step closed with hyp. The effect is that the tip conclusion $F \wedge G$ can see the hypothesis E. The box-and-line translation, and the selection mechanism, had to be modified to deal with intermediate lines like line 3. The GOALPATH navigation mechanism had to be modified to deal specially with cut nodes that can be inserted after a path has been recorded (I'm inordinately proud of the fact that I got that bit to work, but it would be a mistake to try to describe its intricacy here). Otherwise it's all down to CUTIN.

CUTIN in backward steps is usually applied via the fstep tactic. The menu entry for ∧ intro, stripped of its error-handling protection (I'll come back to that) is essentially

```
    SEQ "∧ intro" fstep fstep
```

(see "∧ intro backward" in `I2L_menus.j`). "∧ intro" is the Natural Deduction rule — see `I2L_rules.j`

```
    RULE "∧ intro" IS FROM A AND B INFER A ∧ B
```

— which generates two open subgoals. Fstep takes an open goal, CUTINs its conclusion as a hypothesis, and closes the original goal with hyp. It's in `examples/forwardsteptechnology/forwardstep.j` along with the trueforward tactic which it calls upon:

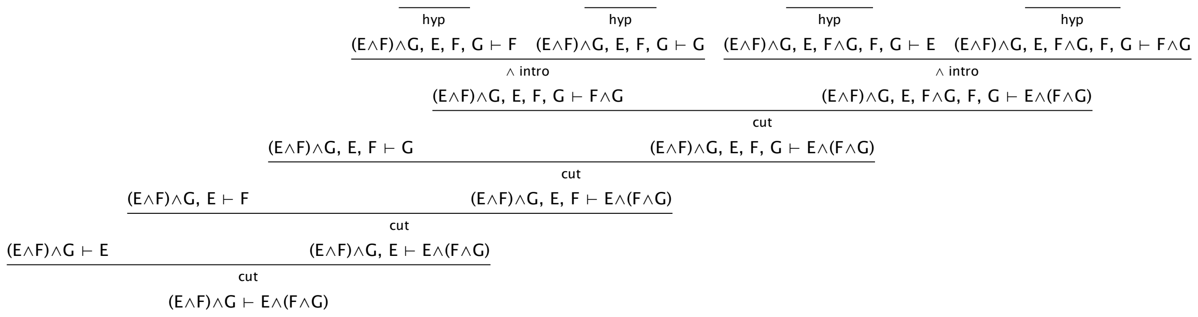


Figure 10.6: A second ∧ intro gives two more cut nodes

1: (E∧F)∧G premise
. . .
2: E
. . .
3: F
. . .
4: G
5: F∧G ∧ intro 3,4
6: E∧(F∧G) ∧ intro 2,5

Figure 10.7: A concise summary of figure 10.6

```
TACTIC fstep IS
  ALT
    (ANY (MATCH hyp))
    (trueforward SKIP)

MACRO trueforward(tac) IS
  LETGOAL _A
    (CUTIN (LETGOAL _B (UNIFY _A _B) tac))
    (ANY (MATCH hyp))
```

Fstep tries to match the open goal with any left-hand-side formula so as to avoid unnecessary duplication of hypotheses (no unification allowed: this must be done without changing the values of any unknowns in the proof tree). If that fails it applies trueforward SKIP. Trueforward records the current goal (LETGOAL _A) and then does a CUTIN. CUTIN looks for the correct place to cut the tree: it's the lowest point below the current node which has the same left-hand-side formulae as the current node, i.e. the lowest place in the tree, and therefore the highest place in the box-and-line display, where all the currently visible hypothesis lines are available. It runs its argument on the left-hand-side of the cut node it inserts there: in this case it records the new goal — the cut formula, which is also the inserted hypothesis on the right-hand-side of the cut node — in _B and unifies it with _A. Then it runs tac, which in this case is SKIP, and CUTIN is finished. Finally, back at the original open goal but now in the new subtree, hyp matches with the hypothesis that must now be there. Apply all that twice, as "∧ intro backward" does, and you get the effect shown in figure 10.5.

Want to see it again? Figure 10.6 shows the effect of applying ∧ intro again, this time to the $F \wedge G$ tip. The CUTIN place in figure 10.5 is the parent node of that tip, and anything in its right-hand subtree will be able to see the CUTIN formula. Once again, two cut nodes; once again, the actual proof step closed by hyp;

once again, the cumbersome tree concisely summarised in box-and-line style (figure 10.7), this time with two more intermediate formulae on lines 2 and 3.

So much for backward steps: they work properly if they use fstep freely. To discuss how forward steps use CUTIN, it's first necessary to discuss formula selection.

10.3 Formula selection

If you click on line 4 of figure 10.7, you are in effect clicking on the tip conclusion G of the tree in figure 10.6. That same click might be interpreted as selecting one of the left-hand-side occurrences of G elsewhere in the tree, but those left-hand selections would be pointless, because they are all of hypotheses in a closed tree, with no open conclusion to which they might correspond. This was one part of the original treatment of cut nodes in box-and-line display: in that tree G is only useful as a conclusion, so when you click on it it's obvious what you mean.

If you click on line 5, Jape rejects the attempt: you are pointing to $F \wedge G$, which occurs only as a conclusion in a closed tree, at the root of a completed $\wedge$ intro step. The same with line 6.

If you click on line 1, it's clear that $(E \wedge F) \wedge G$ occurs only as a left-hand-side formula, but it occurs at every node of the tree. The canonical instance — the one that generates line 1 — is at the root of the tree, so Jape regards that as the location of the selection. If you then conclusion-click on some other line — line 3, for example — the two selections are regarded as taking place in the $(E \wedge F) \wedge G, E \vdash F$ tip. This is the original treatment of hypothesis + conclusion selections in box-and-line displays.

Before CUTIN, all operations on a tree happened at tips, and a click on line 1 of figure 10.7 was by itself ambiguous, because it indicated the root of a tree with three tips (E, F and G). Until this encoding it was regarded as incomplete: "There is more than one unproved conclusion", Jape used to complain; "Please select the one you want to work on". But now we can apply a CUTIN step anywhere in the tree, so `I2L_settings.j` contains

```
    INITIALISE seektipselection false
```

and Jape no longer complains about selections that don't describe tip nodes. You can still tell if the selection is at a tip, because LETGOAL can only match at a tip (LETRHS matches at any node of the tree provided there's a single conclusion formula).

F on line 3 is more of a problem. It occurs at a tip in figure 10.6, but it also occurs as a left-hand-side formula in a tree that isn't closed. So it might be used as a conclusion, or it might be used as a hypothesis somehow to prove G, the only tip in that tree. In the box-and-line display it can be seen as intermediate between lines 1 and 2, a bridge between them. A click on an intermediate line is by itself ambiguous, and such lines are now treated specially. A click in the upper half of the formula produces a selection open towards the top – a conclusion selection, indicating the tip occurrence – and a click in the lower half is a hypothesis selection indicating the occurrence as a left-hand-side formula.

To summarise: a conclusion-click selects a right-hand-side formula at a tip; a hypothesis-click gives you the original left-hand-side occurrence of the formula. This isn't new — it's the way that clicks always worked — but the possibility of accepting a selection outcome which isn't at a tip is new, as is the way of selecting different readings of ambiguous lines.

10.3.1 Forward steps with CUTIN

Given a non-tip selection like line 1 of figure 10.7, a step from the Forward menu invokes (after some checking for errors) ForwardCut in `I2L_menus.j`

```
    TACTIC ForwardCut (n,rule) IS CUTIN (ForwardUncut n rule)
```

and the part of ForwardUncut that matters is

```
    LETHYP _Ah
      (LETGOALPATH G
        (WITHARGSEL Rule)
        (GOALPATH (SUBGOAL G n))
        (WITHHYPSEL hyp)
        (GOALPATH G)
        NEXTGOAL)
```

— apply the rule to the left antecedent of the cut produced by CUTIN, go to its nth antecedent, apply hyp to match the selection, then go back to the root of the left antecedent and look for the next goal. In the ForwardCut use that last bit doesn't matter, because CUTIN returns to the original application position. (And in any case I don't use `autoselect`: the user must always indicate, by selection, where to work next. I think the NEXTGOAL stuff is inherited from some other encoding.)

The second alternative in ForwardUncut deals with the possibility that it's applied without a hypothesis selection: that can happen if there is only one hypothesis formula in the tree. LETLHS _Ah matches just if there is a single left-hand-side formula; what follows imitates what happens with a hypothesis selection.

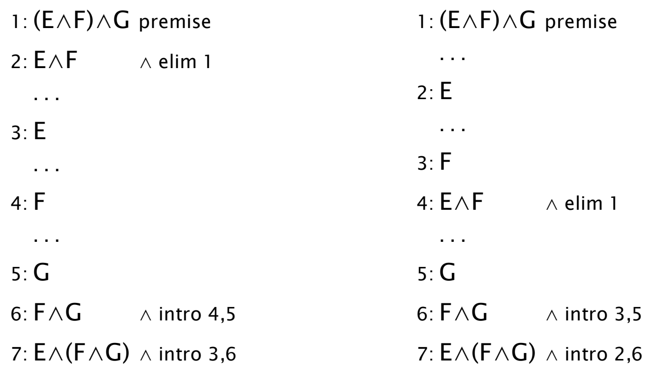
```
    LETLHS _Ah
      (LETGOALPATH G
        (WITHARGSEL Rule)
        (GOALPATH (SUBGOAL G n))
        (LETGOAL _Ag (UNIFY _Ag _Ah) (ANY hyp))
        (GOALPATH G)
        NEXTGOAL)
```

(The UNIFY is used in case Rule introduces any new left-hand-side formulae. Making _Ah an argument to hyp doesn't always work, for reasons which now escape me.)

With a single selection of line 1 in figure 10.7 a step of $\wedge$ elim produces figure 10.8(a) because the cut is applied to the root of the tree in figure 10.6, and the new left-hand-side formula, on line 2, is visible throughout all its right antecedent subtree. With a selection of line 1 and line 4, the cut is applied much higher up the tree, just below the G tip, and the result is figure 10.8(b) where the new left-hand-side formula appears in the new line 4. (It's odd at first to realise that lower down the diagram means higher up the tree, but you get used to it.)

10.4 Multiple hypothesis selections

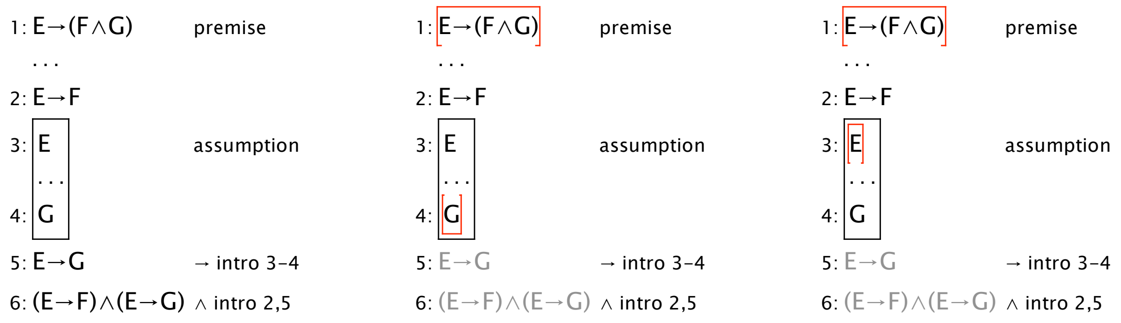
In the sequent calculi which Bernard and I initially encoded, a single left- or right-hand-side formula selection at a tip was enough to indicate the component that was to be worked on by a rule step. This led to a slogan: "point at what you want to work on", and a simple interaction mechanism. Later I had to modify it a bit to allow single hypothesis + single conclusion selections in the box-and-line display — just because a



```
1: (E∧F)∧G  premise           1: (E∧F)∧G  premise
2: E∧F      ∧ elim 1            ...
   ...                        2: E
3: E                            ...
   ...                        3: F
4: F                          4: E∧F      ∧ elim 1
   ...                          ...
5: G                          5: G
6: F∧G      ∧ intro 4,5       6: F∧G      ∧ intro 3,5
7: E∧(F∧G)  ∧ intro 3,6       7: E∧(F∧G)  ∧ intro 2,6
```

(a) selecting only line 1 (b) selecting lines 1 and 4

Figure 10.8: ∧ elim in figure 10.7 after different selections



```
1: E→(F∧G)       premise
   ...
2: E→F
3: E             assumption
   ...
4: G
5: E→G           → intro 3-4
6: (E→F)∧(E→G)   ∧ intro 2,5
```

(a) the problem (b) lines 1 and 4 selected (c) lines 1 and 3 selected

Figure 10.9: ∧ elim: advantages of multi-hypothesis selection

single hypothesis line stands for a multiplicity of left-hand-side occurrences — but we still stuck to single selections on either side of the sequent. Sometimes we smuggled extra hypothesis selections in by demanding a subformula selection, but the cure was worse than the disease, in UI design terms (it was lack of such a selection that reduced that videoed student to tears).

In Natural Deduction several rules used in forward steps — → elim, ¬ elim and ∀ elim at least — have more than one antecedent, and it's unreasonable to force your users to point at only one of them. Consider figure 10.9(a), for example. To make an ∧ elim step in the chapter 5 encoding, you have to point at lines 1 and 4, as in figure 10.9(b). In the `I2L.jt` encoding of this chapter, you click line 1, shift-click line 3 (or, of course, the other way round) as in figure 10.9(c). Much more natural, much easier to explain to a novice (especially with good error reporting: see section 10.6).

Jape has two mechanisms to deal with multi-hypothesis selections. LETHYPS2 *pat1 pat2* recognises exactly two selections; LETHYPS *pat* recognises more than one selection (sorry!) and binds them as a tuple to *pat*. (In this encoding LETHYPS is mostly used for error analysis, but it gets a bit more of an outing in chapter 12.) The → elim menu entry in `I2L_menus.j` leads to the tactic "→ elim forward", which has the structure

```
TACTIC "→ elim forward" IS
  WHEN
    (LETHYP2 _A (_A→_B) ...  normal ...)
    (LETHYP2 _C (_A→_B) ...  error ...)
    (LETHYP2 _A _B ...)
    (LETHYP (_A→_B)
```

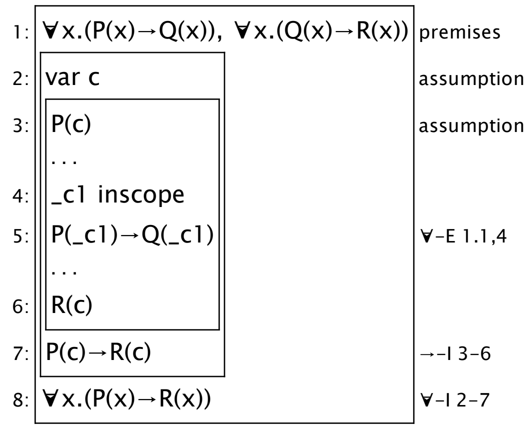

1: ∀x.(P(x)→Q(x)), ∀x.(Q(x)→R(x)) premises
2: var c assumption
3: P(c) assumption
...
4: _c1 inscope
5: P(_c1)→Q(_c1) ∀-E 1.1,4
...
6: R(c)
7: P(c)→R(c) →-I 3-6
8: ∀x.(P(x)→R(x)) ∀-I 2-7

Figure 10.10: A tear-provoking, or fury-provoking, mis-step

```
  (WHEN
    (LETCONC _C ...  accepted ...)
    (...  error ...))
(LETHYP _A ...  error ...)
(LETHYPS _A ...  error ...)
(...)
```

The first line recognises two hypothesis selections which match the antecedents of $\rightarrow$ elim (the order of selection doesn't matter). The second matches two selections which don't quite match: one of them is an $\rightarrow$ formula, but the other doesn't fit its right-hand side. The third alternative matches two hypothesis selections in which neither is an $\rightarrow$ formula. The fourth alternative matches a single selection which is an $\rightarrow$ formula: that's acceptable if there's also a conclusion selection (LETCONC), otherwise not.[1] The fifth alternative deals with a single selection which isn't an $\rightarrow$ formula. The sixth handles the case that there are more than two hypothesis selections. The last complains that there are no selections at all.[2]

The techniques used in the tactic are dealt with in the discussion of error checking and reporting (section 10.6).

10.5 Only complete steps allowed

One of the things that James Aczel's excoriating videos demonstrated to me was the difficulty students had with what he called 'incomplete steps'. Jape had been designed from the first to allow exploration, and in particular to insert unknowns if the user didn't or couldn't supply enough information to describe a definite instance of a rule step.

The $\forall$ elimination step in the chapter 5 encoding, for example,

```
RULE "∀-E"(c) IS FROM ∀x.  A(x) AND c inscope INFER A(c)
```

[1] The half-forward, half-backward step which allows us to deduce B from $A \rightarrow B$, preceding the deduction with an open conclusion A, is useful and I decided to permit it. But I insisted that there had to be a way of signalling that the user knew that just that step was being carried out.

[2] Other forward steps are allowed if there's only a single left-hand-side formula which matches the rule — LETLHS would check for that — but in this case I thought it wasn't appropriate.

might almost have been designed to generate incomplete steps. The 'c inscope' antecedent was normally hidden. The rule was applied from the menu by the tactic

```
TACTIC "∀-E with side condition hidden" IS
  LAYOUT "∀-E" (0) (WITHARGSEL "∀-E")
```

and there was a rule

```
RULE "inscope" IS INFER var x ⊢ x inscope
```

which was applied automatically and silently, like hyp, before the effect of a user's action was displayed. The idea was that the user would subformula-select something to correspond to c when applying the rule. (I realised at the time that I was in effect smuggling in an extra hypothesis selection, but I didn't realise what harm that could do.)

One tearful novice generated a proof display like that shown in figure 10.10 simply by making a ∀-E step without the necessary subformula selection of c on line 2. The nonsense on line 4 was beyond her comprehension — quite reasonably as I now see it — and she couldn't see through that to realise that her problem was to pick something to match _c1.

Aside from the var/inscope silliness in the chapter 5 encoding, unknowns are useful tools, even essential tools. But to novices they are an intrusion, something that Jape uses but the logic doesn't, and they become a barrier to learning. When necessary (see chapter 12) even novices must be able to deal with them, but then they take a bit of explaining. When unnecessary, as in this example, the logic designer must squirm with embarrassment. I squirmed.

After squirming, I took James's criticisms to heart and designed the new encoding to deny the possibility of incomplete steps. The flexibility that novices lose was in any case inaccessible to them while they were still struggling with the meaning of logic and the notion of proof search. "Do it this way" was a possible injunction and a useful one (users might not find out so much about what Jape can do, but so what?).

The ∀ elimination step in the new encoding is

```
RULE "∀ elim" IS FROM ∀x. P(x) AND actual i INFER P(i)
```

and it's applied by a tactic which requires that you select something for both antecedents:

```
TACTIC "∀ elim forward" IS
  WHEN
    (LETHYP2 (actual _i) (∀_x._A) ... normal ...)
    (LETHYP (∀_x._A) ... not enough ...)
    (LETHYP (actual _i) ... not enough ...)
    (LETHYPS _As ... not correct ...)
    (LETGOAL (∀_x._A) ... perhaps ∀ intro is called for ...)
    (... no selections ...)
```

That sort of thing, applied in every step, seems to be effective.[3] It's tedious and difficult, and it feels quite ad-hoc while you're doing it, but I don't know a better way. I can't imagine that there's a better way available within the architecture of Jape.

[3] Nobody has yet conducted educational experiments to check this assertion. Volunteers welcome.

10.6 Error checking and reporting

One thing which James's videos impressed on me was the inutility of Jape's error messages. They were expressed in language which the students couldn't understand, talking of antecedents, goals, tips, and I don't know what else. Students responded by ignoring all error messages, which didn't help the learning experience. So I fixed the error messages. It was a lot of work, and it appeared to help.

10.6.1 Enhanced alerts

The ALERT tactic takes a variety of arguments. The basic shape is

> ALERT *message button ...*

Message describes the text of the alert. The simplest *message* is a string, but there's also a `printf`-style variation: a tuple of a string and arguments. Format specifiers in the string are expanded using the arguments: %s for a string; %t for a formula (a 'term'); %l for a 'list' (actually a tuple: sorry). In the %l case the argument can be ((t0,t1,...,tn), sep) or it can be ((t0,t1,...,tn), sep1, sep2): the first alternative gives you t0<sep>t1<sep>...<sep>tn; the second gives you t0<sep1>t1<sep1>...<sep2>tn. You can put newlines in the string with \n.

Each *button* is a pair of a button label (which is a *message*) and a tactic. You get a button labelled with the label, which runs the tactic when you press it.

Here's an example:

```
ALERT
  "To make a ∀ elim step forward, you must select a hypothesis \
  \of the form ∀x.A, and also a pseudo-assumption of the form \
  \actual i.  You didn't select any hypotheses, but the current \
  \conclusion is %t, which could be used to make a ∀ intro step \
  \backwards.\
  \\nDid you perhaps mean to make a backward step with \
  \∀ intro?", ∀_x._A)
  ("OK",STOP) ("Huh?",SEQ Explainhypothesisandconclusionwords STOP)
```

10.6.2 Careful step-checking

Most of the work went into checking the way that rules are applied, and customising alerts for them. The tactic applied when you ask for ∧ intro, for example is

```
BackwardOnlyC
  (QUOTE (_A∧_B)) (Noarg "∧ intro backward" "∧ intro") "∧ intro" "A∧B"
```

BackwardOnlyC checks that you are applying the step without any confusing hypothesis selection:

```
TACTIC BackwardOnlyC (pattern, action, stepname, shape) IS
  BackwardOnly pattern action stepname shape
    "can also be used forward - see the Forward menu"
```

BackwardOnly (in `backwardstep.j` in `examples/backwardstep_technology`) is

```
MACRO BackwardOnly (pattern, action, stepname, shape, explain) IS
  WHEN
    (LETGOAL pattern (BackwardOnly2 stepname action explain))
    (LETGOAL _A (ComplainBackwardWrongGoal stepname shape))
    (ComplainBackward pattern stepname shape)
```

(it's a MACRO rather than a TACTIC because you don't want to evaluate the arguments, just use them). The first alternative tests that you are at a tip whose right-hand-side formula matches _A∧_B; if you are at some other kind of tip the second alternative generates a message; if you aren't at a tip it's the third alternative and an alternative message. (For details of what those messages are, look in `explain_selections.j` in `examples/explain_technology`.)

If you are at the right kind of tip, BackwardOnly2 checks that you haven't for some reason selected a hypothesis formula or two:

```
TACTIC BackwardOnly2 (stepname, action, explain) IS
  WHEN
    (LETHYPS _Ahs
      (WHEN
        (LETCONC _Ac
          (BackwardOnly3 stepname action explain _Ahs _Ac ", selecting"))
        (LETRHS _Ac /* can't fail */
          (BackwardOnly3 stepname action explain _Ahs _Ac " from"))))
    (WITHSELECTIONS action)
```

The second alternative applies the original action

```
Noarg "∧ intro backward" "∧ intro"
```

but the error checking isn't over yet: Noarg (in `I2L_menus.j`) checks that you haven't tried to give an argument to the step:

```
TACTIC Noarg (rule, stepname) IS
  WHEN
    (LETARGTEXT arg
      (ALERT
        ("The %s rule doesn't need an argument, but you subformula-\
        \selected %s.  Do you want to go on with the step, \
        \ignoring the subformula selection?",
        stepname, arg)
        ("OK", rule) ("Cancel", STOP)))
    (LETMULTIARG args
      (ALERT
        ("The %s rule doesn't need an argument, but you subformula-\
        \selected %l.  Do you want to go on with the step, \
        \ignoring all the subformula selections?",
        stepname, (args, ", ", " and "))
        ("OK", rule) ("Cancel", STOP)))
    rule
```

The first alternative picks out a single selection (it doesn't have to be a subformula selection, because LETARGTEXT will take anything) and complains. The second picks out a multiple selection. The third is the step, at last.

All the other steps are treated similarly. It's hard work — there is only limited commonality that can be exploited — but it produces a satisfactory result.

10.7 Patching internal alerts

This takes yet another custom mechanism (sigh!) called PATCHALERT. For example, in `I2L_menus.j`,

```
PATCHALERT "double-click is not defined"
  "Double-clicking a formula doesn’t mean anything in I2L Jape:  \
  \you have to select (single-click) and then choose a step \
  \from the Backward or Forward menu."
  ("OK") ("Huh?", HowToFormulaSelect)
```

The effect is to replace any built-in message which starts “double-click is not defined” — there are lots of them, all phrased to describe selection in sequent trees — with a single catch-all that talks in language that my users might understand. The catch-all is, in effect

```
ALERT
  "Double-clicking a formula doesn’t mean anything in I2L Jape:  \
  \you have to select (single-click) and then choose a step \
  \from the Backward or Forward menu."
  ("OK", SKIP) (“Huh?”, (SHOWHOWTO FormulaSelect))
```

— a two-argument tactic application in which the first argument, a string, is split across several lines. The other arguments are button descriptions for the alert window.

PATCHALERT is a hack, implemented at the lowest level, but I wanted to give it some of the capabilities of ALERT. In particular, I wanted to get access to the GUI texts which explain mouse gestures: selection, subformula selection, dragging. So it’s possible to add HowToFormulaSelect or HowToTextSelect or HowToDragFormulae or HowToDragDisproofStuff to a button description, but that’s all.

At present alerts which have no specified buttons are the only ones which can be patched. There used to be more patching in this encoding, and there probably ought to be more again.

Chapter 11

Disproof with Kripke trees

I have only one explanation: I needed disproof, it had to be integrated with Jape, it had to be Kripke trees, it was fun doing it.

There's a file `I2L_disproof.j`, containing

```
SEMANTICTURNSTILE ⊢ IS ⊨

FORCEDEF ⊤ IS ALWAYS /* top everywhere */
FORCEDEF ⊥ IS NEVER /* bottom nowhere */

FORCEDEF A∧B IS BOTH (FORCE A) (FORCE B)
FORCEDEF A∨B IS EITHER (FORCE A) (FORCE B)
FORCEDEF A→B IS EVERYWHERE (IF (FORCE A) (FORCE B))
FORCEDEF ¬A IS NOWHERE (FORCE A)

FORCEDEF ∀x.P(x) IS EVERYWHERE (ALL (actual i) (FORCE (P(i))))
FORCEDEF ∃x.P(x) IS SOME (actual i) (FORCE (P(i)))
```

These forcing definitions are interpreted by Jape, and they give you the capabilities described in the `disproof_howto` manual, distributed with modern releases.

Chapter 12

Encoding Hoare logic

This was always the target. I haven't solved the obvious arithmetic problem, nor even linked up to a finite arithmetic oracle (help welcome!), but I went some way to make it easy to use. It's in `examples/hoare_logic` in `hoare.jt`. It's closely tied to the treatment in my book (Proof and Disproof in Formal Logic, OUP, published July 2005).

12.1 Syntax (HOARE_SYNTAX.J)

Variables can start with any letter. Formula names start with any letter except K. Constant names start with K.

```
CLASS VARIABLE a b c d e f g h i j k l m n o p q r s t u v w x y z
CLASS FORMULA A B C D E F G H I J L M N O P Q R S T U V W X Y Z
CLASS CONSTANT K
CLASS BAG FORMULA Γ
```

mod is used in various examples. *length* is used when dealing with arrays.

```
CONSTANT mod length
CONSTANT ⊥ ⊤
```

`simplifiesto`, `equivto`, `conjoins` and `dependson` are operators used in the simplification of array-assignment preconditions (section 12.2.3). The treatment of quantifiers comes from the natural deduction encoding of chapter 10, but‘integer is used in place of actual; I have to define actual as well because of the requirements of PUSHSYNTAX (section 12.1.2). `computes` and `defines` are used in the treatment of definedness (section 12.2.1).

```
INFIX 5L ≜ /* equals def */
INFIX 5L simplifiesto equivto conjoins dependson /* see hoare_arith.j */

PREFIX 10 actual integer /* actual not used, but we have to satisfy PUSHSYNTAX
*/
POSTFIX 10 computes defined
```

Semicolon and becomes are essential operators.

```
    INFIX 10 L ;
    INFIX 12 L :=
```

Circleplus and mapsto are used in the treatment of array assignment (section 12.2.3).

```
    INFIX 50 L ⊕ /* circle plus */
    INFIX 60 L ↦ /* maps to */
```

The infix connectives and leftfix quantifiers:

```
    INFIX 100R → ↔ /* implies, iff */
    INFIX 120L ∨
    INFIX 140L ∧
    LEFTFIX 180 ∀ .
    LEFTFIX 180 ∃ .
```

Relational and arithmetic operators:

```
    INFIX 300L < > ≤ ≥ ≠ =
    INFIX 400 L + -
    INFIX 410 L × ÷
    INFIX 420 R ↑
```

Prefix negation:

```
    PREFIX 1200 ¬
```

Juxtaposition and substitution:

```
    JUXTFIX 9000
    SUBSTFIX 10000 « E / x »
```

How quantifiers bind (only single variables considered):

```
    BIND x SCOPE A IN ∀x .  A
    BIND x SCOPE A IN ∃x .  A
```

Sequents (including a turnstile):

```
    SEQUENT IS BAG ⊢ FORMULA
```

Shorthand rule definition variables, very much as usual:

```
    INITIALISE autoAdditiveLeft true /* allow rules to be stated without an
    explicit left context */
    INITIALISE interpretpredicates true
```

Essential bracketing forms:

```
    OUTFIX   /* for assertions */
    OUTFIX [ ] /* for indexing */
```

Instruction forms (I prefer ‘tilt’ to ‘abort’, but I don’t think I used it all in the end):

```
OUTFIX if then else fi
OUTFIX while do od
CONSTANT skip tilt
```

Transitive reasoning is possible (see chapter 6 — but I don't think I exploited it):

```
INITIALISE hidetransitivity true
INITIALISE hidereflexivity true
```

Finally, the menu of difficult-to-type symbols that the GUI shows you when you have to type a formula:

```
KEYBOARD → ↔ ∧ ∨ ¬ ⊥ ∀ ∃ ⊢ ⊨ ≤ ≥ ≠ ≜ × ÷ ↦ ⊕ « »
```

12.1.1 Juxtaposition and Hoare triples

I'd love to have defined the syntax so that, for example

```
{Q} x:=E {R}
```

was parsed properly, but I couldn't. It's parsed, currently, as

```
({Q} x):=(E {R})
```

— a juxtaposition of {Q} and x on the left of the infix operator :=, a juxtaposition of E and {R} on the right! Oh dear.

There is no solution to this problem: it's inherent in Jape's simple-minded bottom-up priority-based parser generator. Put JUSTFIX high, and it gets `a[i]:=7` right. Put it low, and it would get `{Q} x:=E {R}` right. I thought that bracketing array element expressions would be unbearable, so I chose high. You have to bracket single assignments, and sequences, to avoid the consequential problem with triples. Conditionals and loops have built-in brackets, so they are automatically ok.

12.1.2 PUSHSYNTAX

I wanted to use a couple of encodings, or partial encodings, that had been developed before. However these things each have their own syntax, and in particular their own hierarchy of operator priorities. Aligning yourself with one previously-defined hierarchy may be painful, but is possible. Aligning yourself with two may be impossible.

`hoare.jt` contains, therefore, a couple of uses of PUSH/POPSYNTAX. Provided that the 'calling' encoding defines all the operators of the 'called' encoding, and with the same meaning in each case (e.g. $\wedge$ is INFIX in both), the ordering of priorities doesn't matter: formulae will be read and printed (outside the PUSH/POPSYNTAX bracketing) using the 'calling' theory's definitions.

In loading the I2L encoding, I took the opportunity to change the treatment of quantifiers. Hoare logic quantifies only over integers: `I2L_integer_quantifiers.j` replaces 'actual' with 'integer' and also acknowledges that you can provide a subformula

12.2 Rules

Skip and tilt (tilt isn't in the menus):

```
RULE "skip" IS {A} skip {A}
RULE "tilt" IS {⊥} tilt {A}
```

Sequence:

```
RULE sequence(C) IS FROM {A}F{B} AND {B}G{C} INFER {A}(F;G){C}
```

It's very convenient sometimes to be able to insert intermediate assertions into a sequence (a kind of cut). This could be done in various ways, but I did it bluntly, so that I could write tactics to take Ntuples apart no matter how large. It exploits the fact that triples are parsed as juxtapositions (e.g. A{B}C{D} is parsed as ((A{B})C){D}, and A will be something like {A0}A1{A2}...An):

```
RULES "Ntuple" ARE
  FROM A{B} AND {B}C{D} INFER A{B}C{D}
AND FROM {A}B{C} AND {C}D{E} INFER {A}(B{C}D){E}
END
```

Consequence is straightforward:

```
RULE "consequence(L)" IS FROM A→B AND {B} F {C} INFER {A} F {C}
RULE "consequence(R)" IS FROM {A} F {B} AND B→C INFER {A} F {C}
```

After that it gets interesting.

12.2.1 Variable assignment, definedness and simplifiesto

In my book the assignment axiom is $\{(E \text{ computes}) \wedge A^x_E\}\, x := E\, \{A\}$, where E computes is supposed to pick out problems like division / remaindering by zero and array element misaddressing. I wanted to do the same thing in Jape: $x + 1$ computes, for example, is just true, and should vanish from the proof. So I implemented some tactical machinery.

The axiom is replaced by a rule, with the simplification in an antecedent:

```
RULE "variable-assignment" IS
  FROM R«E/x»∧(E computes) simplifiesto Q
  INFER {Q} (x:=E) {R}
```

This is a very non-committed rule — all it insists on is a match for x:=E — and the action is all in the tactic that's applied to the antecedent.

The menu entry for assignment is

```
ENTRY "variable-assignment" IS
  BackwardOnlyA
    (QUOTE ({_A} (_x := _E) {_B}))
    (Noarg (perhapsconsequenceL (SEQ "variable-assignment" simpl)))
    "variable-assignment"
    "A(x:=E)B"
```

which, when the error-checking is stripped away (see chapter 10 for explanation) is

```
perhapsconsequenceL (SEQ "variable-assignment" simpl)
```

PerhapsconsequenceL tries to apply its tactic, and if that fails inserts a consequence step:

```
TACTIC perhapsconsequenceL (tac) IS
  ALT
    tac
    (SEQ "consequence(L)" fstep (trueforward tac))
```

(again, see chapter 10 for explanation of fstep and trueforward — but I don't recall why it uses trueforward at all). It isn't as wasteful as it looks — it doesn't run simpl twice, because the rule won't match if you need consequence.

Simpl, then, is the tactic that's applied to the simplifiesto formula in the antecedent. It's a structural recursion on the left-hand-side of the antecedent, which takes care not to unify any unknowns it comes across. The tactic itself is in `hoare_arith.j`:

```
TACTIC simpl IS
  LAYOUT HIDEROOT
    (ALT
      (LETGOAL (_E∧(_x computes) simplifiesto _F)
        (UNIFY _F _E) (MATCH "arith_var"))
      (LETGOAL (_E∧(_K computes) simplifiesto _F)
        (UNIFY _F _E) (MATCH "arith_const"))
      (SEQ (MATCH "arith_index") simpl equiv equiv)
      (SEQ (MATCH "arith_single") simpl)
      (SEQ (MATCH "arith_double") simpl simpl)
      (SEQ (MATCH "arith_div") simpl simpl equiv))
```

HIDEROOT hides a step and displays only its antecedents. Because it's applied at every simpl step, they all disappear. Because each step is applied with MATCH, it won't alter any unknowns in the tree.

The first alternative eliminates variables; the second eliminates constants; the third deals with array-element formulae; the fourth with prefix and postfix operators; the fifth with infix operators other than div and mod; the last with div and mod.

The rules it uses do the work. Before `arith_var` and `arith_const` a UNIFY step is necessary to define the output unknown _F; without it the following MATCH step would fail.

```
RULE "arith_var" IS INFER A∧(x computes) simplifiesto A
RULE "arith_const" IS INFER A∧(K computes) simplifiesto A
```

`Arith_single` only has to deal with negation:

```
RULE "arith_single" IS
  FROM E∧(A computes) simplifiesto F
  INFER E∧(¬A computes) simplifiesto F
```

`Arith_double` deals with all the infix operators, simplifying each side separately:

```
RULES "arith_double" ARE
  FROM E∧(A computes) simplifiesto F
  AND F∧(B computes) simplifiesto G
  INFER E∧(A → B computes) simplifiesto G
AND
  FROM E∧(A computes) simplifiesto F
  AND F∧(B computes) simplifiesto G
  INFER E∧(A ↔ B computes) simplifiesto G
AND
...
END
```

In effect `arith_div` converts A ÷ B computes into B≠0, and treats A mod B similarly, but it generates an antecedent G∧B≠0 equivto H, to which simpl applies the equiv tactic (discussed below) in order not to insert too many copies of B≠0.

```
RULES "arith_div" ARE
  FROM E∧(A computes) simplifiesto F
  AND F∧(B computes) simplifiesto G
  AND G∧B≠0 equivto H
  INFER E∧(A ÷ B computes) simplifiesto H
AND
  FROM E∧(A computes) simplifiesto F
  AND F∧(B computes) simplifiesto G
  AND G∧B≠0 equivto H
  INFER E∧(A mod B computes) simplifiesto H END
```

`Arith_index` converts a[F] computes into 0≤F∧F<length(a), and generates equivto antecedents that are dealt with by two calls of equiv.

```
RULE "arith_index" IS
FROM E∧(F computes) simplifiesto G
  AND G∧0≤F equivto H
  AND H∧F<length(a) equivto I
  INFER E∧(a[F] computes) simplifiesto I
```

The equiv tactic looks for occurrences of F inside E, and if it finds one, declares E∧F equivto E. Then it replaces ⊤∧E with E (this can happen in the while rule):

```
TACTIC equiv IS
  SEQ
    (LAYOUT HIDEROOT (ALT (SEQ "arith_dup" conjoinstac) SKIP))
    (LAYOUT HIDEROOT (ALT "true_equiv" "equiv_default"))
```

`Conjoinstac` is another structural recursion, but only dealing with conjunctions:

```
TACTIC conjoinstac IS
  LAYOUT HIDEROOT
    (ALT
      (MATCH "arith_conjoins0")
      (SEQ (MATCH "arith_conjoinsL") conjoinstac)
      (SEQ (MATCH "arith_conjoinsR") conjoinstac))

RULES "arith_conjoins0" ARE
  INFER E∧F conjoins F
AND INFER F conjoins F
END
RULE "arith_conjoinsL" IS
  FROM E conjoins G INFER E∧F conjoins G
RULE "arith_conjoinsR" IS
  FROM F conjoins G INFER E∧F conjoins G
```

It's a lot of work, but Jape gets through it (slowly: you may notice a pause when you apply the assignment step; this tactic is the main reason why the Jape engine is now compiled rather than interpreted by OCaml) and it means that the definedness issues of assignment can be addressed.

12.2.2 Choice

Choice (if-then-else-fi) is dealt with in much the same way as variable assignment. The rule is

```
RULE "choice" IS
  FROM (E→A)∧(¬E→B)∧(E computes) simplifiesto G
  AND {A} F1 {C}
  AND {B} F2 {C}
  INFER {G} if E then F1 else F2 fi {C}
```

and the menu entry boils down to

```
perhapsconsequenceL (SEQ "choice" simpl fstep fstep)
```

12.2.3 Array-element assignment

Array-element assignment is similar to variable assignment, but it generates a more intricate treatment of computes:

```
RULE "array-element-assignment" IS
  FROM B«a⊕E↦F/a»∧(a[E] computes) simplifiesto C
  AND C∧(F computes) simplifiesto D
  INFER {D}(a[E]:=F){B}
```

The menu tactic, shorn of its error checking, is

```
perhapsconsequenceL
  (QUOTE
    (LETGOAL ({_A} (_a[_E] := _F) {_B})
      "array-element-assignment"
      ("length_simpl" _a _E _F)
      simpl
      simpl))
```

Length_simpl replaces formulae of the form length(a⊕E↦F) with length(a) (they are equivalent by definition, and visual clutter without the simplification). It's done with a straightforward rewrite rule:

```
TACTIC "length_simpl"(a,E,F) IS
  LAYOUT HIDEROOT ("arith_length" a E F)
RULE "arith_length"(a,E,F,OBJECT x) IS
  FROM A«length(a)/x» simplifiesto B
  INFER A«length(a⊕E↦F)/x» simplifiesto B
```

12.2.4 While

While loops are dealt with in the same sort of way:

```
RULE "while"(I, M, OBJECT Km) WHERE FRESH Km IS
  FROM ⊤∧(E computes) simplifiesto G
  AND I→G
  AND {I∧E}F{I}
  AND I∧E→M>0
  AND integer Km ⊢ {I∧E∧M=Km}F{M<Km}
  INFER {I}while E do F od{I∧¬E}
```

The menu tactic boils down to

```
perhapsconsequenceLR (SEQ "while" simpl maybetrueimpl fstep fstep)
```

The last antecedent isn't fstepped, because it's hypothetical (I don't know how to integrate it into the proof properly: there's no hyp for boxes; can you see how to do it?). Maybetrueimpl hides I->G in case G is $\top$. (proving it with the rules, no cheating!):

```
TACTIC maybetrueimpl IS
  ALT
    (SEQ (LAYOUT HIDEROOT) (MATCH "→ intro") (LAYOUT HIDEROOT) (MATCH "truth"))
    SKIP
```

12.3 An arithmetic sledgehammer

Arithmetic and natural deduction are what you actually do in a Hoare-logic proof: the program-logic steps are trivial. The ND steps, and the substitutions in program logic produce situations like $\Gamma, a[i] = 2 \vdash a[i] + 1 = 3$, which is so obviously true. Jape might expect you to prove it in Peano arithmetic, but life's too short; it might hitch up with a finite arithmetic oracle, but I haven't seen a way past licensing problems with any of those and therefore haven't seriously investigated the possibility.

So I borrowed an idea of Krysia Broda's, and implemented an 'obviously' step, to be used to prove obvious arithmetic truths. Of course it can be misused, and of course it makes the whole encoding completely unsound, but that has to be ok in this case. The rules are

```
RULE "obviously0" IS INFER A
RULE "obviously1" IS FROM A INFER B
RULE "obviously2" IS FROM A AND B INFER C
RULE "obviously3" IS FROM A AND B AND C INFER D
RULE "obviously4" IS FROM A AND B AND C AND D INFER E
```

and the tactic which applies them is

```
TACTIC obviouslytac IS
  WHEN
  (LETHYPS _A /* 1 or more selections, as a tuple */
    (LETLISTMATCH _A1 _B _A /* _A=(_A1, ...), _B=(...)  */
      (WHEN
        (LETLISTMATCH _B1 _C _B /* _B=(_B1,...), _C=(...)  */
          (WHEN
            (LETLISTMATCH _C1 _D _C ...  etc.  ...)
            (LAYOUT "obviously, from" ALL
              "obviously2" (WITHHYPSEL (hyp _A1)) (WITHHYPSEL (hyp _B1)))))
        (LAYOUT "obviously, from" ALL
          "obviously1" (WITHHYPSEL (hyp _A1))))))
  (LAYOUT "obviously" ALL "obviously0")
```

12.4 N-ary intro and elim steps

If you have a conclusion which is a long conjunction, it takes lots of $\wedge$ intro steps to break it part, and

Arithmetic and natural deduction are what you actually do in a Hoare-logic proof: the program-logic steps

are trivial. The ND steps, and the substitutions in program logic produce situations like $\Gamma, a[i] = 2 \vdash a[i] + 1 = 3$, which is so obviously true. Jape might expect you to prove it in Peano arithmetic, but life's too short; it might hitch up with a finite arithmetic oracle, but I haven't seen a way past licensing problems with any of those and therefore haven't seriously investigated the possibility.

So I borrowed an idea of Krysia Broda's, and implemented an 'obviously' step, to be used to prove obvious arithmetic truths. Of course it can be misused, and of course it makes the whole encoding completely unsound, but that has to be ok in this case. The rules are

```
RULE "obviously0" IS INFER A
RULE "obviously1" IS FROM A INFER B
RULE "obviously2" IS FROM A AND B INFER C
RULE "obviously3" IS FROM A AND B AND C INFER D
RULE "obviously4" IS FROM A AND B AND C AND D INFER E
```

and the tactic which applies them is

```
TACTIC obviouslytac IS
  WHEN
  (LETHYPS _A /* 1 or more selections, as a tuple */
    (LETLISTMATCH _A1 _B _A /* _A=(_A1, ...), _B=(...)  */
      (WHEN
        (LETLISTMATCH _B1 _C _B /* _B=(_B1,...), _C=(...)  */
          (WHEN
            (LETLISTMATCH _C1 _D _C ...  etc.  ...)
            (LAYOUT "obviously, from" ALL
              "obviously2" (WITHHYPSEL (hyp _A1)) (WITHHYPSEL (hyp _B1)))))
        (LAYOUT "obviously, from" ALL
          "obviously1" (WITHHYPSEL (hyp _A1))))))
  (LAYOUT "obviously" ALL "obviously0")
```

12.5 N-ary intro and elim steps

If you have a conclusion which is a long conjunction, it takes lots of ∧ intro steps to break it apart, and they clutter up the proof horribly. It's reasonable to hide the internal reasoning, and provide a single step which appears to be an N-ary ∧ intro. It's all done with LAYOUT COMPRESS, which will take a root with label X and make an N-ary node by hiding any immediate child steps which have the same label.

First I mess with the Backward menu inherited from `I2L.jt`, to get the new entry in a prominent place:

```
MENU Backward IS
  BEFOREENTRY "∧ intro"
    ENTRY "∧ intro (all at once)" IS
      BackwardOnlyC
        (QUOTE (_A∧_B))
        (Noarg "∧ intro*" "∧ intro")
        "∧ intro"
        "A∧B"
  RENAMEENTRY "∧ intro" "∧ intro (one step)"
END
```

∧ intro* is the tactic that does the work. In essence it is

```
TACTIC "∧ intro*" IS
  WHEN
    (LETGOAL (_P∧_Q)
      (LAYOUT COMPRESS "∧ intro")
      "∧ intro"
      "∧ intro*"
      "∧ intro*")
    SKIP
```

The actual tactic does stuff with tacticresult (a global variable) to try to deliver the tree path that corresponds to the leftmost conjunct in the original formula. But I don't use it, because I don't use autoselect, and in any case it doesn't work at present (the correct result is delivered, but you can't use it in a GOALPATH tactic).

Similar tricks are pulled with sequences and Ntuples of commands.

Multiple compressed forward steps are harder. The menu tactic boils down to

```
LETHYP _P (MATCH ("∧ elim*" _P))
```

The tactic is

```
TACTIC "∧ elim*"(P) IS
  WHEN
    (LETMATCH (_P∧_Q) P
      ("∧ elim* step" _P "∧ elim(L)" P)
      ("∧ elim*" _P)
      ("∧ elim* step" _Q "∧ elim(R)" P)
      ("∧ elim*" _Q))
    SKIP
```

— it matches its argument with _P∧_Q and then works on _P and _Q. It doesn't worry about tree paths because it's working on left-hand formulae throughout. Its helper tactic is

```
TACTIC "∧ elim* step"(P, rule, H) IS
  WHEN
    (LETMATCH (_P∧_Q) P
      (CUTIN
        (LAYOUT HIDEROOT)
        rule
        (LETGOAL _A (UNIFY _A H) hyp)))
    (CUTIN
      (LAYOUT "∧ elim")
      rule
      (LETGOAL _A (UNIFY _A H) hyp))
```

— hide intermediate steps (those working on a formula that will be broken down further); label terminal steps ∧ elim.

12.6 And that's it

Apart from the Make Lemma action, which I (finally!) introduced with this encoding, there's nothing else new. The CUTIN action doesn't solve all forward proof problems, but multiple use of boxes is pretty unlikely, so it doesn't matter much.

The next steps in Jape development, if there ever are any, might be to build a proper box-and-line calculator — or they might be to devise a framework for building Jape-like devices, so that there can be a modal logic calculator, and a separation logic calculator, and I can hardly imagine what else.

Appendix A

The paragraph and formula languages

The paragraph language is the one in which logics, their syntax, their rules, the tactics you intend to use and the menus of commands you intend to display are all defined. It uses a lot of reserved words: I add to the list as the need arises but all are multi-letter upper-case words, so it is a good idea to avoid use of that kind of word in your encodings.

At the time of writing the complete list of reserved words is

> ABSTRACTION, AND, ARE, AUTOMATCH, AUTOUNIFY, BAG, BEFOREBUTTON, BEFOREENTRY, BIND, BUTTON, CHECKBOX, CHILDREN, CLASS, COMMAND, CONCFRESH, CONCHIT, CONJECTUREPANEL, CONSTANT, CURRENTPROOF, CUT, DERIVED, DISTINCT, DISPROOF, END, ENTRY, FONTS, FORCEDEF, FORMULA, FORMULAE, FRESH, FROM, GIVENPANEL, HYPFRESH, HYPHIT, IDENTITY, IMPCONCFRESH, IMPFRESH, IMPHYPFRESH, IN, INFER, INFIX, INFIXC, INITIALLY, INITIALISE, IS, JUXTFIX, KEYBOARD, LABEL, LABELS, LEFTFIX, LEFTWEAKEN, LIST, MACRO, MENU, MENUKEY, MIDFIX, NOTIN, NOTONEOF, NUMBER, OBJECT, OUTFIX, PATCHALERT, POPSYNTAX, POSTFIX, PREFIX, PROOF, PUSHSYNTAX, RADIOBUTTON, REFLEXIVE, RENAMEENTRY, RIGHTFIX, RIGHTWEAKEN, RULE, RULES, SCOPE, SEMANTICS, SEMANTICTURNSTILE, SEPARATOR, SEQUENT, STRING, STRUCTURERULE, SUBSTFIX, TACTIC, TACTICPANEL, THEOREM, THEOREMS, THEORY, TRANSITIVE, UMENU, UNIFIESWITH, USE, VARIABLE, VIEW, WEAKEN, WHERE, WORLD

A.1 Directives

In this description I use [... | ...] to describe alternatives, { ... } to describe optional components, and ellipsis to denote optional repetition. I've used $\vdash$ as an entailment symbol: you choose the actual symbol in a SEQUENT directive (see section A.4).

ABSTRACTION decorates a parameter in a RULE or THEOREM directive. When the rule is instantiated, applications of this parameter to arguments are translated into substitutions, with a substitution variable which is made an OBJECT parameter of the rule. The effect is to simulate predicate notation with that parameter.

AND separator.

ARE separator.

AUTOMATCH *tacticname ... tacticname* at the end of each proof step, run the tactics (usually they are rules) specified over each open tip of the tree, but only allow them to work by 'matching' — that is, don't allow any unknowns in the tree to change as a result of running the tactic (see also MATCH in the tactic language).

AUTOUNIFY *tacticname ... tacticname* same as AUTOMATCH but without the restriction on working by 'matching'. This directive is less used than AUTOMATCH, chiefly because it is easy to make automatic steps which make large and/or unexpected and/or unhelpful changes to a proof. But sometimes it is the right thing: see for example the way that the Hindley-Milner algorithm encodings use AUTOUNIFY to determine the type of constants.

BAG { <kind> } *names* see section A.4.

BEFOREENTRY *label* ENTRY inserts an entry into a menu at a particular point.

BIND *variable ... variable* SCOPE *name ... name* IN *formula* see section A.4.

BUTTON *label* { IS } *command* allows you to attach a command to a label in a menu or a button on a panel (see appendix C).

CHECKBOX *variable label* { INITIALLY [true | false] } a checkbox is created associated with the named variable. If the variable doesn't exist in Jape's default environment this directive declares it, and its range of values will be true and false; if it exists, those must already be its range. The initial value, if included, is immediately assigned to the variable.

In a menu, *label* appears ticked or unticked according to whether or not the value of *variable* is true or false; in a panel you see a proper user-interface checkbox with that label.

CHILDREN see WORLD.

CLASS <kind> *names* see section A.4.

COMMAND in a panel button description, replaced by the command of the selected entry.

CONCFRESH *variable* proviso that *variable* doesn't occur free in any right-hand-side formula of the consequent of a proof step. See FRESH.

CONCHIT { { *formula1* } |- } *formula2* IS *tactic* if the user double-clicks on a right-hand-side formula matching *formula2* then run *tactic*. If *formula1* also appears, then either the sequent must have a single left-hand-side formula matching it, or the user must also have selected a left-hand-side formula matching it. See also HYPHIT.

CONJECTUREPANEL *name* { IS } [*entry* | *button*]* END build a panel of conjectures. Each *entry* is one of THEOREM, THEOREMS, DERIVED RULE, PROOF, CURRENTPROOF (the last two in saved-proof files); each *button* is either ENTRY or BUTTON. Entries add to the list of conjectures in the panel.

In addition to the buttons explicitly described, every conjecture panel always has "New. . . ", "Prove" and "Show Proof" buttons, and if there isn't a description of an "Apply" button then one is added as if you had written "BUTTON Apply IS apply COMMAND". Like MENU, a panel description can be divided into sections, and the complete description is just the concatenation of the various parts.

CONSTANT *name ... name* the *name*s are defined to have the syntactic class CONSTANT. See section A.4.

CURRENTPROOF *name sequent* { WHERE *provisos* } { FORMULAE *formulae* } IS *tactic* same as PROOF, except that the proof of *sequent* built by *tactic* need not be complete, is not recorded in the proof store, and is displayed on the screen.

CUT *rulename* synonym for STRUCTURERULE CUT; see section **??**.

DERIVED used in DERIVED RULE to specify a rule that needs proof.

DISTINCT *variable, ..., variable* synonym for lots of NOTIN provisos. Used, especially, to shorten proof display of provisos.

DISPROOF obsolete; see CURRENTPROOF and PROOF.

END closer in lots of directives.

ENTRY *name* { { IS } *tactic* } { MENUKEY *letter* } describes an entry in a menu or in the list of a panel. May only appear as part of a MENU or PANEL directive; MENUKEY is permitted when part of a MENU directive. The label is *name*; if the *tactic* component is omitted then the tactic expression *name* is used; if the MENUKEY component is included then *letter* is used as the 'command key' of that label. When the label is selected in a menu, the command "apply *tactic*" is transmitted to the proof engine; when the label is selected in a panel, there is no effect until a BUTTON is pressed.

FONTS *name* Obsolete.

FORCEDEF *formula* [IS] *forcing_definition* Part of the way that Kripke semantics is described. See `I2L_disproof.j`.

FORMULA *name ... name* the *name*s are declared to be in the syntactic class FORMULA. See section A.4.

FORMULAE separator in PROOF and CURRENTPROOF.

FRESH *variable* proviso in a rule or theorem. *Variable* mustn't appear free in any hypothesis or conclusion of the sequent to which the rule or theorem is applied. This proviso is translated into NOTIN provisos for each of the formulae of that sequent.

FROM separator.

GIVENPANEL you can make panels of 'given' antecedents when proving derived rules.

HYPFRESH *variable* *variable* doesn't occur free in any hypothesis of the problem sequent. See FRESH.

HYPHIT *formula1* |- { *formula2* } IS *tactic* if the user double-clicks on a left-hand-side *formula*1 then run *tactic*. If *formula*2 appears then either the sequent must have a single right-hand side which matches *formula*2, or the user must also select a right-hand-side formula matching *formula*2 in order for *tactic* to fire.

IDENTITY *rulename* synonym for STRUCTURERULE IDENTITY; see section **??**.

IN connective in binding directive.

IMPCONCFRESH, IMPFRESH, IMPHYPFRESH something internal to do with the processing of FRESH provisos. I can't remember what they do; I hope you never see one.

INFER connective in RULE directive.

INFIX ***precedence* [L | R | T] *operatorname* ... *operatorname*** the names are declared to be infix binary operators with the given precedence; L means left-associative, R right-associative, T tupling. Instances of formulae such as *A op B* are then treated internally as if they were ‘uncurried’ function applications — that is, as if you had written (*op*)(*A*,*B*). Note no commas. See section A.4.

INFIXC ***precedence* [L | R | T] *operatorname* ... *operatorname*** similar to INFIX, but parsed ‘curried’ so that *A op B* is then treated internally as if you had written (*op*) *A B*.

INITIALLY part of the RADIOBUTTON and CHECKBOX directives.

INITIALISE ***variablename value*** the variable named is assigned the value given. See the discussion of variables in appendix C.

IS connective, often omitted.

JUDGEMENT Obsolete.

JUXTFIX ***precedence*** defines syntactic precedence of juxtaposition: see section A.4.

KEYBOARD ***symbol ... symbol*** defines the symbols to be included on the on-screen keyboard in appropriate input windows (Unify, Text Command, etc.). In the absence of a KEYBOARD directive, you get all the symbols that don’t appear on the average real keyboard, in no particular order.

LABEL in a panel button description, insert the label of the selected entry. Cf. COMMAND.

LABELS part of the notation used to define Kripke trees in saved proofs/disproofs.

LEFTFIX ***precedence bra* { *punct*1 ... *punctN* }** defines bracketed form which misses a closing bracket. See section A.4.

LEFTWEAKEN ***rulename*** synonym for STRUCTURERULE LEFTWEAKEN; see section **??**.

LIST **<kind>** ***names*** the names are declared to be in the syntactic class LIST; see section A.4.

MACRO a variant of TACTIC. A MACRO’s arguments aren’t evaluated; they are useful mostly because you can pass patterns like _A=_B as arguments.

MENU ***name* IS *entry ... entry* END** the effect of the entries (which can be RULE, RULES, DERIVED RULE, TACTIC, THEOREM, THEOREMS, THEORY, PROOF, CURRENTPROOF, ENTRY, BUTTON, RADIOBUTTON, CHECKBOX or SEPARATOR) are added to the menu named *name*. MENU directives for the same menu are accumulated in sequence, and need not be given all in one place.

MENUKEY ***letter*** part of the ENTRY directive when used inside a MENU description.

MIDFIX ***priority symbol ... symbol*** like LEFTFIX and RIGHTFIX, but without an opening or closing bracket, only separators.

name* NOTIN *formula a proviso that *name* must not occur free in *formula*. Often generated as the result of a FRESH, HYPFRESH or CONCFRESH proviso; sometimes included in its own right.

NUMBER ***name ... name*** the *name*s are declared to be in the syntactic class NUMBER. See section A.4.

NOTONEOF : directive is *variable* IN *pattern* NOTONEOF *listformula*. It’s a terrible hack, used (so far only in) the Hindley-Milner encoding to try to implement an environment stack in the context (oh dear!). Example:

```
RULE "C ⊢ x⇒S" WHERE x IN x⇒S′ NOTONEOF C′ IS INFER C,x⇒S,C′ ⊢ x⇒S
```

OBJECT *name* decorates a parameter in a RULE or THEOREM directive. When the rule is instantiated, the parameter is replaced by a newly-minted variable rather than an unknown (e.g. x rather than _x), unless this default assignment is overridden by provision of an argument formula.

OUTFIX *bra* { *punct1 ... punctN* } *ket* bracketed form with optional internal separators; see section A.4.

PATCHALERT for overriding alert messages. See chapter 12.

POPSYNTAX see PUSHSYNTAX.

PUSHSYNTAX *syntax definitions* POPSYNTAX It's possible to inherit syntactic descriptions by enclosing them in PUSHSYNTAX / POPSYNTAX brackets. The outer syntax must express all the symbols of the inner, but not necessarily with the same syntactic priorities.

POSTFIX *precedence operator ... operator* postfix operators; see section A.4.

PREFIX *precedence operator ... operator* prefix operators; see section A.4.

PROOF *name sequent* { WHERE *provisos* } { FORMULAE *formulae* } IS *tactic* generated when you save proofs. *Sequent* (together with *provisos* if present) is a statement of the conjecture named *name*, and *tactic*, when run, will produce a proof of that conjecture. *Formulae*, if present, are a numbered list of the formulae that occur in *tactic*, provided to reduce the size of the directive and thus the cost of reading it all in (really!). If it all works out: if *sequent* unifies with the statement of conjecture *name*, if *tactic* produces a completed proof without introducing any additional unifications or inventing more or less provisos, then the resulting proof is stored under *name* in the proof store.

RADIOBUTTON *variablename* { IS } *label* { IS } *value* { AND *label* { IS } *value* }* { INITIALLY *value* } END a radio button with the list of labels given is associated with the named variable. If that variable doesn't exist it is declared by this directive, and its range of possible values is those given here; if it does exist the values given here must be in its range. If an initial value is given, the variable is assigned that value immediately.

In a menu a radio button is shown as a sequence of labels, one of which is ticked according to the value of the variable. Currently you can't put a radio button in a panel (sorry).

REFLEXIVE *rule name* declares rôle of a rule. Rule must be of the form A *op* A.

RENAMEENTRY *oldlabel newlabel* change the labelling in a menu.

RIGHTFIX *precedence* { *punct*1 *... punctN* } *ket* defines bracketed form with closing bracket but no opening bracket. See section A.4.

RIGHTWEAKEN synonym for STRUCTURERULE RIGHTWEAKEN; see section **??**.

RULE *rule* definition of a rule. Puts a *rule* into the tactic store with name *name*; if it appears inside a MENU or PANEL definition then the effect is also of ENTRY *name* IS *name*. See section A.2.

RULES *name* { (*params*) } { WHERE *provisos* } ARE *rule1* AND … AND *ruleN* END definition of a number of rules, organised automatically into an ALT tactic.

Each *rule* is an unnamed rule definition (see section A.2); each is considered to be qualified by *params* and *provisos* from the head of the directive, filtered according to the names that occur in each rule (that is, if a particular parameter doesn't occur in a rule, you don't get that parameter declaration with that rule, and you don't get any provisos that mention it). The rules are entered into the tactic store under the names *name'1 ... name'N*; at the same time a tactic ALT *name'1 ... name'N* is entered under *name*.

If it occurs in a MENU or a PANEL directive, RULES *name ...* has the effect also of ENTRY *name*: several rules are defined, but only one entry appears.

SCOPE part of the BIND directive.

SEMANTICS see WORLD (I'll add more here soon).

SEMANTICTURNSTILE ENTAILSSYMBOL SEMANTICTURNSTILE defines the semantic turnstile associated with a particular entails symbol (syntactic turnstile).

SEPARATOR used in the definition of a MENU, gives a division between entries.

SEQUENT IS [BAG | LIST | FORMULA] |- [BAG | LIST | FORMULA] definition of sequent syntax. |- is declared as the entails symbol. See section A.4.

STRING *name ... name* names defined to be in syntactic class STRING; see section A.4.

STRUCTURERULE [CUT | IDENTITY | LEFTWEAKEN | RIGHTWEAKEN | WEAKEN] IS *name* the rule called *name* is defined to have a particular property. See section **??**.

SUBSTFIX *precedence* { *bra fst sep snd ket* } defines the syntactic precedence of substitution forms and, optionally, their appearance as well. See section A.4.

TACTIC *name* { (*name1, . . . , nameN*) } { IS } *tactic* puts a tactic with name *name* and parameters *name1*, . . . , *nameN* into the tactic store. If it appears inside a MENU or PANEL definition then the effect is also of ENTRY *name*.See section A.2.

TACTICPANEL *name* { IS } [*entry* | *button*]* END very like CONJECTUREPANEL, except that there are no extra buttons and no default buttons and each entry labels a command, rule or tactic rather than a conjecture.

THEOREM *conjecture* puts a conjecture into the tactic store. Jape's conjectures are always 'theorem schemata', in the sense that they stand for any substitution-instance of the theorem's sequent. See section A.2.

THEOREMS *name* { *params* } { WHERE *provisos* } ARE *theorem1* AND ... AND *theoremN* END define a collection of conjectures which are organised into an ALT tactic.

Each *theorem* is an unnamed conjecture (see section A.2); each theorem is added to the tactic store prefixed by *params* and *provisos* in the same way as in the RULES directive; at the same time a tactic ALT *theorem1 . . . theoremN* is added to the tactic store under *name*. If included in a menu or panel description, an ENTRY is created for each conjecture. The effect is to define a number of conjectures with evocative names, and to allow searching of the collection if desired. See also THEOREM.

THEORY *name* { IS } *directive ... directive* END the directives may be RULE, RULES, TACTIC, THEOREM, THEOREMS or THEORY; an ALT tactic is made of the various directives and added to the tactic store under *name*. If THEORY occurs in a menu or panel description, the effect is as if the directives occurred separately (that is, Jape does not add an entry corresponding to the overall ALT tactic).

TRANSITIVE like REFLEXIVE (I'll put more here soon).

UMENU for the life of me I can't remember.

formula*UNIFIESWITH *formula a proviso which requires that the two formulae should unify. Used to delay unification when difficult substitutions get in the way.

collection* UNIFIESWITH *collection a proviso which helps when contexts are split. See chapter 3.

USE *“filename”* same effect as C’s #include “*filename*”.

VARIABLE *name ... name* names defined to be of syntactic class VARIABLE; see section A.4.

WEAKEN synonym for LEFTWEAKEN.

WHERE prefixes a list of proviso declarations.

WORLD something to do with Kripke trees. When I remember anything about it, I’ll come back and fill this in.

A.2 Rules, tactics and conjectures

The syntax of rules and conjectures (confusingly called THEOREMS — a battle with Bernard that I lost) can take various forms. The syntax is:

{ *name* } { (*param*,..., *param*) } { WHERE *proviso* AND . . . AND *proviso* } { IS } *body*

Note that almost every part is optional, apart from the body of the rule or conjecture. Note also that, for obvious reasons, if parameters or provisos are included then either the IS word must be included, or else *body* must start with FROM or INFER .

In a conjecture, *body* is

{ INFER } *consequent*

In a rule, *body* is

{ FROM *antecedent* AND … AND *antecedent* } INFER *consequent*

where *antecedent*s and *consequent* are sequents. If a conjecture or rule is un-named, its name is taken to be *consequent*.

Each *param* is [OBJECT | ABSTRACTION] *name*. The name must have been declared in a CLASS directive.

Each proviso is
CONCFRESH *name*,..., *name* or
HYPFRESH *name*,..., *name* or
FRESH *name*,..., *name* or
name NOTIN *formula* or
formula UNIFIESWITH *formula*
where the names used must be parameters of the rule or conjecture.

A.2.1 The meaning of a RULE directive

Ignoring parameters for the moment, the rule

$$\frac{\mathit{ante1} \quad \ldots \quad \mathit{anteM}}{\mathit{conse}}\ (\mathit{prov1},\ldots,\mathit{provN})\ \mathit{rule}$$

with name *rule*, antecedent sequents *ante1* ... *anteM*, consequent *conse* and provisos *prov1* ... *provN*, is stated as the rule directive

RULE *rule* WHERE *prov1* AND … AND *provN* IS FROM *ante1* AND …. AND *anteM* INFER *conse*

A rule is schematic in all names appearing in the antecedents, consequents or provisos which are declared in a CLASS directive. When the rule is applied to a problem sequent, a version of the rule is produced in which all the schematic names have been replaced by new unknowns, and then the consequent of that version is unified with the problem sequent.

If either of the *autoAdditive* variables is set to true (see appendix C) then rules can be defined as they often are in natural deduction presentation, without mentioning unmatched hypotheses and/or conclusions. For example, if *autoAdditiveLeft* is true, the rule

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C} \rightarrow\vdash$$

can be stated as

RULE "→|-" IS FROM *A* AND *B* |- *C* INFER *A*→*B* |- *C*

A.2.2 Parameters in RULE directives

Rule parameters are used for a number of reasons:

- On instantiation the first parameter is replaced by the user's text selection, rather than an unknown. This is useful when that parameter doesn't appear in the consequent or is the replacing formula in a substitution form in the consequent (see the discussions of substitution matching somewhere or other);
- A parameter which is decorated with OBJECT is replaced with a freshly-minted name, rather than a new unknown;
- A parameter which is decorated with ABSTRACTION is treated as a predicate, and juxtapositions involving that parameter are translated into substitution forms;
- Parameters are used to drive the 'proof reload' mechanism (no idea what this means).

A.2.3 Instantiating a rule, including interpretation of predicate notation

When a rule is instantiated, each of its schematic names is replaced by a freshly-minted unknown, whose name is based on the schematic name itself. This instantiation is modified in case the name is a parameter, if an `autoAdditive` parameter is set to true, or if `interpretpredicates` is true.
When an `autoAdditive` parameter is true, a rule is automatically extended so that all the hypotheses (`autoAdditiveLeft`) and/or the conclusions (`autoAdditiveRight`) are extended with a freshly-minted segment variable, and all the antecedents likewise; then the augmented rule is instantiated normally.

When `interpretpredicates` is true, juxtapositions in the rule are interpreted as predicate formulae and replaced by substitution forms; at the same time the parameter list is extended with OBJECT parameters as appropriate and the rule is augmented with invisible provisos. For example, a juxtaposition P(*x*,*y*) will be replaced by $P\,[u, v \backslash x, y]$; the parameter list will be extended with OBJECT *u*, OBJECT *v*, and an invisible proviso *x*,*y* NOTIN P will be added to the rule; every other 'predicate' application of P in the rule then has to have exactly two 'arguments' and the same variables *u* and *v* will be used in those cases as well. Jape avoids substitution forms wherever possible by noting the context: for example, it will translate $\forall x.\forall y.P\,(x, y)$ as $\forall x.\forall y.P$ and then would translate $P\,(E, F)$ as $P\,[x, y \backslash E, F]$ and in such an example there is no need for extra OBJECT parameters. For example, it will translate $\dfrac{P\,(E)}{\exists x.P\,(x)}$ into $\dfrac{P\,[x \backslash E]}{\exists x.P}$.

The latter form of rule is well-suited to backwards reasoning. The effect is to allow a kind of predicate notation while preserving Jape's existing unification mechanisms.

A.2.4 The meaning of provisos in RULE and THEOREM directives

Provisos are side conditions. At present we have a small number of built-in provisos, listed in section ??. All the forms of FRESH provisos are automatically translated into a collection of NOTIN provisos — one for each unmatched hypothesis and/or conclusion as appropriate.

Each time a rule is applied, its provisos are added to the set which is displayed in the bottom pane of the proof window. At the end of each proof step, that set of provisos is checked, and if any proviso is violated, the proof step is cancelled (memo to implementers: change the proof engine so that this is more obviously the way that things happen ...). At the same time provisos which are satisfied are deleted from the set. The ones that are left are those whose status can't be decided, because of the presence of unknowns, substitution forms or names over which the conjecture being proved is quantified.

A.2.5 The meaning of conjectures (stated in THEOREM directives)

A conjecture is stated as a rule without antecedents. Normally the first thing you do with a conjecture is to try to prove it. If that proof is successful, you can store it in the proof store and it will appear in the conjecture panel as a proved theorem. The provisos of a proved theorem are those given in the statement of the conjecture, plus any which arise and aren't satisfied during the proof.

Jape will normally refuse to apply a conjecture until it is proved, but you can tell it not to be so cautious if you wish by setting `applyconjectures` true.

If the consequent of a theorem matches a problem sequent, but in so doing it doesn't use up all the hypotheses and conclusion formulae of the problem, then Jape is cautious. If the logic you are using has a declared LEFTWEAKEN rule and there are too many hypotheses, then you could have first eliminated the extra hypotheses by applications of that rule, and then the theorem would have exactly used up all the remaining ones; similarly if it has a declared RIGHTWEAKEN rule and there are too many conclusions. But unless all that applies, the theorem will be said to be inapplicable.

A.2.6 Proof by resolution

There is a facility to use a sort of resolution step when applying a theorem. If a theorem's conclusion(s) match but its hypotheses don't, and if there is a declared CUT rule, then it would be possible to use a sequence of cuts to introduce the necessary extra hypotheses. In those circumstances Jape can introduce an antecedent for each of the hypotheses, and label the step with the name of the theorem. This feature is turned on and off by assigning to `tryresolution` (see appendix C). The effect is that if you have both right-weakening and cut, we treat a theorem $H_1, ..., H_n \vdash C$ as equivalent to the rule $\dfrac{\Gamma \vdash H_1 \quad ... \quad \Gamma \vdash H_n}{\Gamma \vdash C}$ provided that $\Gamma, H_1, ..., H_n \vdash C$ is a theorem; if you have cut but not right-weakening, we treat it as equivalent to $\dfrac{\Gamma \vdash H_1 \quad \Gamma, H_1 \vdash H_2 \quad ... \quad \Gamma, H_1, ..., H_{n-1} \vdash H_n}{\Gamma \vdash C}$ with the same proviso.

A.2.7 The meaning of STRUCTURERULE directives

It is necessary for the application of conjectures as rules, and for the proper operation of the box display mechanism, for Jape to be informed of the presence of certain kinds of structural rules in the logic. The rules we cater for are the various kinds of identity (hypothesis, axiom), cut and weakening rules.

CUT declares a 'cut' rule. Applications of the rule will normally be hidden in the box display mode of Jape.

This directive is required before Jape will properly interpret `tryresolution` (see appendix C). Cut rules have to be one of the following forms:

- (B) FROM $\Gamma \vdash B$ AND $\Gamma, B \vdash C$ INFER $\Gamma \vdash C$;
- (B) FROM $\Gamma \vdash B, \Delta$ AND $\Gamma, B \vdash \Delta$ INFER $\Gamma \vdash \Delta$;
- (B) FROM $\Gamma \vdash B, \Delta$ AND $\Theta, B \vdash \Pi$ INFER $\Gamma, \Theta \vdash \Delta, \Pi$.

identity declares an 'identity' rule. Instances of the application of this rule are normally hidden in box display mode. Allowable forms are:

- $\Gamma, B \vdash B$;
- $\Gamma, B \vdash B, \Delta$
- $\Gamma, B, \Delta \vdash B$.

leftweaken declares a 'left weakening' rule. Essential if Jape is to be able to apply theorems which don't have enough hypotheses to match the whole of the problem sequent. Valid forms are:

- (B) FROM $\Gamma \vdash C$ INFER $\Gamma, B \vdash C$;
- (B) FROM $\Gamma \vdash \Delta$ INFER $\Gamma, B \vdash \Delta$.

rightweaken similar to LEFTWEAKEN, and plays a similar rôle in theorem application. The only allowable form of the rule is (B) FROM $\Gamma \vdash \Delta$ INFER $\Gamma \vdash B, \Delta$

weaken see LEFTWEAKEN.

A.2.8 Substitution matching

Substitution forms are used in Jape to describe the operation of rules. They aren't intended to be interpreted as themselves — that is, they are not a special sort of formula with associated introduction and elimination rules. There are instead mechanisms inside the proof engine designed to eliminate substitution forms whenever they arise by carrying out the substitutions they describe, and the intention is that substitution forms should normally be read as naming the formula to which they simplify.

For example, consider Hoare logic's variable-assignment rule $\overline{\{R^x_e\}x := e\{R\}}$. This can be expressed (see `hoare_rules.j`) in Japeish as `RULE ":=" IS INFER {R«E/x»} (x:=E) {R}`.

Now consider the problem sequent `{_Q} (x:=1) {x=1}`. If we apply the variable-assignment rule then a version will be generated expressed in terms of new unknowns, which will be `{_R«_E/_x1»} (_x1:=_E) {_R}`. However the unification proceeds, it will eventually have unified `(x=1)«1/x»` with `_Q`. Before that formula is displayed to the user it will be simplified to `1=1`, and the rule will have done its job.

If the problem sequent had been `{1=1} (x:=1) {x=1}`, then the unification process might first come across the problem of unifying `_R«_E/_x1»` with `1=1`. Since that involves a substitution which won't simplify, it is deferred until later on. Unification of `x:=1` with `_x1:=_E` and of `x=1` with `_R` mean that when the deferred problem must finally be considered, it has been transformed into that of unifying `(x=1)«1/x»` with `1=1`: the substitution form simplifies to `1=1` and the unification is trivial.

Not every use of substitution forms in rules gives so little difficulty. When you define a rule with a substitution form in the consequent, and there aren't other occurrences of the components of the substitution form which will help to simplify it, matching becomes a problem.

For example, consider the natural deduction $\forall$ elimination rule $\dfrac{\Gamma \vdash \forall x.P}{\Gamma \vdash P[x\backslash E]}$, which can be expressed in Japeish as

```
RULE "∀ elim"(E, OBJECT x) IS FROM Γ |- ∀ x.P INFER Γ |- P[x\E]
```

If the problem sequent is `a=b,b=c |- (a+b)+c=a+(b+c)`, then there are fourteen significantly different ways in which the consequent of the rule can match the problem:

1. P : $(a+b)+c=a+(b+c)$; x : x; E : E
2. P : $(x+b)+c=a+(b+c)$; x : x; E : a
3. P : $(a+b)+c=x+(b+c)$; x : x; E : a
4. P : $(x+b)+c=x+(b+c)$; x : x; E : a
5. P : $(a+x)+c=a+(b+c)$; x : x; E : b
6. P : $(a+b)+c=a+(x+c)$; x : x; E : b
7. P : $(a+x)+c=a+(x+c)$; x : x; E : b
8. P : $(a+b)+x=a+(b+c)$; x : x; E : c
9. P : $(a+b)+c=a+(b+x)$; x : x; E : c
10. P : $(a+b)+x=a+(b+x)$; x : x; E : c
11. P : $x+c=a+(b+c)$; x : x; E : $a+b$
12. P : $x=a+(b+c)$; x : x; E : $(a+b)+c$
13. P : $(a+b)+c=a+x$; x : x; E : $b+c$
14. P : $(a+b)+c=x$; x : x; E : $a+(b+c)$

It's clearly necessary to say in just which way the formula should match. Jape has two mechanisms which help. The first mechanism, called 'abstraction', finds matches in which every instance of a particular subformula is replaced by the substitution variable: for example, numbers 4, 7, 10, 11, 12, 13 and 14 in the list above. The abstraction mechanism is a kind of higher-order unification. To use this mechanism effectively you nominate the formula which should match E by text-selecting it. It's unnecessary to nominate anything to match x, since it's an OBJECT parameter. You don't provide an argument for P, since it's P that the unification in effect discovers. Given an E, Jape finds the corresponding match which replaces *all* occurrences of E in the consequent by x to give P.

The abstraction mechanism doesn't work at all well if you don't subformula-select an E: an unknown in E's position just makes it impossible to identify subformulae of the problem sequent which should be replaced by x.

Suppose, now, that you text-select *a* and apply the rule. The generated consequent will be Γ |-_P[*x**a*], and the problem sequent $a = b, b = c \vdash (a + b) + c = a + (b + c)$. Jape will try to unify _P[*x**a*] with $(a+b)+c=a+(b+c)$ and, by default, will try to turn the problem formula into a substitution by finding every occurrence of *a* (the replacement formula in the substitution form) in the problem formula and replacing each of them by *x* (the replacement variable in the substitution form). It succeeds, producing the formula $(x+b)+c=x+(b+c)$, which it unifies with _P. That is, of course, does *not* generate a most general unifier of the original pair of formulae, but pragmatically it often finds the one which you want.

If Jape can't find every instance of the replacement formula in the problem formula — for example, if there are unknowns in either or both of them, or if either or both of them contain names over which the theorem being proved is quantified — then it will generate a 'deferred unification' proviso. That's a proviso *formula* UNIFIESWITH *formula*. When those appear it is often because you have done something silly: either you are going through the proof in an unhelpful order, or there is some additional information which could help to clarify the situation and avoid the deferred unifications.

The second mechanism is user-defined substitution matching. The mechanism works together with the WITHSUBSTSEL tactic (see appendix B and chapter 6). The user must text-select all the subformulas in the problem sequent which they want to be considered — both the *a*s, for example, or one of the *b*s. Then the tactic WITHSUBSTSEL("∀ elim") firsts constructs a 'stable substitution form' based on those selections: if both the *a*s it would build $((\_v+b)+c=\_v+(b+c))[\_v\backslash a]$; if the first *b* it would build $((a+\_v)+c=a+(b+c))[\_v\backslash b]$. Then it applies the rule "∀ elim", which causes it to unify the new substitution form with $\_P[x\backslash\_E]$ from the rule: the unification process recognises the stable substitution form as something special, and pattern-matches the two, unifying $\_P$ with its body, *x* with *v* and $\_E$ with its replacement formula (*a* in the first example, *b* in the second). Jape hasn't constructed the most general unifier this time either, but it has justification, because it constructs the one which you asked for.

A.2.9 The tactic store

Theorems are a kind of rule; rules are a kind of tactic. Tactics are programs whose primitive proof steps are the application of rules and/or theorems to problem sequents. The tactic store therefore contains all three in a single soup, indexed by name.

A.3 Forcing semantics for Kripke trees

shouldn't there be something in here?

A.4 Flexible syntax

There are various ways in which you control what formulae can be written down and how they will be interpreted by Jape.

A.4.1 Symbols

The rules of a logic are written in terms of various symbols — logical operators and connectives as well as identifiers like *A*, *B* or *x*. Jape reserves a few special characters — they are double-quote, underscore, opening or closing parenthesis (round bracket), space, newline or tab — which can't be used in symbols.

Jape recognises four distinct kinds of symbol:

identifiers, which are rather like programming language identifiers or mathematical variable names: sequences of characters which start with an alphabetic character and optionally continue with any sequence of alphabetic and/or numeric characters and/or primes (ASCII single quotes). Actually Jape's identifiers can start with any defined sequence of characters, given in a CLASS directive, though some choices — like using commas or arithmetic operator characters — may be more confusing than useful.

unknowns, which are are written as an underscore followed by an identifier.

numbers, which are sequences of numeric characters.

strings, which start with an ASCII double-quote, continue with any sequence of characters not including newline or double-quote, and end with an ASCII double-quote.

special symbols, which are user-defined sequences of characters containing anything other than Jape's reserved characters. Everything else goes, though some choices — like using primes or commas inside a special symbol — may be more confusing than useful. Special symbols are defined in INFIX, PREFIX, POSTFIX, LEFTFIX and OUTFIX directives.

Special symbols are always used as constants, and usually as operators or some kind of brackets, and they are defined implicitly by including them in various kinds of syntactic definitions. Identifiers can also be used as operators or brackets, by using them in just the same kind of syntactic definition.

Numbers, strings and special symbols are always constants of a logic — they stand for themselves and not for anything else. Identifiers can be constants or they can stand for classes of things, like formulae, variables, or whatever[1].

A.4.2 Juxtaposition may need care

Jape's syntax allows juxtaposition of formulae. You may have to use white space (blanks, spaces, newlines) to separate juxtaposed identifiers in some way — *xy*, without spacing, is usually a single identifier, whereas *x y* is usually two juxtaposed identifiers and is equivalent to *x*(*y*), (*x*)*y* or (*x*)(*y*). Similarly, *x*1 is usually a single identifier, whereas *x* 1 is an identifier followed by a number. The syntactic priority of juxtaposition is user-defined.

Usually you can juxtapose special symbols without separation. If you define ¬ to be a special symbol, for example, and you don't also define ¬¬, then ¬¬*x* is read as two ¬ symbols followed by an identifier.

A.4.3 Identifier classes

Typically, you start the definition of a logic by saying what the various identifiers you are going to use 'stand for' or 'range over'. You can say that an identifier ranges over formulae, variables, numbers, strings, or constants; you can say that any identifier which starts with a particular prefix ranges over one of those categories. The directives are

```
<kind> name ...  name
CLASS <kind> name ...  name
```

where <kind> is FORMULA, VARIABLE, CONSTANT, STRING, NUMBER, BAG <kind> or LIST <kind> and *names* is a comma-separated list of identifiers or identifier prefixes.

The unprefixed directives — such as, for example, CONSTANT *map, fold, filter* — define particular identifiers to be of a particular class. They are 'object language' names and when they appear in rules or theorems, they won't be instantiated with anything. But they will unify with unknowns of the same kind.

The prefixed directives — such as, for example, CLASS VARIABLE *x, y, z* — define identifier prefixes which are of a particular kind. Every identifier or unknown which starts with one of those prefixes is of the specified kind and they are all 'general' or 'schematic' or 'meta-language' names: when they appear in rules they are always instantiated with an unknown or an argument of the same kind.

[1] We haven't done the 'whatever' bit, or not very much of it. But we know what we want to do, and we know how to do it ...

Unknowns follow the same rules as identifiers: given the same directives, *_map* would be an unknown that would only unify with constants, *_x33* would be one that unified only with variables.

There isn't, at present, any way of defining a name that is of a kind which is a mixture of primitive kinds (VARIABLE and CONSTANT, for example), but FORMULA includes all the other kinds.

A.4.4 Syntactic hierarchy

Jape has a built-in notion of certain syntactic forms:

- identifiers — like *A, ABC, A1, x, y, y37f, ...*;
- strings — "*anything at all*";
- numbers — 1, 2, 46;
- fully-bracketed formulae — (*formula*);
- substitutions — by default *formula* [*variable list* \ *formula list*], but the order of *variable list* and *formula list* can be reversed if you wish, and you can choose different symbols in place of '[', '\' and ']';
- juxtaposition (like function application in functional languages) — *formula formula*;
- prefix operators — *op formula*;
- postfix operators — *formula op*;
- infix operators — *formula op formula*;
- LEFTFIX bracketing — *bra* f_1 *sep*$_1$ f_2 *sep*$_2$...*sep*$_{n-1}$ f_n;
- OUTFIX bracketing — *bra* f_1 *sep*$_1$ f_2 *sep*$_2$...*sep*$_{n-1}$ f_n ket;
- RIGHTFIX bracketing — f_1 *sep*$_1$ f_2 *sep*$_2$...*sep*$_{n-1}$ f_n ket.

In addition, comma (',') is always a zero-precedence tupling operator, so that tuples — *formula*, *formula*,..., *formula* — are automatically available, with or without brackets.

Both substitution and juxtaposition associate to the left[2]; you define the associativity of infix operators as well as their precedence. Prefix operators, postfix operators, substitution forms, juxtaposition and LEFTFIX bracketed forms all have user-defined precedence.

There are well-known pitfalls in the definition of flexible precedence grammars (but probably no deeper than the holes beneath other forms of grammar). If your definition falls into a hole, Jape may not give much assistance, nor even provide readable parsing diagnostics.

[2] Substitution *has* to associate to the left, but we can imagine right-associative juxtaposition. Another enhancement for the fugure (sigh).

A.4.5 Bracketed formulae

Jape recognises bracketed formulae which use round brackets (parentheses). You can define other kinds of brackets for yourself in LEFTFIX and OUTFIX directives.

OUTFIX directives allow you to define new kinds of opening and closing brackets together with internal punctuation as well. You list the opening bracket, the internal separators and the closing bracket. For example you might write

```
OUTFIX if then else fi
```

and then Jape will recognise

```
if formula then formula else formula fi
```

At present the parser allows you to bring in the closing bracket early, before the list of internal punctuation symbols is exhausted, so that given the OUTFIX directive above any of the following will be recognised as a formula:

```
if fi
if formula fi
if formula then formula fi
if formula then formula else formula fi
```

This is a temporary hack, pending a more flexible parser-generator.

LEFTFIX directives allow you to define opening brackets which have no corresponding closing bracket: you list the syntactic precedence, the opening bracket and the separating symbols. For example you might write

```
LEFTFIX 100 letrec be in
```

and then Jape will recognise formulae of the form

```
letrec formula be formula in formula
```

LEFTFIX formulae are notoriously ambiguous — experts will recognise this as the ‘dangling else’ problem. In effect the final separator has the priority given in the declaration, and Jape will not allow the opening bracket to be preceded by an operator which is of higher priority than that given in the declaration. For example, if you have

```
INFIX 120 ∧
LEFTFIX 100 ∀.
```

then you could write $\forall x.A\wedge B$, but not $C\wedge\forall x.A\wedge B$. That restriction, we hope, eliminates visual ambiguity in the use of bracketed forms without a closing bracket. If you want to write a formula which breaks these rules, you can always use brackets, as for example in $C\wedge(\forall x.A\wedge B)$.

RIGHTFIX is very like LEFTFIX. I have no memory of how it works: see examples to see if it’s ever used.

A.4.6 Substitution and juxtaposition forms

You can define the relative priority of substitution and juxtaposition as well as that of operators. Normally substitution is the highest priority form, and juxtaposition is either the next or follows some prefix/postfix operators, but the choice is yours. You write

```
JUXTFIX precedence
SUBSTFIX precedence
SUBSTFIX precedence bra id1 sep id2 ket
```

The second form of SUBSTFIX allows you to define the syntax of a substitution form, choosing opening bracket (by default '['), separator (by default '/') and closing bracket (by default ']'). At the same time you choose whether the variable list or the formula list comes first, by putting a variable identifier and a formula identifier in place of *id1* and *id2*. Because Jape uses this directive as a definition of some of the symbols, there must always be white space between its various components.

A.4.7 Operator syntax

You define connectives and other such symbols in your logic by defining (unary) PREFIX, (unary) POSTFIX operators and (binary) INFIX operators together with their syntactic precedence; in addition infix operators need an associativity. You write

```
PREFIX precedence op op ...
POSTFIX precedence op op ...
INFIX precedence <associativity> op op ...
INFIXC precedence <associativity> op op ...
```

The *op*s are special symbols, but they may be made up of any characters that you wish — they don't have to be made up of non-alphanumeric characters. <Associativity> is a single character: L means left-associative, so that *A op B op C* means (*A op B*) *op C*; R means right-associative, so that *A op B op C* means *A op* (*B op C*); T means tupling or non-associative, so that *A op B op C* means *A op B op C*. Mixing operators of the same precedence and different associativity may cause confusion, but Jape doesn't prohibit it. The difference between INFIX and INFIXC is to do with the way that formulae are parsed.

As in many modern programming languages, we permit a bracketed operator as a formula, so you can write formulae like (+), (∧), (++) once those symbols have been defined as operators. Operation formulae are parsed as juxtapositions, so that *prefixop formula* is parsed as the juxtaposition (*prefixop) formula*, *formula postfixop* is parsed as (*postfixop*) *formula*, *f1 infixop f2* (INFIX *infixop*) is parsed as (*infixop*) (*f1*,*f2*) and *f1 infixCop f2* (INFIXC *infixCop*)is parsed as (*infixCop*) *f1 f2*; the reverse transcription is made when the formulae are printed out.

A.4.8 Binding structure

Binding structure is defined by pattern: you give some variable names and some formula names and then give a pattern using those names. Any formula or subformula which matches the pattern is automatically a binding formula. Because substitution or unification mustn't be allowed to change the structure of a formula, Jape checks for 'near miss' patterns and complains if it finds them.

The sort of thing you write is

```
BIND x SCOPE P IN ∃x.  P
BIND y SCOPE P IN { y | P }
BIND x, y SCOPE P IN ∀x,y.  P
```

It's normal to use LEFTFIX or OUTFIX patterns, as in these examples, but it isn't obligatory.

The last of the three examples above defines a parallel binding: one that at the same time binds two variable names. At present Jape has no means of defining families of parallel binding formula structures except by exhaustively listing each alternative. And it has no way of defining serial bindings at all.

A.4.9 Sequent structure

At present sequents are always double-sided, and each side is one of

- an optionally-empty comma-separated bag/multiset of formulae — say BAG or BAG FORMULA;
- an optionally-empty comma-separated list/sequence of formulae — say LIST or LIST FORMULA;
- a single formula — say FORMULA

The SEQUENT directive gives you the opportunity to say what can appear on either side, and what the entailment symbol is. You write

```
SEQUENT lhs entailment rhs { AND lhs entailment rhs ...  }
```

You can have as many different kinds of sequent as you wish, provided that their entailment symbols are unique.

A.4.10 Future work

If I knew how to do more, I would. Over to you?

Appendix B

The tactic language

There are no reserved words in the tactic language. It is written in a very restricted sub-dialect of the formula language, without the restriction that the class of every identifier must be pre-declared. When it comes to the application of a rule — the simplest kind of tactic — then the arguments must be stated in the formula language.

Although there aren't any reserved words, there are a lot of tactic language verbs. As in the paragraph language, these are all in upper case. You don't have to avoid these names in the statement of rules and theorems, but if you start using them as tactic parameter names you might confuse things.

Since version 3.0, tactic applications are written in 'curried' style: *verb arg1... argN*, where each argument is bracketed if necessary[1]. If a tactic starts with a verb which isn't one of those listed below, it is treated as an application of a named tactic, rule or conjecture. The verb and arguments of a tactic application are evaluated in the current environment — that means that any names they contain which are parameters of the current tactic, or parameters of the current LET… tactical, are replaced by the corresponding formulae.

When a named tactic is applied to arguments, a new environment is created by zipping together tactic parameters and supplied arguments. If there are too few arguments, the remaining parameters are ignored. When a name is evaluated which doesn't have a value in the current environment, the name itself is taken as the result.

B.1 Tactic verbs

ALT *tactic... tactic*: try each of the tactics in turn, until one is found that succeeds. If none succeeds, ALT fails.

APPLYORRESOLVE *tactic*: rules applied by *tactic* will be tried first normally, where both hypotheses and conclusions must match, and then 'by resolution' where only conclusions need match and extra antecedents are inserted to prove each of the hypotheses. The 'resolution' step requires that the logic have a CUT structure rule.

ASSIGN *variable value*: the named variable, which must be part of the global enviroment (see appendix C) is assigned the given value. Some variables can't be altered once anything has been loaded into the tactic/rule/conjecture store[2].

[1] The older 'uncurried' style, with arguments provided as a bracketed tuple, is now withdrawn. That means we have curried applications and uncurried definitions. One day...

[2] Those variables are properly parameters and for clarity we ought to have a syntax for handling them. Patience, patience.

DO *tactic*: apply *tactic* repeatedly until it fails, then DO succeeds.

EVALUATE *formula* : evaluate one of a fixed number of built-in judgements. Where used, this tactic is explained in one of the distributed encodings; at the time of writing it is used only in the functional programming encoding to evaluate ASSOCEQ(*f1*, *f2*), a judgement that two formulae are identical when rewritten with maximal use of associativity laws. EVALUATE is intended to be the basis, one day, for a mechanism of communication with oracle programs.

EXPLICIT *name*,..., *name*: succeeds if every *name* is a parameter for which an argument has been supplied. I think. Opposite of IMPLICIT below. I think.

FLATTEN *formula*: 'flattens out' all subformulae of *formula* in the conclusion of the current problem sequent by rewriting according to the rules of associativity. It's based on the same machinery as ASSOCEQ; see EVALUATE above and chapter 6.

FOLD *rulename tactic*: Automatically 'folds' collections of rules. See chapter 6 above.
FOLDHYP *pattern tactic*: Automatically 'folds' hypotheses. See chapter 6 above.
IF *tactic*: run *tactic*, but succeed even if it fails.

IMPLICIT *name... nameN*: succeeds if none of of *name*1... *nameN* is a parameter for which an argument has been supplied. I think. Opposite of EXPLICIT above. I think.

JAPE *stuff* : probably deserves a section on its own. Was originally called the 'AdHoc' tactic, and it shows. Usually *stuff* is nothing more than "fail *message*", but can also be "showalert *message*" and "write *message*" and lots more which it would be tedious and embarrassing to list. Likely to change soon and without notice.

LAYOUT *pattern numbers tactic ... tactic*: the way in which a tactic can hide part of a proof. *Pattern* is either () or *string1* or (*string1*, *string2*); *numbers* is a tuple of integers. Run *tactic ... tactic* as a sequence and if that succeeds, mark the subtree it produces so that it is displayed in a special way determined by *pattern* and *numbers*. The tree is displayed either in 'hidden' or 'full' form: by double-clicking on the justfication at the root of the tree the user can force Jape to switch between forms. In 'hidden' form only the antecedents selected by *numbers* are shown, and all others are hidden: antecedents are numbered from left to right, starting with 0, so that if *numbers* is (1,3), for example, only the second and fourth antecedents will be shown. In 'full' form all antecedents are shown. In 'hidden' form the justification shown at the root of the subtree is controlled by *string1*, if included, or by the variable 'hiddenfmt' otherwise; in 'full' form that justificaiton is controlled by *string2*, if included, or by the variable 'unhiddenfmt' otherwise. In either form the controlling string is used as a format string rather as in a very simple kind of C printf, and occurrences of %s in that string are replaced by a summary of the justifications on hidden parts of the subtree; in 'full' format occurrences of %s in the controlling string are replaced by the justification of the node at the root of the subtree.

LETARGSEL *pattern tactic... tactic*: One of the 'guarded tactics' for use in WHEN; also a 'binding tactic' (see below). If the user has made a *single* text selection, parse that text and unify it with *pattern*; then proceed as normal for a binding tactic.
LETCONC *pattern tactic... tactic*: One of the 'guarded tactics' for use in WHEN; also a 'binding tactic' (see below). If the user has formula-selected a conclusion, unify it with *pattern*; then proceed as normal for a binding tactic.

LETCONCFIND *pattern tactic... tactic*: One of the 'guarded tactics' for use in WHEN; also a 'binding tactic' (see below). If the user has made a single text selection *fs* in a conclusion formula *C* so that *C* consists of *f1* followed by *fs* followed by *f2,* if the text *f1* (*fs*) *f2* is a parseable formula, and if the formula (*C*, *f1* (*fs*) *f2*) unifies with *pattern*, then: if *C* is not structurally the same formula as *f1* (*fs*) *f2* proceed as normal for a binding tactic; if they are the same formula, succeed silently, without running the sequence *tactic... tactic*.

LETCONCSUBSTSEL *pattern tactic... tactic*: like LETSUBSTSEL (q.v. below) except that the text-selection must be made in a conclusion (right-hand side) formula.

LETGOAL *pattern tactic... tactic*: One of the ‘guarded tactics’ for use in WHEN; also a ‘binding tactic’ (see below). If the current problem sequent has a single conclusion formula, unify it with *pattern*; then proceed as normal for a binding tactic.

LETHYP *pattern tactic... tactic*: One of the ‘guarded tactics’ for use in WHEN; also a ‘binding tactic’ (see below). If the user has formula-selected a hypothesis formula, unify it with *pattern*; then proceed as normal for a binding tactic.

LETHYPFIND *pattern tactic... tactic*: just like LETCONCFIND (q.v. above), except that the single text-selection must be made in a hypothesis formula

LETHYPSUBSTSEL *pattern tactic... tactic*: like LETSUBSTSEL (q.v. below) except that the text-selection must be made in a hypothesis formula of the current problem sequent.

LETMATCH *pat1 pat2 tactic... tactic*: One of the ‘guarded tactics’ for use in WHEN; also a ‘binding tactic’ (see below). If *pat1* unifies with *pat2*, proceed as normal for a binding tactic.

LETMULTISEL *pattern tactic... tactic*: One of the ‘guarded tactics’ for use in WHEN; also a ‘binding tactic’ (see below). Unify all the user’s text-selections, expressed as a tuple of formulae, with *pattern*; then proceed as normal for a binding tactic.

LETSUBSTSEL *pattern tactic... tactic*: One of the ‘guarded tactics’ for use in WHEN; also a ‘binding tactic’ (see below). If the user has made a number of text selections within a single formula, each an instance of an identical sub-formula, convert that to a substitution (see chapter 1) and unify it with *pattern*; then proceed as normal for a binding tactic.

MAPTERMS *tactic*: if the current problem sequent has a conclusion which is a single formula, try to apply *tactic* (which is probably some sort of rewrite rule) to each of the structural subformulae of that conclusion formula.

MATCH *tactic*: runs *tactic* so that any rules which it applies are required to succeed without visibly changing the unification context — that is, without changing the interpretation of any unknowns in the problem sequent.

PROVE *tactic*: detaches the current goal from the proof tree; tries to prove it; and then plugs in the proof if it’s complete, otherwise fails. A way of ensuring that a tactic builds a subtree with no open tips.

REPLAY *tactic*: run *tactic* but use term equality (up to α-conversion and elimination of substitutions) instead of unification. Used in proof loading, because it seems to be a bit faster than the normal mechanism.

RESOLVE *tactic*: rules and theorems applied by *tactic* will all be applied ‘by resolution’ in which only the conclusions need match and extra antecedents are inserted for each hypothesis. See SIMPLEAPPLY below and APPLYORRESOLVE above.

SAMEPROVISOS *tactic*: rules applied by *tactic* mustn’t add or delete any provisos from the current unification context. Used to be used in proof reloading; may now be obsolete.

SEQ *tactic... tacticN*: run the tactics in sequence. Fail if any of them fails.

SIMPLEAPPLY *tactic*: each of the rules applied by *tactic* will be applied in ‘normal’ style, without using the ‘by resolution’ mechanism.

SKIP: succeed.

THEORYALT : generated internally, and used when a particular mechanism, used only in the functional programming encoding, is searching for and caching rules. It’s all rather horrid and extremely *ad hoc*, and no further details will ever be released.

UNFOLD *rulename tactic*: see FOLD above.

UNFOLDHYP *formula tactic*: see FOLDHYP above.

UNIQUE *tactic*: run *tactic* so that any rules it applies are required to succeed in only one way (i.e. prevents application of those rules from offering the user a choice of alternative matches). Used in proof reloading.

WHEN *guardedtactic.... guardedtactic tactic*: try each of the guarded tactics in turn until one is found whose guard unifies, then run the tactics it guards; if none of the guards succeeds, run the final alternative *tactic*. The guarded tactics must each be one of the LET... variety: see 'guarded and binding tactics' below.

WITHARGSEL *tactic*: run *tactic*, giving it as argument the current text-selection, provided that there is only a single text selection and it parses properly as a formula. Fails if there is a single text selection but it can't be parsed.

WITHCONCSEL *tactic*: if the user has formula-selected a conclusion formula, rules applied by *tactic* must consume it (that is, explicitly match it).

WITHCONTINUATION *tactic1 tactic tactic...*: *tactic1* is run so that its continuation is the sequence *tactic tactic....* Has no effect unless *tactic1* ends with an ALT/THEORYALT; then it will ensure that no alternative of that ALT succeeds unless the sequence *tactic tactic...* succeeds afterwards. Makes alternative choice a little more lazy.

WITHFORMSEL *tactic*: a combination of WITHCONCSEL above and WITHHYPSEL below.

WITHHYPSEL *tactic*: if the user has formula-selected a hypothesis formula, rules applied by *tactic* must consume it (that is, explicitly match it).

WITHSELECTIONS *tactic*: a combination of WITHARGSEL, WITHCONCSEL and WITHHYPSEL.

WITHSUBSTSEL *tactic*: normally used inside a LETSUBSTSEL tactical. The user's text selections have to be entirely within one of the hypotheses or conclusions of the current problem sequent: rewrite that hypothesis or conclusion as a substitution form, based on the text selections given, and then run *tactic*. Fails noisily if the text selections don't describe a substitution in just the right way; fails normally if the substitution is described, but *tactic* fails. Rules applied by *tactic* must consume (i.e. explicitly match) the reconstructed formula.

In addition to all those there are two that can occur inside a *formula* inside a tactic:

ANTIQUOTE (*formula*): everything inside *formula* is liable to 'evaluation' in the current tactic environment, unless it is QUOTEd. Arguments in applications of tactics are treated as if they were ANTIQUOTEd.

QUOTE (*formula*): nothing inside *formula* is liable to 'evaluation' unless it is ANTIQUOTEd.

B.2 The 'current problem sequent', the 'goal' and the 'target'

When you start a proof there is only one problem sequent. When you apply a rule with two antecedents, there are two to choose from. When you are well into a proof, there may be many.

Each time Jape makes a proof step (by application of a rule or in a small number of other ways, mostly to do with the more exotic of the tactics like FIND or FLATTEN, and sometimes caused by the dialogue language) it selects a new problem sequent if the current one is closed, or replaced by a subtree. It always finds the 'next rightmost unclosed tip' and makes that the current problem sequent. The 'next rightmost unclosed tip' is the first one in the fringe of the tree to the right of the current one, or the first one in the fringe if there isn't one to the right of the current one.

The current problem sequent is called the 'goal'; the problem sequent from which we moved to the current one because application of a rule succeeded is called the 'target' (not a very good name, 'target', but that's the way it is).

B.3 Guarded and binding tactics

Jape's tactic language is 'eager' — whether it should be so continues to be a matter of debate — with backtracking on failure. If a tactic fails, then the enclosing tactic either fails, or if it is an ALT, tries another alternative starting from the state in which it first applied the sub-tactic that failed. That sort of backtracking search is fine sometimes, but not always. It can be modified — slightly — by WITHCONTINUATION.

The WHEN tactical takes 'guarded tactics' and applies them carefully, accepting the result of the first one of them whose guard matches. Note that the whole guarded tactic may fail after its guard has matched, and in that case WHEN won't backtrack, it will simply fail.

Each of the guarded tacticals — they are all called LET... — takes a *pattern* and a *tactic* sequence. The *pattern* is matched against something by unification: if the unification succeeds then the environment is updated to reflect that unification. Roughly speaking you can assume that unknowns in *pattern* will be added to the environment as parameters corresponding to the stuff they unified with, and if they are used again in the tactic sequence, they will be replaced by that same stuff. You don't have to worry that the unknowns you use might already appear in the unification context: Jape invents new ones, based on the ones you use, so that the effects of a successful binding tactic never leak into the unification context used in proof steps.

Appendix C

The command language, environment variables and the default environment

C.1 The command language

Jape communicates with its graphical interface in a language of ‘words’, space separated unless they are enquoted "...". You may want to attach commands to buttons, you may want to include commands as entries in TACTICPANELS, and you can type commands into the system — on the Mac into the Text Command box, on X into the command window — so here goes with a description. I’ve divided it into two: the ones you might want to use, and the arcana.

Note that the language described here is *ad hoc* and subject to change without notice or any sign of visible regret on the part of the implementors. Be warned.

C.1.1 Commands you might want to use

addnewconjecture *panelname conjecture*: this command is sent by the New. . . button in a conjecture panel, after the user has typed the conjecture into a dialogue box.

apply *tacticexpression*: this command is used a lot: it is the way that menus and panels apply tactics. Don’t forget that a rule name is a tactic expression.

assign *name value*: the way that Jape’s environment variables (q.v. below) are given values.

backtrack: command sent by the Backtrack button in the Edit menu.

closedbugfile: close the top dbug file on the stack of such files (see createdbugfile below); redirect diagnostic output to the file below, or to the console if the stack is empty.

collapse: the way that the Hide/Show detail entry in the Edit menu does its work.

createdbugfile: create a file, using the normal file selection dialogue, and redirect diagnostic input into it. There’s a stack of these dbug files.

layout: has the same effect as double-clicking on the justification of the selected sequent.

lemma *panelname conjecture*: synonym for addnewconjecture above.

print *filename*: generates a listing of the currently-focussed proof in *filename*, in a form suitable for LaTeX processing.

proof finished: (two words) how the Done entry (on the Mac) and the proof finished entry (on UNIX) in the Edit menu does its work.

prove *conjecturename*: how the Prove button in a conjecture panel does its work.

prune: how the Prune entry in the Edit menu does its work.

QUIT: kill the proof engine, after asking whether the user wants to save any proofs.

redo: how the Redo entry in the Edit menu does its work.

refreshdisplay: clear the currently-focussed proof window and redraw the proof it contains.

reset: how the Reset entry (on the Mac) and the ?? entry (on UNIX) in the Edit menu do their work. On the Mac the Reset entry can be greyed-out even though some syntax definitions have been accepted: in that case typing the reset command to a Text Command window can be helpful.

reset;reload: (no spaces, all one 'word' with a semicolon in the middle!) how the Load New Theory entry in the File menu does its work.

showproof *conjecturename*: how the Show Proof button on a conjecture panel does its work; opens a window with a proof of *conjecturename* in it, if there is one in the proof store.

saveengine *filename*: saves the current proof engine, with all its settings, in a file. Useful for creating pre-initialised engines

steps: display the value of the internal variable 'timestotry' in an alert dialogue. See steps *n* below.

steps *n*: set the value of the internal variable 'timestotry' to the integer *n*. This variable will, in the near future, be part of the default environment (q.v. below). The value of the variable controls the number of steps that Jape will allow in a single tactic application before failing with the message "Time ran out".

tellinterface *variablename word word....*: send the current value of the variable *variablename* to the interface, prefixed by the command *word.....*

undo: the way that the Undo entry in the Edit menu and/or the Undo key do their work.

unify *formulae*: Unify the given formulae and all of the user's text-selections. The way that the Unify button does its work.

use *filenames*: open each of the files named, read and execute the Japeish text they contain. The way that proof files are loaded and a new encodings or a modification to the current encoding is interpreted

version: display the current version information of the Jape engine in an alert dialogue.

Arcana

cd *path*: changes the default directory used by the proof engine. Only works in the UNIX implementations; don't use if you don't know what it does.

closeproof *n*: absolutely not to be used.

DRAGQUERY: part of the drag-and-drop interface; don't use it.

DROPCOMMAND: part of the drag-and-drop interface; don't use it.

fonts_reset: command sent by the graphics interface when its fonts are altered. Triggers all sorts of cache mangling, but otherwise harmless.

HITCOMMAND *comm*: absolutely not to be used.

NOHITCOMMAND *comm*: absolutely not to be used.

profile [on | off | reset | report *filename*]: one of the mechanisms with which we debug the proof engine. Only works in specially-instrumented proof engines under UNIX.

quit: kill the proof engine without asking any questions.

saveproofs *word*: absolutely not to be used.

setfocus *n*: absolutely not to be used.

showfile *filename*: possibly obsolete.
emptyword: ignored.

C.2 Variables and the default environment

Jape has a number of 'environment variables' which can be used to modify its behaviour, and can currently be set by the ASSIGN tactic, by the assign command and by INITIALISE, RADIOBUTTON and CHECKBOX directives in the paragraph language. Some of them are of general use; some are horrid debugging switches of interest only to the implementors. Variables can be set from menus and panels: see the various files like 'autoselect_entry' which are distributed with Jape and put entries in the Edit menu.

C.2.1 Useful variables

Variables whose default value is marked with an asterisk are parameters: their value can be altered only if the rule/tactic/conjecture store is empty.

name	*values*	*default value*	*effect*
applyconjectures	true, false	false	when true, allow conjectures (unproved theorems) to be applied as rules.
autoAdditiveLeft	true, false	false*	when true, any rule whose consequent and antecedents all have a bag on their left-hand sides is augmented by the addition of a bag variable (e.g. Γ) to the left-hand side of every consequent and antecedent which doesn't already have one.
autoAdditiveRight	true, false	false*	as autoAdditiveLeft, except that it applies to right-hand sides
autoselect	true, false	false	when true, select the conclusion of the current problem sequent each time a proof is displayed.
collapsedfmt	any string	"[%s...]"	the string used to control the way that a justification is displayed for a subtree shown in 'collapsed' form — for example, after using Hide/Show Subproof on an uncollapsed subtree.
displaystyle	box, tree	tree	selects the display mechanism used to show a proof. Each proof may have an individual setting of this variable. When a new proof is started, its displaystyle is taken from the currently-focussed proof.
hiddenfmt	any string	"{%s}"	the string used to control the way that a justification is displayed for a subtree produced by the layout tactical in 'hidden' form. This string is over-ridden if *string1* is provided in the layout tactical.
hidecut	true, false	true	hide the application of cut rules in box display.
hidehyp	true, false	true	hide the application of identity rules in box display.
interpretpredicates	true, false	false*	on instantiating a rule, interpret juxtapositions as predicate applications; translate them into substitution forms, add new object parameters and invisible provisos.
outermostbox	true, false	true	when true, draw an outermost box in box display when proving a conjecture which has hypothesis formulae.
showallproofsteps	true, false	true	(misnamed — should be showhiddenproofsteps) when true, show

C.2.2 Adding your own variables

You can invent your own environment variables and assign them values. In particular you can define a variable in a RADIOBUTTON or CHECKBOX directive, give the range of possible values that it can take, and allow the user to control that variable. See, for example, the way that the functional programming encoding controls searching by using variables whose values are the names of tactics.

There are at present few ways in which the value of a variable can be used, once set. But watch this space for developments, including at least a form of case-expression value analysis in the tactic language.

C.2.3 Debugging variables

Jape has a number of debugging variables. Setting any of them to true makes it print lots of stuff on the console (which on the Mac is hidden, and needs secret knowledge to find). The variables currently used are

applydebug, bindingdebug, factsdebug, FINDdebug, FOLDdebug, matchdebug, prooftreedebug, rewritedebug, substdebug, symboldebug, tactictracing, thingdebug, unifydebug, eqalphadebug, varbindingsdebug

In this manual we don't explain or admit what these variables do or don't do or how best to use them. Good luck to you if you try to find out.