

Combinatory Logic

Michael Norrish

Abstract

A formalisation of combinatory logic, building principally on a development done by Tom Melham. The complete script for the development is available as `clScript.sml` in the `examples/ind_def` directory of the distribution. It is self-contained and so includes the answers to the exercises set at the end of this document.

1 Introduction

This small case study is a formalisation of combinatory logic. This logic is of foundational importance in theoretical computer science, and has a very rich theory. We aim to put HOL through its paces more than we intend to explore what is such a big field. Nonetheless, we hope that as the formalisation proceeds, the nature of the object of study is reasonably clear.

2 The Type of Combinators

The first thing we need to do is define the type of *combinators*. There are just two of these, K and S, but we also need to be able to *combine* them, and for this we need to introduce the notion of application. For lack of a better ASCII symbol, we will use the hash (#) to represent this in the logic:

```
- Hol_datatype 'cl = K | S | # of cl => cl';  
> val it = () : unit
```

We also want the # to be an infix, so we set its fixity to be a tight left-associative infix:

```
- set_fixity ("#", Infixl 1100);  
> val it = () : unit
```

2

Finally, there's one last piece of book-keeping to be done for our new type. The datatype package defines the constructors in theorems of their own, and the name of the theorem stored to disk is the same as the name of the constructor. SML doesn't allow # to be an identifier so we must change the name of the theorem. We do this with the function `set_MLname`. The first parameter to the function is the old name, and the second is the new name.

```
- set_MLname "#" "HASH";  
> val it = () : unit
```

3

3 Combinator Reductions

Combinatory logic is the study of how values of this type can evolve given various rules describing how they change. Therefore, our next step is to define the reductions that combinators can undergo. There are two basic rules:

$$\begin{aligned} K x y &\rightarrow x \\ S f g x &\rightarrow (fx)(gx) \end{aligned}$$

Here, in our description outside of HOL, we use juxtaposition instead of the #. Further, juxtaposition is also left-associative, so that $K x y$ should be read as $K \# x \# y$ which is in turn $(K \# x) \# y$.

Given a term in the logic, we want these reductions to be able to fire at any point, not just at the top level, so we need two further congruence rules:

$$\frac{x \rightarrow x'}{x y \rightarrow x' y}$$
$$\frac{y \rightarrow y'}{x y \rightarrow x y'}$$

In HOL, we can capture this relation with an inductive definition. First we set our arrow symbol up as an infix to make everything that bit prettier:

```
- set_fixity ("-->", Infix(NONASSOC, 510));  
> val it = () : unit
```

4

(By choosing to make our arrow symbol non-associative, we make it a parse error to write $x \rightarrow y \rightarrow z$. It would be nice to be able to write this and have it mean $x \rightarrow y \wedge y \rightarrow z$, but this is not presently possible with the HOL parser.)

Our next step is to actually define the relation. The function for doing this returns three separate theorems, so we bind each separately:

```

val (redn_rules, redn_ind, redn_cases) = 5
  IndDefLib.Hol_reln
  '(!x y f. x --> y ==> f # x --> f # y) /\
   (!f g x. f --> g ==> f # x --> g # x) /\
   (!x y. K # x # y --> x) /\
   (!f g x. S # f # g # x --> (f # x) # (g # x))';
> val redn_rules =
  |- (!x y f. x --> y ==> f # x --> f # y) /\
    (!f g x. f --> g ==> f # x --> g # x) /\
    (!x y. K # x # y --> x) /\
    !f g x. S # f # g # x --> f # x # (g # x) : thm
val redn_ind =
  |- !-->'.
    (!x y f. -->' x y ==> -->' (f # x) (f # y)) /\
    (!f g x. -->' f g ==> -->' (f # x) (g # x)) /\
    (!x y. -->' (K # x # y) x) /\
    (!f g x. -->' (S # f # g # x) (f # x # (g # x))) ==>
    !a0 a1. a0 --> a1 ==> -->' a0 a1 : thm
val redn_cases =
  |- !a0 a1.
    a0 --> a1 =
    (?x y f. (a0 = f # x) /\ (a1 = f # y) /\ x --> y) \/\
    (?f g x. (a0 = f # x) /\ (a1 = g # x) /\ f --> g) \/\
    (?y. a0 = K # a1 # y) \/\
    ?f g x. (a0 = S # f # g # x) /\ (a1 = f # x # (g # x))
  : thm

```

The induction theorem `redn_ind` looks a little strange because the induction predicate is given the name `-->'`. We can change the name to make things prettier with the function `RENAME_VARS_CONV`, a conversion:

```

- val redn_ind = CONV_RULE (RENAME_VARS_CONV ["P"]) redn_ind; 6
> val redn_ind =
  |- !P.
    (!x y f. P x y ==> P (f # x) (f # y)) /\
    (!f g x. P f g ==> P (f # x) (g # x)) /\
    (!x y. P (K # x # y) x) /\
    (!f g x. P (S # f # g # x) (f # x # (g # x))) ==>
    !a0 a1. a0 --> a1 ==> P a0 a1 : thm

```

We also need to set the ML name for the constant `-->`.¹

```

- set_MLname "-->" "redn"; 7
> val it = () : unit

```

Now, using our theorem `redn_rules` we can demonstrate single steps of our reduction relation:

```

- PROVE [redn_rules] ‘‘S # (K # x # x) --> S # x’’; 8
Meson search level: ...
> val it = |- S # (K # x # x) --> S # x : thm

```

The system we have just defined is as powerful as the λ -calculus, Turing machines, and all the other standard models of computation.

One useful result about the combinatory logic is that it is *confluent*. Consider the term $S z (K K) (K y x)$. It can make two reductions, to $S z (K K) y$ and also to $(z (K y x)) (K K (K y x))$. Do these two choices of reduction mean that from this point on the terms have two completely separate histories? Roughly speaking, to be confluent means that the answer to this question is *no*.

4 Transitive Closure and Confluence

A notion crucial to that of confluence is that of *transitive closure*. We have defined a system that evolves by specifying how an algebraic value can evolve into possible successor values in one step. The natural next question is to ask for a characterisation of evolution over one or more steps of the \rightarrow relation.

¹Normally, `-->` would be a fine name for an ML identifier, but the problem here is that when the theory is compiled, the identifier `-->` is already declared as an infix.

In fact, we will define a relation that holds between two values if the second can be reached from the first in zero or more steps. This is the *reflexive, transitive closure* of our original relation. However, rather than tie our new definition to our original relation, we will develop this notion independently and prove a variety of results that are true of any system, not just our system of combinatory logic.

So, we begin our abstract digression with another inductive definition. Our new constant is RTC, such that $RTC\ R\ x\ y$ is true if it is possible to get from x to y with zero or more “steps” of the R relation. (The standard notation for $RTC\ R$ is R^* .) We can express this idea with just two rules. The first

$$\frac{}{RTC\ R\ x\ x}$$

says that it’s always possible to get from x to x in zero or more steps. The second

$$\frac{R\ x\ y \quad RTC\ R\ y\ z}{RTC\ R\ x\ z}$$

says that if you can take a single step from x to y , and then take zero or more steps to get y to z , then it’s possible to take zero or more steps to get between x and z . The realisation of these rules in HOL is again straightforward:

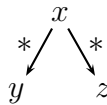
<pre> val (RTC_rules, RTC_ind, RTC_cases) = IndDefLib.Hol_reln ' (!x. RTC R x x) /\ (!x y z. R x y /\ RTC R y z ==> RTC R x z)'; <<HOL message: inventing new type variable names: 'a>> > val RTC_rules = - !R. (!x. RTC R x x) /\ !x y z. R x y /\ RTC R y z ==> RTC R x z : thm val RTC_ind = - !R RTC'. (!x. RTC' x x) /\ (!x y z. R x y /\ RTC' y z ==> RTC' x z) ==> !a0 a1. RTC R a0 a1 ==> RTC' a0 a1 : thm val RTC_cases = - !R a0 a1. RTC R a0 a1 = (a1 = a0) \/\ ?y. R a0 y /\ RTC R y a1 : thm </pre>	9
--	---

Now let us go back to the notion of confluence. We want this to mean something like: “though a system may take different paths in the short-

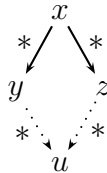
term, those two paths can always end up in the same place”. This suggests that we define confluent thus:

<pre>- val confluent_def = Define 'confluent R = !x y z. RTC R x y /\ RTC R x z ==> ?u. RTC R y u /\ RTC R z u';</pre>	10
---	----

This property states of R that we can “complete the diamond”; if we have



then there must be a u such that



One nice property of confluent relations is that from any one starting point they produce no more than one *normal form*, where a normal form is a value from which no further steps can be taken.

<pre>- val normform_def = Define 'normform R x = !y. ~ (R x y)'; <<HOL message: inventing new type variable names: 'a, 'b>> Definition has been stored under "normform_def". > val normform_def = - !R x. normform R x = !y. ~ R x y : thm</pre>	11
--	----

In other words, a system has an R -normal form at x if there are no connections via R to any other values. (We could have written $\sim?y. R x y$ as our RHS for the definition above.)

We can now prove the following:

```

- g '!R. confluent R ==>
    !x y z.
      RTC R x y /\ normform R y /\
      RTC R x z /\ normform R z ==> (y = z)';
<<HOL message: inventing new type variable names: 'a>>
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
    !R.
      confluent R ==>
      !x y z.
        RTC R x y /\ normform R y /\
        RTC R x z /\ normform R z ==> (y = z)

```

We rewrite with the definition of confluence:

```

- e (RW_TAC std_ss [confluent_def]);
OK..
1 subgoal:
> val it =
  y = z
-----
  0. !x y z. RTC R x y /\ RTC R x z ==>
      ?u. RTC R y u /\ RTC R z u
  1. RTC R x y
  2. normform R y
  3. RTC R x z
  4. normform R z

```

Our confluence property is now assumption 0, and we can use it to infer that there is a u at the base of the diamond:

```

- e ('?u. RTC R y u /\ RTC R z u' by PROVE_TAC []);
OK..
Meson search level: .....
1 subgoal:
> val it =
  y = z
-----
0. !x y z. RTC R x y /\ RTC R x z ==>
    ?u. RTC R y u /\ RTC R z u
1. RTC R x y
2. normform R y
3. RTC R x z
4. normform R z
5. RTC R y u
6. RTC R z u

```

So, from y we can take zero or more steps to get to u and similarly from z . But, we also know that we're at an R -normal form at both y and z . We can't take any steps at all from these values. We can conclude both that $u = y$ and $u = z$, and this in turn means that $y = z$, which is our goal. So we can finish with

```

- e (PROVE_TAC [normform_def, RTC_cases]);
OK..
Meson search level: .....

Goal proved. [...]
> val it =
  Initial goal proved.
  |- !R.
    confluent R ==>
    !x y z.
      RTC R x y /\ normform R y /\
      RTC R x z /\ normform R z ==> (y = z)

```

Packaged up so as to remove the sub-goal package commands, we can prove and save the theorem for future use by:

```

val confluent_normforms_unique = store_thm(
  "confluent_normforms_unique",
  ``!R. confluent R ==>
    !x y z. RTC R x y /\ normform R y /\
      RTC R x z /\ normform R z ==> (y = z)`` ,
  RW_TAC std_ss [confluent_def] THEN
  `?u. RTC R y u /\ RTC R z u' by PROVE_TAC [] THEN
  PROVE_TAC [normform_def, RTC_cases]);

```

...◇...

Clearly confluence is a nice property for a system to have. The question is how we might manage to prove it. Let's start by defining the diamond property that we used in the definition of confluence.

```

- val diamond_def = Define
  'diamond R = !x y z. R x y /\ R x z ==>
    ?u. R y u /\ R z u';
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "diamond_def".
> val diamond_def =
  |- !R.
    diamond R = !x y z. R x y /\ R x z ==>
      ?u. R y u /\ R z u
: thm

```

Now we clearly have that confluence of a relation is equivalent to the reflexive, transitive closure of that relation having the diamond property.

```

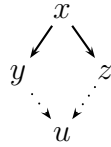
val confluent_diamond_RTC = store_thm(
  "confluent_diamond_RTC",
  ``!R. confluent R = diamond (RTC R)`` ,
  RW_TAC std_ss [confluent_def, diamond_def]);

```

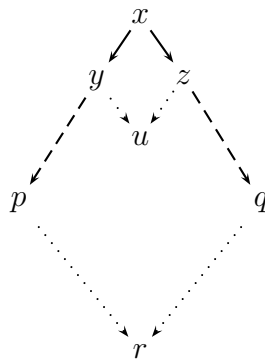
So far so good. How then do we show the diamond property for RTC R ? The answer that leaps to mind is to hope that if the original relation has the diamond property, then maybe the reflexive and transitive closure will too. The theorem we want is

$$\text{diamond } R \supset \text{diamond } (\text{RTC } R)$$

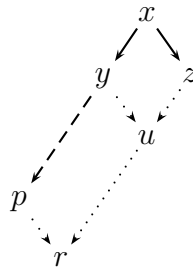
Graphically, this is hoping that from



we will be able to conclude



where the dashed lines indicate that these steps (from x to p , for example) are using RTC R . The presence of two instances of RTC R is an indication that this proof will require two inductions. With the first we will prove



In other words, we want to show that if we take one step in one direction (to z) and many steps in another (to p), then the diamond property for R will guarantee us the existence of r , to which will we be able to take many steps from both p and z .

We take some care to state the goal so that after stripping away the outermost assumption (that R has the diamond property), it will match the induction principle for RTC.²

²In this and subsequent proofs using the sub-goal package, we will present the proof

```

- g '!R. diamond R ==>
      !x p. RTC R x p ==>
          !z. R x z ==>
              ?u. RTC R p u /\ RTC R z u';
<<HOL message: inventing new type variable names: 'a>>
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
    !R.
      diamond R ==>
        !x p. RTC R x p ==> !z. R x z ==>
            ?u. RTC R p u /\ RTC R z u

```

First, we strip away the diamond property assumption (two things need to be stripped: the outermost universal quantifier and the antecedent of the implication):

```

- e (GEN_TAC THEN STRIP_TAC);
OK..
1 subgoal:
> val it =
  !x p. RTC R x p ==> !z. R x z ==> ?u. RTC R p u /\ RTC R z u
  -----
  diamond R

```

Now we can use the induction principle. We use the higher-order backward chaining rule, `HO_MATCH_MP_TAC`, which takes a theorem of the form $\vdash P \supset Q$, tries to instantiate it to make it $\vdash P' \supset Q'$, such that Q' is the same as the goal to be proved, and then requires the user to prove P' .

manager as if the goal to be proved is the first ever on this stack. In other words, we have done a `dropn 1`; after every successful proof to remove the evidence of the old goal. In practice, there is no harm in leaving these goals on the proof manager's stack.

```

- e (HO_MATCH_MP_TAC RTC_ind);
OK..
1 subgoal:
> val it =
  (!x z. R x z ==> ?u. RTC R x u /\ RTC R z u) /\
  !x y z.
  R x y /\ (!z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u) ==>
  !z'. R x z' ==> ?u. RTC R z u /\ RTC R z' u
-----
diamond R

```

Let's strip the goal as much as possible with the aim of making what remains to be proved easier to see:

```

- e (REPEAT STRIP_TAC);
OK..
2 subgoals:
> val it =
  ?u. RTC R z u /\ RTC R z' u
-----
0. diamond R
1. R x y
2. !z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u
3. R x z'

?u. RTC R x u /\ RTC R z u
-----
0. diamond R
1. R x z

```

This first goal is easy. It corresponds to the case where the many steps from x to p are actually no steps at all, and p and x are actually the same place. In the other direction, x has taken one step to z , and we need to find somewhere reachable in zero or more steps from both x and z . Given what we know so far, the only candidate is z itself. In fact, we don't even need to provide this witness explicitly. PROVE_TAC will find it for us, as long as we tell it what the rules governing RTC are:

```

- e (PROVE_TAC [RTC_rules]);
OK..
Meson search level: .....

Goal proved. [...]
Remaining subgoals:
> val it =
  ?u. RTC R z u /\ RTC R z' u
-----
  0. diamond R
  1. R x y
  2. !z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u
  3. R x z'

```

And what of this remaining goal? Assumptions one and three between them are the top of an R -diamond. Let's use the fact that we have the diamond property for R and infer that there exists a v to which y and z' can both take single steps:

```

- e ('?v. R y v /\ R z' v' by PROVE_TAC [diamond_def]);
OK..
Meson search level: .....
1 subgoal:
> val it =
  ?u. RTC R z u /\ RTC R z' u
-----
  0. diamond R
  1. R x y
  2. !z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u
  3. R x z'
  4. R y v
  5. R z' v

```

Now we can apply our induction hypothesis (assumption 2) to complete the long, lop-sided strip of the diamond. We will conclude that there is a u such that $RTC R z u$ and $RTC R v u$. We actually need a u such that $RTC R z' u$, but because there is a single R -step between z' and v we have that as well. All we need to provide PROVE_TAC is the rules for RTC:

```

- e (PROVE_TAC [RTC_rules]);
OK..
Meson search level: .....

Goal proved. [...]
> val it =
  Initial goal proved.
  |- !R.
      diamond R ==> !x p. RTC R x p ==>
                    !z. R x z ==> ?u. RTC R p u /\ RTC R z u

```

Again we can (and should) package up the lemma, avoiding the sub-goal package commands:

```

val R_RTC_diamond = store_thm(
  "R_RTC_diamond",
  ‘‘!R. diamond R ==>
    !x p. RTC R x p ==>
      !z. R x z ==>
        ?u. RTC R p u /\ RTC R z u‘‘,
  GEN_TAC THEN STRIP_TAC THEN HO_MATCH_MP_TAC RTC_ind THEN
  REPEAT STRIP_TAC THENL [
    PROVE_TAC [RTC_rules],
    ‘?v. R y v /\ R z’ v’ by PROVE_TAC [diamond_def] THEN
    PROVE_TAC [RTC_rules]
  ]);

```

...◇...

Now we can move on to proving that if R has the diamond property, so too does $RTC\ R$. We want to prove this by induction again. It's very tempting to state the goal as the obvious

$$\text{diamond } R \supset \text{diamond } (RTC\ R)$$

but doing so will actually make it harder to apply the induction principle when the time is right. Better to start out with a statement of the goal that is very near in form to the induction principle. So, we manually expand the meaning of diamond and state our next goal thus:

```

- g '!R. diamond R ==> !x y. RTC R x y ==>
      !z. RTC R x z ==>
      ?u. RTC R y u /\ RTC R z u';
<<HOL message: inventing new type variable names: 'a>>
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
    !R.
      diamond R ==>
      !x y. RTC R x y ==> !z. RTC R x z ==>
      ?u. RTC R y u /\ RTC R z u

```

Again we strip the diamond property assumption, apply the induction principle, and strip repeatedly:

```

- e (GEN_TAC THEN STRIP_TAC THEN HO_MATCH_MP_TAC RTC_ind THEN
      REPEAT STRIP_TAC);
OK..
2 subgoals:
> val it =
  ?u. RTC R z u /\ RTC R z' u
  -----
  0. diamond R
  1. R x y
  2. !z'. RTC R y z' ==> ?u. RTC R z u /\ RTC R z' u
  3. RTC R x z'

  ?u. RTC R x u /\ RTC R z u
  -----
  0. diamond R
  1. RTC R x z

```

The first goal is again an easy one, corresponding to the case where the trip from x to y has been one of no steps whatsoever.

```

- e (PROVE_TAC [RTC_rules]);
OK..
Meson search level: ...

Goal proved. [...]

Remaining subgoals:
> val it =
  ?u. RTC R z u /\ RTC R z' u
-----
  0. diamond R
  1. R x y
  2. !z'. RTC R y z' ==> ?u. RTC R z u /\ RTC R z' u
  3. RTC R x z'

```

This goal is very similar to the one we saw earlier. We have the top of a (“lop-sided”) diamond in assumptions 1 and 3, so we can infer the existence of a common destination for y and z' :

```

- e ('?v. RTC R y v /\ RTC R z' v'
      by PROVE_TAC [R_RTC_diamond]);
OK..
Meson search level: .....
1 subgoal:
> val it =
  ?u. RTC R z u /\ RTC R z' u
-----
  0. diamond R
  1. R x y
  2. !z'. RTC R y z' ==> ?u. RTC R z u /\ RTC R z' u
  3. RTC R x z'
  4. RTC R y v
  5. RTC R z' v

```

At this point in the last proof we were able to finish it all off by just appealing to the rules for RTC. This time it is not quite so straightforward. When we use the induction hypothesis (assumption 2), we can conclude that there is a u to which both z and v can connect in zero or more steps, but in order to show that this u is reachable from z' , we need to be able to conclude $RTC R z' u$ when we know that $RTC R z' v$ (assumption 5 above) and $RTC R v u$ (our consequence of the inductive hypothesis). We leave the proof of this general result as an exercise, and here assume that it is already proved as the theorem `RTC_RTC`.

```

- e (PROVE_TAC [RTC_rules, RTC_RTC]);
Meson search level: .....

Goal proved. [...]
> val it =
  Initial goal proved.
  |- !R.
      diamond R ==>
        !x y. RTC R x y ==> !z. RTC R x z ==>
          ?u. RTC R y u /\ RTC R z u

```

We can package this result up as a lemma and then prove the prettier version directly:

```

val diamond_RTC_lemma = prove(
  ‘‘!R.
    diamond R ==>
      !x y. RTC R x y ==> !z. RTC R x z ==>
        ?u. RTC R y u /\ RTC R z u‘‘,
  GEN_TAC THEN STRIP_TAC THEN HO_MATCH_MP_TAC RTC_ind THEN
  REPEAT STRIP_TAC THENL [
    PROVE_TAC [RTC_rules],
    ‘?v. RTC R y v /\ RTC R z’ v‘
    by PROVE_TAC [R_RTC_diamond] THEN
    PROVE_TAC [RTC_RTC, RTC_rules]
  ]);
val diamond_RTC = store_thm(
  "diamond_RTC",
  ‘‘!R. diamond R ==> diamond (RTC R)‘‘,
  PROVE_TAC [diamond_def, diamond_RTC_lemma]);

```

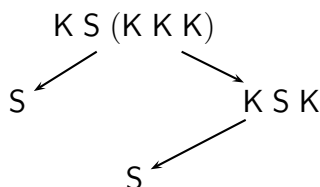
5 Return to the Land of the Combinators

Now, we are in a position to return to the real object of study and prove confluence for combinatory logic. We have done an abstract development and established that

$$\begin{aligned}
 \text{diamond } R &\supset \text{diamond}(\text{RTC } R) \\
 &\quad \wedge \\
 &\text{diamond}(\text{RTC } R) \equiv \text{confluent } R
 \end{aligned}$$

(We have also established a couple of other useful results along the way.)

Sadly, it just isn't the case that \rightarrow , our one-step relation for combinators, has the diamond property. A counter-example is $K S (K K K)$. Its possible evolution can be described graphically:



If we had the diamond property, it should be possible to find a common destination for $K S K$ and S . However, S doesn't admit any reductions whatsoever, so there isn't a common destination.³

This is a problem. We are going to have to take another approach. We will define another reduction strategy (*parallel reduction*), and prove that its reflexive, transitive closure is actually the same relation as our original's reflexive and transitive closure. Then we will also show that parallel reduction has the diamond property. This will establish that its reflexive, transitive closure has it too. Then, because they are the same relation, we will have that the reflexive, transitive closure of our original relation has the diamond property, and therefore, our original relation will be confluent.

5.1 Parallel Reduction

Our new relation allows for any number of reductions to occur in parallel. We use the $-||->$ symbol to indicate parallel reduction because of its own parallel lines:

```

- set_fixity("-||->", Infix(NONASSOC, 510))
> val it = () : unit
33

```

Then we can define parallel reduction itself. The rules look very similar to those for \rightarrow . The difference is that we allow the reflexive transition, and say that an application of $x u$ can be transformed to $y v$ if there are transformations taking x to y and u to v . This is why we must have reflexivity incidentally. Without it, a term like $(K x y) K$ couldn't reduce because while the LHS of the application $(K x y)$ can reduce, its RHS (K) can't.

³In fact our counter-example is more complicated than necessary. The fact that $K S K$ has a reduction to the normal form S also acts as a counter-example. Can you see why?

```

- val (predn_rules, predn_ind, predn_cases) = 34
  IndDefLib.Hol_reln
  '(!x. x -||-> x) /\
   (!x y u v. x -||-> y /\ u -||-> v
    ==>
     x # u -||-> y # v) /\
   (!x y. K # x # y -||-> x) /\
   (!f g x. S # f # g # x -||-> (f # x) # (g # x))';
> val predn_rules =
  |- (!x. x -||-> x) /\
   (!x y u v. x -||-> y /\ u -||-> v ==> x # u -||-> y # v) /\
   (!x y. K # x # y -||-> x) /\
   !f g x. S # f # g # x -||-> f # x # (g # x) : thm
val predn_ind =
  |- !-||->'.
   (!x. -||->' x x) /\
   (!x y u v. -||->' x y /\ -||->' u v ==>
    -||->' (x # u) (y # v)) /\
   (!x y. -||->' (K # x # y) x) /\
   (!f g x. -||->' (S # f # g # x) (f # x # (g # x))) ==>
   !a0 a1. a0 -||-> a1 ==> -||->' a0 a1 : thm
val predn_cases =
  |- !a0 a1.
   a0 -||-> a1 =
   (a1 = a0) \\/
   (?x y u v. (a0 = x # u) /\ (a1 = y # v) /\
    x -||-> y /\ u -||-> v) \\/
   (?y. a0 = K # a1 # y) \\/
   ?f g x. (a0 = S # f # g # x) /\ (a1 = f # x # (g # x))
: thm

```

We again have an induction principle that looks bizarre because of the choice of variable name, so we rename the bound variables.

```

- val predn_ind = 35
  CONV_RULE (RENAME_VARS_CONV ["P"]) predn_ind;
> val predn_ind =
  |- !P.
   (!x. P x x) /\
   (!x y u v. P x y /\ P u v ==> P (x # u) (y # v)) /\
   (!x y. P (K # x # y) x) /\
   (!f g x. P (S # f # g # x) (f # x # (g # x))) ==>
   !a0 a1. a0 -||-> a1 ==> P a0 a1 : thm

```

5.2 Using RTC

Now we can define the reflexive and transitive closures of our two relations. We will use ASCII symbols for both that consist of the original symbol followed by an asterisk. Note also how, in defining the two relations, we have to use the \$ character to “escape” the symbols’ usual fixities. This is exactly analogous to the way in which MLs `op keyword` is used. Finally, because we are defining a constant whose name is symbolic, we have to use `xDefine` rather than `Define`. This is because the latter function likes to try and guess an appropriate name for the definitions that it stores to disk. With symbolic names it doesn’t know how to do this. The first parameter to `xDefine` is an alpha-numeric “stem” which provides the name to use.

```
- set_fixity("-->*", Infix(NONASSOC, 510));  
> val it = () : unit  
  
- val RTCredn_def = xDefine "RTCredn" '$-->* = RTC $-->';  
Definition has been stored under "RTCredn_def".  
> val RTCredn_def = |- $-->* = RTC $--> : thm
```

We do exactly the same thing for the reflexive and transitive closure of our parallel reduction.

```
- set_fixity("-||->*", Infix(NONASSOC, 510));  
> val it = () : unit  
  
- val RTCpredn_def = xDefine "RTCpredn" '$-||->* = RTC $-||->';  
Definition has been stored under "RTCpredn_def".  
> val RTCpredn_def = |- $-||->* = RTC $-||-> : thm
```

Finally, before doing some real proof, let’s generate specialised versions of the RTC theorems for our new constants. This is a straightforward process; we just specialise the R in those theorems with `-->` and `-||->` and then rewrite with the two defining equations above in the RHS-LHS orientation. This will replace instances of $RTC\ R$ with our new constants.

```

- val RTCredn_rules = 38
  REWRITE_RULE [SYM RTCredn_def] (Q.ISPEC '$-->' RTC_rules)
val RTCredn_ind =
  REWRITE_RULE [SYM RTCredn_def] (Q.ISPEC '$-->' RTC_ind)
val RTCpredn_rules =
  REWRITE_RULE [SYM RTCpredn_def] (Q.ISPEC '$-||->' RTC_rules)
val RTCpredn_ind =
  REWRITE_RULE [SYM RTCpredn_def] (Q.ISPEC '$-||->' RTC_ind);
> val RTCredn_rules =
  |- (!x. x -->* x) /\
    !x y z. x --> y /\ y -->* z ==> x -->* z : thm
val RTCredn_ind =
  |- !RTC'.
    (!x. RTC' x x) /\
    (!x y z. x --> y /\ RTC' y z ==> RTC' x z) ==>
    !a0 a1. a0 -->* a1 ==> RTC' a0 a1 : thm
val RTCpredn_rules =
  |- (!x. x -||->* x) /\
    !x y z. x -||-> y /\ y -||->* z ==> x -||->* z : thm
val RTCpredn_ind =
  |- !RTC'.
    (!x. RTC' x x) /\
    (!x y z. x -||-> y /\ RTC' y z ==> RTC' x z) ==>
    !a0 a1. a0 -||->* a1 ==> RTC' a0 a1 : thm

```

Incidentally, in conjunction with PROVE we can now automatically demonstrate relatively long chains of reductions:

```

- PROVE [RTCredn_rules, redn_rules] 'S # K # K # x -->* x'; 39
Meson search level: .....
> val it = |- S # K # K # x -->* x : thm

- PROVE [RTCredn_rules, redn_rules]
  'S # (S # (K # S) # K) # (S # K # K) # f # x -->*
  f # (f # x)';
Meson search level: .....
> val it = |- S # (S # (K # S) # K) # (S # K # K) # f # x -->*
  f # (f # x) : thm

```

(The latter sequence is seven reductions long.)

5.3 Proving the RTCs are the same

We start with the easier direction, and show that everything in $\text{RTC} \rightarrow$ is in $\text{RTC} \dashv\vdash$. Because RTC is monotone (which fact is left to the reader to prove), we can reduce this to showing that $x \rightarrow y \supset x \dashv\vdash y$.

Our goal:

```
- g '!x y. x -->* y ==> x -||->* y'; 40
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
      !x y. x -->* y ==> x -||->* y
```

Now we rewrite with the definitions of our two symbols to expose the fact that they are reflexive, transitive closures:

```
- e (SIMP_TAC std_ss [RTCredn_def, RTCpredn_def]); 41
OK..
1 subgoal:
> val it =
  !x y. RTC $--> x y ==> RTC $-||-> x y
```

We back-chain using our monotonicity result:

```
- e (HO_MATCH_MP_TAC RTC_monotone); 42
OK..
1 subgoal:
> val it =
  !x y. x --> y ==> x -||-> y
```

Now we can induct over the rules for \rightarrow :

```
- e (HO_MATCH_MP_TAC redn_ind); 43
OK..
1 subgoal:
> val it =
  (!x y f. x -||-> y ==> f # x -||-> f # y) /\
  (!f g x. f -||-> g ==> f # x -||-> g # x) /\
  (!x y. K # x # y -||-> x) /\
  !f g x. S # f # g # x -||-> f # x # (g # x)
```

We could split the 4-way conjunction apart into four goals, but there is no real need. It is quite clear that each follows immediately from the rules for parallel reduction.

```

- e (PROVE_TAC [predn_rules]);
OK..
Meson search level: .....

Goal proved. [...]
> val it =
  Initial goal proved.
  |- !x y. x -->* y ==> x -||->* y : goalstack

```

Packaged into a tidy little sub-goal-package-free parcel, our proof is

```

val RTCredn_RTCpredn = store_thm(
  "RTCredn_RTCpredn",
  ‘!x y. x -->* y ==> x -||->* y‘,
  SIMP_TAC std_ss [RTCredn_def, RTCpredn_def] THEN
  HO_MATCH_MP_TAC RTC_monotone THEN
  HO_MATCH_MP_TAC redn_ind THEN
  PROVE_TAC [predn_rules]);

```

...◇...

Our next proof is in the other direction. It should be clear that we will not just be able to appeal to the monotonicity of RTC this time; one step of the parallel reduction relation can not be mirrored with one step of the original reduction relation. It's clear that mirroring one step of the parallel reduction relation might take many steps of the original relation. Let's prove that then:

```

- g ‘!x y. x -||-> y ==> x -->* y‘;
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
    !x y. x -||-> y ==> x -->* y

```

This time our induction will be over the rules defining the parallel reduction relation.

```

- e (HO_MATCH_MP_TAC predn_ind);
OK..
1 subgoal:
> val it =
  (!x. x -->* x) /\
  (!x y u v. x -->* y /\ u -->* v ==> x # u -->* y # v) /\
  (!x y. K # x # y -->* x) /\
  !f g x. S # f # g # x -->* f # x # (g # x)

```

There are four conjuncts here, and it should be clear that all but the second can be proved immediately by appeal to the rules for the transitive closure and for \rightarrow itself. We could split apart the conjunctions and enter a THENL branch. However, we'd need to repeat the same tactic three times to quickly close three of the four branches. Instead, we use the TRY tactical to try applying the same tactic to all four branches. If our tactic fails on branch #2, as we expect, TRY will protect us against this failure and let us proceed.

```

e (REPEAT CONJ_TAC THEN
  TRY (PROVE_TAC [RTCredn_rules, redn_rules]));
OK..
Meson search level: ....
Meson search level: ....
Meson search level: .....
Meson search level: ..
1 subgoal:
> val it =
  !x y u v. x -->* y /\ u -->* v ==> x # u -->* y # v

```

Note that wrapping TRY around PROVE_TAC is not always wise. It can often take PROVE_TAC an extremely long time to exhaust its search space, and then give up with a failure. Here, "we got lucky".

Anyway, what of this latest sub-goal? If we look at it for long enough, we should see that it is another monotonicity fact. In this form, it's not quite right for easy proof. Let's go away and prove RTCredn_ap_monotonic separately. (Another exercise!) Our new theorem should state

```

val RTCredn_ap_monotonic = store_thm(
  "RTCredn_ap_monotonic",
  '(!x y. x -->* y ==> !z. x # z -->* y # z /\ z # x -->* z # y',
  ...);

```

Now that we have this, our sub-goal is almost immediately provable. Using it, we know that

$$\begin{aligned}x u &\rightarrow^* y u \\y u &\rightarrow^* y v\end{aligned}$$

All we need to do is “stitch together” the two transitions above and go from $x u$ to $y v$. We can do this by appealing to our earlier `RTC_RTC` result and reminding `PROVE_TAC` that \rightarrow^* is really just `RTC` \rightarrow .

```
e (PROVE_TAC [RTCredn_def, RTC_RTC, RTCredn_ap_monotonic]); 50
OK..
Meson search level: .....

Goal proved. [...]
> val it =
  Initial goal proved.
  |- !x y. x -||-> y ==> x -->* y : goalstack
```

Odds are that you found that this last step took noticeably longer than previous invocations of `PROVE_TAC`. This is because of the equality in the theorem `RTCredn_def`. (Equality reasoning always slows `PROVE_TAC` down.) Better performance is possible if you instead prove an appropriately specialised version of `RTC_RTC` and use this in place of both `RTC_RTC` and `RTCredn_def`. Let’s go back and do this.

```
- b(); 51
> val it =
  !x y u v. x -->* y /\ u -->* v ==> x # u -->* y # v
```

We need our specialised version of `RTC_RTC`.

```
- val RTCredn_RTCredn = save_thm( 52
  "RTCredn_RTCredn",
  SIMP_RULE std_ss [SYM RTCredn_def] (Q.ISPEC '$-->' RTC_RTC));
> val RTCredn_RTCredn =
  |- !x y z. x -->* y /\ y -->* z ==> x -->* z : thm
```

Now we can finish with:

```

- e (PROVE_TAC [RTCredn_RTCredn, RTCredn_ap_monotonic]) 53
OK..
Meson search level: .....

Goal proved.[...]
> val it =
  Initial goal proved.
  |- !x y. x -||-> y ==> x -->* y : goalstack

```

But given that we can finish off what we thought was an awkward branch with just another application of PROVE_TAC, we don't need to use our fancy TRY-footwork at the stage before. Instead, we can just merge the theorem lists passed to both invocations, dispense with the REPEAT CONJ_TAC and have a very short tactic proof indeed:

```

val predn_RTCredn = store_thm( 54
  "predn_RTCredn",
  ‘‘!x y. x -||-> y ==> x -->* y‘‘,
  HO_MATCH_MP_TAC predn_ind THEN
  PROVE_TAC [RTCredn_rules, redn_rules, RTCredn_RTCredn,
    RTCredn_ap_monotonic]);

```

...◇...

Now it's time to prove that if a number of parallel reduction steps are chained together, then we can mirror this with some number of steps using the original reduction relation. Our goal:

```

- g ‘!x y. x -||->* y ==> x -->* y‘; 55
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
    !x y. x -||->* y ==> x -->* y

```

We use the appropriate induction principle to get to:

```

- e (HO_MATCH_MP_TAC RTCpredn_ind); 56
OK..
1 subgoal:
> val it =
  (!x. x -->* x) /\ !x y z. x -||-> y /\ y -->* z ==> x -->* z

```

This we can finish off in one step. The first conjunct is obvious, and in the second the $x \dashv\vdash \rightarrow y$ and our last result combine to tell us that $x \dashv\vdash \rightarrow^* y$. Then this can be chained together with the other assumption in the second conjunct and we're done.

```

- e (PROVE_TAC [RTCredn_rules, predn_RTCredn,
                RTCredn_RTCredn]);
OK..
Meson search level: .....

Goal proved.[...]
> val it =
    Initial goal proved.
    |- !x y. x -||->* y ==> x -->* y : goalstack

```

Packaged up, this proof is:

```

val RTCpredn_RTCredn = store_thm(
  "RTCpredn_RTCredn",
  ' '!x y. x -||->* y ==> x -->* y' ,
  HO_MATCH_MP_TAC RTCpredn_ind THEN
  PROVE_TAC [predn_RTCredn, RTCredn_RTCredn, RTCredn_rules]);

```

...◇...

Our final act is to use what we have so far to conclude that \rightarrow^* and $\dashv\vdash \rightarrow^*$ are equal. We state our goal:

```

- g '$-||->* = $-->*';
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
        Initial goal:
        $-||->* = $-->*

```

We want to now appeal to extensionality. This is best done with the conversion `FUN_EQ_CONV`, thus:

```

- e (CONV_TAC FUN_EQ_CONV);
OK..
1 subgoal:
> val it =
    !c. $-||->* c = $-->* c

```

This is progress but both “arrows” need another argument. We repeat ourselves (getting rid of extra universal quantifiers along the way):

```

- e (GEN_TAC THEN CONV_TAC FUN_EQ_CONV THEN GEN_TAC);
OK..
1 subgoal:
> val it =
    c -||->* c' = c -->* c'

```

(You might be wondering why it is our variables are suddenly c and c' . This is because they are of type `:c1`, and the code that chooses the name thinks that it's reasonable to use variables named after the type.)

This goal is an easy consequence of our two earlier implications.

```

- e (PROVE_TAC [RTCpredn_RTCredn, RTCredn_RTCpredn]);
OK..
Meson search level: .....

Goal proved. [...]
> val it =
    Initial goal proved.
    |- $-||->* = $-->* : goalstack

```

Packaged, the proof is:

```

val RTCpredn_EQ_RTCredn = store_thm(
  "RTCpredn_EQ_RTCredn",
  '$-||->* = $-->*',
  CONV_TAC FUN_EQ_CONV THEN GEN_TAC THEN
  CONV_TAC FUN_EQ_CONV THEN GEN_TAC THEN
  PROVE_TAC [RTCpredn_RTCredn, RTCredn_RTCpredn]);

```

5.4 Proving a diamond property for parallel reduction

Now we just have one substantial proof to go. Before we can even begin, there are a number of minor lemmas we will need to prove first. These are basically specialisations of the theorem `predn_cases`. We want exhaustive characterisations of the possibilities when the following terms undergo a parallel reduction: $x y$, K , S , $K x$, $S x$, $K x y$, $S x y$ and $S x y z$.

To do this, we will also need the datatype theorems for our `c1` type that tell us that the constructors are distinct and injective:

```

- val cl_11 = theorem "cl_11";
> val cl_11 =
  |- !a0 a1 a0' a1'. (a0 # a1 = a0' # a1') =
                      (a0 = a0') /\ (a1 = a1') : thm

- val cl_distinct0 = theorem "cl_distinct";
> val cl_distinct0 =
  |- ~(S = K) /\ (!a1 a0. ~(S = a0 # a1)) /\
                !a1 a0. ~(K = a0 # a1) : thm

```

We make the latter slightly more applicable by conjoining it with a copy of itself with the equalities reversed:

```

- val cl_distinct =
  CONJ cl_distinct0 (ONCE_REWRITE_RULE [EQ_SYM_EQ] cl_distinct0);
> val cl_distinct =
  |- (~(S = K) /\ (!a1 a0. ~(S = a0 # a1)) /\
      !a1 a0. ~(K = a0 # a1)) /\
      ~(K = S) /\ (!a1 a0. ~(a0 # a1 = S)) /\
      !a1 a0. ~(a0 # a1 = K) : thm

```

Now we can write a little function that derives characterisations automatically:

```

- fun characterise t =
  SIMP_RULE std_ss [cl_11,cl_distinct] (SPEC t predn_cases);
> val characterise = fn : term -> thm

```

For example,

```

- val K_predn = characterise "K";
<<HOL message:
  more than one resolution of overloading was possible>>
> val K_predn = |- !a1. K -||-> a1 = (a1 = K) : thm

- val S_predn = characterise "S";
<<HOL message:
  more than one resolution of overloading was possible>>
> val S_predn = |- !a1. S -||-> a1 = (a1 = S) : thm

```

Unfortunately, what we get back from other inputs is not so good:

```

- val Sx_predn0 = characterise ‘‘S # x‘‘;
> val Sx_predn0 =
  |- !a1.
    S # x -||-> a1 =
    (a1 = S # x) \\/
    ?y v. (a1 = y # v) /\ S -||-> y /\ x -||-> v : thm

```

That first disjunct is redundant, as the following demonstrates:

```

val Sx_predn = prove(
  ‘‘!x y. S # x -||-> y = ?z. (y = S # z) /\ (x -||-> z)‘‘,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [Sx_predn0, predn_rules, S_predn]);

```

Our characterise function will just have to help us in the proofs that follow.

```

val Kx_predn = prove(
  ‘‘!x y. K # x -||-> y = ?z. (y = K # z) /\ (x -||-> z)‘‘,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ‘‘K # x‘‘, predn_rules, K_predn]);

```

What of $K x y$? A little thought demonstrates that there really must be two cases this time.

```

val Kxy_predn = prove(
  ‘‘!x y z.
    K # x # y -||-> z =
    (?u v. (z = K # u # v) /\ (x -||-> u) /\ (y -||-> v)) \\/
    (z = x)‘‘,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ‘‘K # x # y‘‘, predn_rules,
    Kx_predn]);

```

By way of contrast, there is only one case for $S x y$ because it is not yet a “redex” at the top-level.

```

val Sxy_predn = prove(
  ‘‘!x y z. S # x # y -||-> z =
    ?u v. (z = S # u # v) /\ (x -||-> u) /\ (y -||-> v)‘‘,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ‘‘S # x # y‘‘, predn_rules,
    Sx_predn]);

```

Next, the characterisation for $S\ x\ y\ z$:

```

val Sxyz_predn = prove(
  “!w x y z. S # w # x # y -||-> z =
    (?p q r. (z = S # p # q # r) /\
      w -||-> p /\ x -||-> q /\ y -||-> r) \/
    (z = (w # y) # (x # y))”,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ‘‘S # w # x # y‘‘, predn_rules,
    Sxy_predn]);

```

Last of all, we want a characterisation for $x\ y$. What characterise gives us this time can't be improved upon, for all that we might look upon the four disjunctions and despair.

```

- val x_ap_y_predn = characterise ‘‘x # y‘‘;
> val x_ap_y_predn =
  |- !a1.
    x # y -||-> a1 =
      (a1 = x # y) \/
      (?y' v. (a1 = y' # v) /\ x -||-> y' /\ y -||-> v) \/
      (x = K # a1) \/
      ?f g. (x = S # f # g) /\ (a1 = f # y # (g # y)) : thm

```

Our last preliminary before we begin is to derive what is known as the *strong induction principle* for the inductive relation defining $-||->$. This gives us an induction principle where the application case changes from

$$!x\ y\ u\ v. P\ x\ y\ /\ P\ u\ v\ ==>\ P\ (x\ \#\ u)\ (y\ \#\ v)$$

where we can only assume $P\ x\ y$ and $P\ u\ v$ in trying to prove the application case, to the often more useful:

$$!x\ y\ u\ v.
 x\ -||->\ y\ /\ P\ x\ y\ /\ u\ -||->\ v\ /\ P\ u\ v\ ==>
 P\ (x\ \#\ u)\ (y\ \#\ v)$$

Deriving strong induction can be done automatically by the function `derive_strong_induction` found in the `IndDefRules` module. It takes a pair of a list of theorems and another theorem. The list of theorems consists of the rules of the relation split up into individual conjuncts, and the second argument is the normal induction principle.

Thus:

```

val predn_strong_ind =
  IndDefRules.derive_strong_induction (CONJUNCTS predn_rules,
                                       predn_ind);
> val predn_strong_ind =
  |- !P.
    (!x. P x x) /\
    (!x y u v.
      x -||-> y /\ P x y /\ u -||-> v /\ P u v ==>
      P (x # u) (y # v)) /\
    (!x y. P (K # x # y) x) /\
    (!f g x. P (S # f # g # x) (f # x # (g # x))) ==>
    !a0 a1. a0 -||-> a1 ==> P a0 a1 : thm

```

...◇...

Now we are ready to prove the final goal. It is

```

- g ' !x y. x -||-> y ==>
      !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u';
> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
    Initial goal:
    !x y. x -||-> y ==> !z. x -||-> z ==>
      ?u. y -||-> u /\ z -||-> u

```

We now apply the strong induction principle and split the goal into its individual conjuncts:

```

- e (HO_MATCH_MP_TAC predn_strong_ind THEN REPEAT CONJ_TAC); 77
OK..
4 subgoals:
> val it =
  !f g x z. S # f # g # x -||-> z ==>
    ?u. f # x # (g # x) -||-> u /\ z -||-> u

  !x y z. K # x # y -||-> z ==> ?u. x -||-> u /\ z -||-> u

  !x y u v.
    x -||-> y /\
    (!z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u) /\
    u -||-> v /\
    (!z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u) ==>
    !z. x # u -||-> z ==> ?u. y # v -||-> u /\ z -||-> u

  !x z. x -||-> z ==> ?u. x -||-> u /\ z -||-> u

```

The first goal is easily disposed of. The witness we would provide for this case is simply z , but `PROVE_TAC` will do the work for us:

```

- e (PROVE_TAC [predn_rules]); 78
OK..
Meson search level: ...

Goal proved. [...]

```

The next goal includes two instances of terms of the form $x \# y -||-> z$. We can use our `x_ap_y_predn` theorem here. However, if we rewrite indiscriminately with it, we will really confuse the goal. We want to rewrite just the assumption, not the instance underneath the existential quantifier. Starting everything by repeatedly stripping can't lead us too far astray.

<pre> - e (REPEAT STRIP_TAC); OK.. 1 subgoal: > val it = ?u. y # v - -> u /\ z - -> u ----- 0. x - -> y 1. !z. x - -> z ==> ?u. y - -> u /\ z - -> u 2. u - -> v 3. !z. u - -> z ==> ?u. v - -> u /\ z - -> u 4. x # u - -> z </pre>	79
--	----

We need to split up assumption 4. We can get it out of the assumption list using the `Q.PAT_ASSUM` theorem-tactical. We will write

```

Q.PAT_ASSUM 'x # y -||-> z'
(STRIIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn])

```

The quotation specifies the pattern that we want to match. The second argument specifies how we are going to transform the theorem. Reading the compositions from right to left, first we will simplify with the `x_ap_y_predn` theorem and then we will assume the result back into the assumptions, stripping disjunctions and existentials as we go.

We already know that doing this is going to produce four new subgoals (there were four disjuncts in the `x_ap_y_predn` theorem). At least one of these should be trivial because it will correspond to the case when the parallel reduction is just a “do nothing” step. Let’s try eliminating the simple cases with a “speculative” call to `PROVE_TAC` wrapped inside a `TRY`. And before doing that, we should do some rewriting to make sure that equalities in the assumptions are eliminated.

So:

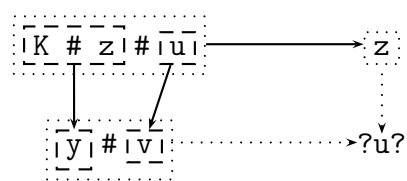
```

- e (Q.PAT_ASSUM 'x # y -||-> z'
      (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn]) THEN
      RW_TAC std_ss [] THEN
      TRY (PROVE_TAC [predn_rules]));
OK..
Meson search level: .....
Meson search level: .....
Meson search level: .....
Meson search level: .....
2 subgoals:
> val it =
  ?u'. y # v -||-> u' /\ f # u # (g # u) -||-> u'
-----
  0. S # f # g -||-> y
  1. !z. S # f # g -||-> z ==> ?u. y -||-> u /\ z -||-> u
  2. u -||-> v
  3. !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u

  ?u. y # v -||-> u /\ z -||-> u
-----
  0. K # z -||-> y
  1. !z'. K # z -||-> z' ==> ?u. y -||-> u /\ z' -||-> u
  2. u -||-> v
  3. !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u

```

Brilliant! We've eliminated two of the four disjuncts already. Now our next goal features a term $K \# z -||-> y$ in the assumptions. We have a theorem that pertains to just this situation. But before applying it willy-nilly, let us try to figure out exactly what the situation is. A diagram of the current situation might look like



Our theorem tells us that y must actually be of the form $K \# w$ for some w , and that there must be an arrow between z and w . Thus:

```

- e ('?w. (y = K # w) /\ (z -||-> w)' by PROVE_TAC [Kx_predn]); 81
OK..
Meson search level: .....
1 subgoal:
> val it =
  ?u. y # v -||-> u /\ z -||-> u
-----
0. K # z -||-> y
1. !z'. K # z -||-> z' ==> ?u. y -||-> u /\ z' -||-> u
2. u -||-> v
3. !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u
4. y = K # w
5. z -||-> w

```

On inspection, it becomes clear that the u must be w . The first conjunct requires $K \# w \# v -||-> w$, which we have because this is what K s do, and the second conjunct is already in the assumption list. Rewriting (eliminating that equality in the assumption list first will make `PROVE_TAC`'s job that much easier), and then first order reasoning will solve this goal:

```

- e (RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]); 82
OK..
Meson search level: ...

Goal proved. [...]
Remaining subgoals:
> val it =
  ?u'. y # v -||-> u' /\ f # u # (g # u) -||-> u'
-----
0. S # f # g -||-> y
1. !z. S # f # g -||-> z ==> ?u. y -||-> u /\ z -||-> u
2. u -||-> v
3. !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u

```

This case involving S is analogous. Here's the tactic to apply:

```

- e ('?p q. (y = S # p # q) /\ (f -||-> p) /\ (g -||-> q)' 83
      by PROVE_TAC [Sxy_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]);
OK..
Meson search level: .....
Meson search level: .....

Goal proved.[...]
Remaining subgoals:
> val it =
      !f g x z. S # f # g # x -||-> z ==>
          ?u. f # x # (g # x) -||-> u /\ z -||-> u

      !x y z. K # x # y -||-> z ==> ?u. x -||-> u /\ z -||-> u

```

This next goal features a $K \# x \# y -||-> z$ term that we have a theorem for already. And again, let's speculatively use a call to `PROVE_TAC` to eliminate the simple cases immediately (`Kxy_predn` is a disjunct so we'll get two sub-goals if we don't eliminate anything).

```

- e (RW_TAC std_ss [Kxy_predn] THEN 84
      TRY (PROVE_TAC [predn_rules]));
OK..
Meson search level: ..
Meson search level: ...

Goal proved. [...]
Remaining subgoals:
> val it =
      !f g x z. S # f # g # x -||-> z ==>
          ?u. f # x # (g # x) -||-> u /\ z -||-> u

```

Better yet! We got both cases immediately, and have moved onto the last case. We can try the same strategy.

```

- e (RW_TAC std_ss [Sxyz_predn] THEN PROVE_TAC [predn_rules]); 85
OK..
Meson search level: ..
Meson search level: .....

Goal proved.[...]
> val it =
  Initial goal proved.
  |- !x y. x -||-> y ==> !z. x -||-> z ==>
    ?u. y -||-> u /\ z -||-> u : goalstack

```

The final goal proof can be packaged into:

```

val predn_diamond_lemma = prove( 86
  ‘!x y. x -||-> y ==>
    !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u‘,
  HO_MATCH_MP_TAC predn_strong_ind THEN REPEAT CONJ_TAC THENL [
    PROVE_TAC [predn_rules],
    REPEAT STRIP_TAC THEN
    Q.PAT_ASSUM ‘x # y -||-> z‘
    (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn]) THEN
    RW_TAC std_ss [] THEN
    TRY (PROVE_TAC [predn_rules]) THENL [
      ‘?w. (y = K # w) /\ (z -||-> w)‘ by PROVE_TAC [Kx_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules],
      ‘?p q. (y = S # p # q) /\ (f -||-> p) /\ (g -||-> q)‘ by
      PROVE_TAC [Sxy_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]
    ],
    RW_TAC std_ss [Kxy_predn] THEN PROVE_TAC [predn_rules],
    RW_TAC std_ss [Sxyz_predn] THEN PROVE_TAC [predn_rules]
  ]);

```

...◇...

We are on the home straight. The lemma can be turned into a statement involving the diamond constant directly:

```

val predn_diamond = store_thm( 87
  "predn_diamond",
  ‘diamond $-||->‘,
  PROVE_TAC [diamond_def, predn_diamond_lemma]);

```

And now we can prove that our original relation is confluent in similar fashion:

<pre>val confluent_redn = store_thm("confluent_redn", 'confluent \$-->' , PROVE_TAC [predn_diamond, RTCpredn_def, RTCredn_def, confluent_diamond_RTC, RTCpredn_EQ_RTCredn, diamond_RTC]);</pre>	88
---	----

Exercises

1. Prove that

$$RTC\ R\ x\ y \ \wedge \ RTC\ R\ y\ z \ \supset \ RTC\ R\ x\ z$$

You will need to prove the goal by induction, and will probably need to massage it slightly first to get it to match the appropriate induction principle. Store the theorem under the name `RTC_RTC`.

2. Another induction. Show that

$$(\forall x\ y. R_1\ x\ y \supset R_2\ x\ y) \supset (\forall x\ y. RTC\ R_1\ x\ y \supset RTC\ R_2\ x\ y)$$

Call the resulting theorem `RTC_monotone`.

3. Yet another RTC induction, but where R is no longer abstract, and is instead the original reduction relation. Prove

$$x \rightarrow^* y \ \supset \ \forall z. x\ z \rightarrow^* y\ z \ \wedge \ z\ x \rightarrow^* z\ y$$

Call it `RTCredn_ap_monotonic`.

4. Come up with a counter-example for the following property:

$$\left(\begin{array}{l} \forall x\ y\ z. R\ x\ y \ \wedge \ R\ x\ z \ \supset \\ \exists u. RTC\ R\ y\ u \ \wedge \ RTC\ R\ z\ u \end{array} \right) \supset \text{diamond}\ (RTC\ R)$$