J. Camilleri and T. Melham, 'Reasoning with Inductively Defined Relations in the HOL Theorem Prover', Technical Report No. 265, University of Cambridge Computer Laboratory (August 1992).

Reasoning with Inductively Defined Relations in the HOL Theorem Prover

Juanito Camilleri

Department of Computer Studies
University of Malta
University Heights
Msida, Malta G.C.

Tom Melham

University of Cambridge
Computer Laboratory
Pembroke Street, Cambridge
England, CB2 3QG.

Abstract: *Inductively defined relations are among the basic mathematical tools of computer science. Examples include evaluation and computation relations in structural operational semantics, labelled transition relations in process algebra semantics, inductively-defined typing judgements, and proof systems in general. This paper describes a set of HOL theorem-proving tools for reasoning about such inductively defined relations. We also describe a suite of worked examples using these tools.*

First printed: August 1992

Parts of this report have previously appeared as: T. Melham, 'A Package for Inductive Relation Definitions in HOL', in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 350–357.

Contents

Introduction

The HOL user community has a tradition of taking a purely *definitional* (or logicist) approach to using higher order logic. That is, the syntax of the logic is extended with new notation not simply by postulating axioms to give meaning to it, but rather by defining it in terms of existing expressions of the logic that already have the required semantics. The advantage of this approach, as opposed to a more axiomatic method, is that each of the primitive rules of definition in the HOL logic—namely, constant definition, constant specification, and type definition—is guaranteed to preserve consistency. The disadvantage is that these rules admit only definitions that satisfy certain very restrictive rules of formation. Definitions expressed in any other form must always be justified formally by deriving them from equivalent, but possibly rather complex, primitive definitions.

The ML metalanguage allows users to implement derived inference rules in the HOL system and thus provides a facility for automating proofs that justify derived rules of definition. For example, recursive definitions are not admitted by the primitive rules of definition of the HOL logic. But certain recursive type definitions and function definitions are supported in the system by derived inference rules written in ML [6, 10]. The details of the primitive definitions that underlie these rules are hidden from the user, and their ML implementations are highly optimized. So these derived principles of definition may simply be regarded as primitive by most users of the system.

This paper describes a set of theorem-proving tools based around a new derived principle of definition in HOL—the *inductive definition* of relations. The key element is a derived rule which allows the user to define a relation by giving a set of rules for generating its elements. Section 1 provides a brief general introduction to the logical basis for inductive definitions. Section 2 then describes some ML functions that have been implemented for reasoning with inductively defined relations in HOL. Finally, sections 3–5 explain some example applications of these HOL tools: the definition of an operational semantics for a simple programming language and a proof that its evaluation relation is deterministic; the definition of a reduction relation for combinatory logic and a proof that it has the Church-Rosser property; the definition of a Hilbert style proof system for minimal intuitionistic logic; the definition of a type system for combinatory logic and a proof of the Curry-Howard isomorphism for combinatory logic and minimal intuitionistic logic; and definitions of the trace and transition semantics for a simple process algebra, together with the proof of a statement of the relationship between them.

1 Inductive definitions

The following is a simple but typical example of a relation defined inductively by a set of rules. Let $R \subseteq A \times A$ be a binary relation on a set A. The reflexive-transitive closure of R can be defined to be the smallest relation $R^* \subseteq A \times A$ for which the following deduction rules hold.

$$\textbf{R1} \quad \frac{}{R^*(x,y)} \; R(x,y)$$

$$\textbf{R2} \quad \frac{}{R^*(x,x)}$$

$$\textbf{R3} \quad \frac{R^*(x,z) \qquad R^*(z,y)}{R^*(x,y)}$$

These rules state precisely the properties required of the reflexive-transitive closure of the relation R. Rule **R1** states that it must be a closure of R, rule **R2** states that it must be reflexive, and rule **R3** states that it must be transitive. The reflexive-transitive closure R^* could therefore simply be defined to be the smallest relation that satisfies these conditions. It then follows immediately that R^* satisfies these rules and is a subset of any other relation that satisfies them. As will be discussed below, the latter property gives rise to an induction principle for reasoning about the relation R^*.

The definition given above is valid because the rules **R1**, **R2**, and **R3** make only positive statements about the elements of R^*. This guarantees that the smallest relation satisfying these rules in fact exists. In particular, if the rules have this form, then one can show that the intersection of any set of relations that satisfy the rules also satisfies the rules. It is therefore legitimate to define the smallest relation that satisfies the rules to be the intersection of all such relations.

In general, an inductive definition of an n-place relation R consists of a collection of rules of the following form.

$$\frac{R(t_1^1,\ldots,t_n^1) \quad \cdots \quad R(t_1^i,\ldots,t_n^i)}{R(t_1,\ldots,t_n)} \; C_1 \cdots C_j$$

The terms above the line are the *premisses* of the rule, each of which makes a positive assertion of membership in the relation R. The term below the line, called the *conclusion* of the rule, likewise asserts membership in R. The terms $C_1,\ldots,C_j$ are *side conditions* on the rule; these may be arbitrary propositions not involving the relation R being defined. A relation R is *closed* under such a rule if whenever the premisses and side conditions hold, the conclusion also holds.

The relation *inductively defined* by a collection of such rules is the smallest relation closed under all the rules. This relation exists because the intersection of any two relations closed under the rules is also closed under the rules. Hence the intersection of the class of all relations closed under the rules is also closed under the rules and is, moreover, the smallest such relation. See [1] for more details about the theory of inductive definitions.

1.1 Rule induction

For every inductively defined relation there is an associated induction principle which holds by virtue of its definition as the *smallest* relation closed under a set of rules. This principle of *rule induction*[1] may be stated briefly as follows. Let R be an n-place relation inductively defined by a set of rules. Suppose we wish to show that every element of R has a certain property:

$$\text{if } R(x_1,\ldots,x_n) \text{ then } P[x_1,\ldots,x_n] \tag{1}$$

Since R is the smallest relation closed under the rules, any relation S which is also closed under the rules has the property that $R \subseteq S$. Now, let

$$S = \{(x_1,\ldots,x_n) \mid P[x_1,\ldots,x_n]\}$$

Then to prove the desired property of R, it suffices to show that the relation S is closed under the rules that define R. For if the relation S in fact is closed under the rules, then we have that $R \subseteq S$ and therefore that every element of R has the defining property of S—i.e. the statement labelled (1) holds of the relation R, as required.

For the reflexive-transitive closure R^*, the principle of rule induction is stated as follows: to prove that a property $P[x,y]$ holds for all x and y for which $R^*(x,y)$, it suffices to show that (i) for all x and y, $R(x,y)$ implies $P[x,y]$, (ii) for all x, $P[x,x]$ and (iii) for all x, y, and z, $P[x,z]$ and $P[z,y]$ imply $P[x,y]$. This is an inductive form of argument; if the property P holds in the 'base cases', corresponding to rules **R1** and **R2**, and if P is preserved

[1] The term 'rule induction' was coined by Glynn Winskel in [11].

by the rule **R3** (the 'step case' of the induction), then every pair in R^* has the property P. A similar induction principle holds for every relation inductively defined by a set of rules.

In addition to rule induction, there is also a slightly stronger induction principle for each inductively defined relation. This principle, which we shall call *strong* rule induction, is in theory dispensable but occasionally very useful in practice. Suppose again that R is an n-place inductively defined relation and that we wish to prove statement (1) shown above. As before, let

$$S = \{(x_1, \ldots, x_n) \mid P[x_1, \ldots, x_n]\}$$

The principle of rule induction says that $R \subseteq S$ provided S is closed under the rules defining R. Suppose that one of these rules is

$$\frac{R(t_1^1, \ldots, t_n^1) \quad \cdots \quad R(t_1^i, \ldots, t_n^i)}{R(t_1, \ldots, t_n)} \; C_1 \cdots C_j$$

Then to prove that S is closed under this rule, we must show that if the side conditions $C_1, \ldots, C_j$ hold and if $P[t_1^1, \ldots, t_n^1], \ldots, P[t_1^i, \ldots, t_n^i]$ hold, then we have that $P[t_1, \ldots, t_n]$ holds. The principle of strong rule induction is based on the observation that in proving this last assertion, we may also assume that $R(t_1^1, \ldots, t_n^1), \ldots, R(t_1^1, \ldots, t_n^1)$ hold, since we could always have taken

$$S = \{(x_1, \ldots, x_n) \mid P[x_1, \ldots, x_n] \wedge R(x_1, \ldots, x_n)\}$$

in the first place. The principle of strong rule induction is just ordinary rule induction with these additional assumptions in the step cases. Some specific examples are given in later sections.

1.2 Inductive definitions in logic

Inductive definitions are based on the notion of a relation being closed under a set of rules. Since rules are essentially implications—*if* the premisses and side conditions hold, *then* the conclusion holds—it is straightforward to express this concept formally in logic.

Consider, for example, the rules given above for reflexive-transitive closure. Let $R : \alpha \rightarrow \alpha \rightarrow bool$ be a fixed but arbitrary relation on α. (Here, a relation is represented by a curried function; but we shall continue to speak loosely of a pair of values x and y as being 'in' the relation R when $R\ x\ y$ holds.) The following formula then asserts that a relation $P : \alpha{\rightarrow}\alpha{\rightarrow}bool$ is closed under the rules defining the reflexive-transitive closure of R:

$$\begin{array}{l}(\forall x\ y.\ R\ x\ y \supset P\ x\ y) \wedge \\ (\forall x.\ P\ x\ x) \wedge \\ (\forall x\ y.\ (\exists z.\ P\ x\ z \wedge P\ z\ y) \supset P\ x\ y)\end{array}$$

Each rule is expressed by a universally quantified implication in which the premisses and side conditions of the rule imply its conclusion. A rule with no side conditions or premisses just becomes a universally quantified assertion. Closure of a relation under any set of rules of the form discussed above can be expressed in logic in a similar way.

Given this scheme for expressing closure under a set of rules generally, one can then define the *smallest* relation closed under a given set of rules by taking the intersection of all such relations. For example, a function

$$\mathsf{Rtc} : (\alpha{\rightarrow}\alpha{\rightarrow}bool) \rightarrow (\alpha{\rightarrow}\alpha{\rightarrow}bool)$$

that maps an arbitrary relation $R : \alpha{\rightarrow}\alpha{\rightarrow}bool$ to its reflexive-transitive closure $\mathsf{Rtc}\ R$ can be defined in the HOL logic by the constant definition:

$$\begin{array}{l}\vdash \forall R\ x\ y.\ \mathsf{Rtc}\ R\ x\ y\ = \\ \quad\quad \forall P.\ ((\forall x\ y.\ R\ x\ y \supset P\ x\ y) \wedge \\ \quad\quad\quad\quad (\forall x.\ P\ x\ x) \wedge \\ \quad\quad\quad\quad (\forall x\ y.\ (\exists z.\ P\ x\ z \wedge P\ z\ y) \supset P\ x\ y)) \\ \quad\quad\quad\quad\quad \supset \\ \quad\quad\quad\quad P\ x\ y\end{array}$$

This definition states that a pair x and y is in the relation $\mathsf{Rtc}\ R$ exactly when it is in every relation P closed under the rules for reflexive-transitive closure. That is, $\mathsf{Rtc}\ R$ is defined to be the intersection of all relations closed under these rules. As will be discussed in the section that follows, this indeed makes $\mathsf{Rtc}\ R$ the smallest such relation, as required.

1.3 Deriving the rules and rule induction

Any relation intended to be defined inductively by a set of rules can be defined formally in the HOL logic by a constant definition of the kind illustrated by the Rtc example given above. Such a definition, however, merely introduces the relation as the intersection of all relations that satisfy the desired set of rules. The proof obligations of a derived principle of inductive definition are, first of all, to show that the resulting relation in fact does satisfy these rules, and secondly to show that it is indeed the smallest such relation. It is these proof obligations which are automated by the HOL inference rule described below in section 2.

For the simple reflexive-transitive closure example, the first proof obligation is to show that:

$$\vdash \forall R\, x\, y.\, R\ x\ y \supset \mathsf{Rtc}\ R\ x\ y$$

$$\vdash \forall R\, x.\, \mathsf{Rtc}\ R\ x\ x$$

$$\vdash \forall R\, x\, y.\, (\exists z.\, \mathsf{Rtc}\ R\ x\ z \wedge \mathsf{Rtc}\ R\ z\ y) \supset \mathsf{Rtc}\ R\ x\ y$$

That is, one must prove that the rules **R1**, **R2**, and **R3** follow from the somewhat indirect formal definition of the relation $\mathsf{Rtc}\ R$ given in the previous section. The second proof obligation is to show that $\mathsf{Rtc}\ R$ is the smallest relation that satisfies these rules:

$$\begin{array}{l} \vdash \forall R\, P.\, ((\forall x\, y.\, R\ x\ y \supset P\ x\ y) \wedge \\ \qquad\quad (\forall x.\, P\ x\ x) \wedge \\ \qquad\quad (\forall x\, y.\, (\exists z.\, P\ x\ z \wedge P\ z\ y) \supset P\ x\ y)) \\ \qquad\qquad \supset \\ \qquad\quad \forall x\, y.\, \mathsf{Rtc}\ R\ x\ y \supset P\ x\ y \end{array}$$

This is the principle of rule induction for the relation $\mathsf{Rtc}\ R$. These four theorems are a complete statement of the defining properties of reflexive-transitive closure. All four can be proved fully automatically in HOL by the derived inference rule described in the next section.

The principle of strong rule induction for $\mathsf{Rtc}\ R$ is

$$\begin{array}{l} \vdash \forall R\, P.\, ((\forall x\, y.\, R\ x\ y \supset P\ x\ y) \wedge \\ \qquad\quad (\forall x.\, P\ x\ x) \wedge \\ \qquad\quad (\forall x\, y.\, (\exists z.\, \mathtt{Rtc}\ R\ x\ z \wedge P\ x\ z \wedge \mathtt{Rtc}\ R\ z\ y \wedge P\ z\ y) \supset P\ x\ y)) \\ \qquad\qquad \supset \\ \qquad\quad \forall x\, y.\, \mathsf{Rtc}\ R\ x\ y \supset P\ x\ y \end{array}$$

Note the additional hypotheses in the third case of the induction—one needs to prove that P is transitive only for pairs also in the relation $\mathsf{Rtc}\ R$, rather than for all pairs, as in ordinary rule induction.

2 Mechanization in HOL

The main component of the HOL tools described in this paper is a function that takes as an argument a list of rules and automatically proves the defining properties of the relation inductively defined by them. More precisely, this derived HOL inference rule builds a term that denotes the smallest relation closed under the rules using the intersection construction described in the previous section. A constant is then introduced to name this relation. The result is a set of theorems stating that the newly-defined relation is the smallest relation closed under the rules supplied by the user.

The ML function that implements this principle of inductive definition is:

```
new_inductive_definition
 : bool ->                          (infix flag)
   string ->                        (definition name)
   (term # term list) ->            (pattern)
   (term list # term) list ->       (rules)
   (thm list # thm)                 (result)
```

The first argument to this function is a boolean flag which indicates if the constant that is defined is to have infix syntactic status of not. The second argument is the name under which the resulting definition will be saved on disk. The third argument is a 'pattern' that supplies information which is needed because this ML function can be used to define classes of inductively defined relations, rather than just single instances of these relations. Details of the purpose and format of this pattern will be explained later. The final argument is a list of rules, each of which is represented by a pair of the form

`([`*premisses and side conditions*`],` *conclusion*`)`

The first component is a list of the premisses and side conditions, which may be arranged in any order. The second component is the conclusion of the rule. Side conditions can be arbitrary boolean terms, provided they do not mention the relation being defined. The premisses and conclusion must be positive assertions of membership in the relation being defined. The precise form that these assertions must take is explained later, but roughly speaking the premisses and conclusion of a rule must be terms of form "$R\ t_1\ \ldots\ t_n$", where R : σ_1 `->` ... `->` σ_n `-> bool` is a variable representing the n-place relation that is to be defined, and each $t_i : \sigma_i$ is an arbitrary term not containing R.

Given an infix flag, a name, a pattern, and a list of rules, the ML function `new_inductive_definition` automatically proves the existence of the smallest relation that satisfies these rules. A constant is then introduced to denote this relation using a constant specification, the result of which is saved on disk under the supplied name. The value returned is a pair consisting of a list of theorems which state that the newly-defined relation satisfies the rules, together with a theorem asserting rule induction for the relation. The result is a complete statement of the defining properties for the smallest relation closed under the specified set of rules.

As stated in section 1.1, there is also a strong principle of rule induction for any inductively defined relation. This can be derived automatically in HOL by the ML function

```
derive_strong_induction : (thm list # thm) -> thm
```

which takes as an argument a pair whose first component is the list of theorems asserting closure under the rules and whose second component is the rule induction theorem returned by `new_inductive_definition`.

2.1 A simple example

The following example HOL session shows how `new_inductive_definition` can be used to inductively define the set of even natural numbers.

```
#let (rules,ind) =                                                   1
   let Even = "Even:num->bool" in
   new_inductive_definition false `Even`
   ("^Even n", [])

   [ [
     % ----------------------------- % ],
                 "^Even 0"           ;

     [           "^Even n"
     % ----------------------------- % ],
               "^Even (n+2)"        ];;
```

The first rule in this definition states that `0` is an even natural number, and the second rule states that if `n` is even then `n+2` is also even. (Antiquotation and ML comments are used to give a readable presentation of these rules.)

Since the even natural numbers are exactly those numbers obtainable from zero by adding multiples of two, these rules inductively define '`Even n`' such that it holds precisely when `n` is even.

The value of the pattern in this example is the pair `("Even n",[])`. The first component of this pair indicates that the constant to be defined, namely `Even`, is a one-place function with typical argument `n`. In general, the second component of a pattern is a non-empty list only when a *class* of relations is being defined (see below). In this example, `Even` is just a simple predicate, and the list component of the pattern is therefore empty.

When the definition in box 1 is evaluated, `new_inductive_definition` automatically proves the existence of the smallest predicate closed under the given list of rules and then defines the constant `Even` to denote this predicate. The following automatically-proved theorems about `Even` are then returned:

```
rules =                                                          2
[⊢ Even 0; ⊢ ∀n. Even n ⊃ Even(n + 2)] : thm list
ind =
⊢ ∀P. P 0 ∧ (∀n. P n ⊃ P(n + 2)) ⊃ (∀n. Even n ⊃ P n)
```

The theorems bound to the ML identifier `rules` state that the required rules hold of the predicate `Even`. And the rule induction theorem bound to `ind` states that the set of numbers for which `Even` holds is the smallest set that satisfies these rules. Having obtained these theorems, strong induction for `Even` can be derived as follows:

```
#derive_strong_induction (rules,ind);;                           3
⊢ ∀P. P 0 ∧ (∀n. Even n ∧ P n ⊃ P(n+2)) ⊃ (∀n. Even n ⊃ P n)
```

An analogous set of theorems can be proved automatically for any particular relation inductively defined by a list of rules. The next section shows how the derived principle of inductive definition in HOL can also be used to define a parameterized class of relations.

2.2 Defining a class of relations

The constant Rtc defined above in section 1.2 is not itself an inductively-defined relation, but rather a function that maps an arbitrary relation R to an inductively-defined relation Rtc R. The function Rtc therefore represents an entire class of inductively-defined relations, one for each possible value of the variable R.

The information required by the derived rule `new_inductive_definition` in order to handle the definition of such functions is supplied by its pattern argument. In the general case, a pattern is a pair of the following form:

("$R\ v_1\ \ldots\ v_n$", ["v_i"; ...; "v_j"])

The first component is an application of the n-place curried function that is to be defined (in this case, R) to n distinct variables v_1, ..., v_n. The second component is a list of those variables that occur at the positions in this application which correspond to the parameters of class of inductively-defined relations, rather than to the arguments to these relations themselves.

An example of the role of the pattern argument is provided by the following definition of reflexive-transitive closure in HOL.

```
#let (rules,ind) =                                              4
   let Rtc = "Rtc:(*->*->bool)->*->*->bool" in
   new_inductive_definition false `Rtc`
   ("^Rtc R x y", ["R:*->*->bool"])

   [ [       "R (x:*) (y:*):bool"
     % ----------------------------- % ],
                "^Rtc R x y"          ;

     [
     %------------------------------ % ],
                "^Rtc R x x"          ;

     [  "^Rtc R x z";  "^Rtc R z y"
     %------------------------------ % ],
                "^Rtc R x y"          ];;
```

The pattern in this case is the pair:

```
("Rtc R x y", ["R:*->*->bool"])
```

The first component of this pattern specifies that the function `Rtc` is to take three arguments in total—a relation `R`, and two values `x` and `y`. The second part of the pattern (the list containing `R`) specifies that the relation argument `R` is to be a parameter to the class of inductively-defined relations that will be represented by `Rtc`. The remaining variables `x` and `y` are assumed to indicate the positions of actual arguments to the predicate that represents these relations.

The result of evaluating this inductive definition in HOL is:

5

```
rules =
[⊢ ∀R x y. R x y ⊃ Rtc R x y;
 ⊢ ∀R x. Rtc R x x;
 ⊢ ∀R x y. (∃z. Rtc R x z ∧ Rtc R z y) ⊃ Rtc R x y]
: thm list
ind =
⊢ ∀R P.
   (∀x y. R x y ⊃ P x y) ∧
   (∀x. P x x) ∧
   (∀x y. (∃z. P x z ∧ P z y) ⊃ P x y)
      ⊃
   (∀x y. Rtc R x y ⊃ P x y)
```

The ML variable `rules` has been bound to a list of theorems which state the three rules that inductively define the reflexive-transitive closure of a relation. In addition, the theorem `ind` states the principle of rule induction for the inductively-defined relation `Rtc R`.

2.3 Stating premisses and conclusions

In addition to the use of the pattern argument, the `Rtc` example also illustrates a restriction on the form in which the premisses and conclusions of rules must be supplied to `new_inductive_definition`. As was mentioned above, premisses and conclusions must be positive assertions of membership of the form

"$R\ t_1\ \ldots\ t_n$"

where R is a variable that stands for the function to be defined. The restriction is that some of the terms among the arguments $t_1, \ldots, t_n$ in such an assertion must be variables—namely, the terms that occur at positions which, according to the supplied pattern, correspond to the parameters of a class of relations. In particular, the terms that occur at these positions must be the same variables given in the pattern itself.

The rules for reflexive-transitive closure shown in box 4 conform to this restriction. The pattern in this case indicates that in the typical assertion of membership `"Rtc R x y"` (i.e. the first component of the pattern), the variable `R` marks the position of a parameter to the class of relations to be defined. Every premiss and conclusion mentioned in the rules must therefore be a term of the form "`Rtc R` t_1 t_2", where the arguments t_1 and t_2 may be arbitrary terms but the parameter `R` must be the variable given in the pattern.

2.4 A tactic for rule induction

In addition to the derived rule of inductive definition itself, there are also several auxiliary functions that support reasoning about inductively-defined relations. The most important of these is the following general tactic for goal-directed proofs by rule induction:

```
RULE_INDUCT_THEN
 : thm ->                     (induction theorem)
   (thm -> tactic) ->         (premiss continuation)
   (thm -> tactic) ->         (side condition continuation)
   tactic                     (resulting tactic)
```

The first argument to this function is the rule induction theorem for a given inductively-defined relation. This may be either the rule induction theorem that is returned by `new_inductive_definition` or the strong rule induction theorem proved by `derive_strong_induction`. Like the general structural induction tactic in HOL, the general rule induction tactic is parameterized by functions that determine what is done with induction hypotheses. These may be either premisses or side conditions, and the user may wish to treat these two kinds of induction hypotheses differently. Two separate theorem continuations are therefore supplied as the second and third arguments to the function `RULE_INDUCT_THEN`.

Given the rule induction theorem for an inductively-defined n-ary relation R, the function described above returns a specialized rule induction tactic that reduces goals of the form:

$$"\forall x_1 \ \ldots\ x_n .\ R\, x_1 \ \ldots\ x_n \supset P[x_1, \ldots, x_n]"$$

to the subgoal(s) of proving that the property P is preserved by the rules that inductively define R. The rule induction theorem for `Rtc`, for example, is:

```
#ind;;                                                              6
⊢ ∀R P.
   (∀x y. R x y ⊃ P x y) ∧
   (∀x. P x x) ∧
   (∀x y. (∃z. P x z ∧ P z y) ⊃ P x y)
      ⊃
   (∀x y. Rtc R x y ⊃ P x y)
```

A rule induction tactic for `Rtc` can be constructed from this theorem by making the simple ML definition:

```
#let Rtc_INDUCT_TAC =                                               7
     RULE_INDUCT_THEN ind ASSUME_TAC ASSUME_TAC;;
Rtc_INDUCT_TAC = - : tactic
```

The use of `ASSUME_TAC` in this definition means that the induction hypotheses arising from the premisses and side conditions of the rules are to be added to the assumptions of the subgoals that are generated. The resulting rule induction tactic for `Rtc` is described by:

$$\frac{\Gamma\ \texttt{?-}\ \forall x\ y.\ \texttt{Rtc}\ R\ x\ y \supset P[x,y]}{\begin{array}{c}\Gamma \cup \{R\ x\ y\}\ \texttt{?-}\ P[x,y] \\ \Gamma\ \texttt{?-}\ \forall x.\ P[x,x] \\ \Gamma \cup \{P[x,z],\ P[z,y]\}\ \texttt{?-}\ P[x,y]\end{array}}\ \texttt{Rtc\_INDUCT\_TAC}$$

This tactic implements the induction scheme described above in section 1.1. It reduces the goal of proving that a property $P[x, y]$ holds for all pairs x and y related by `Rtc` R to showing that this property is preserved by the rules that inductively define this relation.

2.4.1 An example

The following session shows how the rule induction tactic for `Rtc` constructed above can be used to prove a simple theorem about this relation. The aim is to show that the reflexive-transitive closure of a symmetric relation is also symmetric. The proof begins by using the HOL subgoal package (see [10]) to set up an appropriate goal to be proved, as shown below.

```
#set_goal                                                    8
   (["∀x:*. ∀y. R x y ⊃ R y x"],
     "∀x:*. ∀y. Rtc R x y ⊃ Rtc R y x");;
"∀x y. Rtc R x y ⊃ Rtc R y x"
    [ "∀x y. R x y ⊃ R y x" ]

() : void
```

The assumption is that the relation `R` is symmetric, and the conclusion states that the closure `Rtc R` is also symmetric. The conclusion of the goal is in precisely the right form for a proof by rule induction using the induction tactic described above. Applying this tactic results in:

```
#expand Rtc_INDUCT_TAC;;                                     9
OK..
3 subgoals
"Rtc R y x"                                   (subgoal 1)
    [ "∀x y. R x y ⊃ R y x" ]
    [ "Rtc R z x" ]
    [ "Rtc R y z" ]

"∀x. Rtc R x x"                               (subgoal 2)
    [ "∀x y. R x y ⊃ R y x" ]

"Rtc R y x"                                   (subgoal 3)
    [ "∀x y. R x y ⊃ R y x" ]
    [ "R x y" ]

() : void
```

Subgoals 1 and 2 are trivial, since the relation `Rtc R` is transitive and reflexive by definition. The tactic proofs for these subgoals can simply use the rules shown above in box 4. The proof of subgoal 3 is also easy. The proposition `"R y x"` follows immediately from the two assumptions of the subgoal; and this proposition together with the fact that

$\vdash \forall$`R x y. R x y` $\supset$ `Rtc R x y`

directly entail the required conclusion.

The proof sketched above is a trivial example of the kind of reasoning sometimes referred to as induction over the structure (or depth) of derivations in a deductive system stated by a set of rules. This form of inductive argument is very common in certain areas of application—for example, in operational semantics or process algebra. It is made directly accessible in HOL by the tactic just described.

2.5 Generating tactics from rules

In addition to the rule induction tactic, there is also mechanized support for generating tactics from the theorems that state the rules for an inductively-defined relation. This takes the form of an ML function:

```
RULE_TAC : thm -> tactic
```

The theorem argument to this function is expected to be a rule expressed in the form proved by `new_inductive_definition`. Given such a theorem, `RULE_TAC` constructs a tactic that inverts the inference rule stated by it. The resulting tactic reduces goals that match the conclusion of the rule to subgoals consisting of the corresponding instances of its premisses and side conditions.

Consider, for example, the transitivity theorem for `Rtc R`:

$$\vdash \forall \texttt{R x y. } (\exists \texttt{z. Rtc R x z} \wedge \texttt{Rtc R z y}) \supset \texttt{Rtc R x y}$$

When applied to this theorem, `RULE_TAC` returns the tactic described by:

$$\frac{\Gamma \texttt{ ?- Rtc } R\ x\ y}{\Gamma \texttt{ ?- } \exists z.\ \texttt{Rtc } R\ x\ z \wedge \texttt{Rtc } R\ z\ y}$$

This tactic can then be used in goal-directed proofs about membership in the inductively-defined relation `Rtc` R. The other two rules that define `Rtc` R can also be converted into tactics using the function `RULE_TAC`. The result is a complete set of HOL tactics for goal-directed proofs in the deductive system comprising the three rules that define reflexive-transitive closure.

2.5.1 Case analysis

The final major component of the HOL tools for reasoning with inductive definitions is an ML function that proves an exhaustive case analysis theorem for any given relation inductively defined by a set of rules. The name and type of this function are:

```
derive_cases_thm : (thm list # thm) -> thm
```

The arguments to this function are the list of rules satisfied by an inductively defined relation, together with its rule induction theorem—i.e. precisely the defining theorems proved by `new_inductive_definition`. When supplied with these theorems, `derive_cases_thm` proves that if an assertion of membership in the relation holds, then it holds only by virtue of the fact that one of the rules can be used to derive it. This allows one to drive the rules that define a relation 'backwards', inferring from the conclusion of one of the rules that the premisses and side conditions hold.

The following interaction with the HOL system shows the theorem proved by `derive_cases_thm` for the `Rtc` example introduced above. The ML variables `rules` and `ind` are assumed to have the bindings shown above in box 5.

10

```
#derive_cases_thm (rules,ind);;
⊢ ∀R x y.
    Rtc R x y =
      (R x y)  ∨
      (y = x)  ∨
      (∃z. Rtc R x z ∧ Rtc R z y)
```

The resulting theorem states that the most general membership assertion `Rtc R x y` holds precisely when either:

- it is derivable by the closure rule **R1**, in which case `x` and `y` must be related by `R`; or

- it is derivable by the reflexivity rule **R2**, in which case `x` and `y` must be equal; or

- it is derivable by the transitivity rule **R3**, in which case there must be an intermediate value `z` such that `Rtc R x z` and `Rtc R z y`.

A similar case analysis theorem can be proved automatically for any relation defined inductively by `new_inductive_definition`.

3 An operational semantics

Our first example is the definition in HOL of the operational semantics of a simple imperative programming language. The syntax of the language we will consider is defined by

C	$::=$	`skip`	(*does nothing*)
	$\mid$	V `:=` E	(*assignment*)
	$\mid$	C_1 `;` C_2	(*sequence*)
	$\mid$	`if` B `then` C_1 `else` C_2	(*conditional*)
	$\mid$	`while` B `do` C	(*while loop*)

where C, C_1, and C_2 range over commands (i.e. programs), V ranges over program variables, E ranges over natural number expressions, and B ranges over boolean expressions. In what follows, natural number expressions and boolean expressions will just be modelled by total functions from states to numbers and booleans respectively. This simplification will allow us to concentrate on defining the operational semantics of commands.

3.1 Modelling the syntax in HOL

The syntax given above is easily embedded in higher order logic as a logical type `comm` defined using the built-in HOL tools for automatically defining concrete recursive data types (see [6, 10]). The user supplies a specification of the required type in a form similar to the little grammar shown above; the system then constructs an appropriate encoding for values of the required type, defines the type using this encoding and the primitive rule of type definition, and automatically proves an abstract characterization of the newly-defined type. The result is a recursive data type with five (prefix) constructors representing the different syntactic classes of commands, with

`skip`	*represented by*	`skip`
V `:=` E	*represented by*	`assign` V E
C_1 `;` C_2	*represented by*	`seq` C_1 C_2
`if` B `then` C_1 `else` C_2	*represented by*	`if` B C_1 C_2
`while` B `do` C	*represented by*	`while` B C

where we have modelled program variables V by elements of a logical type `string` and where E`:state->num` and B`:state->bool` are natural number expressions and boolean expressions respectively, with `state` an abbreviation for the logical type `string->num`.

The abstract characterization in HOL of the type with these constructors is a theorem which states the admissibility of defining functions over commands by primitive recursion. If for notational clarity we introduce two constants `:=` and `;` as infix abbreviations for `assign` and `seq`, then this theorem is:

```
comm =                                                              1
⊢ ∀e f0 f1 f2 f3.
    ∃! fn.
     (fn skip = e) ∧
     (∀s f. fn(s := f) = f0 s f) ∧
     (∀c1 c2. fn(c1;c2) = f1(fn c1)(fn c2)c1 c2) ∧
     (∀f c1 c2. fn(if f c1 c2) = f2(fn c1)(fn c2)f c1 c2) ∧
     (∀f c. fn(while f c) = f3(fn c)f c)
```

As discussed in [10], a structural induction theorem for commands follows from this theorem. Furthermore, one can (automatically) prove in HOL that the functional constructors for the type `comm` are injective:

```
#let inj = prove_constructors_one_one comm;;                        2
inj =
⊢ (∀s f s' f'. (s := f = s' := f') = (s = s') ∧ (f = f')) ∧
  (∀c1 c2 c1' c2'.
     (c1 ; c2 = c1' ; c2') = (c1 = c1') ∧ (c2 = c2')) ∧
  (∀f c1 c2 f' c1' c2'.
     (if f c1 c2 = if f' c1' c2') =
     (f = f') ∧ (c1 = c1') ∧ (c2 = c2')) ∧
  (∀f c f' c'.
     (while f c = while f' c') = (f = f') ∧ (c = c'))
```

In addition, one can prove that different constructors yield different values:

```
#let dist = prove_constructors_distinct comm;;                      3
dist =
⊢ (∀s f. ¬(skip = s := f)) ∧
  (∀c1 c2. ¬(skip = c1 ; c2)) ∧
    ⋮
   etc.
```

Both these elementary syntactic properties of commands will be used in the proofs that follow.

3.2 Definition of the semantics

The operational semantics is represented in logic as an evaluation relation `EVAL`, defined inductively such that `EVAL` $c\ s_1\ s_2$ holds exactly when executing the command c in the initial state s_1 terminates in the final state s_2. The inductive definition of `EVAL` in HOL is shown below.

```
#let (rules,ind) =                                                    4
    let EVAL = "EVAL: comm -> state -> state -> bool" in
    new_inductive_definition false `EVAL`
    ("^EVAL C s1 s2", [])

    [[
     % --------------------------------------------------- %
     ],                  "^EVAL skip s s"                   ;

     [
     % --------------------------------------------------- %
     ],     "^EVAL (V := E) s (λv. (v=V) => E s | s v)"     ;

     [      "^EVAL C1 s1 s2";         "^EVAL C2 s2 s3"
     % --------------------------------------------------- %
     ],              "^EVAL (C1;C2) s1 s3"                  ;

     [           "^EVAL C1 s1 s2";         "B s1"
     % --------------------------------------------------- %
     ],              "^EVAL (if B C1 C2) s1 s2"             ;

     [          "^EVAL C2 s1 s2";         "¬(B s1)"
     % --------------------------------------------------- %
     ],              "^EVAL (if B C1 C2) s1 s2"             ;

     [                       "¬(B s)"
     % --------------------------------------------------- %
     ],                "^EVAL (while B C) s s"              ;

     [  "^EVAL C s1 s2"; "^EVAL (while B C) s2 s3"; "B s1"
     % --------------------------------------------------- %
     ],              "^EVAL (while B C) s1 s3"          ];;
```

The rules given here are just the standard rules for the operational semantics of `while`-programs (see [4] or [9]).

The HOL theorems that are automatically generated as a result of the above definition of `EVAL` are the following theorems stating the rules

```
rules =                                                              5
[⊢ ∀s. EVAL skip s s;
 ⊢ ∀V E s. EVAL(V := E)s(λv. ((v = V) => E s | s v));
 ⊢ ∀C1 s1 C2 s3.
     (∃s2. EVAL C1 s1 s2 ∧ EVAL C2 s2 s3) ⊃
     EVAL(C1 ; C2)s1 s3;
 ⊢ ∀C1 s1 s2 B.
     EVAL C1 s1 s2 ∧ B s1 ⊃ (∀C2. EVAL(if B C1 C2)s1 s2);
 ⊢ ∀C2 s1 s2 B.
     EVAL C2 s1 s2 ∧ ¬B s1 ⊃ (∀C1. EVAL(if B C1 C2)s1 s2);
 ⊢ ∀B s. ¬B s ⊃ (∀C. EVAL(while B C)s s);
 ⊢ ∀C s1 B s3.
     (∃s2. EVAL C s1 s2 ∧ EVAL(while B C)s2 s3 ∧ B s1) ⊃
     EVAL(while B C)s1 s3]
: thm list
```

together with the following rule induction theorem, which defines `EVAL` to be the smallest relation closed under these rules:

```
ind =                                                                6
⊢ ∀P.
    (∀s. P skip s s) ∧
    (∀V E s. P(V := E)s(λv. ((v = V) => E s | s v))) ∧
    (∀C1 s1 C2 s3.
      (∃s2. P C1 s1 s2 ∧ P C2 s2 s3) ⊃ P(C1 ; C2)s1 s3) ∧
    (∀C1 s1 s2 B.
      P C1 s1 s2 ∧ B s1 ⊃ (∀C2. P(if B C1 C2)s1 s2)) ∧
    (∀C2 s1 s2 B.
      P C2 s1 s2 ∧ ¬B s1 ⊃ (∀C1. P(if B C1 C2)s1 s2)) ∧
    (∀B s. ¬B s ⊃ (∀C. P(while B C)s s)) ∧
    (∀C s1 B s3.
      (∃s2. P C s1 s2 ∧ P(while B C)s2 s3 ∧ B s1) ⊃
      P(while B C)s1 s3) ⊃
    (∀C s1 s2. EVAL C s1 s2 ⊃ P C s1 s2)
```

In addition to these defining theorems for `EVAL`, one can also prove the following case analysis theorem using the ML function `derive_cases_thm` introduced in section 2.5.1.

```
#let cases = derive_cases_thm(rules,ind);;                                7
cases =
⊢ ∀C s1 s2.
    EVAL C s1 s2 =
    (C = skip) ∧ (s2 = s1) ∨
    (∃V E.
      (C = V := E) ∧ (s2 = (λv. ((v = V) => E s1 | s1 v)))) ∨
    (∃C1 C2 s2'.
      (C = C1 ; C2) ∧ EVAL C1 s1 s2' ∧ EVAL C2 s2' s2) ∨
    (∃C1 B C2. (C = if B C1 C2) ∧ EVAL C1 s1 s2 ∧ B s1) ∨
    (∃C2 B C1. (C = if B C1 C2) ∧ EVAL C2 s1 s2 ∧ ¬B s1) ∨
    (∃B C'. (C = while B C') ∧ (s2 = s1) ∧ ¬B s1) ∨
    (∃C' B s2'.
      (C = while B C') ∧ EVAL C' s1 s2' ∧
      EVAL(while B C')s2' s2 ∧ B s1)
```

Many useful theorems can be proved as consequences of this general case analysis theorem. In particular, if the quantified variable `C` is specialized to some specific syntactic form, for example '`C1;C2`', then most of the disjuncts in the conclusion become false because of the syntactic inequality of different commands. These false disjuncts can be discarded by rewriting with the fact that the constructors of `comm` are distinct (the theorem `dist` in box 3 above). With further simplification using the injectivity of constructors (the theorem `inj` in box 2 above), one can prove the following cases theorem for sequencing:

```
⊢ ∀s1 s2 C1 C2.
    EVAL(C1;C2)s1 s2 = (∃s3. EVAL C1 s1 s3 ∧ EVAL C2 s3 s2)
```

Similar theorems can be proved for all the constructors of the type `comm`. Using these theorems, one may infer from an assertion '`EVAL` $C\ s_1\ s_2$', where C is some specific command, that the corresponding instance of the premisses of the rule(s) for C must also hold. This kind of reasoning, in which one drives the rules 'backwards', occurs frequently in proofs about operational semantics.

3.3 An example proof

Given the theorems mentioned in the previous section, a formal proof that the operational semantics of our language is deterministic is relatively straightforward. The standard proof of this property is by structural induction [4, 9], but the proof by rule induction outlined below gives rise to a slightly easier analysis in each case of the induction. There are, however, more cases to deal with in the rule induction—one per rule, rather than one per syntactic constructor.

The goal is to prove the following proposition:

```
∀c st1 st2. EVAL c st1 st2 ⊃
            ∀st3. EVAL c st1 st3 ⊃ (st2 = st3)
```

where we have formulated the proposition that `EVAL` is deterministic in exactly the right form for a rule induction. Using the rule induction tactic generated by `RULE_INDUCT_THEN`, the goal is broken down into seven subgoals—one for each production rule in the definition of `EVAL`. We will consider only one of these here (the proofs of the others proceed on similar lines):

```
"∀st3. EVAL(C1 ; C2)s1 st3 ⊃ (s3 = st3)"                              8
    [ "∀st3. EVAL C1 s1 st3 ⊃ (s2 = st3)" ]
    [ "∀st3. EVAL C2 s2 st3 ⊃ (s3 = st3)" ]
```

We need to prove

```
∀st3. EVAL(C1 ; C2)s1 st3 ⊃ (s3 = st3)
```

under the induction hypotheses

```
∀st3. EVAL C1 s1 st3 ⊃ (s2 = st3)

∀st3. EVAL C2 s2 st3 ⊃ (s3 = st3)
```

The natural way to proceed is to assume that `EVAL(C1 ; C2)s1 st3` holds. One then observes that, by shorter inferences with the rules for `EVAL`, we must have `EVAL C1 s1 s` and `EVAL C2 s st3` for some intermediate state `s`. These transitions, together with the induction hypotheses, imply the required equation `s3 = st3`.

This informal reasoning is simple to mimic in HOL, given the cases theorem for sequencing shown in the previous section. In particular, this theorem is the formal justification for the 'by shorter inferences...' part of the informal proof given above. Rewriting the goal in box 8 using this theorem results in the following transformed goal.

```
"∀st3. (∃s3. EVAL C1 s1 s3 ∧ EVAL C2 s3 st3) ⊃ (s3 = st3)"   9
     [ "∀st3. EVAL C1 s1 st3 ⊃ (s2 = st3)" ]
     [ "∀st3. EVAL C2 s2 st3 ⊃ (s3 = st3)" ]
```

This is straightforward to prove using standard HOL techniques for tactic proofs in the predicate calculus.

3.4 Proving the soundness of Floyd-Hoare rules

Our final example in this section is a proof of soundness for the Floyd-Hoare rules of partial correctness for `while`-programs (see [3]). We are interested in correctness specifications of the form $\{P\}\ C\ \{Q\}$, where C is a command and P and Q are conditions on the values of the program variables in C. We will represent such a correctness specification in logic by the proposition '`SPEC P C Q`', the meaning of which is defined by

```
⊢ ∀P C Q. SPEC P C Q = ∀s1 s2. (P s1 ∧ EVAL C s1 s2) ⊃ Q s2
```

That is, if (according to our semantics) running the command `C` in a state satisfying the preconditon `P` terminates, then it does so in a state satisfying the postcondition `Q`. This is just the standard meaning of partial correctness in Floyd-Hoare logic.

Consider now the standard proof rule for the `while` construct:

$$\frac{\vdash \{P \wedge B\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{while}\ B\ C\ \{P \wedge \neg B\}}$$

The aim here is to prove the soundness of this rule with respect to our operational semantics. We can express the rule in logic by

```
∀P C. SPEC (λs. P s ∧ (B s)) C P ⊃
      SPEC P (while B C) (λs. P s ∧ ¬B s)
```

The rule says that if `P` is an invariant for one execution of `C` whenever `B` holds, then it is also an invariant for the execution of `while B C`. Moreover, `B` will be false when the loop terminates.

The HOL proof of this rule is done in two steps. First we prove a lemma that states that the condition `B` in `while B C` must be false upon termination of the loop. We express the lemma in a form suitable for rule induction on the rules for `EVAL`:

```
∀C s1 s2. EVAL C s1 s2 ⊃
          ∀B' C'. (C = while B' C') ⊃ ¬(B' s2)
```

Note that we are doing a proof by rule induction in which we effectively consider only the rules for `while`. That is, we formulate the problem to be showing that the set

$$\{(\texttt{while}\ B\ C, s_1, s_2) \mid \neg(B\ s_2)\} \ \cup\ \{(C, s_1, s_2) \mid \neg(C = \texttt{while}\ B'\ C')\}$$

is closed under the rules for the evaluation relation. This makes all the cases in the rule induction except the ones for the `while` rules trivial. Furthermore, the result immediately give us the desired property, namely that

```
⊢ ∀B C s1 s2. EVAL (while B C) s1 s2 ⊃ ¬(B s2)
```

The above approach illustrates a general way of proving a property of some specific class of commands by rule induction. One takes the union of two sets: the set containing triples (C, s_1, s_2) whose command component C ranges over the commands of interest and which have the desired property, and the set of all other triples whose command component is not one of the commands of interest. The proof that this set is closed under the rules holds vacuously for all but the rules for the commands of interest.

For the particular lemma shown above, all but the two cases dealing with the rules for `while` are trivial; the subgoals for these cases are implications with false antecedents of the form ($C =$ `while B' C'`) where C is not a `while` command. Showing that the required property is preserved by the remaining two rules is also completely straightforward.

The second lemma deals with the invariant part of the Floyd-Hoare proof rule for `while` commands. The goal is to show that if `P` is an invariant of `C`, then it is also an invariant of `while B C`. The proof is essentially an induction on the number of applications of the evaluation rule for `while`-commands. This is expressed as a rule induction, which establishes that the set

$$\{(\texttt{while}\ B\ C, s_1, s_2) \mid P\ \textit{an invariant of}\ C \supset (P\ s_1 \supset P\ s_2)\}$$

is closed under the rules. As in the proof of the first lemma, the rules for commands other than `while` loops are dealt with by taking the union of this set and

$$\{(C, s_1, s_2) \mid \neg(C = \texttt{while}\ B'\ C')\}$$

Proving closure under evaluation of rules other than the two rules for `while` is then trivial, as outlined earlier.

In the HOL logic, the lemma to be proved is the formula

```
∀C s1 s2.
   EVAL C s1 s2 ⊃
   ∀B' C'. (C = while B' C') ⊃
           (∀s1 s2. P s1 ∧ B' s1 ∧ EVAL C' s1 s2 ⊃ P s2) ⊃
           (P s1 ⊃ P s2)
```

The proof of this lemma proceeds by strong rule induction; with this particular formulation and ordinary rule induction, one obtains hypotheses that are too weak to imply the desired conclusion in the subgoal generated for the second rule for `while`. To see why, suppose one attempts to prove the lemma by rule induction. The only non-trivial case of the induction is the case for `while`, in which we are required to prove

```
P s1 ⊃ P s3
```

under the assumptions

```
1: (∀s1 s2. P s1 ∧ B s1 ∧ EVAL C s1 s2 ⊃ P s2)

2: (∀s1 s2. P s1 ∧ B s1 ∧ EVAL C s1 s2 ⊃ P s2) ⊃
   (P s2 ⊃ P s3)

3: B s1
```

where `s3` is a variable (suitably chosen to avoid name clashes) representing the intermediate state reached by the while loop after executing its body once. From these assumptions, we can prove that `P s2 ⊃ P s3`. However, we need the additional fact that `EVAL C s1 s2` in order to show that `P s1 ⊃ P s2` and hence (by transitivity) that `P s1 ⊃ P s3`.

Using the principle of strong rule induction for this lemma gives us precisely what we need. For the case considered above, we will have the two additional assumptions

`EVAL C s1 s2` and `EVAL (while B C) s2 s3`

The first of these assumptions is exactly the additional fact required for the proof sketched above to go through.

Combining the two lemmas proved above gives us the soundness of the Floyd-Hoare partial correctness rule for `while` commands with respect to the operational semantics presented earlier. Proofs of soundness for the Floyd-Hoare rules of the other constructs in our little language can all be done in a similar way.

4 Combinatory logic in HOL

Our second major example is the definition in HOL of reduction for terms of combinatory logic and the proof that combinator reduction has the Church-Rosser property. This property can be stated in logic for an arbitrary binary relation `R` as follows:

$$\vdash \texttt{CR R} = \forall\texttt{a b. R a b} \supset (\forall\texttt{c. R a c} \supset (\exists\texttt{d. R b d} \wedge \texttt{R c d}))$$

In the following sections, we give an inductive definition of a reduction relation `--->` on terms of combinatory logic and prove that it has the Church-Rosser property as formulated above. The proof presented here follows the same lines as the proof by Tait given in [5].

4.1 The syntax of terms

The syntax of terms in combinatory logic can be represented in the HOL logic by a recursive type `cl`, defined in the usual way using the recursive types package discussed in section 3.1. Informally, the syntax is given by

cl	::=	`s`	(*S combinator*)
	|	`k`	(*K combinator*)
	|	*cl cl*	(*application*)

In the logic, this abstract syntax is represented by a recursive data type `cl` with three constructors, two constants `s:cl` and `k:cl` and a prefix constructor `ap:cl->cl->cl` for application. For notational convenience, the constant `'` is introduced as an infix abbreviation for `ap`. All the usual elementary syntactic properties about terms of combinatory logic (e.g. structural induction) are easily derivable in HOL as theorems about the type `cl`.

4.2 Definition of contraction

Reduction for terms of combinatory logic is defined in terms of the contraction relation `-1->` inductively defined by the rules shown in box 1. This is the *weak contraction* relation of Hindley and Seldin [5]. A redex in this case is a term of the form '`k` x y' or '`s` x y z'. A term u contracts to a term v (i.e. u `-1->` v) if v can be obtained by replacing one occurrence of a redex in u, where `k` x y is replaced by x and `s` x y z is replaced by $(x\ z)y\ z$. The first two rules in the definition below define the contractions of redexes; the second two rules define the contraction of subterms.

```
#let (Crules,Cind) =                                                    1
     let CTR = "-1->:cl->cl->bool" in
     new_inductive_definition true `contract`
     ("^CTR U V", [])

     [[
      % ------------------------------------------------- % ],
                    "^CTR ((k ' x) ' y) x"                 ;
      [
      % ------------------------------------------------- % ],
         "^CTR (((s ' x) ' y) ' z)  ((x ' z) ' (y ' z))"   ;

      [                      "^CTR x y"
      % ------------------------------------------------- % ],
                      "^CTR (x ' z) (y ' z)"               ;

      [                      "^CTR x y"
      % ------------------------------------------------- % ],
                      "^CTR (z ' x) (z ' y)"               ];;
```

As usual, making this definition in HOL results in a list of theorems stating that the above contraction rules hold of the relation `-1->`, together with a rule induction theorem for this relation. In addition, one can automatically generate an exhaustive case analysis theorem for `-1->` (see section 2.5.1).

The rules proved automatically by **`new_inductive_definition`** for the contraction relation `-1->` are the following

```
Crules =                                                          2
[⊢ ∀x y. ((k ' x) ' y) -1-> x;
 ⊢ ∀x y z. (((s ' x) ' y) ' z) -1-> ((x ' z) ' (y ' z));
 ⊢ ∀x y. x -1-> y ⊃ (∀z. (x ' z) -1-> (y ' z));
 ⊢ ∀x y. x -1-> y ⊃ (∀z. (z ' x) -1-> (z ' y))]
: thm list
```

As was discussed in section 2.5, these theorems can be (automatically) converted into tactics for proving assertions of the form u `-1->` v. The third theorem, for example, can be used to create a tactic that reduces any goal of the form (x `'` z) `-1->` (y `'` z) to the goal x `-1->` y. A similar tactic is justified by the fourth theorem. In the case of the first two theorems, the generated tactics just prove goals of the forms `((k '` x`) '` y`) -1->` x and `(((s '` x`) '` y`) '` z`) -1->` (x `'` z) `'` (y `'` z). In the next section, these individual tactics will be used to write a more general tactic for automatically checking the truth of an arbitrary assertion that one term contracts to another.

4.3 Definition of reduction

The *reduction* relation `--->` on terms of combinatory logic is just the reflexive-transitive closure of the contraction relation `-1->`. In other words, `--->` can be defined to be `Rtc(-1->)`, where `Rtc` is the function defined in section 2.2:

```
⊢ ∀U V. (U ---> V) = Rtc -1-> U V
```

As stated earlier, the aim of this example is to prove that this reduction relation is Church-Rosser. We can prove that taking the reflexive-transitive closure of a relation preserves the Church-Rosser property—see section 4.4.2 below. So if we can prove that the contraction relation `-1->` has this property, then we will have shown that the reduction relation `--->` also does. The trouble with this approach, however, is that `-1->` is not Church-Rosser.

A counter-example is the term `(k ' i) ' (i ' i)` where the term `i` is an abbreviation for `(s ' k) ' k`. This contracts both to the term `i` and to the term `(k ' i) ' ((k ' i) ' (k ' i))`. These two terms, however, do not contract to any common term of combinatory logic. In particular, we do have the contraction `(k ' i) ' ((k ' i) ' (k ' i)) -1-> i`, but `i` does not contract to `i` or indeed to any other term.

To derive this counter-example in HOL system, one needs to prove

1: ⊢ `k i (i i) -1-> i`

2: ⊢ `k i (i i) -1-> (k i)((k i) (k i))`

3: ⊢ `k i ((k i)(k i)) -1-> i`

4: ⊢ `¬(i -1-> i)`

Using the tactics mentioned in the previous section, it is straightforward to construct a general-purpose tactic for automatically checking whether one term contracts to another. Given a goal "u `-1->` v", where u and v are terms of combinatory logic, the tactic just carries out a systematic search for a proof of this goal using the rules for `-1->`. If a proof is found, the result is the theorem ⊢ u `-1->` v. The ML definition of this tactic is simply

```
#letrec CONT_TAC g =                                                    3
    FIRST [Cs_TAC;
           Ck_TAC;
           LCap_TAC THEN CONT_TAC;
           RCap_TAC THEN CONT_TAC] g ? failwith `CONT_TAC`
 where
    [Ck_TAC;Cs_TAC;LCap_TAC;RCap_TAC] = map RULE_TAC Crules;;
CONT_TAC = - : tactic
```

where `Crules` has the value shown in the previous section. The first three theorems listed above can be proved automatically using this tactic.

For the proof that ¬(`i -1-> i`) we construct a more general tactic based on the case analysis theorem for the `-1->` relation:

```
#let Ccases = derive_cases_thm (Crules,Cind);;                          4
Ccases =
⊢ ∀U V.
    U -1-> V =
    (∃y. U = (k ' V) ' y) ∨
    (∃x y z.
      (U = ((s ' x) ' y) ' z) ∧ (V = (x ' z) ' (y ' z))) ∨
    (∃x y z. (U = x ' z) ∧ (V = y ' z) ∧ x -1-> y) ∨
    (∃x y z. (U = z ' x) ∧ (V = z ' y) ∧ x -1-> y)
```

This tactic, which we call `EXPAND_CASES_TAC`, uses this theorem to rewrite (once) any subterm of the form u `-1->` v that appears within a goal. It then proceeds to simplify the resulting expression into a form that allows repeated application of the tactic, so that ultimately the truth or falsehood of u `-1->` v can be deduced if possible. If u and v contain free variables then repetitive application of the tactic may diverge, since then rewriting u `-1->` v with `Ccases` can always result in further assertions which involve `-1->` but cannot be proved from the axioms.

The effect of `EXPAND_CASES_TAC` can be illustrated by the way it proceeds to solve the goal

```
"¬(∃U. ((s ' k) ' k) -1-> U)"
```

First, it rewrites the goal with the cases theorem `Ccases` to get:

```
"¬(∃U.
    (∃y. (s ' k) ' k = (k ' U) ' y) ∨
    (∃x y z.
      ((s ' k) ' k = ((s ' x) ' y) ' z) ∧
      (U = (x ' z) ' (y ' z))) ∨
    (∃x y z.
      ((s ' k) ' k = x ' z) ∧ (U = y ' z) ∧ x -1-> y) ∨
    (∃x y z.
      ((s ' k) ' k = z ' x) ∧ (U = z ' y) ∧ x -1-> y))"
```

The equations occuring in this term are then simplified using the injectivity of application and the distinctness of constructors for `cl`, and the result is then rearranged to get

```
"¬((∃U x y z.
     (s ' k = x) ∧ (k = z) ∧ (U = y ' z) ∧ x -1-> y) ∨
   (∃U x y z.
     (s ' k = z) ∧ (k = x) ∧ (U = z ' y) ∧ x -1-> y))"
```

Finally, all redundant equations are eliminated to yield the following subgoal, which is equivalent to the original goal but has simpler terms of `cl` in its assertions about contractions.

```
"¬((∃y. (s ' k) -1-> y) ∨ (∃y. k -1-> y))"
```

Transformation of the goal into this form concludes the first application of `EXPAND_CASES_TAC`. A second application of the tactic yields the following transformation of the goal.

```
"¬((∃y'. s -1-> y') ∨ (∃y'. k -1-> y'))"
```

A third application of the tactic solves the goal. We therefore have that ⊢ `¬∃U. i -1-> U` and in particular we have ⊢ `¬(i -1-> i)`. Hence all the theorems needed to show that `k i (i i)` is a counter-example to `CR(-1->)` have been proved.

4.4 Parallel contraction and reduction of CL terms

In the preceding sections, contraction (`-1->`) and reduction (`--->`) of terms in combinatory logic have been defined and `CR(-1->)` was shown not to hold. We now define a parallel contraction relation `=1=>` which allows any number of redexes among the subterms of a term to be contracted in a single step. This relation *does* have the Church-Rosser property and, moreover, its transitive closure equals `--->`. The HOL definition is:

```
#let (PCrules,PCind) =                                                   5
     let PCTR = "=1=>:cl->cl->bool" in
     new_inductive_definition true `pcontract`
     ("^PCTR U V", [])

     [[
      % --------------------------------------------------- % ],
                          "^PCTR x x"                         ;
      [
      % --------------------------------------------------- % ],
                   "^PCTR ((k ' x) ' y) x"                    ;

      [
      % --------------------------------------------------- % ],
        "^PCTR (((s ' x) ' y) ' z) ((x ' z) ' (y ' z))"       ;

      [             "^PCTR w x";      "^PCTR y z"
      % --------------------------------------------------- % ],
                   "^PCTR (w ' y) (x ' z)"                    ];;
```

Clearly, the counter-example used to show that contraction does not satisfy `CR` does not apply to parallel contraction.

In the sections that follow, the parallel contraction relation just defined is used to prove the main result of this section—namely, that reduction of terms in combinatory logic has the Church-Rosser property. First, we give the HOL proof that the parallel contraction relation `=1=>` satisfies `CR`. We then define transitive closure `Tc` R of a relation R, and prove that taking the transitive closure of a relation preserves `CR`. We conclude that `Tc(=1=>)` (called parallel reduction) satisfies `CR`. Finally, parallel reduction is shown to be equal to reduction of terms in combinatory logic. Hence we deduce that reduction satisfies `CR`.

4.4.1 Parallel contraction is Church-Rosser

This section gives an overview of the HOL proof of the following result.

Theorem 1: ⊢ `CR(=1=>)` (parallel contraction has `CR`)

Proof: The first step is to unfold the definition of `CR`. In other words, we need to prove:

```
∀a b. a =1=> b ⊃ ∀c. a =1=> c ⊃ ∃d. b =1=> d ∧ c =1=> d
```

The proof proceeds by strong rule induction on the relation `=1=>`. The four cases of the induction are:

```
1: "(w ’ y) =1=> c ⊃ (∃d. (x ’ z) =1=> d ∧ c =1=> d)"
      [ "w =1=> x" ]
      [ "∀c. w =1=> c ⊃ (∃d. x =1=> d ∧ c =1=> d)" ]
      [ "y =1=> z" ]
      [ "∀c. y =1=> c ⊃ (∃d. z =1=> d ∧ c =1=> d)" ]

2: "(((s ’ x) ’ y) ’ z) =1=> c ⊃
    (∃d. ((x ’ z) ’ (y ’ z)) =1=> d ∧ c =1=> d)"

3: "((k ’ x) ’ y) =1=> c ⊃ (∃d. x =1=> d ∧ c =1=> d)"

4: "x =1=> c ⊃ (∃d. x =1=> d ∧ c =1=> d)"
```

The implications in cases 2, 3 and 4 are solved by case analysis on the antecedent using the cases theorem for `=1=>`, followed by a straightforward search for the proof of the consequent using the tactics for `=1=>`. The overall proof strategy is similar to the one used in section 3.3 and closely follows what one would naturally do in a pencil-and-paper proof. Case 1 is solved also by first analysing the antecedent using the cases theorem for `=1=>`. For the resulting two sub-cases, however, one needs to do an additional case analysis on the strong induction assumption.

We will consider the proof of subgoal number 3 in more detail. By repetitive expansion of the antecedent `((k ' x) ' y) =1=> c` with the cases theorem for `=1=>`, one can generate the case analysis given by

```
⊢ ((k ' x) ' y) =1=> c =
   (c = (k ' x) ' y) ∨
   (x = c) ∨
   (∃z. (c = (k ' x) ' z) ∧ y =1=> z) ∨
   (∃z z'. (c = (k ' z') ' z) ∧ x =1=> z' ∧ y =1=> z)
```

A general-purpose conversion was written to automate this expansion. It rewrites a term u `=1=>` v with the cases theorem for `=1=>` until the result contains only subterms x `=1=>` y where x and y are both variables.

The above case analysis theorem is used to rewrite the antecedent of subgoal 3. After some minor simplification, the following four subgoals (representing the case analysis) are generated:

```
3a: "∃d. x =1=> d ∧ ((k ' z') ' z) =1=> d"
      [ "x =1=> z'" ]
      [ "y =1=> z" ]

3b: "∃d. x =1=> d ∧ ((k ' x) ' z) =1=> d"
      [ "y =1=> z" ]

3c: "∃d. c =1=> d ∧ c =1=> d"

3d: "∃d. x =1=> d ∧ ((k ' x) ' y) =1=> d"
```

At this stage, an appropriate witness is supplied for the existential quantifier of each subgoal. Then a straightforward search for the proof of each subgoal is done using the tactics justified by the rules that define `=1=>`.

4.4.2 Transitive closure preserves Church-Rosser

The next step is to prove that taking the transitive closure of a relation preserves the Church Rosser property. The transitive closure `Tc` R of a relation R is defined inductively in HOL as shown in the following box.

```
#let (Tcrules,Tcind) =                                              6
     let Tc = "Tc:(*->*->bool)->*->*->bool" in
     new_inductive_definition false `Tc`
     ("^Tc R x y", ["R:*->*->bool"])

     [ [
       % ------------------------------ %  "R x y"],
                 "^Tc R x y"            ;

       [         "^Tc R x z"            ;
       % ------------------------------ %  "R z y"],
                 "^Tc R x y"        ];;
```

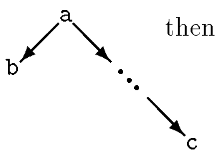
As with the constant `Rtc` discussed in section 2.2, the constant `Tc` is not itself an inductively-defined relation, but rather a function that maps an arbitrary relation R to an inductively-defined relation `Tc` R.

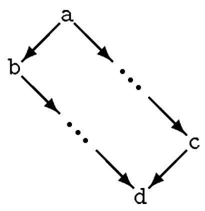
Note that the transitivity rule in the definition given above is not a rule with premisses `"^Tc R x z"` and `"^Tc R z y"` and conclusion `"^Tc R x y"`. This is because the rule actually used in the definition of `Tc` gives a linear structure to rule inductions for transitive closure. This makes the details of these proofs easier to handle than the tree-shaped structure induced by the rule shown above. This is evident in the proof of the following lemma.

Lemma 1: The following theorem holds.

$$\vdash \forall \mathtt{R.\ CR\ R} \supset \forall \mathtt{a\ c.\ Tc\ R\ a\ c} \supset \forall \mathtt{b.\ R\ a\ b} \supset \exists \mathtt{d.\ Tc\ R\ b\ d} \wedge \mathtt{R\ c\ d}$$

Proof: The statement of the lemma can be illustrated as follows. Under the assumption that `CR R` holds, we have that

if 

then ∃ `d` such that: 

Because of our chioce of formulation for the transitivity rule in the definition of `Tc`, the proof of this lemma is an easy rule induction down the `a`-to-`c` leg of the rectangle. The details need not be discussed here.

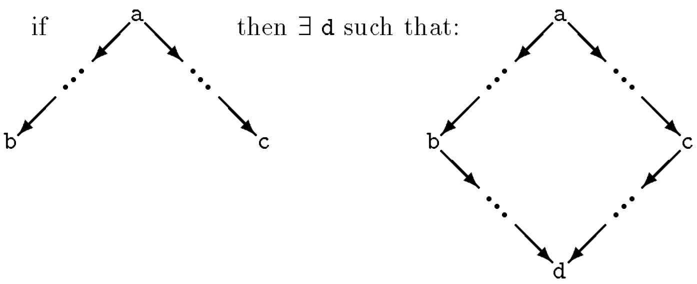
Using Lemma 1, we can prove the main result of this subsection—namely that taking the transitive closure of a relation preserves `CR`.

Theorem 2: ⊢ `∀R. CR R ⊃ CR(Tc R)`

Proof: The HOL proof proceeds by assuming `CR R` and trying to prove the goal `CR(Tc R)` under this assumption. Unfolding the definition of `CR` yields the goal

`∀a b. Tc R a b ⊃ (∀c. Tc R a c ⊃ (∃d. Tc R b d ∧ Tc R c d))`

with the assumption `CR R` unchanged. This goal can be illustrated by:



The proof proceeds by rule induction on `Tc R a b`—i.e. down the `a`-to-`b` leg of the rectangle. Notice that Lemma 1 represents a special case of this goal, namely the case when `R a b` holds. So one can appeal to Lemma 1 in proving the induction subgoal generated for the transitivity rule.

More precisely, applying the rule induction tactic generates two subgoals, one for each rule in the definition of `Tc`. We will consider here only the subgoal generated for the transitivity rule, as the proof of the other subgoal is trivial. The subgoal is

`∃d. Tc R b d ∧ Tc R y d`

with the assumptions:

1: `CR R`

2: `∀b. Tc R x b ⊃ (∃d. Tc R b d ∧ Tc R z d)`

3: `R z y`

4: `Tc R x b`

By Modus Ponens, assumptions 2 and 4 yield two extra facts, namely

5: `Tc R b d'`

6: `Tc R z d'`

where `d'` is suitably chosen to avoid name clashes. The propositions 1, 3 and 6 together with Lemma 1 then yield the results:

7: `Tc R y d''`

8: `R d' d''`

where once again `d''` is suitably chosen to avoid name clashes. We can now supply `d''` as the witness for the existential quantifier of the goal, giving the new goal

```
Tc R b d'' ∧ Tc R y d''
```

The right-hand conjunct is just assumption 7, so this goal reduces to the subgoal `Tc R b d''`. This in turn is simple to prove using assumptions 5 and 8 and the tactic justified by the transitivity rule for `Tc`.

Corollary: ⊢ `CR (Tc(=1=>))`

Proof: Trivial consequence of Theorems 1 and 2.

4.4.3 Reduction is Church-Rosser

The remaining step of the proof of the Church Rosser theorem is to prove that `Tc(=1=>)` equals `--->`.

Theorem 3: ⊢ `∀U V. Tc (=1=>) U V = U ---> V`

Proof: The goal can be reduced to the following subgoals:

```
∀U V. Rtc (-1->) U V ⊃ Tc (=1=>) U V

∀U V. Tc (=1=>) U V ⊃ Rtc (-1->) U V
```

The first can be proved by a rule induction on `Rtc (-1->) U V`, and the second can be proved by a rule induction on `Tc (=1=>) U V`. The proofs are simple and are therefore omitted.

This completes the proof of the Church-Rosser theorem in HOL.

4.5 Minimal Intuitionistic Logic

We conclude this section on combinators with a sketch of the proof in HOL of the Curry-Howard isomorphism [2] for typed combinatory logic and minimal intuitionistic logic (MIL).

The formulas of minimal intuitionistic logic are given by

$$
\begin{array}{rcll}
ty & ::= & \texttt{G *} & (\textit{propositional variables}) \\
 & | & ty \texttt{ -> } ty & (\textit{implication})
\end{array}
$$

which we suppose to be represented in HOL by a recursive data type `(*)ty` defined in the obvious way.

A Hilbert-style proof for a theorem of MIL is a tree built up in the following way. Its leaves are axioms of the two forms

X `->` (Y `->` X), and

(X `->` (Y `->` Z)) `->` ((X `->` Y) `->` (X `->` Z))

The nodes of the proof tree are instances of the inference rule

$$\textbf{Modus Ponens} \quad \frac{A \qquad A \texttt{ -> } B}{B}$$

The root of such a tree is called a *theorem* of MIL.

In HOL, the set of all theorems of MIL can be defined inductively as shown in the following box:

```
#let (Trules,Tind) =                                                    7
     let THM = "THM:(*)ty->bool" in
     new_inductive_definition false `THM_DEF`
     ("^THM p", [])

  [[
   % ----------------------------------------------------- % ],
                    "^THM  (A -> (B -> A))"                 ;

   [
   % ----------------------------------------------------- % ],
      "^THM ((A -> (B -> C)) -> ((A -> B) -> (A -> C)))"    ;

   [         "^THM  (P -> Q)";            "^THM P"
   % ----------------------------------------------------- % ],
                          "^THM  Q"                         ];;
```

The key observation of the Curry-Howard isomorphism is that the types of the combinators `s` and `k` are just the axioms for MIL given above. Moreover, the rule for typing an application u ' v corresponds precisely to the Modus Ponens inference rule in MIL. In HOL, one can define the corresponding typing relation for combinatory logic by the following inductive definition.

```
#let (TYrules,TYind) =                                                    8
     let TY = "IN : cl->(*)ty->bool" in
     new_inductive_definition true `CL_TYPE_DEF`
     ("^TY c t", [])

  [[
   % ---------------------------------------------------- %],
                  "^TY k (A -> (B -> A))"                 ;

   [
   % ---------------------------------------------------- %],
     "^TY s ((A -> (B -> C)) -> ((A -> B) -> (A -> C)))"  ;


   [      "^TY U (t1 -> t2)";          "^TY V t1"
   % ---------------------------------------------------- %],
                     "^TY (U ' V) t2"                     ];;
```

In other words, the types of `cl` terms are just certain propositions of MIL, where implication `->` is viewed as the function type in `cl`.

Given the above definitions, the following statement of the Curry-Howard isomorphism can easily be proved in HOL.

Theorem 4: ⊢ ∀`P:(*)ty. THM P =` ∃`M:cl. M IN P`

Proof: The left-to-right direction is proved by rule induction for the relation `THM`, followed by use of the typing rules (i.e. the tactics for them) to prove the conclusion. The proof in the other direction proceeds by rule induction over the typing relation `IN`. Both inductions are completely straightforward.

5 A rudimentary process algebra

Our final example is the formalization in HOL of a very simple process algebra. Terms of the algebra are called agents. The main result of this section is to prove that there is a correspondence between the maximal traces of agents and the sequences of actions performed by agents according to a labelled transition semantics. See [8] for an introduction to process algebra. For a much more complex example done in HOL, see [7].

The syntax of agents is defined as follows:

$$
\begin{array}{rcll}
A & ::= & \texttt{Nil} & (\textit{does nothing}) \\
 & \mid & \texttt{Pre label } A & (\textit{action prefixing}) \\
 & \mid & \texttt{Sum } A_1 \; A_2 & (\textit{nondeterministic choice}) \\
 & \mid & \texttt{Prod } A_1 \; A_2 & (\textit{parallel composition})
\end{array}
$$

where `label` is the type whose elements represent the names of actions. We assume that `agent` is a recursive type defined in HOL so that it corresponds to this syntax. The type `agent` may, as usual, be defined automatically using the HOL type definition tools.

5.1 The maximal trace semantics

A *trace* of an agent is a sequence of actions that the agent can perform. In HOL, we can represent traces by the type `(action)list`, which we shall abbreviate by `trace`. A *terminal agent* is an agent that may not perform any actions. A *maximal trace* of an agent is a trace containing a sequence of actions that the agent can perform to reach a terminal agent.

The maximal traces for our language of processes are defined informally as follows. The maximal trace of the `Nil` agent is empty. If A is a maximal trace of an agent P then `Cons` a A is a maximal trace of `Pre` a P. Any maximal trace of the left or right operands of a choice is a maximal trace of the choice. If the left and right operands of a parallel composition have the same maximal trace, then this will be a maximal trace of the composition. In other words, we are considering a naive form of parallel composition that requires both operands to execute in 'lock-step' performing the same action at each step. Note that an agent may have several maximal traces.

The maximal trace semantics of agents just descibed can be formalized in HOL by a relation

```
MTRACE : agent -> trace -> bool
```

defined inductively by the rules shown in the box below.

```
#let (trules,tind) =                                              1
     let MTRACE = "MTRACE:agent->Trace->bool" in
     new_inductive_definition false `MTRACE_DEF`
     ("^MTRACE P A", [])

     [ [
       % -------------------------------------------- % ],
                     "^MTRACE Nil []"                  ;

       [               "^MTRACE P A"
       % -------------------------------------------- % ],
              "^MTRACE (Pre a P) (CONS a A)"           ;

       [                 "^MTRACE P A"
       % -------------------------------------------- % ],
                    "^MTRACE (Sum P Q) A"              ;

       [                 "^MTRACE Q A"
       % -------------------------------------------- % ],
                    "^MTRACE (Sum P Q) A"              ;

       [      "^MTRACE P A";        "^MTRACE Q A"
       % -------------------------------------------- % ],
                  "^MTRACE (Prod P Q) A"              ];;
```

It is clear that this formal definition for maximal trace matches the informal one given above.

An agent is *terminal* if the empty trace is a maximal trace of the agent. That is, we define in HOL:

$\vdash$ $\forall$`P. TERMINAL P = MTRACE P []`

Using the techniques described above in section 4.3, it is straightforward to write a tactic to check automatically whether a list of labels A is a maximal trace of an agent P. For example, the theorems

$\vdash$ `MTRACE (Sum (Pre a (Pre b Nil)) Nil) []`

and

$\vdash$ `MTRACE (Sum (Pre a (Pre b Nil)) Nil) [a;b]` can be proved by the application of such a tactic. As expected, however, the tactic would fail to prove:

```
"MTRACE (Sum (Pre a (Pre b Nil)) Nil) [a]"
```

since `[a]` is not a maximal trace of `(Sum (Pre a (Pre b Nil)) Nil)`. Note that the proof-search done by this tactic may involve a certain amount of backtracing, since there are two rules for `Sum` in the definition of `MTRACE`.

5.2 The labelled transition semantics

We now define a labelled transition system in which agents are viewed as states. The transition relation `TRANS : agent->label->agent->bool` is defined inductively as the least set closed under the following rules.

```
#let (lrules,lind) =
     let TRANS = "TRANS: agent->label->agent->bool" in
     new_inductive_definition false `TRANS_DEF`
     ("^TRANS G b E",[])

     [ [
       % ------------------------------------ % ],
                "^TRANS (Pre a Q) a Q"         ;


       [           "^TRANS P a P'"
       % ------------------------------------ % ],
               "^TRANS (Sum P Q) a P'"         ;


       [           "^TRANS Q a Q'"
       % ------------------------------------ % ],
               "^TRANS (Sum P Q) a Q'"         ;


       [  "^TRANS P a P'";    "^TRANS Q a Q'"  ;
       % ------------------------------------ % ],
           "^TRANS (Prod P Q) a (Prod P' Q')"  ];;
```

Informally, an a prefix can perform an a transition. A non-deterministic choice performs an a transition if either its left or right operands can do so. If the operands of a composition perform an a transition to reach states P and Q

respectively, then the composition can perform an a transition to the parallel composition of P and Q.

The transitive closure of the one-step transition relation TRANS defined above is the relation TRANSIT defined as follows:

```
#let (Lrules,Lind) =                                                    2
     let TRANSIT = "TRANSIT: agent->trace->agent->bool" in
     new_inductive_definition false `TRANSIT_DEF`
     ("^TRANSIT X L Y",[])

     [[                                          ],
      % ------------------------------------- %
                  "^TRANSIT P [] P"              ;

      [   "TRANS P a Q";    "^TRANSIT Q B P'"   ],
      % ------------------------------------- %
                "^TRANSIT P (CONS a B) P'"      ];;
```

That is, TRANSIT is defined so that

$\vdash$ TRANSIT P [a_1,...,a_n] Q

holds precisely when there is a series of intermediate states $Q_1, \ldots, Q_n$ such that $Q = Q_n$, TRANS $P\ a_1\ Q_1$ and TRANS $Q_{i-1}\ a_i\ Q_i$ for $1 < i \leq n$. In other words, [a_1,...,a_n] is a trace of actions from P to Q. If the agent Q is a terminal agent, then one would expect [a_1,...,a_n] to be a maximal trace of P.

5.3 The relationship between the two semantics

Given the above definitions of a maximal trace semantics and a transition semantics for agents, one can prove in HOL the following result about the relationship between them.

Theorem 5: $\vdash$ ∀P A Q. TRANSIT P A Q ⊃ TERMINAL Q ⊃ MTRACE P A

That is, if A is a trace of the actions from an agent P to a terminal agent Q, then A is a maximal trace of P.

Proof: The proof is a straightforward rule induction on the inductively defined relation TRANSIT.

An obvious corollary of this theorem is the following.

Corollary: `⊢ ∀P A. TRANSIT P A Nil ⊃ MTRACE P A`

Proof: Trivial consequence of Theorem 5.

We also have a result that states the converse of Theorem 5. That is, if `A` is a maximal trace of `P`, then there is a terminal agent `Q` such that `A` is a trace to `Q` in the transition semantics.

Theorem 6: `⊢ ∀P A. MTRACE P A ⊃ ∃Q. TRANSIT P A Q ∧ TERMINAL Q`

Proof: The proof proceeds by rule induction on `MTRACE`, with an embedded structural induction to solve the subgoal generated by the rule for parallel composition.

5.4 Notions of equivalence

We conclude this section on the process algebra by indicating how one can define notions of equivalence based on the labelled transition semantics and maximal trace semantics defined above.

Maximal trace equivalence. One can define the maximal trace equivalence of two agents P and Q as follows:

```
⊢ MTE P Q = ∀A. MTRACE P A = MTRACE Q A
```

That is, two agents are regarded as being equivalent precisely when they have the same maximal traces.

Bisimulation Equivalence. A *simulation* between agents is a binary relation `S:agent->agent->bool` with the property `SIM` defined by:

```
⊢ SIM S =
    ∀P Q. S P Q ⊃
          ∀a P'. TRANS P a P' ⊃ ∃Q'. TRANS Q a Q' ∧ S P' Q'
```

That is, an agent `Q` simulates another agent `P` if `Q` can do any action that `P` can do and thereby evolve into an agent that can continue to simulate the agent resulting from the action of `P`. Two agents are said to be *bisimilar* if there is a simulation `S` that relates them and whose inverse is also a simulation. That is, we define the equivalence:

```
⊢ BISIMILAR P Q = ∃S. SIM S ∧ S P Q ∧ SIM (λx y. S y x)
```

Having defined these notions of equivalence in HOL, one may then proceed to develop algebraic theories for these equivalence relations. The details of such a development, however, are beyond the scope of this paper.

6 Conclusion

This paper has described a newly-automated mechanism for defining relations inductively in HOL. The theorem-proving tools which have been implemented in HOL based on this mechanism make it simple to define such relations, to use the rules defining them as an interpreter, and to prove properties about them by rule induction—in much the same way as is done on paper. The case studies presented in this paper were chosen to illustrate these points. The HOL sources for all these case studies are available to interested users.

References

[1] P. Aczel, 'An introduction to inductive definitions', in *Handbook of Mathematical Logic*, edited by J. Barwise (North-Holland, 1977), pp. 739–782.

[2] H. Curry and R. Feys, *Combinatory Logic*, vol. 1 (North-Holland, 1958).

[3] M. J. C. Gordon, *Programming Language Theory and its Implementation: Applicative and imperative paradigms*, Prentice Hall International Series in Computer Science (Prentice Hall, 1988).

[4] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics* (Wiley, 1990).

[5] J. R. Hindley and J. P. Seldin, *Introduction to Combinators and λ-Calculus*, London Mathematical Society Student Texts 1 (Cambridge University Press, 1986).

[6] T. F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic', in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P. A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

[7] T. F. Melham, 'A Mechanized Theory of the π-calculus in HOL', Report No. 244, Computer Laboratory, University of Cambridge (January 1992).

[8] R. Milner, *Communication and Concurrency*, Prentice Hall International Series in Computer Science (Prentice Hall, 1989).

[9] G. D. Plotkin, 'A Structural Approach to Operational Semantics', Report No. DAIMI FN–19, Computer Science Department, Aarhus University (September 1981).

[10] University of Cambridge Computer Laboratory, *The HOL System: DESCRIPTION*, revised edition (July 1991).

[11] G. Winskel, 'Introduction to the Formal Semantics of Programming Languages', unpublished lecture notes, University of Cambridge Computer Laboratory (October 1985).