Roll your own Jape logic

Encoding logics for the Jape proof calculator

Richard Bornat and Bernard Sufrin

September 1997 (Jape version 3.2)

Contents

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 we 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 we 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 the various user manuals which we distribute with the example inference systems that we have already encoded.

This manual describes the state of the system in September 1997 (version v3.2f of the proof engine and version 3.2e of the logic encodings). The graphical illustrations are taken from the then current MacOS graphical interface (version 3.2b).

Improvements since 3.0

Since version 3.0 we have implemented proper sequent syntax (Γs and Δs and that sort of thing), plus *autoAdditive* variables for those who want to be able to state rules and theorems natural-deduction style. There is an option to allow interpretation of predicate notation, and we now automatically insert essential but invisible provisos into the statement of some rules and theorems. We've included a drag-and-drop gesture to support multiplicative rules and certain kinds of weakening rules.

Outstanding problems still awaiting a fix

Some of the examples in this manual reveal deficiences with the current Jape system. If we stopped writing in order to improve the system each time we described one of its warts we would never finish. But any experimental system will always, at its frontier, have features which you might wish it didn't. What we list here are the things that we more or less understand how to fix. By putting the statement of problems here, we avoid repetitive apologies in the body of the manual.

- 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 now guard against circularities in proofs).

- Jape can't yet handle sequents in which a side is an optional single formula.
- Jape has no treatment of definitional equality (syntactic equivalence), so you have to handle it with additional rules.
- You can't prove derived rules with antecedents, despite the fact that a theorem is just a kind of derived rule in Jape.

Serious problems which won't be fixed soon

There are things about Jape which are wrong, but which we can't fix in a reasonable time.

- Jape cannot yet distinguish between classes of formulæ in problem sequents, except for variables, constants, numbers and strings.
- Jape has a long-standing problem in that it can't encode 'families of rules'. We have used some ingenuity to allow you to encode such families by finite collections of slightly different rules, but the problem is very unlikely to be fixed without a large research project.
- Jape's parser generator ought to make it possible to distinguish between lots of user-defined syntactic categories. It isn't clear how a parser generator can be both simple enough and powerful enough. We intend to find out, but once again it would have to be part of a large research project.

Future developments

The field of logical systems is huge and growing. Jape doesn't cover it all, and no doubt people can invent logics faster than we can devise ways of encoding them. Nevertheless, we understand how to make Jape do much more than it can at present, and we are developing it constantly. At the time of writing, projects on the immediate horizon include modal logic, linear logic and improved support for equational reasoning as well as work to improve the graphical interface still further.

We are keen to hear from our users about the things that they want Jape to do, so that we can continue to develop it in practically useful directions.

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. Put like that, it may seem rather complicated, but it's really very simple. We decided on unification rather than one-way pattern-matching because it allows us to use Jape as a Prolog-style calculator, solving 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 organise 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 formulæ, 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 Backward proof

Jape always, always, always works backwards, even when its display mechanism tries to produce the illusion that it is working forwards. It always works with trees – Gentzen trees – of sequents, even when its display mechanism is trying to produce the illusion that it is working with Fitch boxes, or something else[1]. If you haven't seen a Gentzen tree, you can learn how Jape handles them if you use one of the distributed logics that uses tree display mode, or you can switch to tree display mode in one of the distributed logics that allows it – for example the single-conclusion sequent calculus encoding.

1 Currently we have a box display which approximates Fitch boxes. We have the beginnings of a more attractive treatment of equational chaining proofs (see, for example, chapter 5) and we dream of a kind of Fitchery for linear logic, and more.

1.3 Inference rule matching: instantiation and unification

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

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

Its rendition in Japeish (see the file SCS_rules.j) is a straightforward linearisation of the original[1]

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

Our interpretation of the rule is that it describes a node in a proof tree by pattern: the consequent at that node has a collection of formulæ on its left-hand side, one of which matches $A \vee B$ for some pair of formulæ *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 which is Γ together with *B*.

Jape makes proofs by replacing tips of the tree with nodes generated from instances of rules, or sometimes with subtrees generated by instances of several rules. Given a rule and a tip (leaf node) in a Gentzen tree, Jape first *instantiates* the rule, generating a version in which the schematic names – in the rule above they are Γ, *A*, *B* and *C* – are replaced by fresh unknowns; then it *unifies* the consequent of the newly-instantiated rule with the sequent at the tip. Typically a rule might unify in more than one way – there might be more than one left-hand side formula, for example, which could unify with $A \vee B$ – and in that case Jape requires the user to decide between the possibilities, either by selecting a preferred principal formula[2] beforehand, or by choosing from a menu of possibilities afterwards.

In the rule above there are various symbols of the logic as well as the schematic identifiers Γ, *A*, *B* and *C*: there is the connective ∨ and there are the punctuation marks ⊢ and comma. Although ∨ is in a sense an identifier, it plays a fixed syntactic rôle in the logic and in the rule; it would be wrong to instantiate it with an unknown. Some other identifiers might be non-schematic: there might 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 constant 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, generated during the instantiation step, may appear in the proof[3].

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
```

1 Earlier versions of Jape required that the segment variable symbol Γ be omitted. Now it may optionally be omitted if every rule is 'additive' in the linear logic sense: for examples see chapter 4.

2 A *principal formula* (sometimes *principal term*) in a rule in a sequent calculus is the one which the rule consumes, or works on. That's the formula which matches $A \vee B$ in this example.

3 This is a *strength* of Jape, not a weakness: we don't require the user to decide prematurely on the identity of those unknowns.

Given a problem sequent $X, Y \rightarrow \neg X \vdash \neg Y$, Jape unifies $\_\Gamma$ with $X, Y \rightarrow \neg X$ and $\_A$ with Y; then it generates the antecedent $X, Y \rightarrow \neg X, Y \vdash \_B \wedge \neg\_B$. If it is given the argument formula X to use instead of $\_B$, it will generate the antecedent $X, Y \rightarrow \neg X, Y \vdash X \wedge \neg X$. In many cases an argument supplied to the application of a rule can prevent a startling proliferation of unknowns in a proof.

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

$$\frac{\Gamma \vdash \exists x.A \quad \Gamma, A[x \setminus c] \vdash C}{\Gamma \vdash C} \text{(FRESH } c, c \text{ NOTIN } \exists x.A) \;\; \exists - E$$

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 are used in other circumstances – in particular, see the discussion of substitution unification below.

1.4 Explicit provisos

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*[1] and violated if it does. 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 formulæ and CONCFRESH for the right-hand side formulæ.

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.

1.5 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 kind of derived rule; if the state variable *applyconjectures* is set to *true* then unproved conjectures can be applied as if they were proved.

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
```

[1] Strictly if it *cannot* appear free, 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* is not automatically nor trivially satisfied.

[2] Conjecture, theorem, which do we mean? There is no split personality here, but there are two Jape authors. One wants to emphasise that it's a conjecture till it's proved; the other wants to emphasise that it is theorems, after all, that you are trying to prove.

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".

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[1]. 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 formulæ; they may therefore be instantiated in the same way as a rule, replacing identifiers in the theorem sequent by unknowns or arbitrary formulæ. 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 $A[x \setminus E]$ can't be replaced by *A* unless 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 are no unknowns in the base sequent of the proof, so that *x* cannot be smuggled into the statement of the theorem we are proving, 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 $A[x \setminus E]$ 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.

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 formulæ to completely match a problem sequent. The theorem "P→(Q→R) ⊦ (P→Q)→(P→R)", for example, matches

[1] Jape sometimes adds invisible provisos which it deduces from the binding structure of the formulæ in the conjecture. Those provisos can be made visible: see 'invisible provisos' below, and also appendix C.

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 formulæ and to set about proving them. That step is justified if the logic contains a ‘cut’ rule which enables you to move formulæ from left- to right-hand side. 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).

1.6 Substitution forms and unification

Jape uses explicit substitution forms – $A[x \setminus c]$, $B[x,y,z \setminus E,F,G]$ – 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 now a mechanism which allows you to write rules and theorems in terms of predicate formulæ. Jape will translate into substitution notation, and construct automatically the additional parameters and fussy NOTIN provisos that it needs – 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

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

then we generate a single antecedent $x > y \vdash (z > y)[z \setminus \_E]$, which immediately simplifies to $x > y \vdash \_E > y$ provided that we know that z and y are necessarily distinct[1].

Much more interesting is what happens when a rule contains an explicit substitution form such as $A[x \setminus E]$ in its *consequent*. When the rule is applied Jape must unify that substitution form – or rather, its instantiated form which in general will be $\_A[\_x \setminus \_E]$ – 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.

i It simplifies substitution forms whenever possible, in order to avoid the problem.

ii 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.

iii If the user provides an argument formula F in place of parameter E, the instantiated form will be $\_A[\_x \setminus F]$; 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 $B'[\_x \setminus F]$ which simplifies to B in presence of the proviso $\_x$ NOTIN B; if successful it will unify $\_A$ with B' in a context

[1] 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.

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 $\_A[z \setminus F]$ where z is a fresh variable; the formula $B'[z \setminus F]$ is easier to construct and to simplify, and the proviso z NOTIN B is easier to check[1].

iv If the user text-selects instances of a sub-formula F of B, then the logic encoding can employ the WITHSUBSTSEL tactical – see chapter 5 and appendix B – to calculate B' by replacing just those instances of F by a fresh unknown $\_y$; $B'[\_y \setminus F]$ necessarily simplifies to B given the proviso $\_y$ NOTIN B; then Jape will unify $\_A$ 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.

v If all else fails, Jape can generate a proviso $\_A[\_x \setminus \_E]$ UNIFIESWITH B, and await developments.

If Jape has to unify two substitution forms which have identical variable lists then it unifies the base formulae and the substituted formulæ. For example, it can unify $A[x,y \setminus A1, A2]$ with $B[x,y \setminus B1, B2]$ by unifying A with B, $A1$ with $B1$, $A2$ with $B2$. This happens rarely and sometimes 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 $A[x \setminus A1]$ with $B[x,y \setminus B1, B2]$ it will try to construct A' such that $A'[y \setminus B2]$ simplifies to A, and then unify $A'[x,y \setminus A1, B2]$ with $B[x,y \setminus B1, B2]$. 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 5 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, we think, inevitable, though we continue to search for ways round the difficulty.

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[2]. 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*.

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 $\forall x.P[v \setminus x]$ 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

[1] 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.

[2] 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 ...

implemented a mechanism which interprets predicate notation, translating it into substitution notation. 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 the necessary proviso. 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 4 and 5 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 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.

1.7 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 $\forall z.A[x \setminus z]$ with $\forall z.B[y \setminus z]$, 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 $(\forall x.A)[x \setminus E]$ always simplifies to $\forall x.A$ and, provided that x doesn't appear free in F, $(\forall x.A)[y \setminus F]$ will simplify to $\forall x.(A[y \setminus F])$; in other circumstances it will simplify to by α-conversion to $\forall z.(A[x,y \setminus z,F])$ together with the proviso z NOTIN A, where z is a fresh variable.

1.8 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.

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 formulæ 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.

1.9 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 – varies between the interfaces, and is not discussed here.

- A user can *select* a formula in a sequent. If a rule is then applied, Jape requires that the selected formula is a principal formula in the rule. Thus, for example, if you select the hypothesis $X \vee Y$ in the sequent $U \vee V, X \vee Y, U \rightarrow X \vdash U \wedge X$ 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 $X \vee Y$ in the sequent, Γ matches $U \vee V, U \rightarrow X$ and, of course, C matches $U \wedge X$. If a tactic is applied it 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 formulæ mentioned in that proviso – unknown segment variables like $\_\Gamma$ or $\_\Delta 1$ – will highlight as the formula is dragged across them.
- A user can *text-select* part – typically, a sub-formula – of a formula in a sequent. If a rule is applied, the text selection is provided as an argument to the application. If a tactic is applied, it can test for text selection, discover the text selected, 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.

1.10 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 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:

- the assumptions – left-hand side formulæ – 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 formulæ, 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 (*axiom, hypothesis*) rules, and it is made to support some forms of forward reasoning (see chapter 4) 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. We are working on the problem. For the moment we provide some assistance by making the underlying tree structure more evident when the user selects an assumption or a conclusion: Jape will 'grey out' lines in the box display which are irrelevant because they are not hierarchically related in the underlying tree.

1.11 Using Jape interactively

Jape normally starts up 'empty', with no theory loaded, although it is possible to save a version of Jape into which a theory has been loaded (using the saveengine command of appendix C: for details of its use see the technical documentation about your version of Jape).

You can load a new theory into Jape by using a command from the File menu (Load New Theory, or something like that). At any time you can add additional bits of Japeish to the brew, by using another of the commands on the File menu (Open Logic file, or something like that). 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 (distributed with Jape in the file MCS.jt and the files it references).

In the distributed files we have described the syntactic rôle of the ≡ connective and included inference rules and conjectures which make use of it, inherited from MacLogic. We haven't included that connective in this discussion.

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

We have encoded a fairly standard version of the sequent calculus. By making our left- and right-hand sides *bags* (aka multisets) of formulæ 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.

axiom

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

Introduction to the right of the turnstile (⊢... rules)

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

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

Introduction to the left of the turnstile (...⊢ rules)

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

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

Structural rules

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

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ \mathit{contract}\vdash \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ \vdash\mathit{contract}$$

The ⊢, →⊢, ∨⊢ and *cut* rules don't split their left- or right-hand side contexts – they are additive rather than 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 (see chapter 3, however, for a discussion of Jape's treatment of context-splitting rules). In practice, the fact that the *axiom* rule ignores unnecessary hypothesis and conclusion formulæ makes context-splitting on either side unnecessary.

Because Jape interprets predicate notation as shorthand for substitution, the *actual* quantifier rules use substitution. These are the rules which Jape employs, translating those above on input:

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

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

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.

2.2 Preliminaries – fonts and syntax

Our presentation uses the Konstanz font encoding, due to Roy Dyckhoff. 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 formulæ[1].

```
FONTS "Konstanz"

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, a name which stands for a variable, or a similar unknown.

The syntax of sequent calculus formulæ is defined as follows:

```
LEFTFIX     20      ∀ .
LEFTFIX     20      ∃ .

INFIX       100 L   ≡
INFIX       110 R   →
INFIX       150 L
INFIX       160 L   ∨

PREFIX      200     ¬

JUXTFIX     300
SUBSTFIX    400
```

1 Note that now there are no commas in these lists of identifier prefixes: in general we have eliminated use of comma as a separator in the paragraph language.

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:

- ¬ *prim*, where *prim* is an atomic formula, a substitution or a juxtaposition (see appendix A);
- *f1* ∨ *f2*, *f1* ∧ *f2* or *f1* → *f2* with the interpretation that ∨ 'operators' have priority over ∧, and both have priority over →;
- ∀ *f1* . *f2* and ∃ *f1* . *f2*.

Note that the leftfix patterns don't constrain you to write ∀ *variable* . *formula*: it is only by defining binding structures and by using variable identifiers in the right way in rule definitions that you can be sure to constrain the use of these structures.

Because there is no closing bracket, formulæ 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 ∀ *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 ∀*x*,y.*E*, ∀*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 ∀*x*.*E*, another for ∀*x*,y.*E*, another for ∀*x*,*y*,*z*.*E* and so on.

Our sequents have bags of formulæ on either side:

```
SEQUENT IS BAG ⊢ BAG
```

2.3 Encoding the inference rules

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 ⊢, 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 formulæ provided via text-selection and/or tactics.

Because we want to write our rules, and prove our conjectures, using predicate notation, we set a global parameter[1]

```
INITIALISE interpretpredicates true
```

With that setting, the rules can be defined directly:

```
RULE   axiom(A)          INFER Γ,A ⊢ A,Δ

RULE   "⊢"               FROM Γ ⊢ A,Δ AND Γ ⊢ B,Δ       INFER Γ ⊢ AB,Δ
RULE   "⊢"               FROM Γ,A, B ⊢ Δ                INFER Γ,AB ⊢ 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 declared to Jape with their proper rôles:

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

2.4 Automatic application of rules

It is possible to require Jape to try to apply 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
```

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

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 formulæ 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   BC      IS "⊢"
CONCHIT   B∨C     IS "⊢∨"
CONCHIT   B→C     IS "⊢→"
CONCHIT   ¬B      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    AB ⊢    IS "⊢"
HYPHIT    ¬A ⊢    IS "¬⊢"
HYPHIT    ∀x.A ⊢  IS "∀⊢"
HYPHIT    ∃x.A ⊢  IS "∃⊢"
```

2.6 Menus

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

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 "∃⊦"

        SEPARATOR

        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 ⊦ AB
        ...

    END
```

then it would have produced exactly the same menu.

2.7 Conjectures

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.

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.

In the distributed version of the multiple-conclusion sequent calculus, two conjectures are stated in this way:

```
    THEOREM     modusponens      IS A, A→B ⊦ B
    THEOREM     contradiction    IS A, ¬A ⊦
```

Note that because this logic includes 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 THEOREMS directive allows you to state a collection of conjectures, each of which will be named by its sequent, in a very economical way. Part of the THEOREMS directive in MCS+SCS_problems.j is:

```
    THEOREMS PropositionalProblems ARE
              P→(Q→R)                                    ⊦ (P→Q)→(P→R)
        AND   P→(Q→R), Q                                 ⊦ P→R
        AND   R→S                                        ⊦ (P→R) → (P→S)
        AND   P→(P→Q)                                    ⊦ P→Q

        ...

        AND   ∀x.¬Q(x), P→(∀x.Q(x))                      ⊦ ¬P
        AND WHERE x NOTIN P INFER
              P∨¬P, ∀x.P→Q(x), ∀x. ¬P→Q(x)               ⊦ ∀x.Q(x)
        AND   R∨¬R, ∀x.R→S(x), ∀x. ¬R→S(x)               ⊦ ∀x.S(x)
        AND   ∀x.P(x)→Q(x), ∀x.Q(x)→R(x)                 ⊦ ∀x.P(x)→R(x)
        AND   ∀x.P(x)→R(x), ∀x.Q(x)→ ¬R(x)               ⊦ ∀x.(P(x)→¬Q(x)) (Q(x)→¬P(x))
        AND   S(m,n), ∀x.P(x) → ¬S(x,n)                  ⊦ ¬P(m)

        ...
```

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 formulæ, 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 The Conjectures panel

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. The distributed version goes as follows. In MCS.jt we have:

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

and in sequent_problems.j we have

```
     CONJECTUREPANEL "Conjectures"
      THEOREMS PropositionalProblems ARE
         P→(Q→R)         ⊦ (P→Q)→(P→R)
      AND P→(Q→R), Q           ⊦ P→R
      AND R→S           ⊦ (P→R) → (P→S)

      ...

      AND ∃y.P          ⊦ ∀y.P
      END
     END
```

Note that additions to a panel or a menu can be made in several lumps: that is, panels and menus can be built up in by disjoint declarations.

2.9 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:

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

2.10 A very small example

Here is the progress of a proof of Pierce’s law in this encoding. As intuitionists, we offer no explanation of the pheonemon. Those who believe, believe.

$$((P\to Q)\to P)\to P$$

$$\dfrac{((P\to Q)\to P)\vdash P}{((P\to Q)\to P)\to P}\;{\scriptstyle \vdash\to}$$

$$\dfrac{\dfrac{\vdash(P\to Q),P \qquad \dfrac{}{P\vdash P}\,{\scriptstyle \text{axiom}}}{((P\to Q)\to P)\vdash P}\;{\scriptstyle \to\vdash}}{((P\to Q)\to P)\to P}\;{\scriptstyle \vdash\to}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P\vdash Q,P}\,{\scriptstyle \text{axiom}}}{\vdash(P\to Q),P}\,{\scriptstyle \vdash\to} \qquad \dfrac{}{P\vdash P}\,{\scriptstyle \text{axiom}}}{((P\to Q)\to P)\vdash P}\;{\scriptstyle \to\vdash}}{((P\to Q)\to P)\to P}\;{\scriptstyle \vdash\to}$$

Chapter 3

Variations on the Sequent Calculus

The sequent calculus of chapter 2 is only one of very many possible variants. In this chapter we discuss the way in which we 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 MSC_LF.j redefines the quantifier rules to allow a more careful treatment of variables[2]. The new rules are

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

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

The intention is that a variable *c* is ‘inscope’ if there is an assumption var *c*; a formula is inscope if its free components are inscope. Note that there is nothing in Jape which demands that we use these words nor this technique: it’s up to the logic decoder.

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

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

and a structural induction to handle formulæ, automatically applied whenever there is an open tip:

```
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 ⊢ AB 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

AUTOMATCH "inscope"
```

Encoding of the rules is then straightforward:

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 at the time of writing, but it will do so Real Soon Now.

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 ∀x.P(x) ⊢ ∃x.P(x), because the proof would require that there be some *m* such that *P*(*m*).

```
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
```

We 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[1].

In order to make these judgements side condidtions we use Jape's LAYOUT tactical: it allows us 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 we would like to display the first antecedent proof (numbered 0) and hide the second (numbered 1); in either case we 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 we put into the menu

```
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"
```

We get all this machinery simply 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, we can show the progress of a proof in which the variable rules are obeyed:

$$\exists x.P(x)Q(x) \ \vdash\ \exists x.Q(x)P(x)$$

$$\dfrac{\text{var } m, P(m)Q(m) \ \vdash \exists x.Q(x)P(x)}{\exists x.P(x)Q(x) \ \vdash \exists x.Q(x)P(x)}\ \exists\vdash$$

$$\dfrac{\dfrac{\text{var } m, P(m)Q(m) \ \vdash Q(m)P(m)}{\text{var } m, P(m)Q(m) \ \vdash \exists x.Q(x)P(x)}\ \vdash\exists}{\exists x.P(x)Q(x) \ \vdash \exists x.Q(x)P(x)}\ \exists\vdash$$

1 I guess they could be provisos one day.

Note that one antecedent of the final step isn't shown. We can see the full display by double-clicking on the justification of that step:

$$\dfrac{\dfrac{\text{var } m, P(m)Q(m) \ \vdash Q(m)P(m) \qquad \dfrac{}{\text{var } m, P(m)Q(m) \ \vdash m \text{ inscope}}\ \text{inscope'0}}{\text{var } m, P(m)Q(m) \ \vdash \exists x.Q(x)P(x)}\ [\vdash\exists]}{\exists x.P(x)Q(x) \ \vdash \exists x.Q(x)P(x)}\ \exists\vdash$$

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. The rest of the proof is straightforward.

Next, the progress of an attempt to prove ∀x.P(x) ⊢ ∃x.P(x), which isn't a theorem in this logic (though it is one in the logic of chapter 2):

$$\forall x.P(x) \vdash \exists x.P(x)$$

$$\dfrac{P(\_B) \vdash \exists x.P(x) \qquad \_B \text{ inscope}}{\forall x.P(x) \vdash \exists x.P(x)}\ \forall\vdash$$

$$\dfrac{\dfrac{P(\_B) \vdash P(\_B1) \qquad P(\_B) \vdash \_B1 \text{ inscope}}{P(\_B) \vdash \exists x.P(x)}\ \vdash\exists \qquad \_B \text{ inscope}}{\forall x.P(x) \vdash \exists x.P(x)}\ \forall\vdash$$

$$\dfrac{\dfrac{\dfrac{}{P(\_B) \vdash P(\_B)}\ \text{axiom} \qquad P(\_B) \vdash \_B \text{ inscope}}{P(\_B) \vdash \exists x.P(x)}\ \vdash\exists \qquad \_B \text{ inscope}}{\forall x.P(x) \vdash \exists x.P(x)}\ \forall\vdash$$

It doesn't matter what we unify with *_B*: the side conditions won't go away, and we don't have a theorem.

Caveat

A deficiency of Jape at present is that it has only one class of formula, but the contexts which will be built up in this encoding include logical formulæ and extra-logical remarks like var *c*. That would permit you, if you were actively incautious, to try to prove nonsense like var *m* ∧ var *n*. We'll fix the problem as soon as possible, but don't hold your breath ...

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:

$$\dfrac{\Gamma, A \vdash}{\Gamma \vdash \neg A, \Delta}\ \vdash\neg \qquad\qquad \dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B, \Delta}\ \vdash\to$$

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

These are defined directly in the file IMCS.j, which you can load after MCS.jt (and before or after MCS_LF.j, if you wish):

```
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, for which intuitionists may thank goodness:

$$((P\rightarrow Q)\rightarrow P)\rightarrow P$$

$$\cfrac{((P\rightarrow Q)\rightarrow P)\vdash P}{((P\rightarrow Q)\rightarrow P)\rightarrow P}\,\vdash\rightarrow$$

$$\cfrac{\cfrac{(P\rightarrow Q) \qquad \cfrac{}{P\vdash P}\,\text{axiom}}{((P\rightarrow Q)\rightarrow P)\vdash P}\,\rightarrow\vdash}{((P\rightarrow Q)\rightarrow P)\rightarrow P}\,\vdash\rightarrow$$

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; we have chosen at the same time to use an axiom rule which doesn't ignore unmatched conclusions:

$$\frac{}{A\vdash A}\,\textit{axiom}$$

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

$$\frac{\Gamma\vdash B,\Delta \quad \Gamma',B\vdash\Delta'}{\Gamma,\Gamma'\vdash\Delta,\Delta'}\,\textit{cut}$$

These rules are defined in MMCS.j, ready to be loaded after MCS.jt:

```
RULE  axiom(A)                                INFER A ⊢ A
RULE  "⊢"      FROM Γ ⊢ A,Δ AND Γ' ⊢ B,Δ'     INFER Γ,Γ' ⊢ AB,Δ,Δ'
RULE  "∨⊢"     FROM Γ,A ⊢ Δ AND Γ',B ⊢ Δ'     INFER Γ,Γ',A∨B ⊢ Δ,Δ'
RULE  "→⊢"     FROM Γ ⊢ A,Δ AND Γ',B ⊢ Δ'     INFER Γ,Γ',A→B ⊢ Δ,Δ'

RULE  cut(A)   FROM Γ ⊢ A,Δ AND Γ',A ⊢ Δ'     INFER Γ,Γ' ⊢ Δ,Δ'
```

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

```
CUT   cut
```

When we use a multiplicative rule, the left and right contexts split. Jape automatically records this fact in a UNIFIESWITH proviso:

$$P\rightarrow(Q\rightarrow R),Q\vdash P\rightarrow R$$

$$\cfrac{P\rightarrow(Q\rightarrow R),Q,P\vdash R}{P\rightarrow(Q\rightarrow R),Q\vdash P\rightarrow R}\,\vdash\rightarrow$$

$$\cfrac{\cfrac{\_\Gamma 1\vdash P,\_\Delta 1 \qquad \_\Gamma 2,(Q\rightarrow R)\vdash\_\Delta 2}{P\rightarrow(Q\rightarrow R),Q,P\vdash R}\,\rightarrow\vdash}{P\rightarrow(Q\rightarrow R),Q\vdash P\rightarrow R}\,\vdash\rightarrow$$

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

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: we can select *P* in the left antecedent and apply axiom:

$$\cfrac{\cfrac{\cfrac{}{P\vdash P}\,\text{axiom} \qquad Q,(Q\rightarrow R)\vdash R}{P\rightarrow(Q\rightarrow R),Q,P\vdash R}\,\rightarrow\vdash}{P\rightarrow(Q\rightarrow R),Q\vdash P\rightarrow R}\,\vdash\rightarrow$$

All the problems are resolved, for the moment, and the rest of the proof can be completed in the same way.

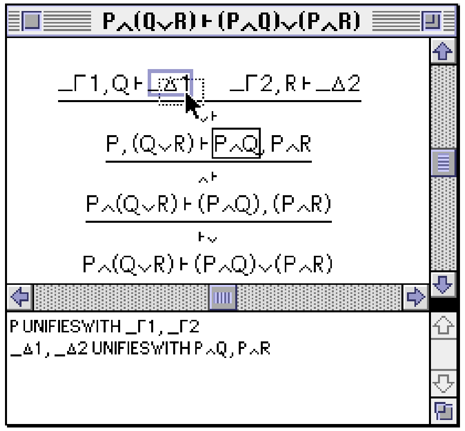
Resolving context-splits with drag-and-drop

Consider the following example:

$$\cfrac{\cfrac{\cfrac{\_\Gamma 1,Q\vdash\_\Delta 1 \qquad \_\Gamma 2,R\vdash\_\Delta 2}{P,(Q\vee R)\vdash PQ\;\;,PR}\,\vee\vdash}{P(Q\vee R)\;\;\vdash(PQ)\;\;,(PR)}\,\vdash}{P(Q\vee R)\;\;\vdash(PQ)\vee(PR)}\,\vdash\vee$$

P UNIFIESWITH _Γ1, _Γ2
_Δ1, _Δ2 UNIFIESWITH PQ, PR

To make progress, we need to send one of the conclusions $P\wedge Q$, $P\wedge R$ into _Δ1 and the other into _Δ2. Jape has a drag-and-drop gesture, designed for this purpose. Dragging $P\wedge Q$ in the MacOS implementation produces this kind of visual feedback, highlighting the dragged formula, the mouse position and potential destinations:



P∧(Q∨R)⊢(P∧Q)∨(P∧R)

$$\dfrac{\dfrac{\dfrac{\_\Gamma1, Q \vdash \_\Delta1 \qquad \_\Gamma2, R \vdash \_\Delta2}{P, (Q\vee R) \vdash \boxed{P\wedge Q}, P\wedge R}\vee\vdash}{P\wedge(Q\vee R) \vdash (P\wedge Q), (P\wedge R)}\wedge\vdash}{P\wedge(Q\vee R) \vdash (P\wedge Q)\vee(P\wedge R)}\vdash\vee$$

```
P UNIFIESWITH _Γ1, _Γ2
_Δ1, _Δ2 UNIFIESWITH P∧Q, P∧R
```

If the mouse is released at the point illustrated, the proof is redrawn and the provisos simplified to match:

$$\dfrac{\dfrac{\dfrac{\_\Gamma1, Q \vdash \_\Delta3, PQ \qquad \_\Gamma2, R \vdash \_\Delta2}{P, (Q\vee R) \vdash PQ\ \ , PR}\vee\vdash}{P(Q\vee R)\ \ \vdash (PQ)\ \ , (PR)}\vdash}{P(Q\vee R)\ \ \vdash (PQ)\vee(PR)}\vdash\vee$$

```
P UNIFIESWITH _Γ2, _Γ1
PR UNIFIESWITH _Δ2, _Δ3
```

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

3.4 Modal logic

It is our intention to enhance Jape so that it can use modal operators, and to further develop this encoding to cover all of linear logic. Most of the code is written and included in Jape, but is lying dormant, so it should be quite soon.

3.5 Single-conclusion sequent calculus (the intuitionistic fragment)

The rules of this logic are very similar to those of chapter 2. 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. The encoding is in the file SCS.jt .

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, we have avoided context-splitting rules, and we have made the left-hand side of a sequent a bag of formulæ.

Negation is normally described by defining it to be equivalent to implication of absurdity: $\neg x$ is just a way of writing $x \rightarrow \bot$. Jape can't handle definitional equality of formulæ yet, and therefore we give rules which implement that equality. With that exception, the rules are more or less the rules of the sequent calculus with the symbol Δ deleted.

hypothesis

$$\dfrac{}{\Gamma, A \vdash A}\ \mathit{hyp}$$

Introduction to the right of the turnstile (⊢... rules)

$$\dfrac{\Gamma\vdash A \quad \Gamma\vdash B}{\Gamma\vdash A\wedge B}\vdash\wedge \qquad \dfrac{\Gamma, A\vdash B}{\Gamma\vdash A\rightarrow B}\vdash\rightarrow \qquad \dfrac{\Gamma\vdash A}{\Gamma\vdash A\vee B}\vdash\vee_L \qquad \dfrac{\Gamma\vdash B}{\Gamma\vdash A\vee B}\vdash\vee_R \qquad \dfrac{\Gamma\vdash A\rightarrow\bot}{\Gamma\vdash\neg A}\vdash\neg$$

$$\dfrac{\Gamma\vdash P(m)}{\Gamma\vdash\forall x.P(x)}\ (\text{FRESH } m)\ \vdash\forall \qquad \dfrac{\Gamma\vdash P(B)}{\Gamma\vdash\exists x.P(x)}\vdash\exists$$

Introduction to the left of the turnstile (...⊢ rules)

$$\dfrac{\Gamma, A, B\vdash C}{\Gamma, A\wedge B\vdash C}\wedge\vdash \qquad \dfrac{\Gamma\vdash A \quad \Gamma, B\vdash C}{\Gamma, A\rightarrow B\vdash C}\rightarrow\vdash \qquad \dfrac{\Gamma, A\vdash C \quad \Gamma, B\vdash C}{\Gamma, A\vee B\vdash C}\vee\vdash \qquad \dfrac{\Gamma, A\rightarrow\bot\vdash B}{\Gamma, \neg A\vdash B}\neg\vdash$$

$$\dfrac{}{\Gamma, \bot\vdash A}\bot\vdash \qquad \dfrac{\Gamma, P(B)\vdash C}{\Gamma, \forall x.P(x)\vdash C}\forall\vdash \qquad \dfrac{\Gamma, P(m)\vdash C}{\Gamma, \exists x.P(x)\vdash C}\ (\text{FRESH } m)\ \exists\vdash$$

structural rules

$$\dfrac{\Gamma\vdash B \quad \Gamma, B\vdash C}{\Gamma\vdash C}\ \mathit{cut} \qquad \dfrac{\Gamma\vdash C}{\Gamma, A\vdash C}\ \mathit{weaken} \qquad \dfrac{\Gamma, A, A\vdash C}{\Gamma, A\vdash C}\ \mathit{contract}$$

LF-style variables

We haven't encoded a multiplicative single-conclusion calculus, but 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.

Syntax

Formula syntax, and use of names, is exactly as in the multiple-conclusion sequent calculus.

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

```
SEQUENT IS BAG ⊢ FORMULA
```

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.

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[1]. 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 is a number of entries, one of which is ticked.

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

3.6 Box display mode and the *hyp* rule

Unlike the multiple-conclusion sequent calculus, the single-conclusion calculus can reasonably be used in the 'box display' mode, simply as a screen-space saving device. Proofs such as

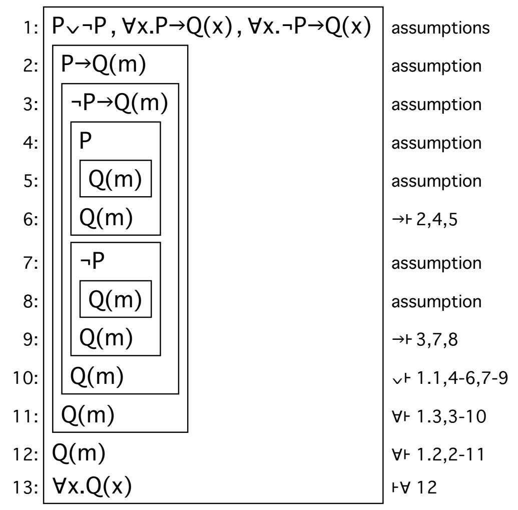
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg P\to Q(m), P\vdash P}\,\text{hyp}\quad\dfrac{}{\neg P\to Q(m), P, Q(m)\vdash Q(m)}\,\text{hyp}}{P\to Q(m), \neg P\to Q(m), P\vdash Q(m)}\,{\to}{\vdash}\qquad\dfrac{\dfrac{}{P\to Q(m), \neg P\vdash \neg P}\,\text{hyp}\quad\dfrac{}{P\to Q(m), \neg P, Q(m)\vdash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

x NOTIN P

are over-large because the hypotheses are written out many times, once in each sequent which they occur.

[1] 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.

Box display of the same proof is much more economical of screen space:



1:	$P\vee\neg P, \forall x.P\to Q(x), \forall x.\neg P\to Q(x)$	assumptions
2:	$P\to Q(m)$	assumption
3:	$\neg P\to Q(m)$	assumption
4:	P	assumption
5:	$Q(m)$	assumption
6:	$Q(m)$	→⊢ 2,4,5
7:	$\neg P$	assumption
8:	$Q(m)$	assumption
9:	$Q(m)$	→⊢ 3,7,8
10:	$Q(m)$	∨⊢ 1.1,4-6,7-9
11:	$Q(m)$	∀⊢ 1.3,3-10
12:	$Q(m)$	∀⊢ 1.2,2-11
13:	$\forall x.Q(x)$	⊢∀ 12

x NOTIN P

and the gain is of course more dramatic in the case of larger proofs.

Part of the gain is produced by hiding applications of *hyp*. A reference to a line proved by *hyp* can be replaced by a reference to the hypothesis used in the *hyp* step – this happens, for example, on lines 4, 5, 9 and 10 of the box display above. All that is necessary is to declare the *hyp* rule and its structural rôle to Jape, which we do as follows:

```
RULE hyp(A) INFER A ⊢ A

STRUCTURERULE IDENTITY    hyp
```

Chapter 4

Encoding natural deduction

The sequent calculus encodings above 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. The challenger isn't finished yet: we can only imitate some kinds of forward step, and some features of the background tree are still traceable in the box display.

This encoding has been used in a first-year course at QMW for three years, with increasing user satisfaction as our encoding has more nearly approached the treatment used by the course lecturer. That lecturer chose the rules we encoded and, in particular, he chose to use a particular classical treatment of negation, not at all the one which we would have chosen for ourselves nor even the particular classical encoding which we would have preferred.

4.1 Inference rules

The rules of natural deduction are not normally stated in terms of sequents, but in a notation which is silent about the hypotheses. In fact the rules were presented to us in Fitch box form and we immediately, almost without thought, transcribed them into a sequent presentation. The rules that we were asked to encode were as follows (plus reiteration, which we don't list).

Elimination rules

i: A ... ... j: $A{\to}B$ ... ... k: B $\to{-}E\ i,j$	i : $A\wedge B$ ... ... j : A $\wedge-E(L)\ i$	i : $A\wedge B$ ... ... j : B $\wedge-E(R)\ i$	i : $\neg\neg A$ ... ... j : A $\neg-E\ i$
i : $A\vee B$ ... ... j: $\boxed{A}$ assumption ... k : $\boxed{C}$ ... ... l : $\boxed{B}$ assumption ... m : $\boxed{C}$ ... ... n : C $\vee-E\ i,j..k,l..m$	i : $\forall x.F(x)$ ... ... j : $F(A)$ $\forall-E\ i$	i : $\exists x.F(x)$ ... ... j : c $\boxed{F(c)}$ assumption ... k : $\boxed{A}$ ... ... l : A $\exists{-}E\ i,j..k$	

Introduction rules

i : $\boxed{A}$ assumption ... j : $\boxed{B}$ ... ... k : $A\to B$ $\to-I\ i..j$	i: A ... ... j : B ... ... k : $A\wedge B$ $\wedge-I\ i,j$	i : $\boxed{A}$ assumption ... j : $\boxed{B\wedge\neg B}$ ... ... k : $\neg A$ $\neg-I\ i..j$	
i : A ... ... j : $A\vee B$ $\vee-I(L)\ i$	i : B ... ... j : $A\vee B$ $\vee-I(R)\ i$	c $\boxed{...}$ i : $\boxed{F(c)}$ ... ... j : $\forall x.F(x)$ $\to-\forall-I\ i$	i : $F(A)$ ... ... j : $\exists x.F(x)$ $\exists{-}I\ i$

Themarginal c in the $\exists$-E and $\forall$-I rules indicates a proviso that the name c should not appear free outside the 'scope box' which it labels.

These Fitch-box rules have well-known tree equivalents, where reiteration is replaced by a hypothesis rule, and the proviso on the $\exists$-elimination rule is more extensive than you might at first suppose.

Hypothesis (tree)

$$\frac{}{A}\,hyp$$

Elimination rules (tree)

$$\frac{\begin{array}{cc}\vdots & \vdots\\ A & A\to B\end{array}}{B}\to-E \qquad \frac{\begin{array}{c}\vdots\\ A\wedge B\end{array}}{A}\wedge-E(L) \qquad \frac{\begin{array}{c}\vdots\\ A\wedge B\end{array}}{B}\wedge-E(R) \qquad \frac{\begin{array}{ccc} & [A] & [B]\\ \vdots & \vdots & \vdots\\ A\vee B & C & C\end{array}}{C}\vee E \qquad \frac{\begin{array}{c}\vdots\\ \neg\neg A\end{array}}{A}\neg-E$$

$$\frac{\begin{array}{cc}\vdots & \vdots\\ \forall x.F(x) & B\ \text{inscope}\end{array}}{F(B)}\forall-E \qquad \frac{\begin{array}{cc} & [\text{var } c, F(c)]\\ \vdots & \vdots\\ \exists x.F(x) & A\end{array}}{A}\,(\text{FRESH } c,\ c \text{ NOTIN } \exists x.F(x))\ \exists-E$$

Introduction rules(tree)

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A\to B}\to-I \qquad \frac{\begin{array}{cc}\vdots & \vdots\\ A & B\end{array}}{A\wedge B}\wedge-I \qquad \frac{\begin{array}{c}\vdots\\ A\end{array}}{A\vee B}\vee-I(L) \qquad \frac{\begin{array}{c}\vdots\\ B\end{array}}{A\vee B}\vee-I(R) \qquad \frac{\begin{array}{c}[A]\\ \vdots\\ B\wedge\neg B\end{array}}{\neg A}\neg-I$$

$$\frac{\begin{array}{c}[\text{var } c]\\ \vdots\\ F(c)\end{array}}{\forall x.F(x)}\,(\text{FRESH } c)\ \forall-I \qquad \frac{\begin{array}{cc}\vdots & \vdots\\ F(B) & B\ \text{inscope}\end{array}}{\exists x.F(x)}\exists-I$$

To implement the scope boxing mechanism of the original presentation we have used pseudo-predicates var and inscope; the intention is that all the names in the formula B in $\exists$-I and $\forall$-E should be mentioned in var predicates in the hypotheses. This is not quite the same as scope boxing: in effect we say that the name c may be used within a particular scope, but we don't say that there may not be other scopes in the proof where the same name is used. (In practice, because the rules which introduce the var pseudo-predicate, used normally, always introduce names new to the proof, the distinction won't be noticed.)

These rules have obvious sequent-calculus equivalents.

Hypothesis (sequent)

$$\frac{}{\Gamma, A \vdash A}\ hyp$$

Elimination rules (sequent)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash A \to B}{\Gamma \vdash B} \to -E \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge -E(L) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge -E(R) \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee E$$

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \neg - E \qquad \frac{\Gamma \vdash \forall x.A(x) \quad \Gamma \vdash B \text{ inscope}}{\Gamma \vdash A(B)} \forall - E$$

$$\frac{\Gamma \vdash \exists x.A(x) \quad \Gamma, \text{var } c, A(c) \vdash B}{\Gamma \vdash B} (\text{FRESH } c, c \text{ NOTIN } \exists x.A) \ \exists - E$$

Introduction rules (sequent)

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to -I \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge -I \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee -I(L) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee -I(R) \qquad \frac{\Gamma, A \vdash B \wedge \neg B}{\Gamma \vdash \neg A} \neg - I$$

$$\frac{\Gamma, \text{var } c \vdash A(c)}{\Gamma \vdash \forall x.A(x)} (\text{FRESH } c) \ \forall - I \qquad \frac{\Gamma \vdash A(B) \quad B \text{ inscope}}{\Gamma \vdash \exists x.A(x)} \exists - I$$

Plainly it is a difficulty, in a backwards-reasoning tool, that each of these is a right-hand rule. Yet if we are to be faithful to our customer's intention, these are the rules that we must encode. In order to understand how to do that, it is necessary to understand how they are intended to be used.

4.2 Syntax

The syntax of formulæ in natural deduction is, of course, just like that in the sequent calculus: see chapter 2 for a discussion of the way that syntax is described in ItL_syntax.j. 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 interpretpredicates   true
```

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 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.

4.3 Forward reasoning

The sort of step that a natural-deduction reasoner might want to make is best illustrated by example. Consider the problem of proving P→Q, Q→R, P⊢R (the second problem in the Conjectures panel defined in the file ItL_problems.j). In box display mode this is shown as

```
1: | P→Q, Q→R, P | assumptions
   | …           |
2: | R           |
```

To anyone used to forward reasoning, the first step is clear: on line 1 there is P and there is also P→Q; use the →-E rule to conclude Q. In the ItL.jt encoding of natural deduction that step can be made by first selecting P→Q on line 1

```
1: | [P→Q], Q→R, P | assumptions
   | …             |
2: | R             |
```

and then applying →-E from the Rules menu

```
1: | P→Q, Q→R, P | assumptions
2: | Q           | →-E 1.3,1.1
   | …           |
3: | R           |
```

It looks like a forward step, and it quacks like a forward step: now you can select Q→R on line 1

```
1: | P→Q, [Q→R], P | assumptions
2: | Q             | →-E 1.3,1.1
   | …             |
3: | R             |
```

and apply →-E again

```
1: | P→Q, Q→R, P | assumptions
2: | Q           | →-E 1.3,1.1
3: | R           | →-E 2,1.2
```

The proof is complete, and has apparently used forward reasoning. Yet in fact it was all done with right-hand side rules and backward reasoning. It is also possible to start by eliminating the arrow in Q→R, but this isn't the manual for that discussion: see "Using ItL Jape" which you can get from the MacOS Web site at QMW.

Cut and forward reasoning

There is a well-known and obvious correspondence between a proof which uses forward reasoning in natural deduction and one which uses *cut* – between $\begin{array}{c}\vdots\\ B\\ \vdots\\ C\end{array}$, on the one hand, and $\dfrac{\begin{array}{c}\vdots\\ B\end{array} \quad \begin{array}{c}[B]\\ \vdots\\ C\end{array}}{C}$ or $\dfrac{\Gamma \vdash B \quad \Gamma, B \vdash C}{\Gamma \vdash C}$ on the other. The proof above is based on a similar correspondence between

$$\begin{array}{c}\vdots\\ \dfrac{P \quad P \to Q}{Q} \to -E\\ \vdots\\ R\end{array}$$

and

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

In the sequent proof, reading downwards, the *hyp* step moves $P \rightarrow Q$ from right to left; the $\rightarrow$-*E* step generates Q, and the *cut* step moves Q from right to left, making it available as an hypothesis for use in the rest of the proof. The sequence "*hyp*; rule; *cut*" must be reversed in a backwards reasoning engine like Jape, but in principle that is all there is to forward reasoning in Jape at present, together with box display mechanisms which hide both *hyp* and *cut* steps.

In order to program this mechanism in Jape's tactic language, we proceed step by step. We include a *hyp* rule, we declare it so that its application is hidden in box display, and we automatically apply it at the end of every proof step (all this could be done anyway, and has nothing essential to do with the forward reasoning mechanism):

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

Similarly, we include and declare *cut*:

```
RULE cut(B) IS FROM B AND B ⊢ C INFER C
STRUCTURERULE CUT cut
```

The elimination rules are the ones which are usually used forward. Each rule is defined in the usual way, just as it would be if it were to be used only as a backwards reasoning rule. For example we encode →-E as follows, giving *A* as an argument because it isn't a subformula of the consequent pattern:

```
RULE "→-E"(A) IS FROM A AND A→B INFER B
```

When this rule is applied it is necessary to distinguish 'backward' from 'forward' application. We have done this by testing if a left-hand side formula has been selected – which we take as a signal for 'forward' reasoning – or not – which is a signal for 'backward' reasoning. The entry for →-E in the Rules menu applies a tactic, giving the name of a tactic and of the rule as arguments:

```
ENTRY "→-E"      IS ForwardOrBackward ForwardCut 1 "→-E"
```

Here *ForwardOrBackward* is the tactic which detects whether to use a forward or a backward step; *ForwardCut* does the necessary work with *cut*, the rule itself, and *hyp*. It includes a step which selects the antecedent to which *hyp* is to be applied:

```
TACTIC ForwardCut (n,Rule)
 SEQ cut (WITHARGSEL Rule) (JAPE (SUBGOAL n)) (WITHHYPSEL hyp)
```

WITHARGSEL applies a tactic taking account of any text-selections which the user might have made – that is, adding an argument to the rule if the user text-selects an argument; JAPE(SUBGOAL *n*) selects the *n*th antecedent (they are numbered 0, 1, ...) of the last applied rule; WITHHYPSEL applies a tactic taking account of any left-hand side selection, and since there will always be one when we call this tactic from *ForwardOrBackward* , the effect is that the user's selected formula is used in the *hyp* step.

ForwardOrBackward tests whether a left-hand-side formula is selected or not, and chooses to call its first argument or its second accordingly. Stripped to its bones it is simply:

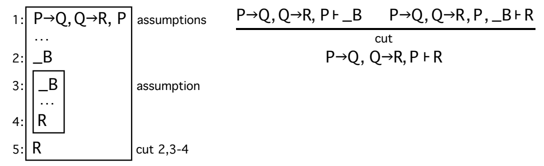
```
TACTIC ForwardOrBackward (Forward, n, Rule) IS
        WHEN   (LETHYP _P (Forward n Rule))
               (WITHSELECTIONS Rule)
```

The WHEN tactical takes a number of tactics, each of which except the last must be guarded: it finds and executes the first guarded tactic whose guard succeeds, or executes its last argument otherwise. LETHYP is a guarded tactic whose guard succeeds if the user has selected a left-hand side formula which unifies with its first argument, and fails otherwise. When it succeeds it executes its second argument in the context produced by the successful unification. In this case, since *_P* will unify with any formula, the effect of the whole tactic is to test whether any left-hand side formula is selected and if so, to execute the tactic corresponding to Forward (in our case *ForwardCut*) or if not, to execute the tactic corresponding to

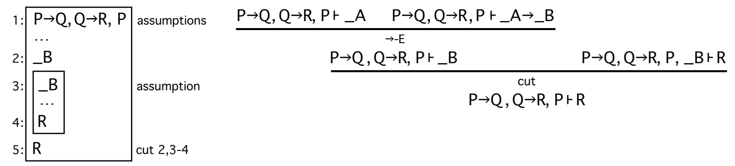
Rule (in our case the rule →-E), taking account of any user gestures, such as argument selection, that may have been made.

In conjunction with AUTOMATCH *hyp*, which automatically closes tips that can be trivially closed, and hiding of *hyp* and *cut* lines, this mechanism has the effect illustrated in the example proof above. To see how it works, we show what would be seen if the steps of the tactic were carried out one by one, without any special display aids, in both tree and box form (you can do this for yourself by loading the file 'displaystyle in Edit menu' from the 'useful buttons' example folder, and then removing the ticks from the 'hide cut lines' and 'hide identity lines' entries). Notice how the unknowns introduced in steps 1 and 2 are all resolved by *hyp* in step 3. That application is on the second antecedent of the →-E, and is constrained to use the originally-selected left-hand side formula, which in this case is P→Q.

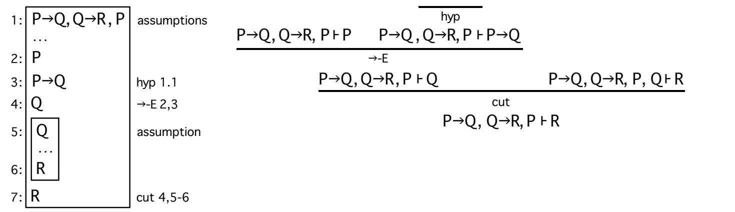
1. *cut*



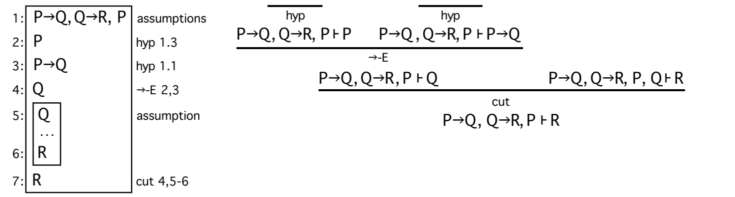
2. WITHARGSEL "→-E"



3. JAPE(SUBGOAL 1), WITHHYPSEL *hyp*



4. AUTOMATCH *hyp*



Because the effect is produced by a tactic the user doesn't see the intermediate steps of the process, and because box display has been instructed to hide *hyp* and *cut* lines, all the user sees, as the original example shows, is a picture in which lines 2 and 3 have been deleted, with references to them converted to references to assumptions 1.3 and 1.1, and in which the *cut* step has been hidden by overlaying line 5 with line 4 and line 7 with line 6.

The principle, then, is to use *cut* to implement a kind of forward reasoning. Not every 'forward' step requires a cut, so we have another auxiliary tactic:

```
TACTIC ForwardUncut (n,Rule)  SEQ (WITHARGSEL Rule) (JAPE (SUBGOAL n)) (WITHHYPSEL hyp)
```

The complete ForwardOrBackward tactic attempts some error reporting if a rule application fails:

```
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))
                    )
            )
```

ALT is a tactic that tries its arguments in turn until one of them is completely successful; FAIL *x* is defined by

```
TACTIC FAIL(x) IS JAPE (fail x)
```

and simply puts up message *x* as an error alert in the graphical interface.

The rest of the rules of the system can now be stated simply, and it's straightforward to organise them into a menu (see ItL_menus.j)

```
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 tac"
  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 "∀-E tac"
  ENTRY "∃-E"         IS ForwardOrBackward ForwardUncut 0 "∃-E"
  SEPARATOR
  ENTRY hyp
END
```

Two of the entries are given indirectly, because they can be overriden in other files which form part of the ItL.jt collection:

```
TACTIC "∀-E tac" IS ForwardOrBackward ForwardCut 0 "∀-E"
TACTIC "∃-I tac" IS "∃-I"
```

The rules are given in ItL_rules.j (note that this collection, for historical reasons, doesn't implement scope boxing; that's done in ItL_LF.j and is described below). Because we have set the *autoAdditiveLeft*

variable to *true*, we can define these rules natural-deduction style, mentioning nothing but the principal formulæ, and Jape will automatically insert a context variable Γ into every sequent:

```
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"(B)                                   IS FROM ∀x. A(x) INFER A(B)
RULE "∃-E"(OBJECT c) WHERE FRESH c AND c NOTIN ∃x.A
                                                IS FROM ∃x.A(x) AND 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 A(c) INFER ∀x .A(x)
RULE "∃-I"(B)                                   IS FROM A(B) INFER ∃x.A(x)
```

Scope boxing is done by adding the file ItL_LF.j, which includes the syntax of the pseudo-predicates, plus new versions of the quantifier rules:

```
PREFIX     10     var
POSTFIX    10     inscope

RULE "∀-I"(OBJECT c) WHERE FRESH c              IS FROM var c ⊢ A(c) INFER ∀x .A(x)
RULE "∃-I"(B)                                   IS FROM A(B) AND B inscope INFER ∃x.A(x)

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

Just as in chapter 3, we define a collection of rules for the inscope judgement, which Jape automatically arranges into an ALT tactic called "inscope", and then we require that the tactic be tried at the end of every proof step:

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

AUTOMATCH "inscope"
```

Finally we provide tactics which use the LAYOUT tactical to require that only the first antecedent (labelled 0) should normally be displayed (see chapter 3 for a little more explanation, and/or appendix B); then we ensure that these tactics are referenced from the menu:

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

TACTIC "∀-E tac" IS ForwardOrBackward ForwardCut 0 "∀-E with side condition hidden"
TACTIC "∃-I tac" IS "∃-I with side condition hidden"
```

There is also a file ItL_hits.j, which implements double-clicking: we don't reference this in ItL.jt, because we don't want to provide it by default to our novice students.

4.4 The Conjectures panel

Since we can apply rules either forward or backward, it would be irksome if we could only apply theorems backward. We 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 formulæ 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 chapter 1), and finally tries to apply it forwards, one way or the other. All of the steps are made 'WITHSELECTIONS' – that is, using any argument 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. The panel is defined in the following manner (for a full listing see the file ItL_problems.j):

```
CONJECTUREPANEL Conjectures
  THEOREMS NaturalDeductionConjectures ARE
        P, P → Q ⊢ Q
  AND   P → Q, Q → R , P ⊢ R
  ...
  END

  THEOREM "G(c) ⊢ ∀x.F(x) → G(x) NOT"                     IS G(c) ⊢ ∀x.F(x) → G(x)
  THEOREM "(∀x.F(x)) → (∀x.G(x)) ⊢ ∀x.F(x) → G(x) NOT"    IS (∀x.F(x)) → (∀x.G(x)) ⊢ ∀x.F(x) → G(x)
  THEOREM "(∃x.F(x))  (∃x.G(x)) ⊢ ∃x.F(x)  G(x) NOT"      IS (∃x.F(x))  (∃x.G(x)) ⊢ ∃x.F(x)  G(x)

  PREFIXBUTTON Apply IS apply TheoremForwardOrBackward

  END
```

The last three conjectures are specially named because they are intended 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 and then the tactic takes over.

See appendix C for a complete listing of commands that can be attached to panel buttons.

4.5 An alternative natural deduction encoding

There is an alternative natural deduction encoding, distributed as jnj.jt[1], and described in "Using J'n'J in Jape", available from the Oxford Jape web site. It will be described in a future edition of this manual.

[1] It doesn't work under MacOS yet, because of problems with font encoding.

Chapter 5

Encoding equational reasoning in functional programs

Previous chapters have dealt with the encoding of logics which are, more or less, variations on the sequent calculus. This chapter describes our treatment of a very different logic. The problem here is to control a large number of equations used to reason about functional-programming formulæ, 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 the files referenced by the file functions.jt.

5.1 Syntax

Jape provides juxtaposition as a primitive syntactic construction, and so it is convenient to represent function application as juxtaposition. In the same way we take the syntax of tupling directly from Jape.

```
FONTS       "Konstanz"

INITIALISE autoAdditiveLeft true

SEQUENT IS BAG ⊢ FORMULA

USE         "equality_rules.j"
USE         "equality_menus.j"
USE         "functions_rules.j"
USE         "functions_menus.j"

AUTOMATCH "= reflexive"
```

Note that we don't automatically translate predicate notation in this encoding: we use juxtaposition mostly to represent function application. We use the ABSTRACTION keyword to label those few parameters of rules which are to be treated as predicates: see below for examples.

The file equality_rules.j covers more than simple equality, since it is 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, we define some binary operators, all left-associative, then we define 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. We have chosen to make *formula* { *variables* \ *formulæ* } the syntax of a substitution form, because that liberates square brackets for use in their conventional rôle in functional programming as list brackets[1].

Then there are some simple definitions, intended to make what follows easier to read:

```
TACTIC FAIL(x)              IS JAPE(fail x)
TACTIC FAILREASON(x)        IS JAPE (failgivingreason x)
```

and 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, we can only give a tuple-equality rule for a fixed finite number of tuple sizes, and here we restrict ourselves to pairs[2]:

$$\frac{}{\Gamma \vdash X = X}\;{=}\,\text{reflexive} \qquad \frac{\Gamma \vdash X = Y \quad \Gamma \vdash Y = Z}{\Gamma \vdash X = Z}\;{=}\,\text{transitive} \qquad \frac{\Gamma \vdash X = Y}{\Gamma \vdash Y = X}\;{=}\,\text{symmetric}$$

$$\frac{\Gamma \vdash X0 = X1 \quad \Gamma \vdash Y0 = Y1}{\Gamma \vdash (X0, X1) = (Y0, Y1)}\;{=}(,)$$

Although most of these rules can be derived from '= reflexive' plus the rewrite rules given below, it is convenient to have them available directly. In any case, Jape doesn't yet have facilities to prove derived rules with antecedents. 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 )
```

Extensionality rules are straightforward but because each implicitly incorporates a step of generalisation, we are careful to include FRESH provisos:

$$\frac{\Gamma \vdash F(x) = G(x)}{\Gamma \vdash F = G}\;(\text{FRESH } x)\ \text{ext} \qquad \frac{\Gamma \vdash F(x,y) = G(x,y)}{\Gamma \vdash F = G}\;(\text{FRESH } x,y)\ \text{ext 2}$$

In the Japeish version we use OBJECT parameters so that the rules, normally used backwards, introduce new identifiers rather than unknowns:

```
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
```

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

5.2 The rewrite rule and user definition of substitutions

We base our treatment of equational reasoning on rewrite rules. We can replace occurrences of a sub-formula *X* within a formula *A* by an alternative sub-formula *Y*, provided only that we can prove $X = Y$. Because an equality can be used to rewrite in either direction we include two rules, whose names are arbitrarily chosen:

$$\frac{\Gamma \vdash X = Y \quad \Gamma \vdash A\{x \backslash Y\}}{\Gamma \vdash A\{x \backslash X\}}\;\text{rewrite} \qquad \frac{\Gamma \vdash X = Y \quad \Gamma \vdash A\{x \backslash X\}}{\Gamma \vdash A\{x \backslash Y\}}\;\text{rewritebackwards}$$

1 We would have reversed the order of formulæ and variables had the author of the encoding not already hijacked the '/' operator.

2 But see our treatment of BAN logic in a later chapter: we are beginning to be able to handle some simple families of rules.

In Japeish, as by now you must expect, we give the first rule a parameter *X* and the second a parameter *Y*. Just for fun we write the rules in predicate notation, which Jape immediately translates into the rules written above.

```
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[1].

In principle, and in practice, it is possible to use the rewrite rule by providing just the argument corresponding to *X* in the rewrite rule or *Y* in the rewritebackwards rule: Jape will search for instances of that argument on the right-hand side of the problem sequent and, in effect, construct a substitution which it unifies with $\_A\{x \backslash \_X\}$ or $\_A\{x \backslash \_Y\}$. That process finds every instance of the argument formula in the right-hand side of the problem sequent, and sometimes that is just what is required.

When we want finer control, which we do when working under direct user control rather than via a search controlled by a tactic, we use the LETSUBSTSEL and WITHSUBSTSEL tacticals. The basis of the technique which we use in encoding equational reasoning with functional programs is exemplified by the following fragment

```
WHEN (LETSUBSTSEL _A (WITHSUBSTSEL rewrite))
```

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

- the user has made at least one text-selection;
- all text-selections are of identical subformulæ *E* within the same formula *F* in the current goal sequent;
- it is possible to construct a substitution form $F'\{v \backslash E\}$, where *v* is a fresh variable, such that $F'\{v \backslash E\}$ reduces to *F* in the presence of the proviso *v* NOTIN *F*;
- *pattern* unifies with $F'\{v \backslash E\}$, without simplifying the substitution unless it is unified with a non-substitution form.

If all four conditions are satisfied, the sequence *tactic* … is executed within the context created by the unification of *pattern* with $F'\{v \backslash E\}$. The substitution form $F'\{v \backslash E\}$ is specially marked so that it is not simplified during the unification process unless it is matched with a non-substitution form: the effect is that it will be unified by structure-matching with a substitution form in *pattern*, if one is provided. $F'\{v \backslash E\}$ is equivalent to the original formula *F*, and if it isn't used immediately in a unification it is simplified out of existence: that's the reason that we have to have both LETSUBSTSEL, to check if we can make the substitution form at all, and then WITHSUBSTSEL, to make it again and to use it immediately in a rule application.

WITHSUBSTSEL *tactic* …is a tactic which requires that the first three of the LETSUBSTSEL conditions are satisfied and, if they are, it applies the sequence *tactic* …to a copy of the problem sequent in which formula *F* has been replaced by $F'\{\_v \backslash E\}$. this time using a fresh unknown *_v* to facilitate unification with the consequent of a rule which uses a corresponding OBJECT parameter. As before, the substitution form $F'\{\_v \backslash E\}$ is specially marked so that it can be unified component-by-component with a corresponding substitution form in the consequent of a rule, and it's eliminated very easily by simplification, so the first rule of *tactic* … is usually the only one that gets a bite at it.

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.

1 We are beginning to realise how to display linear equational proofs using transitivity in much the same way that forward reasoning steps use cut. Once we have perfected the technique our rewrite rules will have to take a different form.

5.3 Hiding parts of proofs: the LAYOUT tactical

When we use the rewrite rule in this logic-encoding, for the most part we employ straightforward function definitions for the left-hand antecedent $X = Y$. These definitions – 'facts' like *map f* [] = [] – are supposed to be well-known to the user, and are therefore best kept as marginal notes in the proof. Our eventual goal is to be able to show a linear equational proof like those in Bird and Wadler, in which every step transforms a formula by equality-substitution:

rev(rev [])	= *rev* []	by *rev* [] = []
	= []	by *rev* [] = []
	= *id* []	by *id x* = *x*

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. The facilities of Jape don't quite stretch to linear equational proofs yet, but we're close.

What we 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 we use in our encoding, for example, reads as follows:

```
TACTIC UnfoldOneSel(x) IS
  WHEN  (LETSUBSTSEL _A (LAYOUT "Fold %s" (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 – should be shown (antecedents are numbered 0, 1, ...); the first argument says that it should be shown with a text which starts with Fold[1] and continues 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[2].

Here is an example proof using our encoding, after two steps:

```
   ...
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
```

Lines 1, 2 and 3 can be read as a partly-completed linear equational proof, up the left-hand side and down the right:

1 A backwards step of unfolding is a folding step when read forwards. Proofs in this encoding are made backwards but read forwards: we have labelled our buttons and named our 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.

2 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 subformulæ 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.

In practice the tactic's error message is often inappropriate, but we show it as it is in order to illustrate the difficulty.

(*rev* • *rev*) *x*	= *rev*(*rev x*)	by (*f* • *g*) *x* = *f*(*g x*)
	= *x*	by ...
	= *id x*	by *id x* = *x*

LAYOUT only hides antecedents, it doesn't destroy them: by double-clicking on the justification of line 2 or line 3 the hidden detail can be revealed. Here is what you see if you double-click on line 3:

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

5.4 Selecting a subformula: LETHYPFIND, LETCONCFIND, ASSOCEQ and FLATTEN

LETSUBSTSEL and WITHSUBSTSEL don't solve all the problems of rewriting, because Jape has a very simple-minded treatment of subformula selection. It provides only character-sequence selection, and the user can select any sub-sequence of the characters which make up a formula. It is possible to select a section of text which isn't a formula at all – $a+b)$, for example, in $x+(a+b)$. Worse, it is possible to select text which is a formula but not a proper subformula – $x+y$, for example, in $f\ x+y$. There are well-known user-interface solutions to this problem, exploiting the syntactic structure of a formula to guide selection, but we haven't implemented any of them. The reason is partly lack of effort, but we have our eyes on a higher prize: we want eventually to include a proper treatment of subformula selection in logics which include associative operators: those which, like + and × in school algebra, don't need to be bracketed when they occur in sequence.

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 × has higher priority than +, is treated internally just like $(A\times B)+C$ but displayed in its unbracketed form[1]. Since we treat all operators as either binary or unary, 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 – can be read as a formula even though it is not a structural subformula of the whole.

We might hope to tell Jape that the operator + is neither left- nor right-associative but *associative* in the mathematical sense, so that $A+B+C$ should 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 that + is commutative as well, so that $A+B+C$ can be read as $(A+C)+B$ if that is what you wish. We intend that a future version of Jape will incorporate a more seamless syntactic treatment of associative and commutative operators that will allow some of these alternative readings, based on the mechanisms which already underly our treatment of BAGs and LISTs in sequents. For the time being we provide support for the explicit manipulation of associative operators in the tactic language.

Our treatment is based on the principle that a formula whose operator is associative can be rewritten in a canonical form, and we provide means to access an internal mechanism of Jape which converts formulæ to their canonical form via the built-in judgement ASSOCEQ(*formula1*, *formula2*) and the tactic FLATTEN *formula*.

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

The first problem is to convert a formula so that the selected text is a proper sub-formula. For example, consider the following proof-in-progress of one of the conjectures from functions.jt:

```
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 Theorem F=id•F 3
```

Next we want to use the second assumed equality, to replace $H \bullet F$ with $H \bullet G$. But the • operator in this encoding is left-associative, and to make the step we must first change the structure of the conclusion formula on line 2, changing its structure from the left-associative form $(J \bullet H) \bullet F = G$ into $J \bullet (H \bullet F) = G$. The step won't work unless there is a proof that • is associative – i.e. unless a conjecture with the form $(F \bullet G) \bullet H = F \bullet (G \bullet H)$ or one with the form $F \bullet (G \bullet H) = (F \bullet G) \bullet H$ exists and is either proved or can be assumed proved because 'apply conjectures and theorems' is ticked in the Edit menu.

We text-select $H \bullet F$ and apply Find from the Rules menu[1]to alter the structure of the formula:

```
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 Theorem F=id•F 4
```

Now the $H \bullet F = H \bullet G$ equality can be used:

```
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 Theorem F=id•F 5
```

Now we would like to apply the first assumption again, but $J \bullet H$ isn't a textual subformula as the formula is written, so we have first to modify the conclusion. Flatten from the Rules menu does the trick:

```
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 Theorem F=id•F 6
```

1 Or from either of the panels – one of us doesn't think that this is good GUI / HCI practice, but the other one made the encoding.

The rest of the proof is straightforward. It remains to explain how all this is done.

The LETHYPFIND and LETCONCFIND tacticals allow the user to rebracket a formula. LETHYPFIND (*old,new*) *tactic* ... *tactic* succeeds if

- the user has made a single text-selection in a hypothesis formula, dividing it in effect into *before*, *middle* and *after* texts;
- the hypothesis formula unifies with the pattern *old*;
- *middle* is a valid formula[1];
- the text *before* (*middle*) *after* is a valid formula and unifies with *new*;
- the sequence *tactic* ... *tactic* succeeds in the context produced by those unifications.

(LETCONCFIND is similar, but demands a selection in a conclusion formula.) The tactical succeeds silently, without running *tactic* ... *tactic*, if *before* (*middle*) *after* turns out to be structurally equal to the original unmodified formula – a test which does not call upon information about associativity. So LETHYPFIND and LETCONCFIND match, and run their argument tactics, if your text selection reorganises the structure of the formula.

In functions_menus.j an entry is put in the Rules menu, and an associated tactic is defined:

```
MENU Rules IS
    ENTRY       "Find"        IS FindSelection
    ...

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)))
             )
        )
```

The FindSelection tactic calls either rewriteHypotheticalEquation or rewriteEquation: those rules are[2]

```
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 ASSOCEQ judgement flattens its arguments, using any relevant theorems / rules about associativity. Each of these rules therefore replaces an equation with a provably equivalent equation. The

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

2 The fact that FindSelection splits the selected formula into two, and the rules pick up that split, is an artefact of the way that ASSOCEQ is currently implemented; we will fix the problem Real Soon Now. The existence of two rewrite rules, rather than a single one plus a tactic that can use cut, is because the encoder doesn't want the kind of ugly trees that result from that kind of simulated forward reasoning.

EVALUATE tactic interprets the judgement; 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 the FLATTEN tactic. The menu entry indexes 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; subformulæ of which that is the operator alone are flattened[1].

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

5.5 Induction in Jape

Jape makes no special treatment of induction. It is handled in the same way as any other logical generalisation rule, using the FRESH proviso. We encode a form of list induction which uses concatenation rather than *cons*[2]:

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

We have collapsed into one step that which usually takes two (by an induction principle prove $\forall x.A(x)$; then infer $A(B)$ by specialisation). There is no need to introduce quantification into equational reasoning, and our one-step rule is perfectly convenient. We encode 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. We could allow both mechanisms, activated by different entries in a menu, but we have instead required our users always to select the particular instances of a subformula which they 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))
```

1 This is the reason that, at present, the FindSelection mechanism splits the formula to which it is matched. It's a bug in our existing mechanism, which will be fixed.

2 Definining lists with concatentation 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 (Richard is ashamed to admit that he 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 a different story!

5.6 Controlling collections of rules

One of the problems of reasoning in functional programming, as we have set it up in this encoding, is that each function definition corresponds to a number of individual statements of equality. The definition of *map*, for example, gives three:

$$map\ f\,[\,] = [\,]$$

$$map\ f\,[x] = f\ x$$

$$map\ f\,(xs+\!\!+ys) = map\ f\ xs +\!\!+ map\ f\ ys$$

It would be tedious to be required to give a name to each individual equality, and in any case we expect our 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 of rules 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:

```
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 our collections of rules, we group 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
```

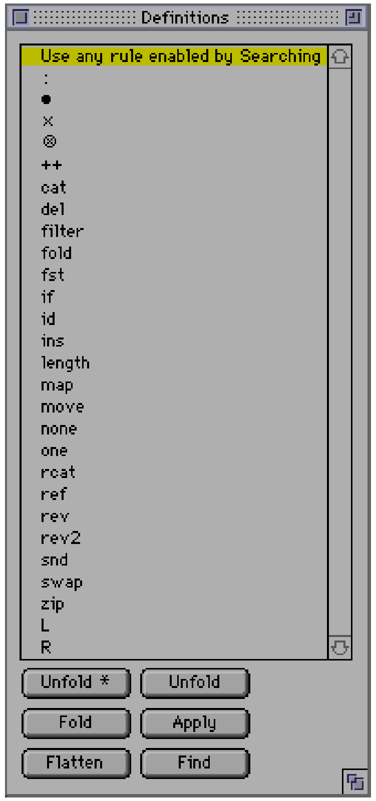
The effect of THEORY is to define all the rules and tactics described by its components, plus a tactic which allows search of those components. In this case the tactic is

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

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

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

The effect, on the Macintosh, is a panel which looks like this:



We discuss the effect of the tactics bound to the buttons and entries below.

5.7 Searching collections of rules and theorems: the FOLD and UNFOLD tacticals

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 we 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.

In the future Jape will support such searching by a mechanism based on mapping tactics over a list of subformulæ of a formula. For the moment our support is more ad-hoc: although based on the same principles, it is closely adapted to the particular problem of equational rewriting.

Jape's support for search in equational reasoning is at present the FOLD, UNFOLD, FOLDHYP and UNFOLDHYP tacticals. The 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 formulæ *L* and *R* and a binary operator *op*[1]; they search for subformulæ 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 UNFOLDHYP are similar, except that they take a pattern which allows the user to define *op*, and they search the list of hypotheses rather than a collection of rules.

However ad-hoc, these techniques are fast and they work well. In this encoding our 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*++*Ys*), for example, then unfolding with the rule *map F* (*Xs*++*Ys*) = *map F Xs*++ *map F Ys* is probably worthwhile.

Our search mechanism, then, is based on the tactic

```
TACTIC Unfold(x) IS LAYOUT "Fold %s" (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 we also use 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 still in their infancy, we have done this in the most naive way possible, using RADIOBUTTONs in a special Searching menu[2].

We have grouped the equality rules into three theories: List (illustrated above), Functions and Reflect. We have grouped conjectures into collections, some of which we are prepared 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)
```

[1] The rules – actually rules and theorems – are doubly filtered because we eliminate all unproved conjectures unless the applyconjectures variable is set to true.

[2] These controls would have been easier to use if they had been simple checkboxes, but at present Jape can't make much use of the values of variables during tactics. This will be remedied soon.

```
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.

Our basic technique at present[1] 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

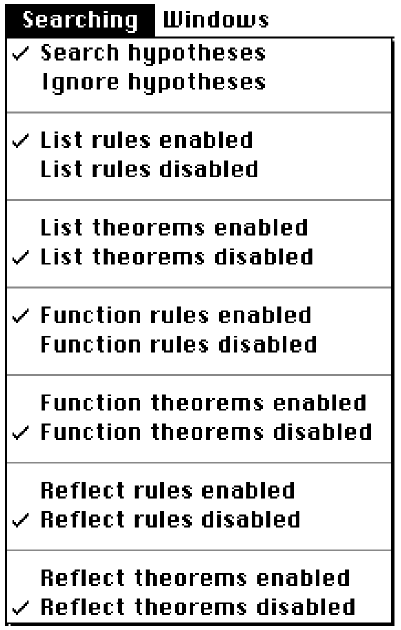
```
MENU "Searching" IS
    RADIOBUTTON dohyp IS
    "Search hypotheses" IS DoHyp
    AND "Ignore hypotheses" IS JUSTFAIL
    INITIALLY DoHyp
    END

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


END
```

1 This paragraph reflects a temporary hack to get around the fact that Jape doesn't yet have any analogue of the ML *case* expression, which we can use to direct the activity of a tactic according to the value of a variable. It probably took longer to type this footnote than to implement the mechanism – but the manual must come first!

On the Macintosh this produces a menu



The AUTO tactic is set up either to unfold or to fold – though for the reasons given above, we never actually use it 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, we wanted to make the button fail noisily if there was nothing at all that it could do; hence the double invocation of the tactic.

We use 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 this 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 contrariwise.

All of the other techniques that we have used are discussed in earlier chapters.

Chapter 6

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 subformulæ with rewrite rules. This chapter shows how the same techniques can be used to support the encoding of a very naive version of axiomatic set theory, which uses rewriting to support equality-style reasoning in both forward and backward steps. Our treatment was inspired by that of David Schmidt ("Natural Deduction Theorem Proving in Set Theory", CSR-142-83, Edinburgh).

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* ≙ *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.

This is 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. We may have gone too far with some of the user interface tricks we have used, and the encoding can hardly be described as 'transparent'. The tactic programming is, indeed, at times rather subtle. We expect, as we learn from this and other examples under development, to be able to generalise and therefore simplify it.

6.1 The natural deduction encoding

This is contained in the files 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 4 above, with the addition of rules for a bi-implication operator, a falsity constant, equality, and a unique-existence operator:

$$\frac{\Gamma, A\vdash B \quad \Gamma, B\vdash A}{\Gamma\vdash A\leftrightarrow B}\leftrightarrow -I \qquad \frac{\Gamma\vdash B \quad \Gamma\vdash A\leftrightarrow B}{\Gamma\vdash A}\leftrightarrow -E(L) \qquad \frac{\Gamma\vdash A \quad \Gamma\vdash A\leftrightarrow B}{\Gamma\vdash B}\leftrightarrow -E(R)$$

$$\frac{\Gamma\vdash P \quad \Gamma\vdash\neg P}{\Gamma\vdash\bot}\bot - I \qquad \frac{\Gamma\vdash\bot}{\Gamma\vdash P}\bot - E \qquad \frac{}{\Gamma\vdash A=A}A=A$$

$$\frac{\Gamma\vdash A[x\setminus B] \quad \Gamma, A[x\setminus c]\vdash B=c}{\Gamma\vdash\exists! x.A}(\text{FRESH}\,c,\,c\,\text{NOTIN}\,B)\ \exists!-E \qquad \frac{\Gamma\vdash\exists! x.A}{\Gamma\vdash\exists x.A}\exists!-E$$

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

$$\frac{\Gamma\vdash A\leftrightarrow B \quad \Gamma\vdash P[v\setminus B]}{\Gamma\vdash P[v\setminus A]}\mathit{rewrite}\leftrightarrow« \qquad \frac{\Gamma\vdash A\leftrightarrow B \quad \Gamma\vdash P[v\setminus A]}{\Gamma\vdash P[v\setminus B]}\mathit{rewrite}\leftrightarrow»$$

$$\frac{\Gamma\vdash A=B \quad \Gamma\vdash P[v\setminus B]}{\Gamma\vdash P[v\setminus A]}\mathit{rewrite}={«} \qquad \frac{\Gamma\vdash A=B \quad \Gamma\vdash P[v\setminus A]}{\Gamma\vdash P[v\setminus B]}\mathit{rewrite}={»}$$

These are encoded, completely straightforwardly, in the file BnE-Fprime_rules.j.

The rules are inserted into the menu as

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

    SEPARATOR

    ENTRY "→-E"          IS FOB "→-E forward" "→-E"
    ENTRY "↔-E(L)"       IS FOB "↔-E(L) forward" "↔-E(L)"
    ENTRY "↔-E(R)"       IS FOB "↔-E(R) forward" "↔-E(R)"
    ENTRY "-E(L)"        IS FOB ForwardCut "-E(L)"
    ENTRY "-E(R)"        IS FOB ForwardCut "-E(R)"
    ENTRY "∨-E"          IS FOB ForwardUncut "∨-E"
    ENTRY "¬-E"          IS FOB ForwardCut "¬-E"
    ENTRY "⊥-E"          IS FOB ForwardCut "⊥-E"
    ENTRY "∀-E"          IS FOBSS ForwardCut "∀-E"
    ENTRY "∃-E"          IS FOB ForwardUncut "∃-E"

    SEPARATOR

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

Here FOB is essentially the tactic ForwardOrBackward of chapter 4, ForwardCut and ForwardUncut are also as described in chapter 4, 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 5, 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 5 to allow rewriting in forward as well as backward reasoning style. We require that the user must text-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-

1 Perhaps, at this point, you might begin to wonder whether the complexity of our tactic programming doesn't undermine the claim that Jape is simple and easy to program. Our answer is twofold: first, this is work in progress, it is much simpler than

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 text-selections describe a substitution in a hypothesis (left-hand side) formula; LETCONCSUBSTSEL succeeds when they describe a substitution in a conclusion (right-hand side) formula.

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 does a *cut* and then 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 text-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 hypothesis[1]. The automatic WITHHYPSEL enables us, as in this example, to distinguish between two selected hypotheses: the one selected for application as an equality, and the one text-selected for rewriting.

6.2 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 at present, so we have made some drastic simplifications.

The form of a set abstraction, in this encoding, is { *variable* | *formula* }, and the occurrence of the variable to the left of the bar is a binding occurrence; we also allow { <*variable*,*variable*> | *formula* }. We include, therefore, in set_syntax.j

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

PREFIX    1000  Pow
PREFIX    800   ⋒
```

it used to be, and that we are still working on it. But second, we 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.

1 It 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?

```
POSTFIX  800   ⊥
INFIX    700L  ∩ ⁻
INFIX    720L  •
INFIX    740L  ×
INFIX    600L  ⊆
INFIX    500L  ∊ ¬∊
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. We misuse the linear logic ⊥ symbol as our representation of set negation, but we do use it as a postfix operator[1] .

Given the OUTFIX and BIND directives above, together with the standard interpretation of comma as a zero-priority associative operator, we allow the following as formulæ:

{}	which we interpret as the empty set;
{ *formula* }	which we interpret as a singleton set;
{ *formula*, ... , *formula* }	which we interpret as a literal description of a set;
{ *variable* | *formula* }	which we interpret as a set abstraction;
{ <*variable*, *variable*> | *formula* }	which we interpret as a set abstraction, a set of pairs.

Allowing set brackets with and without the vertical bar is a trick of which we are slightly ashamed. In future we hope that these shapes of formulæ, and more, will be recognised by a more principled parser.

6.3 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. Schmidt’s treatment was based on Gödel-Bernays set theory and had a judgement “Set *A*”, which we have not carried forward into our treatment, principally because our client didn’t want us to.

The axioms of comprehension and extension in this naive treatment are

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

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

Of course the axiom of comprehension, stated as above, isn’t first order, but that doesn’t bother Jape. We haven’t yet found a way to incorporate comprehension as a single rule, just because of the existence operator, and so we have followed Schmidt and incorporated it as two rules for each of our set-abstraction forms:

$$\frac{\Gamma \vdash P(A)}{\Gamma \vdash A \in \{y \mid P(y)\}} \qquad \frac{\Gamma \vdash A \in \{y \mid P(y)\}}{\Gamma \vdash P(A)} \qquad \frac{\Gamma \vdash P(A,B)}{\Gamma \vdash \langle A,B\rangle \in \{\langle y,z\rangle \mid P(y,z)\}} \qquad \frac{\Gamma \vdash \langle A,B\rangle \in \{\langle y,z\rangle \mid P(y,z)\}}{\Gamma \vdash P(A,B)}$$

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
```

[1] Putting a smiley face here, in Windings font, adds about 300k bytes to the PostScript version of this file. Consider yourself smiled at.

```
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
```

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))))
```

We can incorporate extension, however, as an axiomatic definition. We don’t include the outer quantification, as our rules are schemata. The rule is

$$\frac{}{A = B \triangleq \forall y.y \in A \leftrightarrow y \in B}$$

encoded as[1]

```
RULE (OBJECT y) IS INFER 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 axiomatic definition above. 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.

[1] It’s obvious from this example that Jape needs a simple way of expressing rules whose name is just the consequent of the rule. It will have it, one day.

We include the rule as part of a Definitions panel, then, 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:

```
PREFIXBUTTON "A≙…" IS apply ForwardSubstHiding "rewrite ≙ «" "rewrite ≙ »"
PREFIXBUTTON "…≙B" IS apply ForwardSubstHiding "rewrite ≙ »" "rewrite ≙ «"
```

The tactic ForwardSubstHiding is rather subtle, because it allows the user to rewrite

- either a hypothesis or a conclusion;
- after text-selecting a number of instances of a subformula, just those instances;
- without text-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 () (1) ruleRL thm (WITHSUBSTSEL hyp)))
          (LETCONCSUBSTSEL _P (LAYOUT () (1) (WITHSUBSTSEL ruleLR) thm))
          (LETHYP _P cut (LAYOUT () (1)  ruleRL thm
                                         (LETGOAL (_P'[_v\_Q]) (WITHHYPSEL(hyp _Q)))))
          (LETGOAL _P (LAYOUT () (1) (ruleLR _P) thm))
```

The first alternative in the WHEN is activated when the user has text-selected in a hypothesis: it cuts, uses one of the rewrite rules, closes the first antecedent with the theorem, and the second using the text-selection that the user made. The second alternative is activated when there is a text-selection in a conclusion: it uses the other rewrite rule followed by the theorem. The last alternative is activated when there is no recognisable text-selection[1] 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 nothing to work on, so that rule will necessarily succeed by deferred unification of the consequent of the rewrite with the conclusion. Then it closes the first antecedent of the rewrite with the theorem: that alters the consequent of the rewrite, but won't introduce enough constant material to enable the deferred unification to be resolved. Somehow we have to unify the selected hypothesis with one side or other of the theorem, just as in the fourth alternative. The trick is to realise that after the theorem is applied, the second antecedent of the rewrite step will be a substitution: we take the substituting formula from that substitution and, using *hyp*, unify that with the whole substitution and the originally-selected hypothesis. The effect is like magic: the whole of the selected hypothesis is unified with one side or the other of the theorem, just as in the fourth alternative[2].

Each of the entries in the Definitions panel is intended to be used as a two-way rewrite rule, using the buttons above. One entry in the Definitions panel is given in BnE-Fprime_menus.j (where also the buttons are defined):

```
RULE IS A≠B ≙ ¬(A=B)
```

1 Actually, and unfortunately, when there is no *valid* text selection.
2 It's quite a clever bit of tactic programming, and that's the problem. In the future we hope to be able to allow *either* of the formulæ – *A* or *B* – in the rewrite rule to be provided as argument.

This definition makes it unnecessary to have rules for ≠[1]. The others are in set_menus.j:

```
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}
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 AB ≙ { y | y∊A∨y∊B }
RULE (OBJECT y) IS A∩B ≙ { y | y∊Ay∊B }
RULE (OBJECT y) IS A-B ≙ { y | y∊Ay¬∊B }
RULE (OBJECT y) IS A⊥ ≙ {y | y¬∊A}
RULE (OBJECT x, OBJECT y) IS (C) ≙ { x | ∃y. x∊yy∊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∊Ay∊B }
RULE (OBJECT x, OBJECT y, OBJECT z) IS A•B ≙ { <x,z> | ∃y.<x,y>∊A<y,z>∊B }
```

6.4 The non-axiomatic rules

A proof using the axioms will typically introduce and then eliminate logical connectives. Here is the beginning of such an axiomatic proof:

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

1 In the future we hope to be able to handle this sort of definition by 'definitional equality', where you write *A≠B* and Jape interprets it as ¬(*A*=*B*) but displays it as *A≠B*; compare the treatment of ¬*A* as equivalent to *A*→⊥ in many treatments of the intuitionistic sequent calculus, which we also can't handle at the moment as transparently as we would wish.

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:

$$\frac{\Gamma, c \in A \vdash c \in B}{\Gamma \vdash A \subseteq B}\,(\text{FRESH } c) \subseteq -I \qquad \frac{\Gamma \vdash C \in A \quad \Gamma \vdash A \subseteq B}{\Gamma \vdash C \subseteq B} \subseteq -E$$

$$\frac{\Gamma \vdash A \subseteq B \quad \Gamma \vdash B \subseteq A}{\Gamma \vdash A = B} = -I \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash A \subseteq B} = -E(L) \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B \subseteq A} = -E(R)$$

and here is the proof completed using these rules, rather than the axiomatic definitions:

```
1: | A=B        | assumption
2: | A⊆B        | =-E(L) 1
3: | B⊆A        | =-E(R) 1
4: | A⊆BB⊆A     | -I   2,3

5: | A⊆BB⊆A     | assumption
6: | A⊆B        | -E(L)   5
7: | B⊆A        | -E(R)   5
8: | A=B        | =-I 6,7
9: A=B↔A⊆BB⊆A     ↔-I 1-4,5-8
```

Somewhat simpler! The rules are encoded in the obvious way[1] and likewise organised into a menu.

Naturally we regret that Jape cannot yet deal with proofs of derived rules such as these.

[1] Jape is currently unequipped to allow the user to prove derived rules from the axioms. We intend that in the near future it should permit it – the mechanism we have for proving theorem schemata is almost all that we need.

Chapter 7

Encoding the Hindley-Milner type-assignment algorithm

We consider a version of the algorithm for the lambda calculus with tuples and *let*/*letrec* bindings.

$$\frac{C, x : T \vdash E : T'}{C \vdash \lambda x.E : T \rightarrow T'}\ \lambda - I \qquad \frac{C \vdash F : T \rightarrow T' \quad C \vdash G : T}{C \vdash F\,G : T'}\ application - I$$

$$\frac{C \vdash E1 : T1 \quad C \vdash E2 : T2}{C \vdash (E1, E2) : (T1 \times T2)}\ tuple - I$$

$$\frac{C \vdash E : T1 \quad C \vdash T1 \prec S \quad C, x : S \vdash F : T}{C \vdash \text{let}\, x = E \,\text{in}\, F \,\text{end} : T}\ \text{let}- I \qquad \frac{C, x : T1 \vdash E : T1 \quad C \vdash T1 \prec S \quad C, x : S \vdash F : T}{C \vdash \text{letrec}\, x = E \,\text{in}\, F \,\text{end} : T}\ \text{letrec}- I$$

$$\frac{C(x) \mapsto S \quad S \succ T}{C \vdash x : T}\ identifier\ type$$

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 $C(x) \mapsto \ldots$ 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 formulæ.

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.

7.2 Syntax

We represent λ formulæ as a LEFTFIX formula, and we give that formula a lower priority than the colon operator, so that we don't unnecessarily have to bracket λ formulæ. The type-tupling operator × is treated as an associative operator, rather like comma. We need an == operator (blechh!) because = is used in the *let / letrec* syntax. We use « for generalisation and » for specialisation. We use a double-arrow operator rather than a colon in the contexts, for no particularly good reason that we can remember. We have included additional operators • and ◃ which are used in the generalisation-step induction.

We have represented type schemes which include schematic variables– so-called polytypes – as $\forall t.T$ or $\forall t1, t2.T$ and so on, with up to four schematic variables. Those which have no schematic variables – so-called monotypes – as #*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 $\forall ().T$.

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

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
```

7.2 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 formulæ.

```
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 we apply whenever possible – in this case AUTOUNIFY seems to be the best mechanism:

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

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

```
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
```

Reading the context and specialising a type scheme

Things get more interesting when we consider how to handle the context-evaluation step $C(x) \mapsto S$: C maps 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, we can use a NOTIN proviso to help us to read the context in this way. Because variables and constants are different syntactic classes, we 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
```

We declare these two 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"
```

We have rules for the types of the constant function identifiers which we have used:

```
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 we 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)
                                 (JAPE(fail(x:T can only be applied to either variables or
                                            constants: _E is neither)))
                      )
                      (LETGOAL   _E (JAPE(fail(conclusion _E is not a ' name:type ' judgement))))
              )
        )
        "S»T"
```

The generalisation step

The technique used here is to perform a structural induction on the type *T* in order to calculate its schematic variables. These will be unknowns, because of course we don't judiciously introduce type variables when running the algorithm (though we might): we simply introduce unknowns as necessary, as we go.

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 these 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 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•..."
```

Automatic search

In this chapter we are dealing with an encoding of an *algorithm*, not simply a 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.

7.3 An example

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

```
 1: C                                                         assumption
 2:   map⇒#(t1→t2)→[t1]→[t2], f⇒#num→num                       assumptions
 3:     f⇒#t1→t2                                              assumption
 4:       xs⇒#[t1]                                            assumption
 5:       (==):[t1]→[t1]→bool                                 constant
 6:       xs:[t1]                                             C(x)⇒S; S»T
 7:       (==)xs:[t1]→bool                                    F G : T 5,6
 8:       nil:[t1]                                            constant
 9:       xs==nil:bool                                        F G : T 7,8
10:       nil:[t2]                                            constant
11:       (::):t2→[t2]→[t2]                                   constant
12:       f:t1→t2                                             C(x)⇒S; S»T
13:       hd:[t1]→t1                                          constant
14:       xs:[t1]                                             C(x)⇒S; S»T
15:       (hd xs):t1                                          F G : T 13,14
16:       f(hd xs):t2                                         F G : T 12,15
17:       (::)(f(hd xs)):[t2]→[t2]                            F G : T 11,16
18:       map:(t1→t2)→[t1]→[t2]                               C(x)⇒S; S»T
19:       f:t1→t2                                             C(x)⇒S; S»T
20:       map f:[t1]→[t2]                                     F G : T 18,19
21:       tl:[t1]→[t1]                                        constant
22:       xs:[t1]                                             C(x)⇒S; S»T
23:       (tl xs):[t1]                                        F G : T 21,22
24:       map f(tl xs):[t2]                                   F G : T 20,23
25:       f(hd xs)::map f(tl xs):[t2]                         F G : T 17,24
26:       if xs==nil then nil else f(hd xs)::map f(tl xs)fi:[t2]   if E then ET else EF fi : T 9,10,25
27:     λxs.if xs==nil then nil else f(hd xs)::map f(tl xs)fi:[t1]→[t2]   λx.E : T1→T2 4-26
28:   λf.λxs.if xs==nil then nil else f(hd xs)::map f(tl xs)fi:(t1→t2)→[t1]→[t2]   λx.E : T1→T2 3-27
29:   map⇒#(t1→t2)→[t1]→[t2], f⇒#num→num                       assumptions
30:     x⇒#num                                                assumption
31:     (+):num→num→num                                       constant
32:     x:num                                                 C(x)⇒S; S»T
33:     (+)x:num→num                                          F G : T 31,32
34:     x:num                                                 C(x)⇒S; S»T
35:     x+x:num                                               F G : T 33,34
36:   λx.x+x:num→num                                          λx.E : T1→T2 30-35
37: (t1→t2)→[t1]→[t2]«∀t1,t2.(t1→t2)→[t1]→[t2]                generalise
38: num→num«#num→num                                          generalise
39:   map⇒∀t1,t2.(t1→t2)→[t1]→[t2], f⇒#num→num                 assumptions
40:   map:(num→num)→[num]→[num]                               C(x)⇒S; S»T
41:   f:num→num                                               C(x)⇒S; S»T
42:   map f:[num]→[num]                                       F G : T 40,41
43:   (::):num→[num]→[num]                                    constant
44:   0:num                                                   n:num
45:   (::)0:[num]→[num]                                       F G : T 43,44
46:   (::):num→[num]→[num]                                    constant
47:   1:num                                                   n:num
48:   (::)1:[num]→[num]                                       F G : T 46,47
49:   (::):num→[num]→[num]                                    constant
50:   2:num                                                   n:num
51:   (::)2:[num]→[num]                                       F G : T 49,50
52:   nil:[num]                                               constant
53:   2::nil:[num]                                            F G : T 51,52
54:   1::2::nil:[num]                                         F G : T 48,53
55:   (0::1::2::nil):[num]                                    F G : T 45,54
56:   map f(0::1::2::nil):[num]                               F G : T 42,55
57: 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 2-28,29-36,37,38,39-56
```

This example shows that it is necessary for Jape to learn how to fold long formulæ when displaying a proof (it can fold long lists of formulæ – see, for example, the BAN logic encoding).

7.4 Jape’s treatment of type-theoretic logics

In simple, ‘pure’ logics, we can reasonably claim that Jape can transparently encode the inference rules, and all the magic is hidden in its treatment of substitutions, bindings and unification. In the case of the Hindley-Milner logic and, we surmise, other type-theoretic logics, that isn’t so. We’ve made some creative choices and had to program an encoding of the treatment of contexts. If the treatment in this chapter is to serve as a model of how Jape can encode type-theoretic logics, there are a number of questions which have to be answered.

First, and trivially, we ought to able to deal with the monotype / polytype distinction without the ugly syntactic mechanism we have used here. That’s a matter of improving our parser generator, we believe, and is simply a question of development.

More seriously, our treatment has no judgement equivalent to ‘C is a context’, and we have pushed the question into the context-interpretation rule, treating the context as a mapping and making sure with a proviso that we aren’t overlooking a later binding. Meta-theoretically it is clear that the context might easily be formed by ensuring that every name it contains is distinct; the necessary α-conversion, however, makes it hard for a human prover to keep track of what is going on. It seems to us, therefore, that we are pragmatically correct to treat the context as a mapping. Also, our rules are context-validity preserving. But it is still possible to state a conjecture with a nonsense context and yet prove it in our system: $GARBAGE \vdash \lambda x.x : T \rightarrow T$ will be a theorem. It would be absurdly inefficient to check the validity of the context at every rule application; nevertheless, we must find ways in which we can check its validity at crucial points in a proof.

We intend, in future work on type-theoretic logics, to continue to develop the approach used here. We expect to invent proviso mechanisms which allow us to state that the names in some type judgement are not rebound by the context to their right, or something similar. We dream, even, of user-defined provisos which will allow close control of the meaning of such provisos. We hope to find the right place to put ‘C is a context’ judgements.

Chapter 8

Encoding Hoare logic

This chapter has very little to say. The encoding defined in the file hoare.jt and the files that it invokes is chiefly interesting for what it *doesn’t* do. Jape is perfectly capable of encoding the program syntax and the rules of inference about predicates, but it falls down when it tries to handle arithmetic. You could, in principle, prove that $x<x+1$ by induction (really!) but induction is absolutely no help when it comes to deciding that 3<4. Experience with this encoding shows that Jape needs an ‘arithmetic oracle’, and one is under construction.

The problem of arithmetic is tricky, and we realise that provision of an arithmetic oracle won’t make it go away. Jape lacks a ‘the user is an oracle’ mechanism, as for example is provided in the Imperial College proof editor Pandora. Such a mechanism would make it possible to certify certain steps in a proof and for the steps to be accepted without further ado. That makes difficult arithmetic the responsibility of the user: certainly not sound, but far more convenient than Jape’s current incapacity.

Chapter 9

Encoding BAN logic

Burroughs-Abadi-Needham (BAN) logic is a logic of authentication-protocols. It's of interest to us chiefly because it is a logic in which the rules don't fit into a tidy introduction / elimination structure, so that we have to use 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 allows us to demonstrate 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 which we managed to add to our Konstanz font. We've had to transform some of the notation to linearise it: for example, we have made $A\overset{K}{\leftrightarrow}B$ (A and B share private key K) into $(A,B)\leftrightarrow K$ and we've made $\{X\}_K$ into $\{X\}K$. We've used $K^{\perp}$ rather than K^{-1}. Otherwise, we believe, we have faithfully described the syntax, even if we have had to guess at the syntactic hierarchy of operators.

```
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    ⊥
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 depicted 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 we haven't reproduced (and which is discussed in the paper though not depicted there). The rules are given natural-deduction style, without mentioning a context of hypotheses:

$$\frac{P\mid\!\equiv Q\overset{K}{\leftrightarrow}P \quad P\triangleleft\{X\}_K}{P\mid\!\equiv Q\mid\!\sim X} \qquad \frac{P\mid\!\equiv \overset{K}{\mapsto}Q \quad P\triangleleft\{X\}_{K^{-1}}}{P\mid\!\equiv Q\mid\!\sim X} \qquad \frac{P\mid\!\equiv Q\overset{Y}{\rightleftharpoons}P \quad P\triangleleft\langle X\rangle_Y}{P\mid\!\equiv Q\mid\!\sim X}$$

$$\frac{P\mid\!\equiv \#X \quad P\mid\!\equiv Q\mid\!\sim X}{P\mid\!\equiv Q\mid\!\equiv X} \qquad \frac{P\mid\!\equiv Q\mid\!\Rightarrow X \quad P\mid\!\equiv Q\mid\!\equiv X}{P\mid\!\equiv X}$$

$$\frac{P\mid\!\equiv X \quad P\mid\!\equiv Y}{P\mid\!\equiv(X,Y)} \qquad \frac{P\mid\!\equiv(X,Y)}{P\mid\!\equiv X} \qquad \frac{P\mid\!\equiv Q\mid\!\equiv(X,Y)}{P\mid\!\equiv Q\mid\!\equiv X} \qquad \frac{P\mid\!\equiv Q\mid\!\sim(X,Y)}{P\mid\!\equiv Q\mid\!\sim X}$$

$$\frac{P\triangleleft(X,Y)}{P\triangleleft X} \qquad \frac{P\triangleleft\langle X\rangle_Y}{P\triangleleft X}$$

$$\frac{P\mid\!\equiv Q\overset{K}{\leftrightarrow}P \quad P\triangleleft\{X\}_K}{P\triangleleft X} \qquad \frac{P\mid\!\equiv \overset{K}{\mapsto}P \quad P\triangleleft\{X\}_K}{P\triangleleft X} \qquad \frac{P\mid\!\equiv \overset{K}{\mapsto}Q \quad P\triangleleft\{X\}_{K^{-1}}}{P\triangleleft X}$$

$$\frac{P\mid\!\equiv \#X}{P\mid\!\equiv \#(X,Y)}$$

$$\frac{P\mid\!\equiv R\overset{K}{\leftrightarrow}R'}{P\mid\!\equiv R'\overset{K}{\leftrightarrow}R} \qquad \frac{P\mid\!\equiv Q\mid\!\equiv R\overset{K}{\leftrightarrow}R'}{P\mid\!\equiv Q\mid\!\equiv R'\overset{K}{\leftrightarrow}R} \qquad \frac{P\mid\!\equiv R\overset{X}{\rightleftharpoons}R'}{P\mid\!\equiv R'\overset{X}{\rightleftharpoons}R} \qquad \frac{P\mid\!\equiv Q\mid\!\equiv R\overset{X}{\rightleftharpoons}R'}{P\mid\!\equiv Q\mid\!\equiv R'\overset{X}{\rightleftharpoons}R}$$

$$\frac{P\mid\!\equiv\forall v_1\ldots v_n.(Q\mid\!\Rightarrow X)}{P\mid\!\equiv Q\mid\!\Rightarrow X[v_1\ldots v_n\backslash Y_1\ldots Y_n]}$$

The rules which don't deal with tuples are very 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⊥ ⇒ P⫢Q|~X" IS FROM P⫢Q↦K AND P◁{X}K⊥ 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⊥ ⇒ P◁X" IS FROM P⫢Q↦ K AND P◁{X}K⊥ 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)
```

We've had to include *hyp* so that we can use assumptions. *Cut* allows us to mimic forward proof. Left-weakening means that we can use theorems which don't match all the hypotheses:

```
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
```

Putting rules into menus

Organising these into menus is quite a problem. We’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◃{X}K ⇒ P⊨Q|~X" is in the menus for ↔, ◃ and |~. Only *hyp* and the rule dealing with ∀ are in a menu labelled ‘Logic’.

We have implemented forward reasoning in the style of chapter 4; then, for example when "P⊨(Q,P)↔K, P◃{X}K ⇒ P⊨Q|~X" is included in the ↔ menu we have

```
ENTRY "P⊨(Q,P)↔K, [P◃{X}K] ⇒ P⊨Q|~X"
    IS ForwardOrBackward ForwardCut 0 "P⊨(Q,P)↔K, P◃{X}K ⇒ P⊨Q|~X"
```

in the ◃ menu we have

```
ENTRY "P◃{X}K, [P⊨(Q,P)↔K] ⇒ P⊨Q|~X" IS
    ForwardOrBackward ForwardCut 1 "P⊨(Q,P)↔K, P◃{X}K ⇒ P⊨Q|~X"
```

The square-bracketted 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_menus.j. We may not have included the rules in enough menus or enough times (for example, we probably ought to have "P⊨(Q,P)↔K, P◃{X}K ⇒ P⊨Q|~X" in the |~ menu twice, focussing once on each antecedent). We haven’t had enough users to know if we have got this bit of user interaction right.

Dealing with tuples

We’ve generalised some of the BAN rules: for example, we have implemented

$$\frac{P \vDash X_1 \quad \ldots \quad P \vDash X_n}{P \vDash \left(X_1, \ldots, X_n\right)} \qquad\qquad \frac{P \vDash (\ldots, X, \ldots)}{P \vDash X}$$

for 2-, 3- and 4-tuples. We’ve done it, as you ought to expect, by listing each version of the rule and combining them with the RULES directive:

```
RULES "... P⊨X ... ⇒ P⊨(...,X,...)" 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
```

```
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 us an interesting forward proof problem. We would like to be able to select an item of a tuple and pick it out using one of these rules. To do so we need to be able to search the collection. Since our forward proof steps are all sequences “*cut; rule; select subgoal; hyp*” we have to make sure on the second step that we select the right rule. We don’t have a very good mechanism for that in our tactic language at present. The best we have come up with is a sort of automatic backtracking using 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 $tactic_0$ $tactic_1$ ... $tactic_n$. We make our forward step tactics use WITHCONTINUATION:

```
TACTIC ForwardCut (n,Rule)
    SEQ cut (WITHCONTINUATION (WITHARGSEL Rule) (JAPE (SUBGOAL n)) (WITHHYPSEL hyp))
```

```
TACTIC ForwardUncut (n,Rule)
    WITHCONTINUATION (WITHARGSEL Rule) (JAPE (SUBGOAL n)) (WITHHYPSEL hyp)
```

Then we include in the ⊨ menu

```
ENTRY "P⊨Q⊨(...,X,...) ⇒ P⊨Q⊨X"
    IS ForwardOrBackward ForwardCut 0 "P⊨Q⊨(...,X,...) ⇒ P⊨Q⊨X"
```

and Bob’s your uncle.

9.3 Conjectures with long assumption lists

On educational grounds we thought it best to include lots of assumptions in each conjecture, simply because the problem for novices is to decide which assumptions are relevant and how. 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
```

Jape automatically folds long assumption lists in a box display to fit the proof window. The proof of this conjecture, in a moderately-sized window, is

```
 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|~X 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'6 9
11: A⊨S⤇(A,B)↔Kab                                           P⊨∀x.X(x) ⇒ P⊨X(Y) 2.3
12: A⊨(A,B)↔Kab                                             P⊨Q⤇X, P⊨Q⊨X ⇒ P⊨X 11,10
```

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: we 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, BIND, BUTTON, CHECKBOX, CLASS, CONCFRESH, CONCHIT, CONJECTUREPANEL, CONSTANT, CURRENTPROOF, CUT, END, ENTRY, FONTS, FORMULA, FRESH, FROM, HYPFRESH, HYPHIT, IDENTITY, IN, INFER, INFIX, INFIXC, INITIALLY, INITIALISE, I S, JUDGEMENT, JUXTFIX, LEFTFIX, LEFTWEAKEN, LIST, MENU, MENUKEY, NOTIN, NUMBER, OBJECT, OUTFIX, POSTFIX, PREFIX, PREFIXBUTTON, PROOF, RADIOBUTTON, RIGHTWEAKEN, RULE, RULES, SCOPE, SEPARATOR, SEQUENT, STRING, STRUCTURERULE, SUBSTFIX, TACTIC, TACTICPANEL, THEOREM, THEOREMS, THEORY, UNIFIESWITH, USE, VARIABLE, VIEW, WEAKEN, WHERE

A.1 Directives

In this description I use [... | ...] to describe alternatives, { ... } to describe optional components and ellipsis to denote optional repetition.

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 the discussion of flexible syntax below.

BIND *variable ... variable* SCOPE *name ... name* IN *formula*: see the discussion of flexible syntax below.

BUTTON: this allows you to attach a command to a label in a menu or a button on a panel. See appendix C for information on the command language.

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.

CLASS <kind> *names*: see the discussion of flexible syntax below.

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* } <entails> } *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 below.

CONJECTUREPANEL *name* { IS } [*entry* | *button*]* END: build a panel of conjectures. Each *entry* is one of ENTRY, THEOREM, THEOREMS, PROOF, CURRENTPROOF; each *button* is one of BUTTON, PREFIXBUTTON, RADIOBUTTON, CHECKBOX. ENTRY, THEOREM and THEOREMS add entries to the list of conjectures which forms part of the panel; PROOF and CURRENTPROOF are used in the proof save and reload mechanism.

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 "PREFIXBUTTON Apply IS apply". Like MENU (q.v.), 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 the discussion of flexible syntax below.

CURRENTPROOF *name sequent* { WHERE *proviso* AND ... AND *proviso* } IS *tactic*: same as PROOF (q.v. below), 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; declares that the rule called *rulename* is a 'cut' rule, provided that it meets certain conditions (see below). Applications of the rule will normally be hidden in the box display mode of Jape. This directive is required before Jape will properly interpret the 'tryresolution' variable (see appendix C).

END: closer in lots of directives.

ENTRY *name* { { IS } *tactic* } { MENUKEY *letter* }: used to describe 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 only 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 PREFIXBUTTON (q.v.) is pressed.

FONTS *name*: the font-encoding name *name* is transmitted to the graphical interface. At the time of writing our interfaces only recognise "Konstanz", but watch this space ...

FORMULA *name ... name*: the *name*s are declared to be in the syntactic class FORMULA. See discussion of flexible syntax below.

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; is translated into NOTIN provisos for each of the formulæ of that sequent.

FROM: separator in RULE directive.

HYPFRESH *variable*: *variable* doesn't occur free in any hypothesis of the problem sequent. See FRESH.

HYPHIT *formula*1 <entails> { *formula*2 } 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 the directive to fire.

IDENTITY *rulename*: synonym for STRUCTURERULE IDENTITY; the rule named *rulename* is declared to be a 'identity' rule. Instances of the application of this rule are normally hidden in box display mode.

IN: connective in binding directive.

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 formulæ 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*). See discussion of flexible syntax below.

INFIXC *precedence* [L | R | T] *operatorname* ... *operatorname*: very similar to INFIX (q.v. above), 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 IS [BAG | LIST | FORMULA] *turnstile* [BAG | LIST | FORMULA]: same as SEQUENT directive (see below), except that in box display a JUDGEMENT form is always displayed complete, on a conclusion line. That is, the left-hand side is not treated as a collection of hypotheses in box display.

JUXTFIX *precedence*: defines syntactic precedence of juxtaposition: see discussion of flexible syntax below.

LEFTFIX *precedence leftbra punct*1 ... *punctN*: defines syntactic precedence and form of bracketed form which misses a closing bracket. See discussion of flexible syntax below.

LEFTWEAKEN *rulename*: synonym for STRUCTURERULE LEFTWEAKEN; declares that the rule called *rulename* is 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.

LIST <kind> *names*: see the discussion of flexible syntax below.

MENU *name* IS *entry* .... *entry* END: the effect of the entries (which can be RULE, RULES, 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.

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 the discussion of flexible syntax below.

OBJECT *name*: decorates a parameter in a RULE or THEOREM directive. When the rule is instantiated, the parameter is replaced by a newly-minted object variable rather than an unknown, unless this default assignment is overridden by provision of an argument formula.

OUTFIX *leftbra punct*1 ... *punctN rightbra*: see the discussion of flexible syntax below.

POSTFIX *precedence operator* ... *operator*: see discussion of synctactic directives below

PREFIX *precedence operator* ... *operator*: see discussion of synctactic directives below.

PREFIXBUTTON *label* { IS } *command*: in a panel, a button with label *label* is added. When the button is pressed and an entry which indexes *text* is selected, then the string "*command text*" is passed to the proof engine.

PROOF *name sequent* { WHERE *proviso* AND ... AND *proviso* } IS *tactic*: both PROOF and CURRENTPROOF are generated when you save proofs. You will probably never want to write one yourself, but this is what PROOF means: *sequent* is a statement of the conjecture named *name*, and *tactic*, when run, will produce a proof of that conjecture with the given provisos. 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; in a panel it is a proper Macintosh-style radio button.

RIGHTWEAKEN: same as STRUCTURERULE RIGHTWEAKEN. Similar to LEFTWEAKEN above, and plays a similar rôle in theorem application.

RULE *rule*: definition of a rule. Puts a rule with name *name* into the tactic store; if it appears inside a MENU definition then the effect is also of ENTRY *name* IS *name*. See 'rules, tactics and conjectures' below for more explanation.

RULES *name* { (*params*) } { WHERE *provisos* } ARE *rule*1 AND ... AND *ruleN* END: definition of a number of rules, organised automatically into an ALT tactic.

Each *rule* is an unnamed rule definition (see 'rules, tactics and conjectures below); 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, RULES *name* ... has the effect also of ENTRY *name*: that is, only one entry appears in the menu.

SCOPE: part of the BIND directive.

SEPARATOR: used in the definition of a MENU, gives a division between entries. On a panel, probably has no effect.

SEQUENT IS BAG | LIST | FORMULA] <entails> BAG | LIST | FORMULA]: see discussion of flexible syntax below.

STRING *name*, ... , *name*: see the discussion of flexible syntax below.

STRUCTURERULE *kind* IS *name*: *kind* must be CUT, IDENTITY, LEFTWEAKEN, RIGHTWEAKEN or WEAKEN; *name* must be the name of a rule in the tactic store. See 'structural rules' below, and see the discussion of each of the *kind*s in this list.

SUBSTFIX *precedence* { *bra fst sep snd ket* }: defines the syntactic precedence of substitution forms and, optionally, their appearance as well. See the discussion of flexible syntax below.

TACTIC *name* { (*name*1, ... , *nameN*) } { IS } *tactic*: puts a tactic with name *name* and parameters *name*1, ... , *nameN* into the tactic store. If it appears inside a MENU or PANEL definition then the effect is also of ENTRY *name*; in a TACTICPANEL it has the effect of ENTRY *name* IS apply *name*. See ‘rules, tactics and conjectures’ below for more explanation.

TACTICPANEL *name* { IS } [*entry* | *button*]* END: very like CONJECTUREPANEL above, except that there are no extra buttons and no default buttons and each entry labels a command rather than a conjecture. Tactics, rules and conjectures included in a TACTICPANEL description are associated with the command “apply *name*” where *name* is the name of the tactic, rule or 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 *sequent*. See ‘rules, tactics and conjectures’ below for more explanation.

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 ‘rules, tactics and conjectures’ below); each theorem is added to the tactic store prefixed by *params* and *provisos* in the same way as in the RULES directive (q.v. above); 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 above.

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).

formula UNIFIESWITH *formula*: a proviso which requires that the two formulæ 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*: see the discussion of identifier classes below.

WEAKEN: synonym for LEFTWEAKEN (q.v. above).

WHERE: prefixes a list of proviso declarations.

A.2 Rules, tactics and conjectures

The syntax of rules and conjectures 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 } *sequent*

In a rule, *body* is

{ FROM *antecedent* AND ... AND *antecedent* } { INFER } *consequent*

If the conjecture is un-named, its name is taken to be *sequent*; if a rule is un-named, its name is taken to be *consequent*

Each *param* is either a name, OBJECT followed by a name or ABSTRACTION followed by a name: in every case the name must have been declared in a CLASS directive. Each proviso is either

CONCFRESH *name*, ... , *name*
HYPFRESH *name*, ... , *name*
FRESH *name*, ... , *name*
name NOTIN *formula*
formula UNIFIESWITH *formula*

where the names used must be parameters of the rule or conjecture.

The meaning of a RULE *directive*

Ignoring parameters for the moment, the rule

$$\frac{ante1 \quad \ldots \quad anteM}{conse}\,(prov1,\ldots,provN)\ rule$$

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 *ante*1 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.

Rule matching is linear – formulæ that are matched in the consequent are used up, and aren’t copied to the antecedent unless you write them there. For example the sequent calculus ‘→ on the left’ rule

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\to\vdash$$

is stated as

RULE “→⊢” IS FROM Γ ⊢ *A*, Δ AND Γ, *B* ⊢ Δ INFER Γ, *A*→*B* ⊢ Δ

and the ‘multiplicative’ or ‘context-splitting’ version of that rule

$$\frac{\Gamma 1 \vdash A, \Delta 1 \quad \Gamma 2, B \vdash \Delta 2}{\Gamma 1, \Gamma 2, A \to B \vdash \Delta 1, \Delta 2}\to\vdash$$

as

RULE “→⊢” IS FROM Γ1 ⊢ *A*, Δ2 AND Γ2, *B* ⊢ Δ2 INFER Γ1, Γ2, *A*→*B* ⊢ Δ1, Δ2

If you want a version in which the implication formula is not used up

$$\frac{\Gamma, A \to B \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\to\vdash$$

then you can have it:

RULE “→⊢” IS FROM Γ, *A*→*B* ⊢ *A*, Δ AND Γ, *B* ⊢ Δ INFER Γ, *A*→*B* ⊢ Δ

If either of the *autoAdditive* variables is set to *true* (see appendix C) then some of these rules can be defined as rules often are in natural deduction presentation, without mentioning unmatched hypotheses and/or conclusions. For example, if *autoAdditiveLeft* is *true*, the rule

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

can be stated as

RULE "$\rightarrow\vdash$" IS FROM A AND $B \vdash C$ INFER $A \rightarrow B \vdash C$

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 below and in various chapters above);
- 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 – see below.
- parameters are used to drive the 'proof reload' mechanism.

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 (see above), 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 formulæ 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.

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 above. 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.

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 the variable *applyconjectures* (see appendix C).

If the consequent of a theorem matches a problem sequent, but in so doing it doesn't use up all the hypotheses and conclusion formulæ 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.

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 the *tryresolution* variable (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.

The meaning of STRUCTURERULE directives

It is necessary for the application of conjectures as rules (see above), 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.

At present Jape will recognise a rule as of the right form if it is in one of the following patterns (Γ is a bag of formulæ, Δ a list of formulæ, B and C are formulæ).

CUT (B) FROM Γ ⊦ B AND Γ, B ⊦ C INFER Γ ⊦ C
(B) FROM Γ ⊦ B,Γ' AND Γ, B ⊦ Γ' INFER Γ ⊦ Γ'
(B) FROM Γ ⊦ B,Γ' AND Γ'', B ⊦ Γ''' INFER Γ,Γ'' ⊦ Γ',Γ'''

WEAKEN, LEFTWEAKEN
(B) FROM Γ ⊦ C INFER Γ,B ⊦ C
(B) FROM Γ ⊦ Γ' INFER Γ,B ⊦ Γ'

RIGHTWEAKEN
(B) FROM Γ ⊦ Γ' INFER Γ ⊦ B,Γ'

IDENTITY
Γ,B ⊦ B
Γ,B ⊦ B,Γ'
Δ,B,Δ' ⊦ B

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 Dijkstra's weakest precondition calculus assignment rule $\mathrm{wp}(x := e, R) = R_x^e$, expressed in the notation of Hoare logic:

$$\frac{}{\{R_x^e\}x := e\{R\}} :=$$

This can be expressed in Japeish as follows (see the file hoare_rules.j):

```
RULE ":=" IS INFER <{R[x\E]} x:=E {R}>
```

Now suppose that we apply this rule to the problem sequent `<{_Q} x:=1 {x=1}>`. If we apply the asignment rule then a version will be generated expressed in terms of new unknowns, which will be `<{_R[_x1\_E]} _x1:=_E {_R}>`. However the unification proceeds, it will eventually have unified `(x=1)[x\1]` 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[_x1\_e]` 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)[x\1]` with `1=1`: the substitution form simplifies to `1=1` and the unification is trivial.

But 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 ∀ elimination rule:

$$\frac{\Gamma \vdash \forall x.P}{\Gamma \vdash P[x \setminus E]} \ \forall\,\mathrm{elim}$$

If the problem sequent is, say, $a = b, b = c \vdash (a+b)+c = a+(b+c)$, 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. A statement of the rule which can make use of either mechanism is

RULE "∀ elim"(E, OBJECT x) IS FROM $\Gamma \vdash \forall x.P$ INFER $\Gamma \vdash P[x\backslash E]$

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 lists of matches above. The abstraction mechanism is a kind of higher-order unification.

The first parameter of the rule, *E*, can be replaced by an argument which is provided by text-selection (if you provide no argument then an unknown will silently be used): if you text-select *a*, for example, then *E* must be replaced by *a* in the version of the rule that is generated. The second parameter, *x*, is declared to be an OBJECT. This means that instead of generating an unknown, a freshly-minted identifier will be used in place of *x* in the version of the rule that is generated. The parameter declarations say nothing about the name *P*: it will always be replaced by an unknown.

Suppose, now, that you text-select *a* and apply the rule. The generated consequent will be $\Gamma \vdash \_P[x\backslash a]$, and the problem sequent $a = b, b = c \vdash (a+b)+c = a+(b+c)$. Jape will try to unify $\_P[x\backslash a]$ 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 formulæ, but pragmatically it is often 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 called user-defined substitution matching. The mechanism works together with the WITHSUBSTSEL tactic (see appendix B and chapter 5). 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.

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 Fonts

Presentations of logics in textbooks and technical papers make use of special logical symbols, sometimes invented specially for the purposes of that particular logic, which can't easily be represented in the ASCII character set. In polished presentations of logics in Jape we use special fonts – but it is perfectly possible to use, and in the first instance you may want to use, combinations of ASCII characters to approximate the special characters. Whatever font you use, it will need at least to include the normal ASCII characters for identifiers, numbers, round and square brackets, quotation marks, backslash and comma as well as the special characters you need for your logical connectives and operators.

A description of a logic will therefore often begin by describing the font in which the logic is intended to be viewed. This will necessarily also be the font in which the logic description itself is written, and therefore you will need an 8-bit ASCII text editor to create and modify the logic description[1]. In practice the graphical interface part of Jape may use more than one font and more than one character size to handle proof display, menus and buttons. You therefore use the name of a ‘font encoding’ to describe the whole scheme to Jape. The graphical examples in this manual are taken from the current implementation of Jape on MacOS, and they use Roy Dyckhoff’s Konstanz and Detroit fonts in various sizes[2]. The name of the encoding is Konstanz, and there is a similar encoding in X Jape. The directive used in this and the other encodings in this manual is therefore

```
FONTS "Konstanz"
```

Other encodings are in preparation.

A.4 Flexible syntax

There are various ways in which you control what formulæ can be written down and how they will be interpreted by Jape.

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 formulæ, variables, or whatever[3].

[1] Under MacOS, we’ve found BBEdit Lite (freeware) very useful in preparing Jape logic descriptions, and thank its authors for their skill and philanthropy. Under X, Bernard has produced an 8-bit editor of his own called jed; it’s part of the standard UNIX distribution of Jape.

[2] Konstanz and Detroit fonts were produced by Roy Dyckhoff as part of the MALT project at the University of St Andrews. We are grateful to Roy for permission to use them and to distribute them with MacOS Jape.

[3] 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 ...

Juxtaposition may need care

Jape’s syntax allows juxtaposition of formulæ. 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.

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 formulæ, 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> *names*
CLASS <kind> *names*

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 which are 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.

Unknowns follow the same rules as identifiers: given the directives above, *_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.

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*

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[1]; 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.

Bracketed formulæ

Jape recognises bracketed formulæ 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 formulæ of the form

letrec *formula* be *formula* in *formula*

LEFTFIX formulæ 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 $\forall x.A B$, but not $C\forall\ x.AB$. 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 $C(\ \ \forall x.AB\ \ )$.

[1] Substitution *has* to associate to the left, but we can imagine right-associative juxtaposition. Another enhancement for the fugure (sigh).

Substitution 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.

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 formulæ are parsed.

As in many modern programming languages, we permit a bracketed operator as a formula, so you can write formulæ like (+), (∧), (++) once those symbols have been defined as operators. Operation formulæ 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* is parsed as (*infixop*) (*f1*,*f2*) and *f1 infixCop f2* is parsed as (*infixCop*) *f1 f2*; the reverse transcription is made when the formulæ are printed out.

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.

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 formulæ – 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.

In version 3.2 we have introduced a JUDGEMENT directive. This works in just the same way as SEQUENT, except that in box display a JUDGEMENT is always written on a single line – that is, its lhs is not interpreted as a collection of hypotheses and its rhs a conclusion. We are fairly sure we have chosen the wrong word for this directive (and indeed for SEQUENT). Watch this space or send us a suggestion.

Future work

In the future we intend to provide a more powerful form of syntax definition for Jape's users, providing in particular a more efficient means of defining binding forms and ways of making finer distinctions between syntactic categories. More structure in sequent forms would be a step still further, and we don't yet envisage it.

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 formulæ.

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].

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 formulæ 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.

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.

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 5.

FOLD *rulename tactic*: Automatically 'folds' collections of rules. See chapter 5 above.

FOLDHYP *pattern tactic*: Automatically 'folds' hypotheses. See chapter 5 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 formulæ, 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 subformulæ 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.

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 *formulæ*: Unify the given formulæ 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.

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 formulæ.
showallproofsteps	true, false	true	(misnamed – should be showhiddenproofsteps) when true, show proof steps hidden by LAYOUT tacticals.
showallprovisos	true, false	false	(misnamed – should be showhiddenprovisos) when true, show hidden provisos, marked as <proviso>.
tryresolution	true, false	true	apply theorems and antecedent-free rules in ‘resolution’ style if the conclusions of the consequent match but the hypotheses don’t.
uncollapsedfmt	any string	"%s"	string that controls the display of a subtree that was once collapsed and is now reinflated.
unhiddenfmt	any string	"[%s]"	string that controls the display of a subtree produced by the LAYOUT tactical and displayed in ‘normal’ form. This string is over-ridden if *string2* is provided in the LAYOUT tactical.

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.

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.