Animating Operational Semantics with JAPE

Bernard Sufrin*
Richard Bornat†

Revision 1.25‡

Abstract

In this note we give a brief introduction to the ideas of operational semantics and show how to use JAPE to animate the operational semantics of a couple of simple programming languages. We give the syntax and inference rules for each language in JAPE's metalanguage, then define tactics suitable for automating the choice of rules during the simulated execution of a program.

We assume that the reader has used JAPE to do formal proofs in a conventional logic, but we do not expect detailed knowledge or understanding of JAPE's metalanguage or tactic language.

1 Introduction

JAPE was designed to make it easy to work with inference systems. Although our main aim was to help people to learn how to do fully formal proofs in more-or-less standard logics, an important subsidiary goal was to provide a means of bringing other kinds of formal system to life. Amongst other applications we had in mind were type-inference, Plotkin-style Structured Operational Semantics (small-step semantics), and Kahn-style Natural Semantics (big-step semantics). In this note we show how to use JAPE to animate the operational semantics of a couple of simple programming languages.

The most notable early approaches to specifying the semantics of programming languages operationally did so by defining abstract machines with instructions which were directly derived from program phrases – for example [Lan64]. Such approaches have the advantage of being very nearly direct interpreters for the language being defined; on the other hand, the order in which essential computations take place tends to be overspecified (relative to a more abstractly given semantics) by such an approach.

*Programming Research Group and Worcester College, Oxford
†Queen Mary and Westfield College, London
‡14:55 22nd September 2000

Plotkin's method of Structured Operational Semantics[Plo81], and Kahn's refinement, Natural Semantics[Kah87] are a more abstract approach, in which computations are defined by inference systems and the ordering of essential computations is specified only implicitly by the dependency relations between parts of an inference rule.

In the next section we illustrate both Structural and Natural semantic methods by using them to define computations in a declarative language based on pure lambda calculus.

2 The Lambda Language

2.1 Syntax

LAMBDA is a family of languages based on the λ calculus. Its forms of expression are outlined below (we use names based on uppercase E to denote expressions, names based on lowercase x, y, z to denote variables, and names based on *num* to denote numbers).

$\lambda\, x \bullet E$	Abstraction
$E_1\ E_2$	Application
let $x = E_1$ **in** E_2	Block
$E_1 + E_2$	Sum
x	Variable
num	Number

The first kind of expression denotes a function abstraction: "that function of x whose value is E"; the second kind of expression denotes a function application: "apply the function E_1 to the operand E_2"; the value of a block is the value of E_2 in a context where x is taken to have the value of E_1. As usual, function application is syntactically more binding to the left than to the right, so that $(E_1\ E_2)\ E_3$ may be written without parenthesis as $E_1\ E_2\ E_3$. In order to simplify our presentation we will equip LAMBDA with only one arithmetic operation, namely integer addition.

2.2 Substitution Semantics of LAMBDA

Computation takes place as a sequence of steps, each of which transforms the whole program in some way. The relation $\rightarrow$, defined inductively in Figures 1 through 3, characterises a single such step. As always in an operational semantics there are two main categories of rule. The *computation rules* (Figure 1) describe what happens when a function is applied – whether it be an abstraction[1] or the built-in arithmetic primitive. The *structural rules* (Figure 2) describe the way in which computations may take place *within* program structures as well as at the top level.

[1] The notation $E[E'/x]$ stands for the expression E with all *free* occurences of x replaced by E'.

$$\frac{}{(\lambda x \bullet E)\ E' \rightarrow E[E'/x]}\ \mathit{Beta}$$

$$\frac{(\mathit{num}' \text{ is the sum of the numbers } \mathit{num}_1, \mathit{num}_2)}{\mathit{num}_1 + \mathit{num}_2 \rightarrow \mathit{num}'}\ \mathit{Addition}$$

Figure 1: Computation rules for LAMBDA

$$\frac{E_1 \rightarrow E_1'}{E_1\ E_2 \rightarrow E_1'\ E_2}\ \mathit{Operator}$$

$$\frac{E_2 \rightarrow E_2'}{E_1\ E_2 \rightarrow E_1\ E_2'}\ \mathit{Operand}$$

$$\frac{E_1 \rightarrow E_1'}{E_1 + E_2 \rightarrow E_1' + E_2}\ \mathit{Add}_L$$

$$\frac{E_2 \rightarrow E_2'}{E_1 + E_2 \rightarrow E_1 + E_2'}\ \mathit{Add}_R$$

Figure 2: Structural rules for LAMBDA

The last remaining rule (Figure 3) is neither structural nor computational in content: it defines the meaning of a block in terms of an abstraction and an application. It is called a *sugaring rule.*

$$\frac{}{\textbf{let}\ x =\ E'\ \textbf{in}\ E \rightarrow (\lambda x \bullet E)(E')}\ \mathit{Let}$$

Figure 3: Sugaring rule for LAMBDA

A computation rule may be directly applicable to the top-level of a program, in which case no structural rules need be applied.

Example 2.2.1: A computation step using *Addition*

$$\frac{}{4 + 4 \rightarrow 8}\ \mathit{Addition}$$

Example 2.2.2: A computation step using *Beta*

$$\frac{}{(\lambda x \bullet x + x)(4) \rightarrow 4 + 4}\ \mathit{Beta}$$

On the other hand computation rules are not *always* applicable at the top level of the program, and structural rules must, in general, be used to locate an expression at which a computation can take place.

Example 2.2.3: A computation step using *Operand* and *Addition*

$$\dfrac{\dfrac{}{(2+2)\rightarrow 4}\ \textit{Addition}}{(\lambda x \bullet x + x)(2+2)\rightarrow(\lambda x \bullet x + x)(4)}\ \textit{Operand}$$

2.3 Irreducible Expressions and Normal Forms

An expression is said to be *reducible* if at least one of the above rules apply to it, and *irreducible* otherwise. It may be irreducible because the computation from which it arose has finished, (in which case it is said to be in *normal form*), or because it is in some sense erroneous (in which case it is said to be *stuck*). The normal forms of LAMBDA are defined to be numbers and abstractions – all other irreducible forms are stuck.

Example 2.3.1: $3 + 3$ is not irreducible.

Example 2.3.2: 3 is a normal form.

Example 2.3.3: ˜3 (negative 3) is a normal form.

Example 2.3.4: $\lambda x \bullet x + x$ is a normal form.

Example 2.3.5: $\lambda x \bullet 3 + 3$ is a normal form.

The expression to the left of the arrow in the consequent of a rule is called the *subject* of the rule. An expression is obviously irreducible if it does not match the subject of any of the LAMBDA rules, but it may do so and still be irreducible.

Example 2.3.6: 3 3 is stuck, despite matching the subject of both *Operator* and *Operand*. This is because neither its operator nor its operand are reducible, so the antecedents of the rules whose subjects it matches are unprovable.[2]

2.4 Computations

A *finite computation* is a sequence of expressions $E_0, ...E_n$ such that $0 \leq i < n \Rightarrow E_i \rightarrow E_{i+1}$. This is true if and only if $0 \leq i \leq j \leq n \Rightarrow E_i \overset{*}{\rightarrow} E_j$, where $(\overset{*}{\rightarrow})$ is the reflexive transitive closure of the single-step relation. This relation is defined inductively by the rules in figure 4.

[2]The form of meta-logical reasoning employed here can in general be used to prove many kinds of property of a language defined by an inference system, but we shall not do so in this note.

$$\frac{E_1 \rightarrow E_2}{E_1 \overset{*}{\rightarrow} E_2}\ \textit{Step}$$

$$\frac{E_1 \overset{*}{\rightarrow} E_2 \quad E_2 \overset{*}{\rightarrow} E_3}{E_1 \overset{*}{\rightarrow} E_3}\ \textit{Transitive}$$

$$\frac{}{E \overset{*}{\rightarrow} E}\ \textit{Identity}$$

Figure 4: Transitive Computation Rules

It can be shown that the rules which define LAMBDA are nondeterministic – in the sense that the sequence of expressions in a computation which starts at a given expression is not strictly determined. For example, the distinct computations below start at the same expression

$$(3 + 4) + (5 + 6) \rightarrow 7 + (5 + 6) \rightarrow 7 + 11 \rightarrow 18$$
$$(3 + 4) + (5 + 6) \rightarrow (3 + 4) + 11 \rightarrow 7 + 11 \rightarrow 18$$

as do

$$(\lambda x \bullet x + x)(3 + 4) \rightarrow (\lambda x \bullet x + x)(7) \rightarrow 7 + 7 \rightarrow 14$$
$$(\lambda x \bullet x + x)(3 + 4) \rightarrow (3 + 4) + (3 + 4) \rightarrow 7 + (3 + 4) \rightarrow 7 + 7 \rightarrow 14$$
$$(\lambda x \bullet x + x)(3 + 4) \rightarrow (3 + 4) + (3 + 4) \rightarrow (3 + 4) + 7 \rightarrow 7 + 7 \rightarrow 14$$

On the other hand it can also be shown that if two computations starting with the same expression both reach a normal form, then the normal forms are the same. In other words the normal form reached by a terminating computation *is* determined by the starting expression. It would be easy to modify the rules to make computations deterministic, and we shall eventually show how to do so. But first we will explore the present collection of rules using JAPE — designing tactics which characterise the two principal *systematic* methods of mechanising the inferences by which LAMBDA computations can be conducted, namely *normal order evaluation* and *applicative order evaluation*.

2.5 Structural Semantics of LAMBDA in JAPE

Defining the Notation

When presenting an inference system to JAPE we first describe its syntax. Since we will be using a nonstandard character coding to write the expressions of our language, we start by declaring this fact:

```
FONTS "Konstanz"
```

We adopt the convention that (identifiers beginning with) lowercase letters stand for variables, and (identifiers beginning with) uppercase letters for formulæ of some kind.[3] Names beginning with *num* will stand for numbers.

```
CLASS   VARIABLE a b c d e f g h i j k l m n o p q r s t u v w x y z
CLASS   FORMULA  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
CLASS   NUMBER   num
```

The phrases of any object language which we study with JAPE are a subset of JAPE's generic expression language. This language admits expression-forming operators of various fixities and arities in addition to the usual atomic forms. An object language is defined by declaring the fixity, arity and syntactic scopes (binding powers) of its operators. Here we declare the main composite forms of the LAMBDA notation.

```
LEFTFIX 20 λ •
INFIX   1T → →*
INFIX   100T =
INFIX   200T +
PREFIX  1000 ~
LEFTFIX 10   let in
```

We also need to declare all the variable binding forms in our object language and indicate the scope of the variables bound therein.

```
BIND x SCOPE T IN λ x • T
BIND x SCOPE T IN let x=S in T
```

JAPE's rules are expressed in the form of sequents. Here we declare the principal form of sequent we will use. For (trivial) technical reasons JAPE presently requires that sequents be two-sided, but in this work, the LIST of formulæ to the left of the turnstile will always be empty.

```
SEQUENT IS LIST ⊢ FORMULA
```

Finally we declare the notation we shall be using to denote substitutions, and its syntactic binding power – higher than that of any other symbol used in this theory.

```
SUBSTFIX 500000 [ S / s ]
```

Coding the Rules

A direct JAPE encoding of the rules outlined above is straightforward to construct. Here we do so in the scope of a declaration which places buttons which invoke them on a menu called `Lambda Rules`.

[3] Here and subsequently JAPE input will be presented in a typewriter-like face.

```
MENU "Lambda Rules"
    RULE  Beta    ⊢ (λ x • S) T  →  S[T/x]
    RULE  Let     ⊢ let x=T in S →  (λ x • S)(T)
    ENTRY Addition IS Add
    SEPARATOR
    RULE  Rator  FROM  ⊢ S→S'  INFER ⊢ S T →  S' T
    RULE  Rand   FROM  ⊢ T→T'  INFER ⊢ S T →  S  T'
    RULE  AddL   FROM ⊢ E1→E1' INFER ⊢ E1+E2 → E1'+E2
    RULE  AddR   FROM ⊢ E2→E2' INFER ⊢ E1+E2 → E1+E2'
    SEPARATOR
    RULE Transitive  FROM  ⊢ A→*B AND ⊢ B→*C INFER ⊢ A→*C
    RULE Step        FROM  ⊢ A→B INFER ⊢ A→*B
    RULE Identity    INFER ⊢ A→*A
END
```

The only notable thing here is the use of the tactic `Add` to implement addition. It does so by invoking the `Addition` rule, which transforms a goal of the form `T1+T2→T3` into one of JAPE's built-in arithmetic goals `"ADD"(T1,T2,T3,(~))`. `Add` then solves the latter by invoking the built-in decision procedure for such goals, namely `EVALUATE`. The use of the decision procedure is concealed in the proof by embedding its work in `LAYOUT` tactics.

```
TACTIC Add()    IS (LAYOUT "Add" () Addition (LAYOUT "" () EVALUATE))
RULE   Addition IS FROM  ⊢ "ADD"(T1, T2, T3, (~)) INFER  ⊢ T1+T2→T3
```

Exploring the reduction relation

Before defining systematic computation strategies, we will proceed for a while to explore the computation relation *ad-hoc*. The way to do this in JAPE is to build a panel of conjectures of the form $E \stackrel{*}{\to} \_T$ and attempt to prove them. The "$\_T$" is a "proof unknown" – a (meta-) variable whose value will be determined during the proof.

```
   CONJECTUREPANEL "Structural Semantics"
    THEOREMS AdHoc
    ARE ⊢ let f = (λ x • x) in f f 1 →* _T
    AND ⊢ (1+2)+(3+1)→*_T
    AND ⊢ (1+2)+(~4)→*_T
    AND ⊢ let x = 1+2 in x+x →*_T
    AND ⊢ let c = (λ f • λ g • λ a • f(g a)) in
          let f = (λ a • a) in c f f →* _T
    AND ⊢ let g = (λ f • f f) in g g →* _T
    AND ⊢ let g = (λ f • f f) in
          let k = (λ a • λ b • a) in k 2 (g g) →* _T
    END
END
```

After initializing JAPE with the theory **lambda.jt**, we can select conjectures from the panel for proof, then invoke rules from the "Lambda Rules" menu. Below we give an account of two such proofs.

Example 2.5.1: A normal form for `(1+2)+(3+1)`

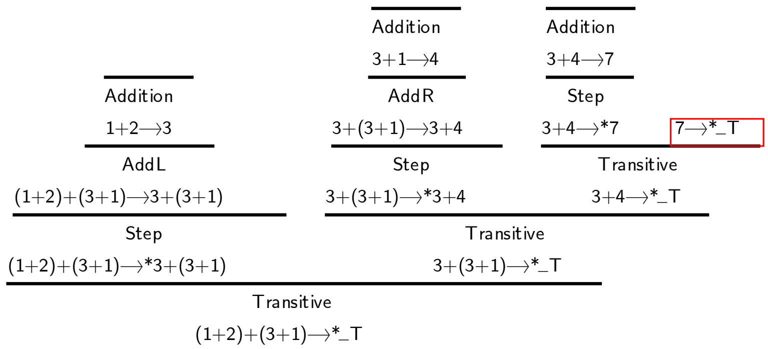

Figure 5: Near-complete derivation of a normal form for $(1 + 2) + (3 + 1)$ (sum0)

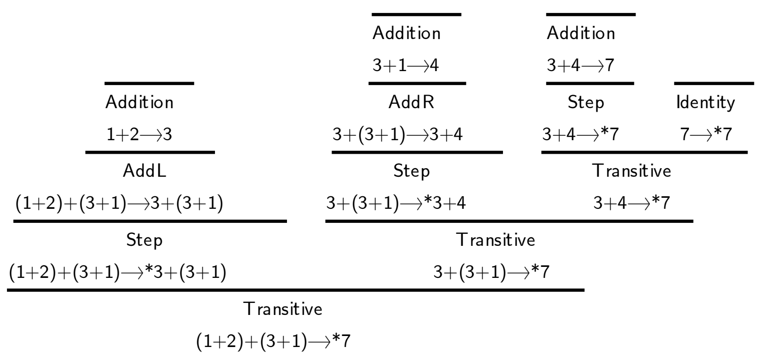

Figure 6: Completing the derivation (sum1)

In Figure 5 we show a nearly-complete derivation of a normal form for $(1+2)+(3+1)$. Notice that we have been systematic in our derivation: each computation step, including the last, is set up by an application of the *Transitive* rule followed by an application of the *Step* rule. Once the expression has been reduced to normal form this way, the computation is completed (Figure 6) by an application of the *Identity* rule.

Example 2.5.2: A normal form for `let f = (λ y • y) in f f 1`

Figures 7 through 12 show how a computation of a normal form for a more complicated expression might proceed *under human guidance.*

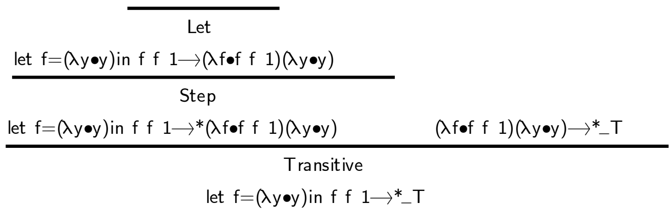

Figure 7: First computation step (lam-adhoc0)

We set up the first computation step, a *Let*, by applying *Transitive* then *Step* (Figure 7).

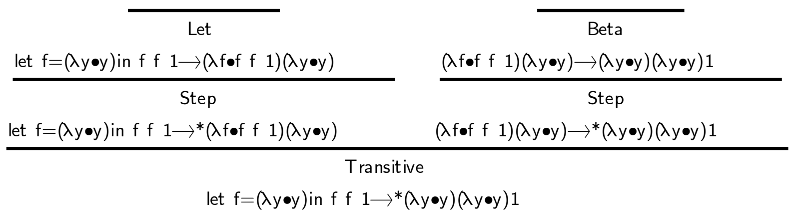

Figure 8: Mistaken second computation step (lam-adhoc1)

We then make a mistake, and erroneously omit the use of *Transitive* in setting up the second step (Figure 8), so although this step can itself be completed with *Beta*, it leaves the result in reducible form but with no scope for further work.

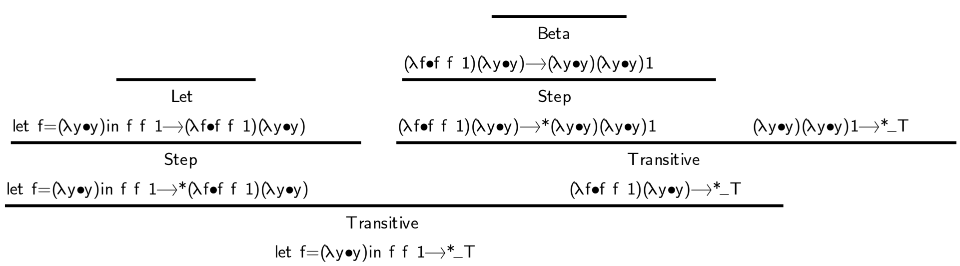

Figure 9: Correct second computation step (lam-adhoc2)

In Figure 9 we correct the earlier mistake – this time setting up the computation properly with *Transitive*.

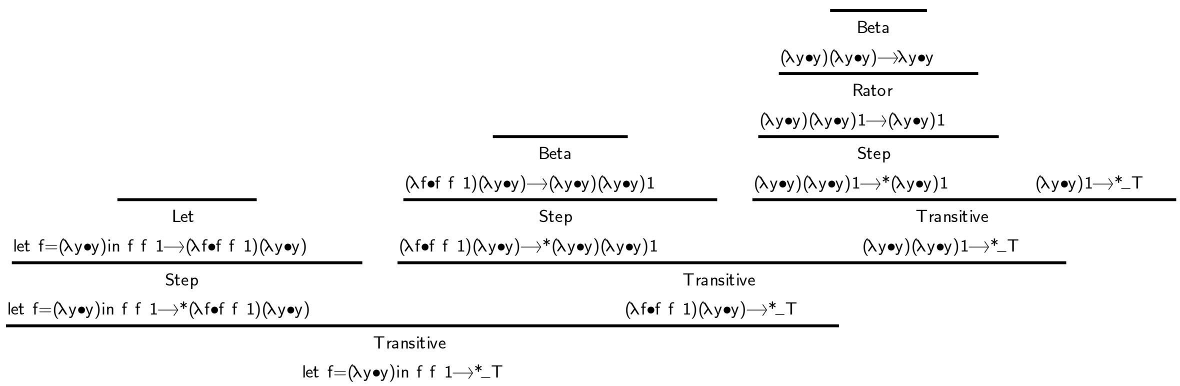

Figure 10: Just one *Beta* step to termination (lam-adhoc3)

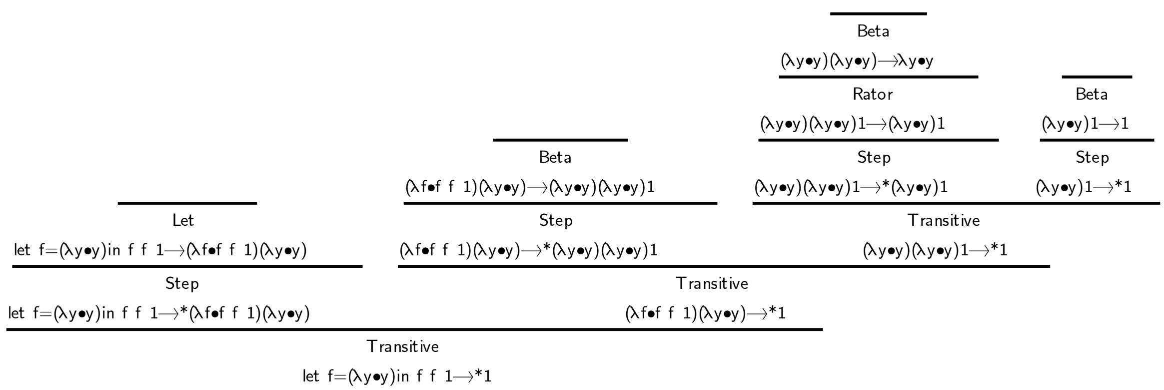

Figure 11: Non-systematic, but succesful, application of `Step,Beta` (lam-adhoc4)

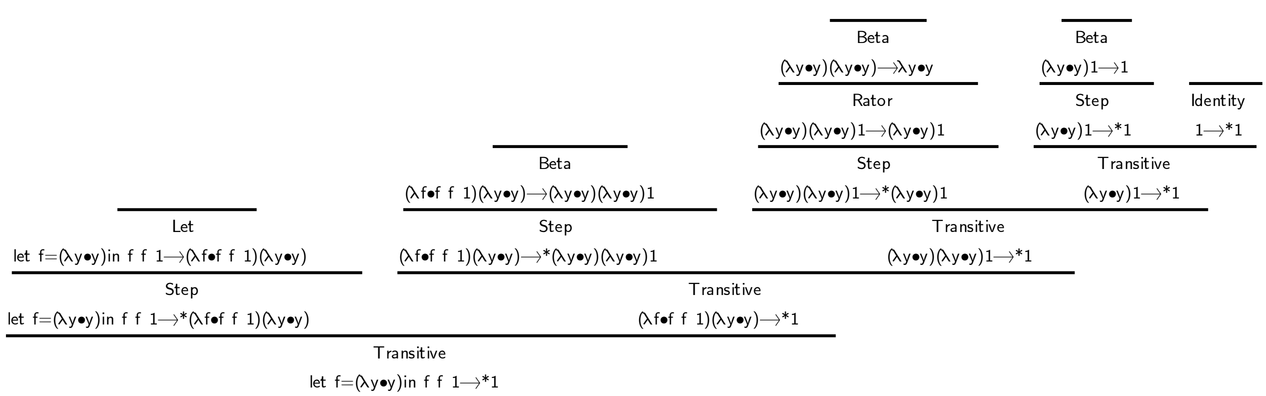

Figure 12: The systematically completed computation (lam-adhoc5)

A few more inferences bring us to the situation shown in Figure 10. At this point we notice that a single *Beta* step will complete the computation, so do not bother to apply *Transitive* before setting it up. This leads to the completed proof in Figure 11.

Whilst it's fairly realistic to expect a human to have the insight to recognise a situation which is one computation step away from a normal form, it is unreasonable to this to be *cheaply* automatable. So it is worth exploring the systematic use of *Transitive* to set up each computation step, including the last. It turns out that for this computation it takes only a couple more inferences than the computation shown in Figure 11. The final step of the derivation is the identity step, which is taken when the expression has reached normal form. The result of doing so is shown in Figure 12.

This preliminary exploration of the reduction relation shows that computation *can* be conducted step-by-step by under human guidance in much the same way as proofs can be discovered. But if we are to justify the rules we have given as an *operational semantics*, then we shall need to automate the process of computation-by-proof so that it can be conducted without insight but without mistakes.

A superficial interlude – improving the interface

When JAPE is used this way to animate a program of nontrivial size it can be distracting to have to look at the details of the application of the structural rules. In most circumstances someone who is exploring the semantics or behaviour of a program can understand the computation steps which have taken place by reading the sequence of expressions in the (top-level) computation. JAPE provides machinery which lets us automatically conceal parts of a completed derivation, and in this section we shall show how to deploy that machinery by building an interaction tactic which will conceal most of the distracting clutter. At the same time we will will automate the application of *Transitive* and *Step* rules when finding a computation sequence.

The interaction tactic `TransitiveStep(rule)` arranges for its argument, *rule*, to be applied. It applies *rule* after *Transitive* and *Step* if it is used when the current goal is a $\rightarrow*$-judgement. On the other hand, it applies only *rule* if used "inside" a computation step, *i.e.* when the current goal is a $\rightarrow$-judgement. It declares, (using `LAYOUT`) that the details of a computation step set up in this way can be suppressed once the step is complete (*i.e.* the proof tree above *Step* is closed).

```
TACTIC TransitiveStep(rule)
(WHEN (LETGOAL (_E →* _T) Transitive (LAYOUT "%s" () Step rule))
      (LETGOAL (_E →  _T) rule))
```

The tactic `Finished` applies the `Identity` rule – thereby terminating the computation – if the subject expression is in normal form.

```
TACTIC Finished()
(WHEN (LETGOAL (_num        →* _T) Identity)
      (LETGOAL (~ _num      →* _T) Identity)
      (LETGOAL ((λ _x • _E) →* _T) Identity))
```

We provide an improved interface to the semantics by adding buttons which invoke rules under the control of `TransitiveStep`.

```
CONJECTUREPANEL "Structural Semantics"
     BUTTON Beta      IS apply TransitiveStep Beta
     BUTTON Let       IS apply TransitiveStep Let
     BUTTON Rator     IS apply TransitiveStep Rator
     BUTTON Rand      IS apply TransitiveStep Rand
     BUTTON AddL      IS apply TransitiveStep AddL
     BUTTON AddR      IS apply TransitiveStep AddR
     BUTTON Addition  IS apply TransitiveStep Add
     BUTTON Finished  IS apply Finished
END
```

Example 2.5.3: Normal form for a simple block

Figures 13 through 16 show, using JAPE's Fitch-box display style, the derivation of a normal form for a simple block. Each proof step was made using the buttons on the conjectures menu to invoke the rules. The first step taken is to desugar the *Let*, and this leaves the situation shown in Figure 13. After we decide to simplify the operand of the resulting application and apply *Rand*, the situation is as shown in Figure 14. The goal for us to address is on line 2, and the *Step* and *Rand* inferences we took to get to this state are shown explicitly – because this branch of the proof is not yet closed. We close it by invoking *Add*, and JAPE reports the situation as shown in Figure 15. Notice that the details of the proof branch have been suppressed, and that only the "bottom line" (line 2 of Figure 15) is shown.

1	$\mathsf{let\ x = 1+2\ in\ x+x} \overset{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x+x})(1+2)$	Let
	...	
2	$(\lambda \mathsf{x} \bullet \mathsf{x+x})(1+2) \overset{*}{\to} \_\mathsf{T}$	
3	$\mathsf{let\ x = 1+2\ in\ x+x} \overset{*}{\to} \_\mathsf{T}$	Transitive 1, 2

Figure 13: After *Let* (let-improved0)

1	$\mathsf{let\ x = 1+2\ in\ x+x} \overset{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x+x})(1+2)$	Let
	...	
2	$1+2 \to \_\mathsf{T}'$	
3	$(\lambda \mathsf{x} \bullet \mathsf{x+x})(1+2) \to (\lambda \mathsf{x} \bullet \mathsf{x+x})\_\mathsf{T}'$	Rand 2
4	$(\lambda \mathsf{x} \bullet \mathsf{x+x})(1+2) \overset{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x+x})\_\mathsf{T}'$	Step 3
	...	
5	$(\lambda \mathsf{x} \bullet \mathsf{x+x})\_\mathsf{T}' \overset{*}{\to} \_\mathsf{T}$	
6	$(\lambda \mathsf{x} \bullet \mathsf{x+x})(1+2) \overset{*}{\to} \_\mathsf{T}$	Transitive 4, 5
7	$\mathsf{let\ x = 1+2\ in\ x+x} \overset{*}{\to} \_\mathsf{T}$	Transitive 1, 6

Figure 14: After *Rand* (let-improved1)

If we focus on the top two lines of this figure, we see the essence of the computation so far. There is a lot of noise and repetition in this presentation, and it would be much easier to understand if the deduction were shown as follows

1	$\mathsf{let\ x = 1 + 2\ in\ x + x} \stackrel{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})(1 + 2)$	Let
2	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})(1 + 2) \stackrel{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})3$	Rand, Add
	...	
3	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})3 \stackrel{*}{\to}$ _T	
4	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})(1 + 2) \stackrel{*}{\to}$ _T	Transitive 2, 3
5	$\mathsf{let\ x = 1 + 2\ in\ x + x} \stackrel{*}{\to}$ _T	Transitive 1, 4

Figure 15: After *Add* (let-improved2)

```
let x=1+2 in x+x
    →* (λ x • x+x)(1+2)
    →* (λ x • x+x) 3
    ...
    →* _T
```

As we shall soon see, JAPE can easily be instructed to present proofs involving transitive relations in this way.

Continuing the derivation by invoking *Beta* then *Add* leads, with no surprises, to the situation in Figure 16, from which the derivation can be formally completed by the *Identity* rule.

1	$\mathsf{let\ x = 1 + 2\ in\ x + x} \stackrel{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})(1 + 2)$	Let
2	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})(1 + 2) \stackrel{*}{\to} (\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})3$	Rand, Add
3	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})3 \stackrel{*}{\to} 3 + 3$	Beta
4	$3 + 3 \stackrel{*}{\to} 6$	Add
	...	
5	$6 \stackrel{*}{\to}$ _T	
6	$3 + 3 \stackrel{*}{\to}$ _T	Transitive 4, 5
7	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})3 \stackrel{*}{\to}$ _T	Transitive 3, 6
8	$(\lambda \mathsf{x} \bullet \mathsf{x} + \mathsf{x})(1 + 2) \stackrel{*}{\to}$ _T	Transitive 2, 7
9	$\mathsf{let\ x = 1 + 2\ in\ x + x} \stackrel{*}{\to}$ _T	Transitive 1, 8

Figure 16: After *Beta* and *Add* (let-improved6)

Figure 16 makes it even more apparent that the essence of a computation (the succession of top-level expressions) can be obscured by the style in which it is being presented. Apart from the fact that each intermediate program term appears three times, the spine of the computation appears in two different places and in both directions. This kind of presentation cannot illuminate very much for us.

Fortunately we can arrange for JAPE to display transitive derivations more conveniently. We do so by declaring (using the form "`TRANSITIVE RULE ...`") the name of each transitive rule in which we are interested, and setting an internal flag "`hidetransitivity`" when we require this form of display. By adding a checkbox bound to it which is controlled from the Edit menu we can arrange for the user to change the state of the flag at will. It turns out that the application of the `Identity` rule can also be usefully suppressed, and an analogous declaration "`REFLEXIVE RULE ...`" and variable "`hidereflexivity`" provide support for this.

```
TRANSITIVE RULE "Transitive"
REFLEXIVE  RULE "Identity"
MENU Edit
SEPARATOR
  CHECKBOX   hidetransitivity "Transitive Rule Display" INITIALLY false
  CHECKBOX   hidereflexivity  "Hide Identity Steps"     INITIALLY false
END
```

The display of the proof from Figure 16 which results from enabling transitive rule display and hiding identity steps is presented in Figure 17, and an alternative derivation (in which the Beta rule is used before the arithmetic is performed) is shown in Figure 18.

1: let x=1+2 in x+x		
2: $\longrightarrow^*$	(λx•x+x)(1+2)	Let
3: $\longrightarrow^*$	(λx•x+x)3	Rand,Add
4: $\longrightarrow^*$	3+3	Beta
5: $\longrightarrow^*$	6	Addition,

Figure 17: Transitive display of the computation in Figure 16 (let-improved4)

1: let x=1+2 in x+x		
2: $\longrightarrow^*$	(λx•x+x)(1+2)	Let
3: $\longrightarrow^*$	(1+2)+(1+2)	Beta
4: $\longrightarrow^*$	3+(1+2)	AddL,Add
5: $\longrightarrow^*$	3+3	AddR,Add
6: $\longrightarrow^*$	6	Addition,

Figure 18: Transitive display of an alternative computation sequence (let-improved5)

Automating single steps

It is easy to automate the choice of rules to apply in a computation step if we observe that a non-stuck expression is either normal, or susceptible to the application of a computation rule, a desugaring rule, or a structural rule. In the case of an application, we need to decide whether to apply the *Rator* or the *Rand* rule, and in the case of a sum, we need to decide whether to apply the *AddL* or the *AddR* rule. We cannot decide arbitrarily, because (for example), there is no point in applying the *AddL* rule in a situation where the left operand of a sum is already a number.

With these ideas in mind, let us consider the design of two tactics which will make the choices for us. There are two principal systematic computation strategies:

1. *Normal order* – in which the leftmost outermost reducible expression in the expression is reduced at each stage.
2. *Applicative order* – in which the operand of an application must be in normal form before the application of a *Beta* rule takes place.

The tactic `NormalStep` first tries all the computation and sugaring rules. If none of these succeed then the subject expression is a sum whose subexpressions are not both numeric, or an application. In the former case an `Add` rule is applied which directs evaluation towards the leftmost non-numeric subexpression. In the latter case the `Rator` rule is applied, on the grounds that the operator of an application must be reduced to normal form before it can be applied.

```
TACTIC NormalStep()
(ALT Beta Add Let
     (WHEN (LETGOAL (_num1 + _E  → _T) AddR)
           (LETGOAL (_E1   + _E2 → _T) AddL)
           (LETGOAL (_F _A → _T) Rator)))
```

The tactic `ApplicativeStep` first tries the addition and sugaring rules. If neither of these succeeds then the subject expresssion is either a sum whose subexpressions are not both numeric, or an application whose operator is not in normal form. In the former case an `Add` rule is applied which directs evaluation towards the leftmost non-numeric subexpression. In the latter case, the operator may be in normal form, in which case a `Beta` move may be performed providing the operand is also in normal form, otherwise the process of normalizing the operand must begin with a `Rand` move. If the operator is not in normal form, then the process of normalizing it must begin with a `Rator` move.

```
TACTIC ApplicativeStep()
(ALT Add Let
     (WHEN (LETGOAL ((λ _x • _E) _A → _T) (WhenNormal _A Beta Rand))
           (LETGOAL (_num _A        → _T) FAIL)
           (LETGOAL (_num1 + _E     → _T) AddR)
           (LETGOAL (_E1   + _E2    → _T) AddL)
           (LETGOAL (_F _A          → _T) Rator)))

TACTIC WhenNormal(expr, tactic, othertactic)
(WHEN  (LETMATCH (λ _x • _S) expr tactic)
       (LETMATCH _num        expr tactic)
       (LETMATCH (~ _num)    expr tactic)
       (LETMATCH _x          expr tactic)
       othertactic)
```

Now we are in a position to automate the computation completely. The tactic `RepeatStep(tactic)` sees whether the expression is in normal form, and if so applies the *Identity* rule; otherwise it applies its `tactic` argument, then recursively works on the resulting expression. The tactic `SingleStep(tactic)` applies its `tactic` argument once, unless it is in normal form.

```
TACTIC RepeatStep(tactic)
(WHEN (LETGOAL (_num        →* _T) Identity)
      (LETGOAL (~ _num      →* _T) Identity)
      (LETGOAL ((λ _x • _E)   →* _T) Identity)
      (SEQ (TransitiveStep tactic) (RepeatStep tactic)))

TACTIC SingleStep(tactic)
(WHEN (LETGOAL (_num        →* _T) Identity)
      (LETGOAL (~ _num      →* _T) Identity)
      (LETGOAL ((λ _x • _E)   →* _T) Identity)
      (TransitiveStep tactic))
```

All that remains is for us to add the tactics we have just defined to the conjecture panel. We do so by defining a tactic variable, `oneSmallStep`, whose value is set by a radio-button to either `NormalStep` or `ApplicativeStep`, and defining buttons which invoke it either once or repeatedly.

```
CONJECTUREPANEL "Structural Semantics"
   RADIOBUTTON oneSmallStep  "Normal Order"      IS NormalStep
   AND                       "Applicative Order" IS ApplicativeStep
   END

   BUTTON Step    IS apply SingleStep oneSmallStep
   BUTTON "Step*" IS apply RepeatStep oneSmallStep
END
```

Some simple, but interesting computations are recorded in Figures 19 through 21, below.

```
1: let g=(λf•f f)in g g
2:  —>*  (λg•g g)(λf•f f)  Let
3:  —>*  (λf•f f)(λf•f f)   Beta
4:  —>*  (λf•f f)(λf•f f)   Beta
   . . .
5:  —>*  _T
```

Figure 19: Some expressions have no normal form (order0)

```
1: let g=(λf•f f)in(let k=(λa•(λb•a))in k 2(g g))
2:  —>*  (λg•(let k=(λa•(λb•a))in k 2(g g)))(λf•f f)  Let
3:  —>*  let k=(λa•(λb•a))in k 2((λf•f f)(λf•f f))     Beta
4:  —>*  (λk•k 2((λf•f f)(λf•f f)))(λa•(λb•a))         Let
5:  —>*  (λa•(λb•a))2((λf•f f)(λf•f f))                Beta
6:  —>*  (λb•2)((λf•f f)(λf•f f))                      Rator,Beta
7:  —>*  (λb•2)((λf•f f)(λf•f f))                      Rand,Beta
   . . .
8:  —>*  _T
```

Figure 20: Applicative-order may not reach a normal form (order1)

```
1: let g=(λf•f f)in(let k=(λa•(λb•a))in k 2(g g))
2:  —>*  (λg•(let k=(λa•(λb•a))in k 2(g g)))(λf•f f)  Let
3:  —>*  let k=(λa•(λb•a))in k 2((λf•f f)(λf•f f))     Beta
4:  —>*  (λk•k 2((λf•f f)(λf•f f)))(λa•(λb•a))         Let
5:  —>*  (λa•(λb•a))2((λf•f f)(λf•f f))                Beta
6:  —>*  (λb•2)((λf•f f)(λf•f f))                      Rator,Beta
7:  —>*  2                                             Beta
```

Figure 21: Normal order reaches a normal form if there is one (order2)

2.6 Normal Order Semantics of LAMBDA

In the previous section we developed two systematic methods of applying inference rules, one led to normal order and the other to applicative order semantics. In this section we present an alternative collection of rules which admit only one systematic method of application – that which corresponds to normal order semantics.

The first *Apply* rule is identical to the *Beta* rule, and succeeds only if the operator is an abstraction. The second *Apply* rule succeeds, leaving computation in the operator as a subgoal.

```
RULES Apply ARE
  INFER (λ x • E') E → E'[E/x]
AND
  FROM  F → F'
  INFER F E → F' E
END
```

The *Arithmetic* tactic first applies the *Arith* rules to a sum. If its operands are already in numeric form, the first rule will succeed, leaving a subgoal of the form `"ADD"(num1, num2, K, (~))`, which is suitable for solution by JAPE's built-in decision procedure `EVALUATE`. If the first operand is in normal form, then the second rule will succeed, leaving computation in the second operand as a subgoal. If neither operand is in numeric form then the last rule will succeed, leaving computation in the first operand as a subgoal.

```
RULES Arith ARE
  FROM   "ADD"(num1, num2, K, (~))
  INFER  num1 + num2 → K
AND
  FROM   E2 → E2'
  INFER  num + E2 → num + E2'
AND
  FROM   E1 → E1'
  INFER  E1 + E2 → E1' + E2
END
TACTIC Arithmetic() IS (SEQ Arith (IF (LAYOUT "Arithmetic" () EVALUATE)))
```

The *NormalStep* tactic applies either the *Let* rule, the *Apply* rule, or the *Arithmetic* tactic. Its repeated transitive application will eventually compute the normal form of an expression, if it has one.

```
TACTIC NormalStep() IS (ALT Let Apply Arithmetic)
```

The behaviour of the normal-order semantics can be explored from the menu defined below.

```
MENU "Normal Order Rules"
   BUTTON Apply    IS apply TransitiveStep Apply
   BUTTON Arith    IS apply TransitiveStep Arithmetic
   BUTTON Let      IS apply TransitiveStep Let
   BUTTON Finished IS apply Finished
   BUTTON Step     IS apply SingleStep NormalStep
END
```

2.7 Natural Semantics of Lambda in Jape

The substitution semantics outlined above is a small-step semantics, in the sense that each of the rules which characterise $\rightarrow$ describes only a single step in the evaluation of the program. We start our description of a big-step (Natural) semantics by noting that the following derived rules can be proven in the small-step semantics. What is more, the derived rules are still valid if we add the requirement that S', T', and U are normal forms.

$$\frac{S \xrightarrow{*} S' \quad S'\ T \xrightarrow{*} U}{S\ T \xrightarrow{*} U}\ \textit{Operator}$$

$$\frac{T \xrightarrow{*} T' \quad S\ T' \xrightarrow{*} U}{S\ T \xrightarrow{*} U}\ \textit{Operand}$$

$$\frac{S[T/x] \xrightarrow{*} U}{(\lambda x \bullet S)(T) \xrightarrow{*} U}\ \textit{Beta}$$

$$\frac{(\lambda x \bullet T)(S) \xrightarrow{*} U}{\textbf{let}\ x = S\ \textbf{in}\ T \xrightarrow{*} U}\ \textit{Let}$$

$$\frac{S \xrightarrow{*} N_1 \quad T \xrightarrow{*} N_2 \quad N_1 + N_2 \rightarrow U}{S + T \xrightarrow{*} U}\ \textit{Add}$$

Figure 22: Derived Rules

Below we encode the stronger derived rules for Jape, interpreting the judgement $S \Rightarrow S'$ as: "the expression S has normal form S' if it doesn't get stuck." The `NormalForm` rules complete the description of this judgement.

```
INFIX 1T ⇒

MENU "Natural Semantic Rules"
 RULE  "Rator⇒"   IS FROM ⊢ S⇒S’  AND ⊢ S’ T  ⇒ U INFER ⊢ S T ⇒ U
 RULE  "Rand⇒"    IS FROM ⊢ T⇒T’  AND ⊢ S  T’ ⇒ U INFER ⊢ S T ⇒ U
 RULE  "Beta⇒"    IS FROM  ⊢ S[T/x]         ⇒ U INFER ⊢ (λ x •S) T  ⇒ U
 RULE  "Let⇒"     IS FROM  ⊢ (λ x •T)(S)   ⇒ U INFER ⊢ let x=S in T ⇒ U
 RULE  "Add⇒"     IS FROM  ⊢ S⇒N1 AND T⇒N2 AND N1+N2→N INFER S+T⇒N
 ENTRY "Addition" IS Add
 RULES NormalForm
 ARE   ⊢ λ x • S ⇒ λ x • S
 AND   ⊢ num     ⇒ num
 AND   ⊢ ~num    ⇒ ~num
 END
END
```

In essence what is going on here is that the transitivity of computation sequences is being built in to the big-step rules. Judgements are being made about complete computations, rather than about individual computational steps.

We can again characterise the two main evaluation strategies by defining appropriate single-step tactics.

```
TACTIC BigNormalStep() IS
(ALT NormalForm
   (WHEN
      (LETGOAL ((λ _x • _S) _T   ⇒  _U) "Beta⇒")
      (LETGOAL (let _x = _S in _T ⇒  _U) "Let⇒")
      (LETGOAL (_S _T             ⇒  _U) "Rator⇒")
      (LETGOAL (_S + _T           →  _U) Add)
      (LETGOAL (_S + _T           ⇒  _U) "Add⇒")))

TACTIC BigApplicativeStep() IS
(ALT NormalForm
  (WHEN
      (LETGOAL ((λ _x • _S) _T    ⇒  _U) (WhenNormal _T "Beta⇒"  "Rand⇒"))
      (LETGOAL (_S _T             ⇒  _U) (WhenNormal _T "Rator⇒" "Rand⇒"))
      (LETGOAL (let _x = _S in _T ⇒  _U) "Let⇒")
      (LETGOAL (_S + _T           →  _U) Add)
      (LETGOAL (_S + _T           ⇒  _U) "Add⇒")))
```

The user interface has a radiobutton to determine which strategy is to be used, and a single step button which takes a big step, using the appropriate strategy.

```
   CONJECTUREPANEL "Natural Semantics"
    RADIOBUTTON oneBigStep "Normal Order"      IS BigNormalStep
    AND                    "Applicative Order" IS BigApplicativeStep
    END

    BUTTON Step IS apply oneBigStep

    THEOREMS BigStepThms
    ARE ⊢ let f = (λ x • x) in f f 1⇒ _T
    AND ⊢ (1+2)+(3+1)⇒_T
    AND ⊢ (1+2)+(~4)⇒_T
    AND ⊢ let x = 1+2 in x+x ⇒_T
    AND ⊢ let c = (λ f • λ g • λ a • f(g a)) in
          let f = (λ a • a) in c f f ⇒ _T
    AND ⊢ let g = (λ f • f f) in g g ⇒ _T
    AND ⊢ let g = (λ f • f f) in
          let k = (λ a • λ b • a) in k 2 (g g) ⇒ _T
    END
END
```

In Figure 23 we show the record of a compuitation in the big-step semantics; the corresponding small-step semantic computation is showm in figures 24 and 25. Not surprisingly, the difference between the two computations viewed as proofs lies only in the details of the application of transitivity rules, and the order in which the computation steps occur in the standardised presentation of the derivation.

1	$\lambda b \bullet 2 \Rightarrow \lambda b \bullet 2$	NormalForm′0
2	$(\lambda a \bullet (\lambda b \bullet a))2 \Rightarrow \lambda b \bullet 2$	Beta $\Rightarrow$ 1
3	$2 \Rightarrow 2$	NormalForm′1
4	$(\lambda b \bullet 2)((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \Rightarrow 2$	Beta $\Rightarrow$ 3
5	$(\lambda a \bullet (\lambda b \bullet a))2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \Rightarrow 2$	Rator $\Rightarrow$ 2, 4
6	$(\lambda k \bullet k\ 2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)))(\lambda a \bullet (\lambda b \bullet a)) \Rightarrow 2$	Beta $\Rightarrow$ 5
7	$\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \Rightarrow 2$	Let $\Rightarrow$ 6
8	$(\lambda g \bullet (\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2(g\ g)))(\lambda f \bullet f\ f) \Rightarrow 2$	Beta $\Rightarrow$ 7
9	$\text{let } g = (\lambda f \bullet f\ f)\text{in}(\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2(g\ g)) \Rightarrow 2$	Let $\Rightarrow$ 8

Figure 23: A computation in big-step semantics (letg-big)

1	$\text{let } g = (\lambda f \bullet f\ f)\text{in}(\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2(g\ g)) \stackrel{*}{\to}$	Let
2	$(\lambda g \bullet (\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2(g\ g)))(\lambda f \bullet f\ f) \stackrel{*}{\to}$	Beta
3	$\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \stackrel{*}{\to}$	Let
4	$(\lambda k \bullet k\ 2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)))(\lambda a \bullet (\lambda b \bullet a)) \stackrel{*}{\to}$	Beta
5	$(\lambda a \bullet (\lambda b \bullet a))2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \stackrel{*}{\to}$	Rator, Beta
6	$(\lambda b \bullet 2)((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \stackrel{*}{\to}$	Beta
7	2	Identity

Figure 24: A computation in small-step semantics (letg-small)

8	$(\lambda b \bullet 2)((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \stackrel{*}{\to} 2$	Transitive 6, 7
9	$(\lambda a \bullet (\lambda b \bullet a))2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \stackrel{*}{\to} 2$	Transitive 5, 8
10	$(\lambda k \bullet k\ 2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)))(\lambda a \bullet (\lambda b \bullet a)) \stackrel{*}{\to} 2$	Transitive 4, 9
11	$\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2((\lambda f \bullet f\ f)(\lambda f \bullet f\ f)) \stackrel{*}{\to} 2$	Transitive 3, 10
12	$(\lambda g \bullet (\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2(g\ g)))(\lambda f \bullet f\ f) \stackrel{*}{\to} 2$	Transitive 2, 11
13	$\text{let } g = (\lambda f \bullet f\ f)\text{in}(\text{let } k = (\lambda a \bullet (\lambda b \bullet a))\text{in } k\ 2(g\ g)) \stackrel{*}{\to} 2$	Transitive 1, 12

Figure 25: Details of transitivity in the small-step semantics (letg-small-1)

2.8 Deterministic Semantics

Hitherto we have striven to write the inference rules in our semantics in such a way that both normal and applicative orders of evaluation are permissible. In this section we present a natural semantics for LAMBDA which forces normal order of evaluation. The inference rule for applications reflects the fact that the operator of an application must be reduced to an abstraction before it can be applied. The absence of structural rules for applications reflects the fact that operand substitution must happen once the operator has been reduced to an abstraction. The `Let` rule is similar: substitution must take place as soon as possible. The rules for addition, and the normal form rules are identical to those in the previously-given semantics.

```
MENU "Deterministic Rules"
 RULE   "App∆"   IS FROM ⊢ F ⇒ (λ x  • S) AND ⊢ S[A/x] ⇒ U INFER ⊢ F A ⇒ U
 RULE   "Let∆"   IS FROM ⊢ S[A/x] ⇒ U INFER ⊢ let x=A in S ⇒ U
 TACTIC "Add∆"   IS "Add⇒"
 ENTRY  Addition  IS Add
 ENTRY  NormalForm
END
```

The tactic which takes a single step in the evaluation of an expression can also be simplified, because the form of the expression *uniquely* defines which rule should be applied to it. All that is necessary is to try the rules in some order – indeed *any* order will do.

```
TACTIC BigDeterministicStep() IS
(ALT NormalForm "App∆" "Let∆" Add "Add∆")

CONJECTUREPANEL "Natural Semantics"
  BUTTON "∆Step" IS apply BigDeterministicStep
END
```

Having placed a button on the big-step computations panel which invokes this tactic, we can demonstrate some of the differences between this formulation and the last.

The main observable difference in the derivation, apart from the absence of `Rator` and `Rand` rules, is that during the evaluation of the operator of an application, *proof variables* appear.[4]

The outcome of the evaluation is, of course, the same as before.

2.9 Evaluation Semantics

As our final foray into the semantics of LAMBDA we show how to dispense with substitutions by using an environment to represent the current value of each variable bound in an evaluation context. An environment is a mapping from identifiers to values which are either numbers or function closures (written $[\![\lambda x \bullet E, Env]\!]$).

[4]Proof variables are variables whose names begin with an underscore, and which stand for formulae which have not yet been uniquely identified in the derivation.

1	λa • (λb • a) ⇒ λ_x1 • _S1	
	...	
2	_S1[2/_x1] ⇒ λ_x • _S	
3	(λa • (λb • a))2 ⇒ λ_x • _S	AppΔ 1, 2
	...	
4	_S[(λf • f f)(λf • f f)/_x] ⇒ _T	
5	(λa • (λb • a))2((λf • f f)(λf • f f)) ⇒ _T	AppΔ 3, 4
6	let k = (λa • (λb • a))in k 2((λf • f f)(λf • f f)) ⇒ _T	LetΔ 5
7	let g = (λf • f f)in(let k = (λa • (λb • a))in k 2(g g)) ⇒ _T	LetΔ 6

Figure 26: An incomplete computation in the deterministic semantics (letg-det-1)

1	λa • (λb • a) ⇒ λa • (λb • a)	NormalForm′0
2	λb • 2 ⇒ λb • 2	NormalForm′0
3	(λa • (λb • a))2 ⇒ λb • 2	AppΔ 1, 2
4	2 ⇒ 2	NormalForm′1
5	(λa • (λb • a))2((λf • f f)(λf • f f)) ⇒ 2	AppΔ 3, 4
6	let k = (λa • (λb • a))in k 2((λf • f f)(λf • f f)) ⇒ 2	LetΔ 5
7	let g = (λf • f f)in(let k = (λa • (λb • a))in k 2(g g)) ⇒ 2	LetΔ 6

Figure 27: The completed computation in the deterministic semantics (letg-det-2)

Judgements in this system take the form

$$Env \vdash E \Rightarrow V$$

where E is a LAMBDA expression, V is a value, and Env is a finite mapping from identifiers to values, which is represented as described in Appendix A. They are interpreted as "in environment Env the expression E has value V." The auxiliary judgement form

$$\vdash Env\ x = V$$

is interpreted as "environment Env binds variable x to value V, and the judgement form

$$\vdash N1 + N2 \rightarrow V$$

is interpreted (as before) as "V is the sum of the numbers N_1 and N_2."

Because the theory of mappings is described in the file `mapping.j` (which is derived from the manuscript of Appendix A), we need to incorporate it in the present theory. Moreover, because the material to the left of the turnstile in our main form of judgement is not a hypothesis, but a term representing an environment, we need to adjust the way in which JAPE labels "assumption" lines. The only remaining detail is the declaration of the notation for closures. We are going to invoke the rules from a menu called`Value Rules`.

```
USE "mapping.j"
INITIALISE outerassumptionword "Environment"
```

```
INITIALISE innerassumptionword "Environment"
OUTFIX ⟦ ⟧

MENU "Value Rules"
```

Numbers evaluate to themselves. An abstraction evaluates to a closure which embeds both the abstraction and the environment in which it was abstracted.

```
RULES Constant
ARE   Env ⊢ num    ⇒ num
AND   Env ⊢ ~num   ⇒ ~num
AND   Env ⊢ λ x•S ⇒ ⟦ λ x • S, Env ⟧
END
```

As in the big-step semantics, the value of a sum in an environment is the sum of the values of its summands – which had better be numbers.

```
RULE  Sum
FROM  Env ⊢ S ⇒ N1
AND   Env ⊢ T ⇒ N2
AND       ⊢ N1+N2 → V
INFER Env ⊢ S+T⇒V
```

The value of a variable is the value to which the environment maps it (which might be ⊥).

```
RULE  Variable
FROM      ⊢ Env x = V
INFER Env ⊢ x ⇒ V
```

The operator of an application must evaluate to a closure; if it does so, then the value of the application is the result of evaluating the body of the "enclosed" abstraction in an environment defined by extending the enclosed environment by associating the bound variable with the value of the operand. The value of a block is obtained by evaluating its body in an extension of the current environment.

```
RULE  Application
FROM             Env ⊢ F ⇒ ⟦ λ x • E, Env' ⟧
AND              Env ⊢ A ⇒ V
AND  Env' ⊕ (x==V) ⊢ E ⇒ V'
INFER            Env ⊢ F A ⇒ V'

RULE  Block
FROM             Env ⊢ A ⇒ V
AND   Env ⊕ (x==V) ⊢ E ⇒ V'
INFER            Env ⊢ let x=A in E ⇒ V'
```

The value of an expression at the top level is its value in an empty environment.

```
RULE   TopLevel
FROM   ⟧ ⊢ E ⇒ V
INFER    ⊢ E ⇒ V

SEPARATOR
```

As in the deterministic semantics presented earlier, the form of the expression at each stage *uniquely* determines which rule should be applied to it. Mechanisation of evaluation is, therefore, simply a matter of trying the rules one by one. In the case of the `Variable` rule, which generates a judgement in the theory of mappings, we suppress (using a `LAYOUT` tactic) the details of the solution of this judgement.

```
   TACTIC EvalStep IS
   (ALT Constant
        (LAYOUT "Variable" () Variable Lookup)
        Sum
        Application
        Block
        Add
        TopLevel)

   ENTRY Evaluate IS (DO (EvalStep))
 END
```

Figure 28 shows a derivation in the evaluation semantics. Notice that each subderivation is enclosed in a box, and that the environment in which the subderivation takes place appears as the top line of the box. This requires no intervention on the part of the theory designer – it is simply an artefact of the box-style of proof display.

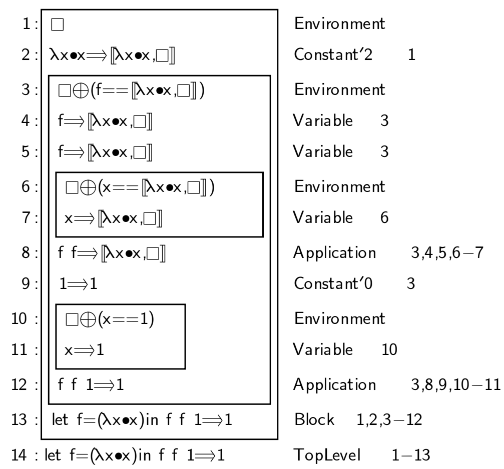

Figure 28: A Derivation in the Evaluation Semantics (letf-val)

3 A Small Imperative Language

3.1 Syntax

IPL is a small expression-based imperative programming language whose forms of expression are outlined below in Figure 29. We use names based on uppercase E to denote expressions, names based on lowercase x, y, z to denote variables, and names based on k to denote (natural number) constants.

$x := E$	Assignment
$E_1; E_2$	Sequential composition
$E_1 ? E_2 : E_3$	Conditional expression: E_2 if $E_1 \neq 0$ else E_3
$E_1 \otimes E_2$	Repeat E_2 until $E_1 = 0$
$\surd$	Skip (do nothing)
$E_1 + E_2$	Sum
$E_1 - E_2$	Difference
$E_1 < E_2$	Less than
k	Nonnegative number constants
$\tilde{}k$	Negative number constants

Figure 29: IPL Syntax

Expressions are integer-valued, and the guards of conditional expressions and loops are expressions which are interpreted as false when zero, and as true otherwise.[5] The normal forms of expression are the numeric constants and $\surd$ – the latter meaning "successful termination."

A simple example program is the following fragment, which multiplies y by 2^n if $n \geq 0$.

$$n \otimes (y := y + y; n := n - 1)$$

3.2 Structural Semantics of IPL in JAPE

Many of the details required to set up the small-step semantics of IPL in JAPE have already been discussed in Section 2, and we will not repeat them here.

First we introduce the syntax of the language

```
CLASS    CONSTANT k
CONSTANT √ .
INFIX    40R ;
```

[5] We do not think this is a particularly good language feature, but it reduces the number of semantic rules we have to explain here.

```
INFIX    50T :=
INFIX    60R ? : ⊗
INFIX    100T =
INFIX    150T <
INFIX    200T + -
PREFIX   1000 ~
USE      "mapping.j"
```

The reduction arrow and "syntactic brackets" are introduced next.

```
INFIX   1T →
LEFTFIX 2  ⟦  ⟧
```

The most important judgement in the semantics takes the form

$$[\![E]\!]S\rightarrow[\![E']\!]S'$$

(where E, E' are expressions and S, S' are mappings from names to values). Such judgements are interpreted as: "A single step in the evaluation transforms expression E in state S to expression E' in state S'." An expression+state pair $[\![E]\!]S$ is called a *configuration*.

As usual, we interpret $[\![E]\!]S \stackrel{*}{\rightarrow} [\![E']\!]S'$ as "Zero or more execution steps transform expression E in state S to the normal form expression E' in state S'. The normal form configurations are those whose expressions are either number constants or the "do nothing" command $\surd$. These configurations require no further computation steps to complete, and will therefore not be the subject of any semantic rules.[6]

All other configurations make progress towards a normal form (if there is one) by means of a sequence (possibly empty) of single computation steps. As in the earlier small-step semantics, we note that the relation between the configurations at the start and end of a (perhaps partial) computation sequence is the transitive closure $\rightarrow*$ of the single computation step relation $\rightarrow$, and accordingly we define

```
INFIX 1T →*
RULE Step        FROM  ⊢  ⟦ E ⟧ S→ ⟦ E' ⟧ S' INFER ⊢  ⟦ E ⟧ S→* ⟦ E' ⟧ S'
RULE Transitive  FROM  ⊢  ⟦ E ⟧ S→* ⟦ E' ⟧ S' AND ⊢  ⟦ E' ⟧ S'→* ⟦ E'' ⟧ S'' INFER ⊢ ⟦ E ⟧ S→* ⟦ E'' ⟧ S''
RULE Identity    INFER ⊢  ⟦ E ⟧ S→* ⟦ E ⟧ S
```

The following declarations let us control the form of display which JAPE uses to show computation sequences.

```
TRANSITIVE RULE Transitive
REFLEXIVE  RULE Identity
MENU Edit
SEPARATOR
  CHECKBOX   hidetransitivity "Transitive Rule Display" INITIALLY true
  CHECKBOX   hidereflexivity  "Hide Identity Steps"     INITIALLY true
END
```

[6] When we come to automate the application of rules it will be necessary to know when no further rules are applicable, and this means that we will have to *recognise* the normal form configurations.

Now we can introduce the rules which characterise single computation steps. The rule for variables is straightforward.

```
MENU "Small-Step Rules"
  RULE  Var
  FROM  S x = V
  INFER ⟦ x ⟧  S → ⟦ V ⟧  S
```

The rule for simplifying the right-hand-side of an assignment is simple – note that side-effects are permitted.

```
  RULE ":=0"
  FROM     ⟦ E ⟧      S → ⟦ E' ⟧     S'
  INFER    ⟦ x:=E ⟧   S → ⟦ x:=E' ⟧  S'
```

The most obvious rule for completing a fully-simplified assignment by updating the store is

$$[\![x := k]\!] \ \ S \rightarrow [\![\surd]\!] \ \ (S \oplus (x == k))$$

The JAPE encoding of this is a bit more complicated:

```
  RULES ":=1"
  FROM  (S⊕(x==k)) = S'
  INFER ⟦ x:=k ⟧     S → ⟦ √ ⟧ S'
  AND
  FROM  (S⊕(x==(~k))) = S'
  INFER ⟦ x:=(~k) ⟧  S → ⟦ √ ⟧ S'
  END
```

The first complication is the presence of an antecedent. This is intended to make it possible for the expression $(S \oplus (x == ...))$, which denotes the updated store, to be simplified just after the rule is applied. Although the antecedent is not formally necessary, simplification turns out to be essential when one wants to animate any but the simplest programs containing assignment. Without simplification the size of the expression denoting the store is linear in the number of assignments which have been made, rather than in number of variables assigned to. Details of the rules which define simplification and the tactic which accomplishes it appear in Section A.4.

The second complication is the fact that there are *two* rules. This is because we have to take account of negative numbers, which have a distinct syntactic representation $(\tilde{k})$. If JAPE's formula classification language were flexible enough to let us define the class of constants syntactically then the second rule wouldn't be necessary.[7]

Sequential composition, and iteration are straightforward; conditional expressions are also complicated a little by the need to account for negative numbers.

[7] We're working on it.

```
RULE ";0"
FROM     ⟦ E ⟧    S → ⟦ E' ⟧    S'
INFER    ⟦ E;F ⟧  S → ⟦ E';F ⟧  S'

RULE ";1"
INFER    ⟦ √;F ⟧  S → ⟦ F ⟧  S

RULE "⊗"
INFER ⟦ E1⊗E2 ⟧  S → ⟦ E1?(E2; E1⊗E2):√ ⟧  S

RULE "?0"
FROM  ⟦ E1 ⟧        S → ⟦ E1' ⟧  S'
INFER ⟦ E1?E2:E3 ⟧  S → ⟦ E1'?E2:E3 ⟧  S'

RULES "?1"
INFER ⟦ 0?E2:E3 ⟧   S → ⟦ E3 ⟧  S
AND
INFER ⟦ k?E2:E3 ⟧   S → ⟦ E2 ⟧  S
AND
INFER ⟦ ~k?E2:E3 ⟧  S → ⟦ E2 ⟧  S
END
```

The rules for scheduling the simplification of arithmetic and relational expressions are just as straightforward.

```
RULES  "+"
FROM  ⟦ F ⟧    S → ⟦ F' ⟧    S'
INFER ⟦ k+F ⟧  S → ⟦ k+F' ⟧  S'
AND
FROM  ⟦ F ⟧     S → ⟦ F' ⟧     S'
INFER ⟦ ~k+F ⟧  S → ⟦ ~k+F' ⟧  S'
AND
FROM  ⟦ E ⟧    S → ⟦ E' ⟧    S'
INFER ⟦ E+F ⟧  S → ⟦ E'+F ⟧  S'
END

RULES  "-"
FROM  ⟦ F ⟧    S → ⟦ F' ⟧    S'
INFER ⟦ k-F ⟧  S → ⟦ k-F' ⟧  S'
AND
FROM  ⟦ F ⟧     S → ⟦ F' ⟧     S'
INFER ⟦ ~k-F ⟧  S → ⟦ ~k-F' ⟧  S'
AND
FROM  ⟦ E ⟧    S → ⟦ E' ⟧    S'
INFER ⟦ E-F ⟧  S → ⟦ E'-F ⟧  S'
END

RULES  "<"
FROM  ⟦ F ⟧    S → ⟦ F' ⟧    S'
INFER ⟦ k<F ⟧  S → ⟦ k<F' ⟧  S'
AND
FROM  ⟦ F ⟧     S → ⟦ F' ⟧     S'
INFER ⟦ ~k<F ⟧  S → ⟦ ~k<F' ⟧  S'
AND
FROM  ⟦ E ⟧    S → ⟦ E' ⟧    S'
INFER ⟦ E<F ⟧  S → ⟦ E'<F ⟧  S'
```

```
END
```

In order to do arithmetic using JAPE's built-in facilities we need two special rules. These transform judgements about expressions into a built-in judgement – such as `"ADD"(k1, k2, K, (~))` – which is interpreted as "the sum of the integers $k1$ and $k2$ is K (if the symbol `~` is interpreted as the unary negation symbol)."

```
RULE  add
FROM  "ADD"(k1, k2, K, (~))
INFER ⟦k1 + k2⟧ S → ⟦K⟧ S

RULE  sub
FROM  "ADD"(k1, ~k2, K, (~))
INFER ⟦k1 - k2⟧ S → ⟦K⟧ S
```

Computing relational answers is more difficult since, by an oversight which will eventually be remedied, there are no built-in judgements which correspond to them. But we can improvise, by doing a subtraction and seeing if the result is negative. The tactic `Comparison` succeeds, with the correct answer, by trying the cases of the `compare` rule in sequence.[8]

```
RULES compare
FROM  "ADD"(k1, ~k2, 0, (~))
INFER ⟦k1 < k2⟧ S → ⟦0⟧ S
AND
FROM  "ADD"(k1, ~k2, ~K, (~))
INFER ⟦k1 < k2⟧ S → ⟦1⟧ S
AND
FROM  "ADD"(k1, ~k2, K, (~))
INFER ⟦k1 < k2⟧ S → ⟦0⟧ S
END

TACTIC Comparison
IS (ALT (LAYOUT "Compare" () compare'0 EVALUATE)
        (LAYOUT "Compare" () compare'1 EVALUATE)
        (LAYOUT "Compare" () compare'2 EVALUATE))
```

The rules (and menu) are now complete.

```
END
```

Automating single steps

Next we turn our attention to automating the choice of evaluation steps.

The tactic `SmallStep` simply tries all the rules in order – trying the less general rule(s) for a given subject before trying the more general rule. Invocations of the variable and the constant assignment rules are followed immediately by invocations of tactics which solve their antecedents. Both such tactics are defined in section A. The former looks up the value of the variable in (the expression denoting) the store; the latter systematically simplifies the expression denoting the store.

[8] The order is important, since the inbuilt judgement `ADD(k,  k,  0, ( ))` will succeed.

```
TACTIC SmallStep IS
(ALT (LAYOUT "Var"  () Var Lookup)
     (LAYOUT ":=1" () ":=1" Update)
     ":=0"
     ";1"
     ";0"
     "?1"
     "?0"
     "⊗"
     (ALT (LAYOUT "Add" () add EVALUATE) "+")
     (ALT (LAYOUT "Sub" () sub EVALUATE) "-")
     (ALT Comparison "<")
     )
```

The last three branches of the `SmallStep` tactic need a little more explanation. Since they are similar, we will just explain the first.

```
(ALT (LAYOUT "Add" () add EVALUATE) "+")
```

This is a complete tactic for taking a single step in the evaluation of a sum when the subject configuration is in the form:

$$[\![N1 + N2]\!]S \rightarrow [\![K]\!]S$$

and K is unknown. It works by applying `add`, then attempting to solve the

```
"ADD"(N1, N2, K, (~))
```

judgement thereby generated using the built-in `EVALUATE` tactic. If `EVALUATE` succeeds – and it can only do so if $N1$ and $N2$ are (possibly negated) numeric constants, then their sum is bound to K, and this solves the subject configuration. If `EVALUATE` fails it is because one of $N1, N2$ is not a constant and exactly one of the three + rules will match the configuration – the `"+"` alternative applies just the right one, thereby choosing one or the other subexpression to evaluate further.

The tactic `OneStep()`, which is defined below, carries out a single step of a computation.

```
TACTIC OneStep()
(ALT
     (WHEN (LETGOAL ( ⟦ _E  ⟧ _S →   ⟦ _E' ⟧ _S') SmallStep)
           (LETGOAL ( ⟦  _k ⟧ _S →* _T) Identity)
           (LETGOAL ( ⟦ ~_k ⟧ _S →* _T) Identity)
           (LETGOAL ( ⟦ √   ⟧ _S →* _T) Identity)
           (LETGOAL ( ⟦ _E  ⟧ _S →*  ⟦ _E' ⟧ _S') Transitive (LAYOUT "%s" () Step SmallStep))))
```

If the current goal is of the form $[\![E]\!]S \rightarrow [\![E']\!]S'$, then the appropriate single step rule is applied by the `SmallStep` tactic. If the current goal is of the form $[\![E]\!]S \xrightarrow{*} [\![E']\!]S'$ and its subject is in normal form, the `Identity` rule is applied – signifying the end of the computation; otherwise we set up the first step of a sequence by applying `Transitive`. This leaves us with a derivation tree of the form

$$\frac{[\![E]\!]S \xrightarrow{*} [\![E'']\!]S'' \quad [\![E'']\!]S'' \xrightarrow{*} [\![E']\!]S'}{[\![E]\!]S \xrightarrow{*} [\![E']\!]S'} \; \textit{Transitive}$$

We now need to execute a single small step, and this is done by the tactic

```
(LAYOUT "%s" () Step SmallStep)
```

which first invokes the `Step` rule, transforming the derivation tree into

$$\dfrac{\dfrac{[\![E]\!]S \rightarrow [\![E'']\!]S''}{[\![E]\!]S \xrightarrow{*} [\![E'']\!]S''}\;\textit{Step} \qquad [\![E'']\!]S'' \xrightarrow{*} [\![E']\!]S'}{[\![E]\!]S \xrightarrow{*} [\![E']\!]S'}\;\textit{Transitive}$$

The **`SmallStep`** tactic now finds the appropriate single-step rule to apply, and the `LAYOUT...` block suppresses (from the proof display) the resulting sub-proof. Figures 30 through 33 show the successive steps in the application of `OneStep` to a simple goal. At each stage the current goal is shown with a box around it – in JAPE's display the box would be coloured red.

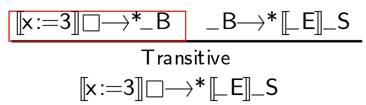

Figure 30: After *Transitive* (ipl1)

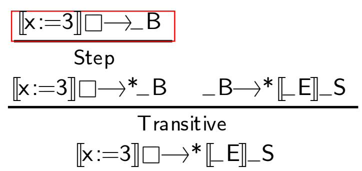

Figure 31: After *Step* (ipl2)

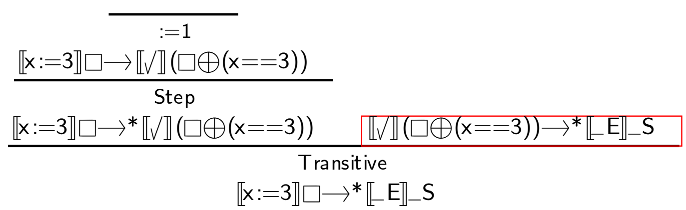

Figure 32: After := .1 (ipl3)

$$\dfrac{\dfrac{}{[\![\mathsf{x{:=}3}]\!]\Box \longrightarrow^{*} [\![\surd]\!](\Box \oplus (\mathsf{x{==}3}))}\ {:=}1'0,\mathsf{Update0} \qquad [\![\surd]\!](\Box \oplus (\mathsf{x{==}3})) \longrightarrow^{*} [\![\_\mathsf{E}]\!]\_\mathsf{S}}{[\![\mathsf{x{:=}3}]\!]\Box \longrightarrow^{*} [\![\_\mathsf{E}]\!]\_\mathsf{S}}\ \mathsf{Transitive}$$

Figure 33: After `(LAYOUT ...)` suppresses detail (ipl4)

The tactic Steps() carries out a complete computation, repeatedly applying rules until the subject is a normal configuration.[9]

```
TACTIC Steps()
(ALT
     (WHEN (LETGOAL (⟦  _k ⟧ _S →* _T) Identity)
           (LETGOAL (⟦ ˜_k ⟧ _S →* _T) Identity)
           (LETGOAL (⟦ √   ⟧ _S →* _T) Identity)
           (SEQ OneStep Steps)))
```

Finally we construct a conjecture panel with a few small programs to run.

```
CONJECTUREPANEL "Small Step Execution"
  BUTTON "Step" IS apply OneStep
  BUTTON "Steps" IS apply Steps
  THEOREMS SmallStepTheorems
  ARE ⟦ x:=3 ⟧  ⟦                                   →* ⟦ _E ⟧  _S
  AND ⟦ y:=3; x:=y ⟧  ⟦                             →* ⟦ _E ⟧  _S
  AND ⟦ y:=x+3; x:=y ⟧  ( ⟦ ⊕(x==4))                 →* ⟦ _E ⟧  _S
  AND ⟦ y:=3; x:=y; z:=x+y ⟧  ⟦                     →* ⟦ _E ⟧  _S
  AND ⟦ t:=x; x:=y; y:=t ⟧ ( ⟦ ⊕(x==k1)⊕(y==k2)) →* ⟦ _E ⟧  _S
  AND ⟦ x:=2; y:=4; x⊗(x:=x-1; y:=y+y) ⟧  ⟦        →* ⟦ _E ⟧  _S
  END
END
```

In Figure 34 we show the complete top-level derivation of a normal configuration for

$$[\![y := x + 3; x := y]\!] ([\![\oplus (x == 4)) [\!]$$

```
1: ⟦y:=x+3;x:=y⟧(□⊕(x==4))
2:  →*  ⟦y:=4+3;x:=y⟧□⊕(x==4)                    ;0,:=0
3:  →*  ⟦y:=7;x:=y⟧□⊕(x==4)                      ;0,:=0
4:  →*  ⟦√;x:=y⟧((□⊕(y==7))⊕(x==4))              ;0,:=1
5:  →*  ⟦x:=y⟧(□⊕(y==7))⊕(x==4)                  ;1
6:  →*  ⟦x:=7⟧(□⊕(y==7))⊕(x==4)                  :=0,Var
7:  →*  ⟦√⟧(□⊕(y==7)⊕(x==7))                     :=1'0,Update1
                     Providing
                     y NOTIN x
```

Figure 34: A complete computation (ipl5)

This presentation could be a little misleading, for it shows only the top-level transitive steps in the computation. If the justifications of these steps are presented in full, as in Figure 35, then a lot more detail appears.

In fact while the justification of a top-level step is being constructed a step at a time, the appropriate subcomputations will be present in the display. For example, in Figure 36 we show the situation after 3 uses of Step in the above computation.

[9] Or JAPE runs out of patience – the number of steps to which JAPE limits itself when performing a computation automatically may be set by the user using the Set Proof Step Count button on the File menu.

1 :	⟦x⟧□⊕(x==4)⟶⟦4⟧□⊕(x==4)	Var
2 :	⟦x+3⟧□⊕(x==4)⟶⟦4+3⟧□⊕(x==4)	+′2 1
3 :	⟦y:=x+3⟧□⊕(x==4)⟶⟦y:=4+3⟧□⊕(x==4)	:=0 2
4 :	⟦y:=x+3;x:=y⟧(□⊕(x==4))⟶⟦y:=4+3;x:=y⟧□⊕(x==4)	;0 3
5 :	⟦4+3⟧□⊕(x==4)⟶⟦7⟧□⊕(x==4)	Add
6 :	⟦y:=4+3⟧□⊕(x==4)⟶⟦y:=7⟧□⊕(x==4)	:=0 5
7 :	⟦y:=4+3;x:=y⟧□⊕(x==4)⟶⟦y:=7;x:=y⟧□⊕(x==4)	;0 6
8 :	⟦y:=7⟧□⊕(x==4)⟶⟦√⟧((□⊕(y==7))⊕(x==4))	:=1
9 :	⟦y:=7;x:=y⟧□⊕(x==4)⟶⟦√;x:=y⟧((□⊕(y==7))⊕(x==4))	;0 8
10 :	⟦√;x:=y⟧((□⊕(y==7))⊕(x==4))⟶⟦x:=y⟧(□⊕(y==7))⊕(x==4)	;1
11 :	⟦y⟧(□⊕(y==7))⊕(x==4)⟶⟦7⟧(□⊕(y==7))⊕(x==4)	Var
12 :	⟦x:=y⟧(□⊕(y==7))⊕(x==4)⟶⟦x:=7⟧(□⊕(y==7))⊕(x==4)	:=0 11
13 :	⟦x:=7⟧(□⊕(y==7))⊕(x==4)⟶⟦√⟧(□⊕(y==7)⊕(x==7))	:=1
14 :	⟦y:=x+3;x:=y⟧(□⊕(x==4))	
15 :	⟶* ⟦y:=4+3;x:=y⟧□⊕(x==4)	[Step] 4
16 :	⟶* ⟦y:=7;x:=y⟧□⊕(x==4)	[Step] 7
17 :	⟶* ⟦√;x:=y⟧((□⊕(y==7))⊕(x==4))	[Step] 9
18 :	⟶* ⟦x:=y⟧(□⊕(y==7))⊕(x==4)	[Step] 10
19 :	⟶* ⟦x:=7⟧(□⊕(y==7))⊕(x==4)	[Step] 12
20 :	⟶* ⟦√⟧(□⊕(y==7)⊕(x==7))	[Step] 13

Providing
y NOTIN x

Figure 35: A complete computation, with justifications (ipl6)

	. . .	
1 :	⟦x⟧□⊕(x==4)⟶⟦_E′3⟧_S′1	
2 :	⟦x+3⟧□⊕(x==4)⟶⟦_E′3+3⟧_S′1	+′2 1
3 :	⟦y:=x+3⟧□⊕(x==4)⟶⟦y:=_E′3+3⟧_S′1	:=0 2
4 :	⟦y:=x+3;x:=y⟧(□⊕(x==4))⟶⟦y:=_E′3+3;x:=y⟧_S′1	;0 3
5 :	⟦y:=x+3;x:=y⟧(□⊕(x==4))	
6 :	⟶* ⟦y:=_E′3+3;x:=y⟧_S′1	4
	. . .	
7 :	⟶* ⟦_E⟧_S	

Figure 36: Three `Step`s into the computation (ipl7)

A Environments and Stores

In this section we outline the JAPE theory we use to describe the finite mappings used to model environments and stores. The theory itself is quite straightforward, and the tactics used to evaluate expressions in the theory may be of some interest – as examples of *tactically controlled logic programming.*

A.1 Syntax

In addition to the conventions established earlier, we declare here that

```
CLASS FORMULA  map
```

A finite mapping is either empty, denoted $[\!]$, a singleton mapping from a variable x to a value V, denoted $x == V$, or the join of mappings *map* and *map'*, denoted $map \oplus map'$. When applied to a variable outside its domain, a mapping yields the value *undefined*, denoted $\bot$.

```
CONSTANT    [] ⊥
INFIX   5L ==
INFIX   3L ⊕
```

A.2 Rules

The empty mapping maps all variables to $\bot$, the singleton mapping $x == V$ maps the variable x to V, and maps all variables distinct from x to $\bot$.

```
RULES Map WHERE  y NOTIN x
ARE   ⊢   [] x = ⊥
AND   ⊢  (x==V)  x = V
AND   ⊢  (x==V)  y = ⊥
END
```

The combination $map \oplus map'$ maps x to the value which *map'* does, providing that value is not $\bot$, otherwise it maps x to the value which *map* does. We define this form of mapping combination by means of a subsidiary selection function, which is also denoted $\oplus$. The rules `Select` define $\oplus$ as selecting its right-hand argument, providing that it is not $\bot$.

```
RULES Select  WHERE W NOTIN ⊥
ARE INFER⊢  V ⊕ ⊥ = V
AND INFER⊢  V ⊕ W = W
END
```

The rule $\oplus$ expresses preference for the result of searching the right component of a composite environment, providing it is non-$\bot$.

```
RULE "⊕"
FROM  ⊢  map' x = V
AND   ⊢  map  x = W
AND   ⊢  W ⊕ V  = X
INFER ⊢ (map⊕map') x = X
```

A.3 Decision Procedure

The tactic `Lookup`, defined below, is a decision procedure for goals of the form *map* $x = V$. If *map* is simple, then the problem will be solved immediately by one of the `Map` rules. If, on the other hand, *map* is composite, then it will be solved by first applying the $\oplus$ rule, then recursively applying `Lookup` (twice) to discover the values of x in each of the components, and finally applying `Select` to choose between these values.

```
TACTIC Lookup IS
(ALT Map (SEQ "⊕" Lookup Lookup Select))
```

Figure 37 and Figure 38 show the proof trees resulting from environment searches. Although

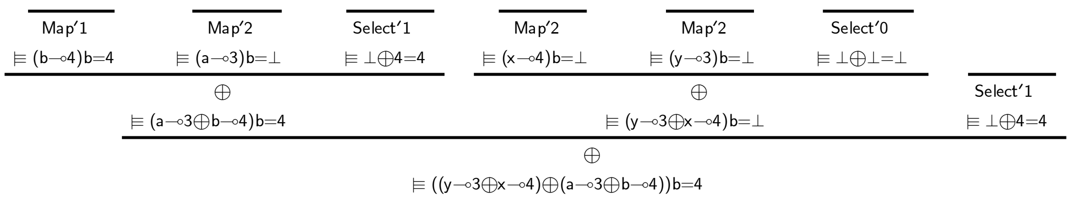

Figure 37: An environment lookup (mapping0)

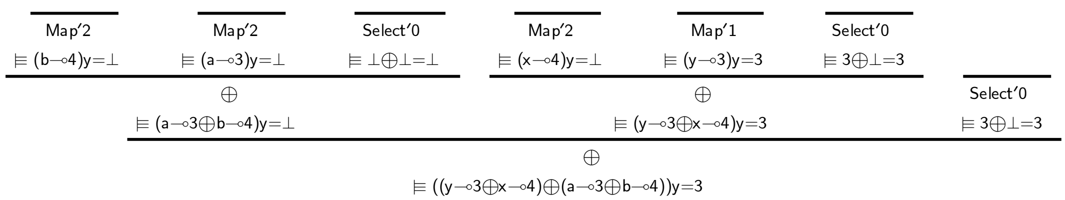

Figure 38: An environment lookup (mapping1)

it is beyond the scope of this note to discuss such matters in detail, it is worth noting here that the number of inferences made by this procedure is *in all cases* the same linear function of the number of $\oplus$-nodes in the subject mapping. Performance can be improved in many cases, but only at the cost of making the mapping rules and decision procedure harder to understand!

A.4 Updating Mappings

When a theory uses a mapping to represent a store, it is convenient to represent the store in minimal form – in other words, with at most one occurence of a form $x == V$ for each x in the store. The updating of a variable x in a store is usually modelled by extending the mapping which represents the store: in other words by transforming *map* to *map* $\oplus$ $(x == V')$. The following rules inductively characterise the equivalences between extended stores, and serve as a basis for the tactic `Update`, which transforms an extended store into its normal form.

```
RULE  Update0  ⊢ ( [] ⊕(x==V)) = ( [] ⊕(x==V))
RULE  Update1  ⊢ (map⊕(x==W)⊕(x==V)) = (map⊕(x==V))
RULE  Update2
FROM  ⊢ (map⊕(x==V)) = map'
INFER ⊢ (map⊕(y==W)⊕(x==V)) = (map'⊕(y==W))

TACTIC Update IS (ALT  Update0 Update1 (SEQ Update2 (PROVE Update)))
```

Contents

References

[Kah87] Gilles Kahn. Natural Semantics. In *Lecture Notes in Computer Science*, volume 247, pages 22–39. Springer, 1987.

[Lan64] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN–19, Aarhus University, September 1981.

opsem.tex 1.25 14:55 22nd September 2000
lambda.tex 1.26 23:38 29th October 1998
ipl.tex 1.9 14:55 22nd September 2000
mapping.tex 1.15 18:42 19th June 1998