The Church-Rosser Theorem in Isabelle: A Proof Porting Experiment

Ole Rasmussen*
Computer Laboratory
University of Cambridge

March 1995

Abstract

This paper describes a proof of the Church-Rosser theorem for the pure λ-calculus formalised in the Isabelle theorem prover. The initial version of the proof is ported from a similar proof done in the Coq proof assistant by Gérard Huet, but a number of optimisations have been performed. The development involves the introduction of several inductive and recursive definitions and thus gives a good presentation of the inductive package of Isabelle.

1 Introduction

In this paper we describe a formal development of the residual theory for the pure λ-calculus leading to a proof of the Church-Rosser theorem. The proof originates from Gérard Huet [Hue94] using the Coq proof assistant [D$^+$93]. In this development we construct the same proof but with the Isabelle theorem prover [Pau94b] using the object-logic of Zermelo-Fraenkel set theory (from now on just called ZF). The project illustrates the use of Isabelle's inductive package [Pau94a] on a large proof development, definition of recursive functions, and how the automation tools of Isabelle can be employed effectively to solve major proof steps.

Formalisation of this proof is not new and the first mechanical proof was done in the Boyer-Moore prover by Shankar [Sha88] and has later been described by Huet as mentioned above and in for example [Pfe92]. The Church-Rosser theorem has also been proved for combinatorial logic in for example [CM92] and by Paulson contained in the Isabelle ZF distribution. The proofs for combinatorial logic are however much simpler. The major goal of this project is not to prove that the Church-Rosser theorem is true but rather to see how much work is required to port a fairly large, non-trivial proof development from Coq to Isabelle ZF. Both systems provide tools for formalising inductive definition but are otherwise quite different. Thus the aim is to investigate if the systems

*From April 95: Department of Computer Science, Technical University of Denmark. E-mail: `osr@id.dtu.dk`

have reached such a high level of abstraction that porting developments is easy. Furthermore we want to pursue some of Huet's suggestions in order to optimise the original proof using features of Isabelle ZF.

The following sections are organised as follows. We begin by giving a brief sketch of the method used to prove the Church-Rosser theorem and introduce various notions rather informally. The next section contains a short introduction to Isabelle to render this paper self-contained. Section 4 presents the final version of the Church-Rosser proof in Isabelle and defines all notions formally in Isabelle syntax. In this section we also show a concrete proof of one of the key theorems as performed in Isabelle. The presentation will not cover all theorems proved and only one proof is shown; however, a full transcript of the proof is available by ftp[1]. The final version of the proof was reached in a number of development steps which we account for in Section 5. There we also explain how we employed the automation tools of Isabelle. In Section 6 we provide some statistics and compare them with the proof in Coq. Note that we do not make a direct comparison between Coq and Isabelle since we believe that this will never be fair, as our knowledge of the two systems is not comparable. The best comparison is obtained by viewing this paper together with Huet's original paper or alternatively viewing the respective transcripts. The paper ends with some concluding remarks.

2 Proof Sketch of the Church-Rosser Theorem

In this section we give a brief proof sketch of the Church-Rosser Theorem for β-reduction in the pure λ-calculus, following the method due to Tait and Martin-Löf. The concrete proof method is of course in debt to the one used by Gérard Huet [Hue94]. At some points we will be rather informal but all notions used will be defined formally in Section 4. We expect some familiarity with the λ-calculus, such as the notion of β-reduction and representing λ-terms by de Bruijn indexes [dB72], although they will also be defined and exemplified in Section 4. A thorough description of the lambda calculus and the Tait – Martin-Löf proof of the Church-Rosser theorem can be found in [Bar84].

The Church-Rosser theorem for β-reduction states that if two terms, m and n, are β-convertible then there exists a common reduct u:

$$m \longleftrightarrow_\beta n \;\Rightarrow\; \exists u \cdot m \longrightarrow u \wedge n \longrightarrow u$$

where $\longrightarrow$ is the transitive closure of one-step β-reduction, $\xrightarrow{1}$, i.e. $0, 1$ or many steps of β-reductions. $\longleftrightarrow_\beta$ is similar $0, 1$ or many steps of β-reductions or anti-reductions. To prove the Church-Rosser property we show that $\longrightarrow$ is *confluent*, or satisfies the diamond property, which can be stated as

$$x \longrightarrow y \wedge x \longrightarrow z \;\Rightarrow\; \exists u \cdot y \longrightarrow u \wedge z \longrightarrow u$$

The two properties are shown in Figure 1(a) and 1(b) respectively. It should be obvious from the figure that the Church-Rosser property follows directly from

[1] ftp address: `ftp.id.dtu.dk:/pub/osr/`

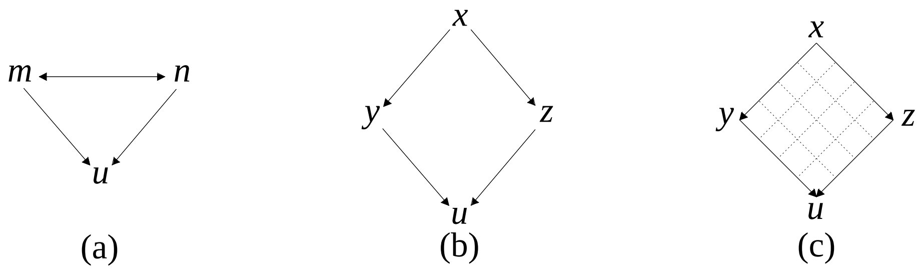

Figure 1: The Church-Rosser and diamond property for β-reduction.

the diamond lemma since conversion is defined in terms of β-reductions and anti-reductions.

A general result is that, if a relation is confluent then so is its transitive closure. The diagram in Figure 1(c) suggests a proof of this theorem: The dotted lines denote the relation and the outer lines its transitive closure. In our case this means that it suffices to show that $\xrightarrow{1}$ is confluent. This is unfortunately not true as can be understood from the following counter-example taken from [Bar84]:

> Let $R \xrightarrow{1} R'$; then the following term can be reduced in two ways:
>
> $$(\lambda x.xx)R \xrightarrow{1} RR \quad \text{or} \quad (\lambda x.xx)R \xrightarrow{1} (\lambda x.xx)R'$$
>
> A common reduct would be $R'R'$, but this cannot be reached in one step from RR. Thus $\xrightarrow{1}$ is not confluent.

As suggested by the example above we need a new reduction relation which is less restricted than $\xrightarrow{1}$, but still has $\longrightarrow$ as its transitive closure. We therefore introduce the notion of *parallel reduction*, written $\overset{1}{\Longrightarrow}$, which is similar to $\xrightarrow{1}$ but β-reductions may now take place at several places simultaneously. As $\overset{1}{\Longrightarrow}$ basically is one or more β-reductions, clearly $\longrightarrow$ is its transitive closure. In the mechanical proof we prove confluence of $\longrightarrow$ in four steps as shown in Figure 2. First we prove, using the so called *strip lemma*, that confluence of $\overset{1}{\Longrightarrow}$ implies confluence of $\Longrightarrow$, where $\Longrightarrow$ is defined as zero or more steps of $\overset{1}{\Longrightarrow}$. The last step shown on the figure is an easy consequence of that $\Longrightarrow$ is in fact equal to $\longrightarrow$. The proof of the Church-Rosser theorem is now boiled down to a proof of confluence of one step parallel reduction, known as the *Parallel Moves* lemma.

Before continuing we introduce the notions of redex-positions and residuals which play an essential part in the following. A *redex-position* is a position in a term where β-reduction can take place, i.e. any subterm of the form $((\lambda x \cdot y)\ z)$. The position of the redexes in a term after one step of parallel reduction is called its *residuals*. To prove the parallel moves lemma we need to keep track of what happens during one step of parallel reduction, i.e. we must be able to trace the residuals. Following Huet we introduce a new datatype, `redexes`, as an enrichment of λ-terms where redex positions can be marked.

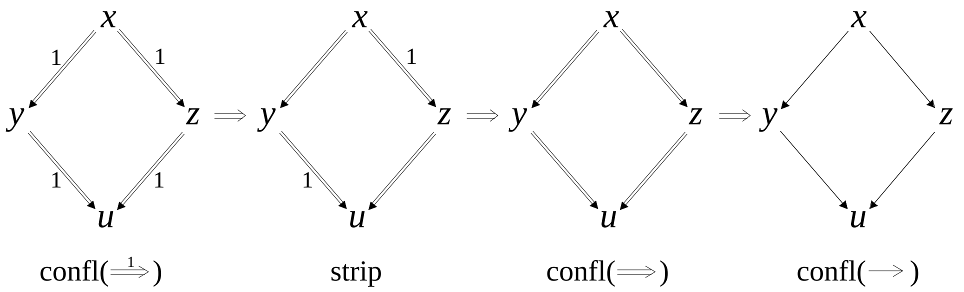

Figure 2: Four steps to prove confluence of β-reduction

The type `redexes` serves *two* purposes since it both represents λ-terms in the proofs and represents a set of redex positions.

We define the *residual function* to simulate parallel reduction, but extended with the ability exactly to state which redex positions are reduced. The residual function is written

$$u \rhd v$$

which returns u reduced by one step of parallel reduction at all the redex positions marked in v. Note that this is a function on u and v as opposed to $\overset{1}{\Longrightarrow}$, which is defined as a relation. This function corresponds to the standard residual map used by Barendregt and to the 3 argument predicate used by Gérard Huet. The function is partial and only defined when the two arguments are *compatible*, i.e. representing the same λ-term structure.

The main result on `redexes` is what Gérard Huet calls the *Prism* theorem, which states that for all compatible sets of redexes u, v and w,

$$v \subseteq u \;\Rightarrow\; w \rhd u = (w \rhd v) \rhd (u \rhd v)$$

Recall that `redexes` both represents λ-terms and sets of redex positions; thus informally the above theorem can be read: If one first reduces both w and u at the redex positions marked in v and then reduces the result, $(w \rhd v)$, with the remaining positions in u, then it equals the term got from reducing w by all the positions marked in u in one step.

As a corollary of the Prism Theorem we first obtain Levy's *Cube* lemma, which states that for all compatible sets of redexes u, v and w,

$$(w \rhd v) \rhd (u \rhd v) = (w \rhd u) \rhd (v \rhd u)$$

which finally leads to the *Paving* lemma needed to prove confluence of one step parallel reduction. For all compatible sets of redexes u, v and w,

$$\exists uv \cdot \exists vu \cdot (w \rhd u) \rhd vu = (w \rhd v) \rhd uv$$

which means that if w is reduced by the redex positions marked in some set u and likewise w is reduced by the redex positions marked in some set v then there always exists further reductions (uv and vu) to make these terms equal.

Description	ASCII notation
Meta typing	`t::a`
Curried function type	`[a1,...,an]=>t`
Curried abstraction	`%x1,...,xn.t`
Curried application	`t(x1,...,xn)`
Meta equality	`P == Q`
Implication	`P ==> Q`
Nested implication	`[|P1;...;Pn|]==> Q`
Nested quantification	`!!x1...xn.P`

Figure 3: Some of the meta level symbols in Isabelle

This is essentially the same property as described by confluence. We prove that the residual function describes the same reduction as one step parallel reduction and thus have a proof of confluence for one step parallel reduction. Referring to Figure 2 we finally have a proof of the Church-Rosser property for β-reduction.

3 Introduction to Isabelle

The purpose of this section is to give a brief introduction to the Isabelle logical framework and the concrete logic used in this work. Isabelle has been described in numerous articles but for a thorough presentation of the system please refer to [Pau94b]. Isabelle is a generic logical framework meant for defining different proof systems. The user must distinguish between two levels of abstraction, the *meta-level* and the *object-level*, where the former is used to define a particular object logic. The basic Isabelle system, called *Pure* Isabelle, defines the meta logic which is a fragment of intuitionistic high-order logic and the meta language which is the simple typed lambda calculus. Inference rules and axioms in Isabelle are all theorems of the meta logic (usually containing a meta implication). Therefore, even when working in a specific object logic, the meta level is always present. The meta level symbols used in this work are described in Figure 3.

Isabelle is implemented in Standard ML (SML) and the proof commands are SML functions changing the current proof state. The major proof method is backward proofs applying *tactics* to the current proof state. The main tactics apply lists of rules to a subgoal using various forms of resolution. The resolution tactics can produce several solutions, since the high-order unification used in the resolution can produce many solutions and several rules may match the subgoal.

Tactics can be composed into new more complex tactics using *tacticals*, which are high-order SML functions. The simple tacticals are used for composing tactics sequentially, alternatively, repeatedly, but also more complex tacticals exist for expressing control structures, depth-first search etc.

Isabelle has a built in generic parsing mechanism allowing a flexible mixfix notation of new defined symbols in the object logic. Furthermore the user may

Description	ASCII notation
Universal quantifier	ALL x.B(x)
Existential quantifier	EX x.B(x)
Implication	A --> B
Set membership	a:A
Subset	A<=B
Pair	<a,b>
Binary product	A * B
Lambda	lam x:A.b(x)
Application	f`a

Figure 4: Notation of selected symbols in ZF

define *translations*, which will only affect the appearance of an object keeping the internal representation unchanged.

3.1 Zermelo-Fraenkel Set Theory

The distribution of Isabelle includes an implementation of Zermelo-Fraenkel set theory built as an extension to classical first-order logic. ZF was chosen as object logic as it offers a great amount of flexibility to express general mathematics and a large number of basic theories already exist. Figure 4 shows a selection of the symbols in ZF used in this work. In ZF all desired typing of variables are stated directly in the goal assumptions. The meta type system is used very little since only two types are defined: `i` for sets and `o` for propositions.

3.2 Generic Packages

Isabelle contains a number a generic packages which can be instantiated to a specific object logic. Three of these have been used extensively in this work; the *simplifier*, the *classical* reasoning package and the *inductive* definition package. These packages are all set up in the standard distribution of ZF and the following sections exemplify their use.

The simplifier performs conditional and unconditional rewriting using contextual information and takes a simplification set as argument. The set contains a number of rewrite rules and possibly the specification of tactics to be applied in specific cases. The simplifier can both rewrite the assumptions and the conclusion.

The classical reasoning package provides a number of tactics to prove theorems in the style of the sequent calculus. They take a rule set as argument, which contains a collection of introduction and elimination rules divided into either safe and unsafe rules. Safe rules can be applied blindly but unsafe rules must be handled more carefully. Two of these tactics has been used extensively in this work:

`step_tac` performs the basic step of the classical proof procedure applying safe rules and one step of unsafe rules. The tactic is for example useful to perform all trivial proof steps in the beginning of a proof.

`fast_tac` applies `step_tac` repeatedly using depth-first search. The tactic fails if the goal cannot be solved completely.

The inductive package is based on a fixedpoint approach and permits the formalisation of all monotone inductive definitions. The inductive definitions are defined as a subset of an existing set given the desired inference rules as Isabelle meta implications and possibly some information about the involved operators. The same package is used to define recursive data structures, here called datatypes, and co-inductive definitions which are not used in this work.

4 The Proof in Isabelle

In the proof we need two representations for λ-terms: the normal de Bruijn notation and the enriched type `redexes` introduced in Section 2. In Isabelle ZF we have no methods to enrich type definitions but easy methods to define subtypes. We therefore start our formal development by defining the largest type `redexes` and prove the major theorems related to it, i.e. the Prism and Parallel Moves theorems. We then define conventional λ-terms as a subtype of `redexes` and continue the proof development by defining the various kinds of reductions finally leading to the Church-Rosser Theorem.

4.1 Redexes

The type `redexes` is defined as an inductive set with three constructors using the Isabelle `datatype` declaration. The constructor `Var(i)` represents a variable bound by the `i`'th abstraction, `Fun(t)` represents a λ-abstraction and `App(b,f,a)` represents an application of `a` to `f`. The boolean variable `b` is used to mark redex positions, so that if `b` is *true* (denoted `1` in ZF) then the node belongs to the set of redex positions and if *false* (denoted `0` in ZF) then it does not. The definition is shown in Figure 5 together with `redex_rec` described below.

```
consts
  redexes     :: "i"
  redex_rec   :: "[i, i=>i, [i,i]=>i, [i,i,i,i,i]=>i]=>i"
datatype
  "redexes" = Var ("n: nat")
            | Fun ("t: redexes")
            | App ("b:bool", "f:redexes", "a:redexes")
  type_intrs "[bool_into_univ]"
defs
  redex_rec_def
  "redex_rec(p,b,c,d) == Vrec(p,%p g.redexes_case(b, %x.c(x,g`x),
                                    %n x y.d(n, x, y, g`x, g`y), p))"
```

Figure 5: Definition of `redexes`

The type `redexes` can be used to represent the abstract syntax of λ-terms in *de Bruijn* notation. The idea is that variables are represented by their relative placement in the term and not by their name and thus eliminates the need for renaming (α-conversions). We give some examples of λ-terms in *de Bruijn* notation and for brevity we have left out the boolean mark in the application nodes:

(1)	$\lambda x \cdot x$	`Fun(Var(0))`
(2)	$\lambda x \lambda y \cdot xy$	`Fun(Fun(App(Var(1),Var(0))))`
(3)	$\lambda x \cdot xy$	`Fun(App(Var(0),Var(1)))`

where (1) shows that the nearest abstraction is numbered 0, (2) shows how more references are constructed, and (3) how free variables (here y) are represented as if an immediately enclosing outer abstraction existed.

A datatype definition in Isabelle returns a number of theorems packed into an SML structure with the same name as the defined type. Three major parts are used in this work:

- `redexes.intrs` contains the introduction rules, i.e. the rules stating that the constructors belong to `redexes`. For example for the `App`-case:

```
[| b:bool; f:redexes; a:redexes|] ==> App(b,f,a):redexes
```

- `redexes.induct` contains the following induction rule permitting structural induction over the defined datatype:

```
[| x:redexes;
   !!n. n:nat ==> P(Var(n));
   !!t.[| t:redexes; P(t) |] ==> P(Fun(t));
   !!a b f.[| b:bool; f:redexes; P(f); a:redexes; P(a)|]==> P(App(b,f,a))
 |] ==> P(x)
```

- Additional rules to allow freeness reasoning e.g. to prove

$$Var(n) \neq Fun(t)$$

Furthermore Figure 5 defines a recursion principle for `redexes` allowing recursive definitions over the three constructors. The definition uses the general recursion operator `Vrec` and is admittedly a bit hard to read but leads to these straight-forward equality rules:

```
redex_rec_Var   redex_rec(Var(n),b,c,d) = b(n)
redex_rec_Fun   redex_rec(Fun(t),b,c,d) = c(t,redex_rec(t,b,c,d))
redex_rec_App   redex_rec(App(m,f,a),b,c,d) =
                     d(m,f,a,redex_rec(f,b,c,d),redex_rec(a,b,c,d))
```

The use of `redex_rec` to define recursive functions is illustrated below.

4.2 Auxiliary definitions on redexes

The type `redexes` viewed as a set of redex positions naturally forms a boolean algebra with a subset relation and a union to join sets. The former is defined as an inductive predicate and the latter as a recursive function. Additionally we inductively define that two sets of `redexes` are compatible and that a set is `regular`.

An inductive definition in Isabelle ZF is constructed by defining a new set as a subset of an existing set.

The definition of the subset relation on `redexes` is shown in Figure 6 and starts with the declaration `inductive` and introduces the set `Ssub` as a subset of pairs of `redexes` as stated in the `domains` part. In order to make the definition and subsequent proofs more readable we define an Isabelle translation enabling us to state the subset relation as a predicate using an infix notation, `a<==b` instead of a set membership notation, `<a,b>:Ssub`. The four inference rules defining the subset relation are given in the `intrs` part and state that `a` is a subset of `b` if they are both variables or are both functions, whose bodies also respect the subset relation. For application nodes there are two cases, either where `a` is unmarked or where they both are marked. In both cases their inner `redexes` must respect the subset relation. The two new symbols `Ssub` and `<==` are defined with the appropriate meta type and notation in the `const` part. Finally the `type_intrs` declaration states appropriate type rules to enable the inductive package to type check the definition. Here the type rules for `redexes` and the boolean constants are supplied.

```
consts
  Ssub       ::   "i"
  "<=="      ::   "[i,i]=>o" (infixl 70)
translations
  "a <== b"     == "<a,b>:Ssub"
inductive
  domains    "Ssub" <= "redexes*redexes"
  intrs
    Sub_Var  "n:nat ==> Var(n) <== Var(n)"
    Sub_Fun  "u<==v ==> Fun(u) <== Fun(v)"
    Sub_App1 "[| u1<==v1; u2<==v2; b:bool |]==>
                        App(0,u1,u2) <== App(b,v1,v2)"
    Sub_App2 "[| u1<==v1; u2<==v2 |]==>
                        App(1,u1,u2) <== App(1,v1,v2)"
  type_intrs "redexes.intrs@bool_typechecks"
```

Figure 6: Inductive definition of the subset relation on `redexes`

Like the datatype definition above; the inductive definition returns an SML structure containing various elements. The most important for `Ssub` are:

- The variable `Ssub.intrs` contains the four defined inference rules.

- An induction rule for the subset relation:

```
[| xa<==x;
   !!n. n:nat ==> P(Var(n),Var(n));
   !!u v. [| u<==v ; P(u, v) |] ==> P(Fun(u),Fun(v));
   !!b u1 u2 v1 v2.[| u1<==v1 ; u2<==v2 ; b:bool;P(u1,v1);P(u2,v2)
                  |] ==> P(App(0,u1,u2),App(b,v1,v2));
    !!u1 u2 v1 v2.[| u1<==v1 ; u2<==v2 ; P(u1,v1); P(u2,v2)
                  |] ==> P(App(1,u1,u2), App(1,v1,v2))
 |] ==> P(xa,x)
```

- A proof function, `Ssub.mk_cases`, can create simplified versions of the general elimination rule given the defined constructors and a specific pattern. For example:

```
- Ssub.mk_cases redexes.con_defs "App(1,f,a) <== v";
[| App(1,f,a) <== v; !!v1 v2.[| f<==v1; a<==v2; v=App(1,v1,v2) |] ==> Q
   |] ==> Q
```

- The variable `Ssub.dom_subset` contains a theorem stating that `Ssub` is a subset of pairs of `redexes`:

```
Ssub <= redexes * redexes
```

In the subsequent inductive definitions we leave out the parts `const` and `type_intrs` to ease the reading. Two `redexes` are compatible, written infix as `~`, if they represent the same λ-term i.e. if we ignore the boolean marks the two terms are equal. An element of `redexes` is `regular` if only redex positions are marked with *true*. We need to state regularity as it is meaningless to mark application nodes which do not represent proper redexes. These two predicates are defined in Figure 7.

The union of two `redexes`, written infix as `un`, is defined as a recursive partial function. It is partial as it is meaningful only for compatible sets of redexes. We first give the definition in SML since definitions of recursive functions are hard to read in Isabelle syntax:

```
fun Var(n) un Var(n) = Var(n)
  | Fun(u) un Fun(v) = Fun(u un v)
  | App(b1,u1,v1) un App(b2,u2,v2) =
                       App(b1 or b2,u1 un u2,v1 un v2)
  | _ un _ = 0
```

where `or` denotes the conventional boolean or-function.

The function is partial in the sense that it returns the empty set, `0`, when not defined. The concrete Isabelle syntax for the definition is shown in Figure 8 using the recursion principle, `redex_rec` defined above for `redexes`.

Although the definition is not easy to follow, it enables us to prove straightforward equality rules for the three distinct cases of compatible redexes. The equalities are similar to the SML definition above and are all proved in one step

```
translations
  "a ~ b"       == "<a,b>:Scomp"
  "regular(a)"  == "a:Sreg"
inductive
  domains    "Scomp" <= "redexes*redexes"
  intrs
    Comp_Var "n:nat ==> Var(n) ~ Var(n)"
    Comp_Fun "u ~ v ==> Fun(u) ~ Fun(v)"
    Comp_App "[|u1~v1; u2~v2 ; b1:bool; b2:bool|]==>
                        App(b1,u1,u2) ~ App(b2,v1,v2)"
inductive
  domains    "Sreg" <= "redexes"
  intrs
    Reg_Var  "n:nat ==> regular(Var(n))"
    Reg_Fun  "regular(u)==> regular(Fun(u))"
    Reg_App1 "[|regular(Fun(u));regular(v)|]==>regular(App(1,Fun(u),v))"
    Reg_App2 "[|regular(u);regular(v)|]==>regular(App(0,u,v))"
```

Figure 7: Inductive definitions of compatible and regular `redexes`

```
defs
  union_def  "u un v == redex_rec(u,
      %i.lam t:redexes.redexes_case(%j.Var(i), %x.0, %b x y.0, t),
      %x m.lam t:redexes.redexes_case(%j.0, %y.Fun(m`y), %b y z.0, t),
      %b x y m p.lam t:redexes.redexes_case(%j.0, %y.0,
                             %c z u. App(b or c, m`z, p`u), t))`v"
```

Figure 8: Definition of union

by the simplifier:

```
union_Var  n:nat==> Var(n) un Var(n) = Var(n)
union_Fun  [|u:redexes;v:redexes|]==> Fun(u) un Fun(v) = Fun(u un v)
union_App  [|b1:bool;b2:bool;u1:redexes;v1:redexes;u2:redexes;
             v2:redexes|]==>  App(b1,u1,v1) un App(b2,u2,v2) =
                              App(b1 or b2,u1 un u2,v1 un v2)
```

Obvious properties such as reflexivity, symmetry and transitivity of compatible `redexes`, are easily proved by induction from the above definitions and is shown in Figure 9.

4.3 Lifting and substitution

In this section we define the notion of substitution. When substituting a term for a variable we have to make sure that the de Bruijn variables still refer to the correct λ-abstraction. Free variables might be captured by surrounding λ-terms and their index should therefore be adjusted accordingly. We may first study some examples to illustrate the substitution process, both in normal mathematical notation and in de Bruijn notation (again ignoring the boolean marks in our representation). The notation `a/b` means that all occurrences of `Var(0)` in the term `b` are replaced by `a`. We first show the redex to be reduced

```
comp_refl    u:redexes ==> u ~ u
comp_sym     u ~ v ==> v ~ u
comp_trans   [| u ~ v; v ~ w|] ==> u ~ w

sub_comp     u <== v  ==> u ~ v
sub_preserve_reg
             u <== v  ==> regular(v) --> regular(u)

union_l      u ~ v ==> u <== (u un v)
union_r      u ~ v ==> v <== (u un v)
union_sym    u ~ v ==> u un v = v un u
union_preserve_regular
             u ~ v ==> regular(u)-->regular(v)-->regular(u un v)
```

Figure 9: Simple properties of compatability, subset and union

and then the substitution to be performed:

- The reference number of a free variable inside an abstraction (here y) is decreased by 1 due to the removal of one λ-abstraction:

  $((\lambda x \cdot xy)\ y) \stackrel{1}{\longrightarrow} xy[y/x] = yy$

  ```
  App(Fun(App(Var(0),Var(1))),Var(0)) -1->
      Var(0) / App(Var(0),Var(1)) = App(Var(0),Var(0))
  ```

- The variable z is lifted by one when instantiated inside the body of a λ-abstraction. y is local and therefore unchanged:

  $((\lambda x \lambda y \cdot xy)\ z) \stackrel{1}{\longrightarrow} \lambda y \cdot xy[z/x] = \lambda y \cdot zy$

  ```
  App(Fun(Fun(App(Var(1),Var(0)))),Var(0)) -1->
      Var(0) / Fun(App(Var(1),Var(0))) = Fun(App(Var(1),Var(0))
  ```

- The variable u is lifted by one when instantiated inside the body of a λ-abstraction. Both y and z are local variables and therefore unchanged:

  $((\lambda x \lambda y \cdot xy)\ (\lambda z \cdot uz)) \stackrel{1}{\longrightarrow} \lambda y \cdot xy[(\lambda z \cdot uz)/x] = \lambda y \cdot (\lambda z \cdot uz)y$

  ```
  App(Fun(Fun(App(Var(1),Var(0)))),Fun(App(Var(1),Var(0)))) -1->
      Fun(App(Var(1),Var(0))) / Fun(App(Var(1),Var(0))) =
      Fun(App(Fun(App(Var(2),Var(0))),Var(0)))
  ```

Having exemplified some of the complications of performing substitution with de Bruijn indexes, we are now ready to define it formally. First we define lifting of free variables in a term by defining the general case of lifting, `lift_rec`, which adds one to all variables in a term greater or equal to a `k`. The function is always called at top level with `k` equal to 0, here called `lift`, and will thus lift all free variables in the term one step. The definition is given in SML syntax:

```
fun lift_rec(Var(i),k)      = if i<k then Var(i) else Var(succ(i))
  | lift_rec(Fun(t),k)      = Fun(lift_rec(t,succ(k)))
  | lift_rec(App(b,t,s),k) = App(b,lift_rec(t,k),lift_rec(s,k))

val lift(t) = lift_rec(t,0)
```

```
translations
  "lift(r)"  == "lift_rec(r,0)"
  "u/v"      == "subst_rec(u,v,0)"
defs
  lift_rec_def  "lift_rec(r,kg) ==
                 redex_rec(r,%i.(lam k:nat.if(i<k,Var(i),Var(succ(i)))),
                             %x m.(lam k:nat.Fun(m`(succ(k)))),
                             %b x y m p.lam k:nat.App(b,m`k,p`k))`kg"
  subst_rec_def "subst_rec(u,t,kg) ==
                 redex_rec(t,%i.(lam r:redexes.lam k:nat.
                                  if(k<i,Var(i#-1),if(k=i,r,Var(i)))),
                             %x m.(lam r:redexes.lam k:nat.
                                  Fun(m`(lift(r))`(succ(k)))),
                             %b x y m p.lam r:redexes.lam k:nat.
                                  App(b,m`r`k,p`r`k))`u`kg"
```

Figure 10: Definition of lifting and substitution

Substitution is likewise first defined in the general case, `subst_rec(u,v,k)` substituting `Var(k)` in `v` with `u`; and the specialised case, where `k` is zero, written in infix notation `u/v`. The `Var(i)`-case consist of three possibilities: if `i` is less than `k` then variable is bound by an outer λ-abstraction and must therefore be subtracted one to compensate for the removed abstraction in the substitution; if `k` equal `i` then it is the variable we want to replace with `u`; and finally if `k` is greater than `i` then the variable is not affected by the substitution. In the `Fun`-case we use `k` to count the number of local abstractions and lift `u` by one:

```
fun subst_rec(u,Var(i),k)      = if k<i then Var(i-1) else
                                          if k=i then u else Var(i)
  | subst_rec(u,Fun(t),k)      = Fun(subst_rec(lift(u),t,succ(k)))
  | subst_rec(u,App(b,f,a),k) = App(b,subst_rec(u,f,p),subst_rec(u,a,p))

fun u / v = subst_rec(u,v,0)
```

The actual definition in Isabelle uses `redex_rec` and is given in Figure 10. The two specialisations `lift` and `/` are conveniently defined as Isabelle translations.

These definitions lead to straight-forward equality rules which can be proved in one step by the simplifier. Here is for example the rule for the `Fun`-case of `subst_rec`:

```
[|p:nat;u:redexes|]==>
        subst_rec(u,Fun(t),p) = Fun(subst_rec(lift(u),t,succ(p)))
```

The most important result concerning substitution, which we need, is the `substitution` lemma, which in mathematical notation may be stated as:

$$v[w/x][(u[w/x])/y] = v[u/y][w/x]$$

expressing how two successive substitutions may be swapped.

Formulated with our definition of substitution using de Bruijn indexes we get:

```
substitution
 [|v:redexes;u:redexes;w:redexes;n:nat|]==>
   subst_rec(w,u,n) / subst_rec(lift(w),v,succ(n))= subst_rec(w,u/v,n)
```

The application of `lift` and `succ` on the left side of the equality reflects the fact that a λ-abstraction is removed on the right side in the substitution `u/v`. The `substitution` lemma is proved as a specialisation of the more general lemma:

```
subst_rec_subst_rec
 v:redexes ==>
   ALL m:nat.ALL n:nat.ALL u:redexes.ALL w:redexes.m le  n -->
    subst_rec(subst_rec(w,u,n),subst_rec(lift_rec(w,m),v,succ(n)),m)=
    subst_rec(w,subst_rec(u,v,m),n)
```

The proof of this theorem is rather long and tedious and requires a number of intermediate lemmas (see Section 6 for a discussion of these proofs).

Additionally we prove a number of preservation theorems stating that lifting and substitution preserve both regularity and compatibility. As an example we show that substitution preserves regularity which is proved by induction over `regular` and then in one step by the simplifier using automatic case-splitting:

```
subst_rec_preserve_reg  regular(v) ==> ALL m:nat.ALL u:redexes.
                          regular(u) --> regular(subst_rec(u,v,m))
```

4.4 Residuals

To define the residual function we first define a three argument predicate, `residuals(u,v,w)`, where `w` is `u` reduced by one step of parallel reduction at all the redex positions marked in `v`. That is, `w` is the residuals of `u` after one step of parallel reduction by the marks in `v`. The inductive definition in Isabelle is shown in Figure 11 and has four inference rules. The two interesting cases are the `app` and the `redex` rules. The former states that if the position is not marked (the boolean variable of `v` is `0`) then nothing happens to this application node, but the inner parts may be reduced. The latter states that if the application node is a marked redex, then we reduce this redex and thus substitute the results of the inner parts.

The `residuals` predicate has a natural formulation as a partial function and we therefore prove that it produces, from any `u` and `v`, a unique `w`:

```
res_is_func  [|residuals(u,v,w); residuals(u,v,w1)|]==> w1 = w
```

which is proved by induction over `residuals`. We use the unique description operator (ι, written `THE` in Isabelle) to define the residual function also shown in Figure 11, where it is written infix as `u|>v`. The Isabelle definition is partial in the sense that `THE` returns the empty set if the predicate is not defined. The residual function is only meaningful if `u` and `v` are compatible sets of redexes and `v` is regular. Under these assumptions we prove the four equality rules corresponding to the inference rules for the predicate and that `u|>v` belongs to

```
translations
  "residuals(u,v,w)"  == "<u,v,w>:Sres"
inductive
  domains       "Sres" <= "redexes*redexes*redexes"
  intrs
    var   "n:nat ==> residuals(Var(n),Var(n),Var(n))"
    fun   "residuals(u,v,w)==> residuals(Fun(u),Fun(v),Fun(w))"
    app   "[|residuals(u1,v1,w1); residuals(u2,v2,w2);b:bool|]==>
               residuals(App(b,u1,u2),App(0,v1,v2),App(b,w1,w2))"
    redex "[|residuals(u1,v1,w1); residuals(u2,v2,w2);b:bool|]==>
               residuals(App(b,Fun(u1),u2),App(1,Fun(v1),v2),w2/w1)"

defs
  res_func_def  "u |> v == THE w.residuals(u,v,w)"
```

Figure 11: Definition of the residual predicate and function

`redexes`:

```
res_Var    n:nat==> Var(n) |> Var(n) = Var(n)
res_Fun    [| u~v; regular(v) |]==> Fun(u) |> Fun(v) = Fun(u|>v)
res_App    [| s~u; regular(u); t~v; regular(v); b:bool |]==>
               App(b,s,t) |> App(0,u,v) = App(b, s|>u, t|>v)
res_redex  [| s~u; regular(u); t~v; regular(v); b:bool |]==>
               App(b,Fun(s),t) |> App(1,Fun(u),v) = (t|>v) / (s|>u)

resfunc_type  u ~ v ==> regular(v) --> u |> v:redexes
```

The equality rules are all proved in one step by `fast_tac` and the type rule in one step by the simplifier using the equality rules.

We may now combine properties of substitution and residuals and prove the `commutation` theorem stating that the residual function commutes with substitution. This theorem is crucial in solving the key step of the prism theorem as shown in the next section.

```
commutation   [| u1~u2; v1~v2; regular(u2); regular(v2) |]==>
                 (v1/u1) |> (v2/u2) = (v1|>v2) / (u1|>u2)
```

It is proved as a specialisation of a more general theorem using `subst_rec` as we did for the substitution lemma above. This general theorem is proved by induction over the compatibility predicate followed by rewriting, where the crucial rules are the `substitution` lemma and the rule expressing that the residual function distributes over `lift_rec`.

Finally we prove that the residual function preserves both compatibility and regularity. The latter is shown below and proved by induction over compatibility of `u` and `v`:

```
residuals_preserve_reg
    u~v ==> regular(u) --> regular(v) --> regular(u|>v)
```

4.5 The prism and cube theorem

As explained in Section 2 the main result on `redexes` is the `prism` theorem formulated in Isabelle syntax as,

```
prism  [| v<==u; regular(u); w~v |]==> w|>u = (w|>v) |> (u|>v)
```

We present the main part of this proof as performed interactively in Isabelle to demonstrate the use of some of the automated proof tools. For clarity we have removed some of the output from the Isabelle session although nothing important is missing. The theorem we prove is a slight reformulation of the prism theorem above where we have used object-logic universal-quantification and implication to enable the induction to instantiate the variables `u` and `v`. Furthermore we have added an extra assumption (`w~u`) which is easily derivable from the other assumptions, but it eases the automation to state it explicitly. We start the proof by entering the theorem and then performing induction over the subset predicate (see page 10) which produces four subgoals. This is shown in Figure 12.

```
goal Residuals.thy
    "!!u.v<==u  ==> regular(u) --> (ALL w.w~v --> w~u -->
                            w|>u = (w|>v) |> (u|>v))";
- by (eresolve_tac [sub_induct] 1);
Level 1
 1. !!u n. n:nat ==>
       regular(Var(n)) -->(ALL w.w~ Var(n) -->w~ Var(n) -->
           w |> Var(n) = w |> Var(n) |> (Var(n) |> Var(n)))
 2. !!u ua v.[| ua<== v;
             regular(v) -->(ALL w.w~ua --> w~v -->
                w |> v = w |> ua |> (v |> ua)) |] ==>
       regular(Fun(v)) --> (ALL w.w~Fun(ua) --> w~Fun(v) -->
           w |> Fun(v) = w |> Fun(ua) |> (Fun(v) |> Fun(ua)))
 3. !!u b u1 u2 v1 v2.[| u1<==v1; u2<==v2; b : bool;
             regular(v1) -->(ALL w.w~u1 --> w~v1 -->
                      w |> v1 = w |> u1 |> (v1 |> u1));
             regular(v2) -->(ALL w.w~u2 --> w~v2 -->
                      w |> v2 = w |> u2 |> (v2 |> u2)) |] ==>
       regular(App(b,v1,v2)) --> (ALL w.w~App(0,u1,u2) -->
                                  w~App(b,v1,v2) -->
           w |> App(b,v1,v2) =
           w |> App(0,u1,u2) |> (App(b,v1,v2) |> App(0,u1,u2)))
 4. !!u u1 u2 v1 v2.[| u1 <==  v1; u2 <==  v2;
             regular(v1) -->(ALL w.w~u1 --> w~v1 -->
                      w |> v1 = w |> u1 |> (v1 |> u1));
             regular(v2) -->(ALL w.w~u2 --> w~v2 -->
                      w |> v2 = w |> u2 |> (v2 |> u2)) |] ==>
       regular(App(1,v1,v2)) --> (ALL w.w~App(1,u1,u2) -->
                                  w~App(1,v1,v2) -->
           w |> App(1,v1,v2) =
           w |> App(1,u1,u2) |> (App(1,v1,v2) |> App(1,u1,u2)))
```

Figure 12: The first proof step

To solve the proof we have constructed a simplification set, `prism_ss`, containing the `commutation` theorem, various preservation lemmas, the equality rules for the residual function etc. Additionally the simplification set will apply elimination rules after each simplification, which in this case are those for compatibility and regularity. The two first subgoals can be solved by the simplifier using

```
- by (asm_simp_tac prism_ss 1);
```

The last two subgoals can be solved by applying the same three tactics to each of them. We show only the proof of the last goal (number 4 above) which is the key step requiring the `commutation` theorem. First we simplify and eliminate as much as possible, using the simplification set which in this step also rewrites with the `commutation` theorem:

```
- by (asm_simp_tac prism_ss 1);
Level 7
 1. !!u u1 u2 v1 v2 w b1 u1a u2a b1a u1b u2b ua ub uc.
       [| uc~ua; regular(ua);regular(v2); u2b~u2; b1a:bool;
          u2<==v2; u2b~v2; ub<==ua; uc~ub;
(*hyp1*)  ALL w.w~u2 --> w~v2 --> w |> v2 = w |> u2 |> (v2 |> u2);
(*hyp2*)  ALL w.w~Fun(ub) --> w~Fun(ua) -->
             w |> Fun(ua) = w |> Fun(ub) |> (Fun(ua) |> Fun(ub))
        |] ==>
       u2b|>u2 |> (v2|>u2) / (uc|>ua) =
       u2b|>u2 |> (v2|>u2) / (uc|>ub |> (ua|>ub))
```

To continue we must use the second induction hypothesis, `hyp2`, where `w` is instantiated by `Fun(uc)`. Instead of manually instantiating `w`, we split the universal-quantification and the two implications into separate subgoals. Unfortunately the constructed tactic also matches the first induction hypothesis which we do not want to use. But by using the `THEN` tactical to compose tactics, we get Isabelle to automatically backtrack to a solution where the next tactic succeeds. As the compatibility introduction rule for the `Fun`-case does not match the first induction hypothesis it will automatically choose the second leading to the following:

```
- by (dresolve_tac [spec RS mp RS mp] 1 THEN
      resolve_tac [Scomp.Comp_Fun] 1 THEN
      resolve_tac [Scomp.Comp_Fun] 2 THEN (REPEAT(assume_tac 1)));
Level 8
 1. !!u u1 u2 v1 v2 w b1 u1a u2a b1a u1b u2b ua ub uc.
       [| uc~ua; regular(ua);
          ALL w.w~u2 --> w~v2 --> w |> v2 = w |> u2 |> (v2 |> u2);
          regular(v2); u2b~u2;b1a:bool; u2<==v2; u2b~v2; ub<==ua; uc~ub;
          Fun(uc) |> Fun(ua) =
          Fun(uc) |> Fun(ub) |> (Fun(ua) |> Fun(ub)) |] ==>
       u2b|>u2 |> (v2|>u2) / (uc|>ua) =
       u2b|>u2 |> (v2|>u2) / (uc|>ub |> (ua|>ub))
```

We now rewrite the induction hypothesis in the assumptions to remove the enclosing `Fun`s and may then use the obtained equality to solve the goal. This is done by the version of the simplifier which also rewrites assumptions:

```
by (asm_full_simp_tac prism_ss 1);
```

This step finishes the proof which is thus completed in 9 steps. Finally we get to the `prism` theorem above proving the extra assumption using the rules `comp_trans` and `sub_comp` in Figure 9.

We get the `cube` theorem as a corollary of the `prism` theorem. However, we first need to prove the `preservation` theorem which states that `u` reduced by the redexes marked in `v` is equal to the union of `u` and `v` reduced by the same redexes. This theorem is obvious since the residual function does not use the marks in its first argument. It is proved by induction over compatibility:

```
preservation   [| u~v; regular(v) |] ==> u|>v = (u un v) |> v
cube           [| u~v; regular(v); regular(u); w~u |]==>
                           (w|>u) |> (v|>u) = (w|>v) |> (u|>v)
```

To prove the `cube` theorem we rewrite the second term on both the right and the left side of the equality (`v|>u` and `u|>v`) once using the `preservation` theorem. We now apply the `prism` theorem on both sides and reach the same term due to the symmetry of union.

The `paving` theorem ends our development on `redexes`. It states that if a term `w` has any two reductions `u` and `v`, there always exist two further reductions reducing them to the same term. As mentioned in the proof sketch above this is essentially the same property as confluence. The theorem is here formulated with additional properties of the two existential variables.

```
paving  [| w~u; w~v; regular(u); regular(v) |]==>
           EX uv.EX vu. (w|>u) |> vu = (w|>v) |> uv &
              (w|>u)~vu & regular(vu) & (w|>v)~uv & regular(uv)
```

It is proved directly by the `cube` theorem and the side conditions from various preservation lemmas such as `residuals_preserve_reg`.

4.6 Lambda terms

We return to our main subject, namely the λ-terms. We now define them inductively as a subtype of `redexes`, where all boolean marks are false i.e. `0`. Furthermore we define a syntactic translation to write application nodes with a false mark as `Apl(n,m)`, hiding the boolean mark, and thus making the term look like conventional λ-term representations. The definition is shown in Figure 13.

To transform `redexes` to `lambda` terms we also in Figure 13 define the recursive function `unmark`. It resets all marks in an element belonging to `redexes` as the following proved equalities show:

```
unmark_Var     unmark(Var(n))      = Var(n)
unmark_Fun     unmark(Fun(t))      = Fun(unmark(t))
unmark_App     unmark(App(b,n,m)) = Apl(unmark(n),unmark(m))
```

Note that we do not need a transformation function in the other direction (i.e. to mark terms) as `lambda`-terms in fact are `redexes`. The `unmark` function will not affect `lambda`-terms and substitution preserves `lambda`-terms which is

```
translations
  "Apl(n,m)" == "App(0,n,m)"
inductive
  domains       "lambda" <= "redexes"
  intrs
    Lambda_Var  "n:nat    ==> Var(n):lambda"
    Lambda_Fun  "u:lambda ==> Fun(u):lambda"
    Lambda_App  "[| u:lambda; v:lambda |]==> Apl(u,v):lambda"
defs
  unmark_def    "unmark(u) == redex_rec(u,%i.Var(i),
                                         %x m.Fun(m),
                                         %b x y m p.Apl(m,p))"
```

Figure 13: Definition of `lambda`-terms

expressed in the following lemmas:

```
lambda_unmark u:lambda ==> unmark(u) = u
substL_type   [|v:lambda; n:nat; u:lambda|]==> subst_rec(u,v,n):lambda
```

They are both proved by induction over `lambda` followed by one step by the simplifier, using the fact that `lift` also preserves `lambda`-terms.

4.7 Reduction on lambda terms

We now axiomatise one step of β-reduction on λ-terms by defining four inference rules. The rule *beta* is where the actual β-reduction takes place and the three others are congruence rules stating that a β-reduction may take place within terms:

$$\frac{}{((\lambda x \cdot m)\ n) \xrightarrow{1} m[n/x]}\ beta \qquad \frac{m \xrightarrow{1} n}{\lambda x \cdot m \xrightarrow{1} \lambda x \cdot n}\ fun$$

$$\frac{m_1 \xrightarrow{1} n_1}{(m_1\ m_2) \xrightarrow{1} (n_1\ m_2)}\ apl\_l \qquad \frac{m_2 \xrightarrow{1} n_2}{(m_1\ m_2) \xrightarrow{1} (m_1\ n_2)}\ apl\_r$$

The inductive definition of β-reduction in Isabelle is shown in Figure 14 together with the definition of its transitive closure. The transitive closure is defined by three inference rules accounting for one step of reduction, no reduction and multi step reductions.

One step of parallel reduction is similarly axiomatised by four inference rules. The rule $beta_p$ defines the β-reduction step, but as opposed to the definition above, further reductions may have taken place in inner terms here defined in a bottom up fashion. Reflexivity is only defined for variables, var_p, but is provable for all terms due to the two last congruence rules:

```
translations
  "a -1-> b" == "<a,b>:Sred1"
  "a ---> b" == "<a,b>:Sred"
inductive
  domains    "Sred1" <= "lambda*lambda"
  intrs
    beta     "[|m:lambda; n:lambda |]==> Apl(Fun(m),n) -1-> n/m"
    rfun     "m-1->n ==> Fun(m) -1-> Fun(n)"
    apl_l    "[|m2:lambda; m1 -1-> n1 |]==> Apl(m1,m2) -1-> Apl(n1,m2)"
    apl_r    "[|m1:lambda; m2 -1-> n2 |]==> Apl(m1,m2) -1-> Apl(m1,n2)"
inductive
  domains    "Sred" <= "lambda*lambda"
  intrs
    one_step "m-1->n ==> m--->n"
    refl     "m:lambda ==> m--->m"
    trans    "[|m--->n; n--->p |]==> m--->p"
```

Figure 14: Definition of β-reduction

$$\frac{m_1 \stackrel{1}{\Longrightarrow} n_1 \quad m_2 \stackrel{1}{\Longrightarrow} n_2}{((\lambda x \cdot m_1)\ m_2) \stackrel{1}{\Longrightarrow} n_1[n_2/x]}\, beta_p \qquad \frac{}{x \stackrel{1}{\Longrightarrow} x}\, var_p$$

$$\frac{m \stackrel{1}{\Longrightarrow} n}{\lambda x \cdot m \stackrel{1}{\Longrightarrow} \lambda x \cdot n}\, fun_p \qquad \frac{m_1 \stackrel{1}{\Longrightarrow} n_1 \quad m_2 \stackrel{1}{\Longrightarrow} n_2}{(m_1\ m_2) \stackrel{1}{\Longrightarrow} (n_1\ n_2)}\, apl_p$$

The inductive Isabelle definition of one step of parallel reduction is shown in Figure 15 together with its transitive closure. For parallel reduction we do not need the reflexive case since this is provable from the one step definition.

```
translations
  "a =1=> b" == "<a,b>:Spar_red1"
  "a ===> b" == "<a,b>:Spar_red"
inductive
  domains    "Spar_red1" <= "lambda*lambda"
  intrs
    betap    "[|m1 =1=> n1; m2 =1=> n2 |]==> Apl(Fun(m1),m2) =1=> n2/n1"
    varp     "n:nat   ==> Var(n) =1=> Var(n)"
    funp     "m=1=>n  ==> Fun(m) =1=> Fun(n)"
    aplp     "[|m1 =1=> n1; m2 =1=> n2 |]==> Apl(m1,m2) =1=> Apl(n1,n2)"
inductive
  domains    "Spar_red" <= "lambda*lambda"
  intrs
    one_step "m=1=>n  ==>  m===>n"
    trans    "[|m===>n; n===>p |]==> m===>p"
```

Figure 15: Definition of parallel reduction

We prove some simple facts about the reductions which were loosely postulated in the proof sketch. We here show three examples, where the first states that $\stackrel{1}{\Longrightarrow}$ is an extension of $\stackrel{1}{\longrightarrow}$, and the two next that parallel reduction and beta reduction are equivalent:

```
red1_par_red1  m -1-> n  ==>  m =1=> n
red_par_red    m ---> n  ==>  m ===> n
par_red_red    m ===> n  ==>  m ---> n
```

These are all proved by induction on the first formula.

The next two lemmas formally relate the residual function to $\stackrel{1}{\Longrightarrow}$. Viewing their respective definitions it should be apparent that they describe the same reductions. Residuals simulates $\stackrel{1}{\Longrightarrow}$, which means that for all $\stackrel{1}{\Longrightarrow}$ reductions there exists a set of redexes performing the same reduction:

```
simulation     m =1=> n ==> EX v.m|>v = n  & m~v & regular(v)
```

We also prove the property the other way around by saying that all reductions performed by `|>` are $\stackrel{1}{\Longrightarrow}$ reductions:

```
completeness   [|u:lambda; u~v; regular(v)|]==> u =1=> unmark(u|>v)
```

`Simulation` is proved by induction over $\stackrel{1}{\Longrightarrow}$; `completeness` is proved by induction over compatibility and by using the fact that the `unmark` function distributes over substitution.

4.8 Confluence and strip lemmas

In Figure 16 we define the confluence and strip predicates introduced in the proof sketch.

```
defs
  confluence_def "confluence(R) ==
                  ALL x y. <x,y>:R --> (ALL z.<x,z>:R -->
                          (EX u.<y,u>:R & <z,u>:R))"
  strip_def      "strip == ALL x y.(x===>y) --> (ALL z.(x=1=>z) -->
                          (EX u.(y=1=>u) & (z===>u)))"
```

Figure 16: Definition of confluence and the strip-lemma

Using the fact that the residual function and $\stackrel{1}{\Longrightarrow}$ describe the same reduction, as expressed by the `simulation` and `completeness` lemmas above, we prove confluence of $\stackrel{1}{\Longrightarrow}$, the `parallel moves` lemma, as a corollary of the `paving` theorem:

```
parallel_moves confluence(Spar_red1)
```

We continue by following the steps shown on Figure 2 in the proof sketch, proving the three lemmas:

```
strip_lemma_r  confluence(Spar_red1) ==> strip
strip_lemma_l  strip==> confluence(Spar_red)
lemma1         confluence(Spar_red)  ==> confluence(Sred)
```

The first two are proved by induction over $\stackrel{1}{\Longrightarrow}$ and the last from the fact that $\Longrightarrow$ and $\longrightarrow$ describe the same reduction.

From the `parallel_moves` lemma and using Modus Ponens on the three lemmas above we finally reach a proof of confluence of β-reduction:

```
confluence_beta_reduction     confluence(Sred)
```

4.9 The Church-Rosser Theorem

Conversion is axiomatised in the same style as reduction, by first defining one step of conversion and then its transitive closure. One step of conversion is defined as one step of either β-reduction or anti-reduction and is shown in Figure 17.

```
translations
  "a<-1->b"    == "<a,b>:Sconv1"
  "a<--->b"    == "<a,b>:Sconv"
inductive
  domains       "Sconv1" <= "lambda*lambda"
  intrs
    red1        "m -1-> n ==> m<-1->n"
    exp1        "n -1-> m ==> m<-1->n"
inductive
  domains       "Sconv" <= "lambda*lambda"
  intrs
    one_step    "m<-1->n  ==> m<--->n"
    refl        "m:lambda ==> m<--->m"
    trans       "[|m<--->n; n<--->p|] ==> m<--->p"
```

Figure 17: Definition of conversion

We can now formulate the Church-Rosser theorem and prove it as a direct consequence of confluence of β-reduction:

```
Church_Rosser    m<--->n ==> EX p.(m --->p) & (n ---> p)
```

The proof is performed by induction first over conversion and then in the first case (the one-step case) by induction over one step conversion.

5 The Development Process

We have now presented the final version of our proof of the Church-Rosser theorem. As explained in the introduction the proof is ported from a similar development in Coq done by Gérard Huet. The development in Isabelle was done in 5 steps: first to port Huet's proof directly and then to investigate some of his suggestions for optimisation together with exploiting some features of Isabelle ZF.

The process can be summarised as follows:

- Copy Huet's proof.
- Remove superfluous type information.

(a)
```
(u,v,w:redexes)(sub v u) -> (uv:redexes)(residuals u v uv) ->
   (wv:redexes)(residuals w v wv) ->
     ((wu:redexes)(residuals w u wu) <->
      (regular u) /\ (residuals wv uv wu))
```

(b)
```
[|u:redexes;v:redexes;w:redexes;uv:redexes;wv:redexes;wu:redexes;
  v<==u; residuals(u,v,uv);residuals(w,v,wv)|]==>
     residuals(w,u,wu) <-> (regular(u) & residuals(wv,uv,wu))
```

(c)
```
[| v<==u; residuals(u,v,uv); residuals(w,v,wv)|]==>
     residuals(w,u,wu) <-> (regular(u) & residuals(wv,uv,wu))
```

(d)
```
[| v<==u; regular(u); w~v |]==> w|>u = (w|>v) |> (u|>v)
```

Figure 18: Four versions of the prism theorem described in the text. (a) Coq version, (b) initial version, (c) less typed version, (d) final version.

- Define residuals and union as partial functions.
- Change the definitions of lifting and substitution.
- Exploit automatic case-splitting using the simplifier.

In the next sections we will explain the above transformation steps and comment on the achieved improvements. Finally at the end of this section we will explain how we have effectively employed the automatic proof tools by following a systematic work process.

5.1 The Initial Transformation

The initial proof development was done by more or less re-doing Huet's proof in Isabelle, by proving the lemmas in the same order. However, we decided to make some initial changes to exploit the use of subsets in ZF. The proof sketch in Section 2 introduces the type `redexes` as an enrichment of λ-terms, but this is unfortunately not possible in ZF. We instead decided to make `redexes` (the "biggest" type) our main type and defined `lambda` as a subtype of `redexes`, which for example saved us from defining substitution twice. Consequently we had to start the proof development from `redexes` and return to `lambda` terms later. Additionally we changed the notation of most predicates to a more readable infix notation and used Isabelle translations to define useful shorthands (e.g. `subst`).

We literally took each definition and transformed it into Isabelle syntax followed by a transformation of appropriate lemmas. This transformation was very easy and almost one-to-one as can be seen from Figure 18. Figure 18(a) shows the original formulation of the prism theorem in Coq, where `(u:redexes)` means universal-quantification of `u` and `/\` means logical 'and'. Figure 18(b) shows the same theorem as first formulated in Isabelle.

The actual Coq proofs were not of much help, although the application of major theorems served as an inspiration in constructing the Isabelle proofs. Maybe the proofs would have helped more if we had had a deeper knowledge of

Coq. The most important aspect, however, was that the formal definitions and the order of lemmas could be used. This allowed the proof development of the initial version to be finished in less than two weeks of normal working hours. Huet used 3 months to construct the proof (or "1 monk month" as he writes) using an existing pre-formal development in ML [Hue92].

5.2 Remove Superfluous Type Information

The observation behind this optimisation is that all inductive predicates are defined as subtypes. For example, the subset relation on `redexes`, `<==`, is defined as a subtype of pairs of `redexes` thus stating that `a<==b` will implicitly imply that both `a` and `b` belong to `redexes`. We therefore removed from both definitions and theorems all type information which could be deduced from the context. Note that this does not mean that the development becomes untyped as the types can still be deduced. This transformation is shown for the prism theorem as the step from Figure 18(b) to (c). As type information was removed throughout the development, extra type information had to be deduced at only a very few places.

Naturally having less type information led to justification of a smaller number of subgoals, but it did not make the proofs notably smaller as most of these subgoals were trivial and were solved automatically. However, this transformation accounted for the largest speed-up in run time of our optimisations since the overall proof state gets smaller.

5.3 Using Partial Functions

The definition of union of `redexes` and the residual function have a natural formulation as partial functions as shown above. However, Huet defined them as three argument predicates since Coq insists on all functions being total. We defined union directly as a recursive function but for the residual function we exploited the existing inductive definition of `residuals`. We proved that `residuals` was functional and employed the unique description operator `THE` to transform the predicate into a function.

The functional definitions lead to a more natural formulation of the theorems much closer to conventional mathematical notation. The transformation used on the prism theorem is shown in Figure 18 as the step from (c) to (d). An example of the reformulation of union is shown in the following two formulations of the `preservation` theorem. The first being the relational and the second the functional:

```
[| union(u,v,w); residuals(u,v,uv) |]==> residuals(w,v,uv)

[| u~v; regular(v) |]==> u|>v = (u un v)|>v
```

where the first is the relational and the second the functional version. The relation `union(u,v,w)` means that `w` is the union of `u` and `v`. At first sight it is not obvious that these two formulations are equal!

The functional formulation of residuals leads to a relatively short proof of the `prism` theorem. The original relational formulation required the introduction of an extra inductive predicate to enable simultaneous induction over three variables and two separate proofs of the right-left, left-right part of the equality. The quite complicated prism proof decreased to 1/4 of its original size.

5.4 Changing the Definition of Substitution

Huet originally defined substitution slightly differently due to computational considerations originating from his pre-formal ML development. So instead of lifting a term at every abstraction node he first lifts it, with the inner number of λ-abstractions, when the variable to be replaced is encountered:

```
fun subst_rec(u,Var(i),k)     = if k<i then Var(i-1)
                                else if k=i then lift(k,u) else Var(i)
  | subst_rec(u,Fun(t),k)     = Fun(subst_rec(u,t,succ(k)))
  | subst_rec(u,App(b,f,a),k) = App(b,subst_rec(u,f,p),subst_rec(u,a,p))
```

This requires that the `lift` function takes an extra argument, `k`, to lift a term `k` steps by use of addition. Our formulation is similar to the one used by Shankar [Sha88] and completely eliminates the use of addition thus saving a large number of lemmas. The size of the substitution proofs decreased by over 1/3 after changing the definitions to the current formulation, thus computational considerations can be expensive in proofs. However, as the substitution proofs are quite tricky it took relatively long time to make the change and get all sub-lemmas right.

5.5 Employing Automatic Case Splitting

This last optimisation does not involve changing the formulation of the theorems but is performed in order to simplify the proofs. The substitution proofs are quite big and involve a lot of case splitting due to the if-terms in the definitions. Our aim was to simplify these proofs by employing the automatic case-splitting feature of the simplifier. It works by splitting if-terms into the conjunction of two implications having the true and false condition as their assumption respectively. The type rule for `lift_rec` was originally proved using manual case splitting by the tactic `excluded_middle_tac`:

```
goal Substitution.thy
 "!!n.u:redexes==>ALL k:nat.lift_rec(u,k):redexes";
by (eresolve_tac [redexes.induct] 1);
by (resolve_tac [ballI] 1);
by (excluded_middle_tac "na < k" 1);
by (ALLGOALS(asm_full_simp_tac lift_ss));
```

but could be solved automatically by the case-splitting simplification set:

```
by (eresolve_tac [redexes.induct] 1);
by (ALLGOALS(asm_full_simp_tac lift_split_ss));
```

It turned out that automatic case-splitting was only usable when both cases (after splitting) was immediately solvable by the simplifier (as above). Unfortunately most proofs involves reasoning with transitivity of less-than which cannot be handled by the simplifier, thus the automatic case-splitting could not help us in solving the major substitution proofs. To shorten the substitution proofs we need access to arithmetic decision procedures.

5.6 Automatic Proof Tools

In this section we will comment on how we employed the automation tools of Isabelle effectively. Setting up proper simplification sets and classical rules sets can speed up the development process considerably. We therefore gradually added new rules to the initial rule sets in ZF. Large rule sets do not seem to slow down the proof tools; thus the following strategy was used:

Recursive functions mostly lead to simplifier proofs. We prove proper equality rules and a type rule and add them to the simplification set. Always prove the equality rules first and use them in all subsequent proofs.

Inductive predicates mostly lead to proofs by the classical prover but also to simplifier proofs. For every inductive definition (say `X`) we

- Add `X.intrs` and type rules to the simplification set.
- Add `X.intrs` with `addIs` to the classical rule set.
- Create instances of elimination rules using `mk_cases` with appropriate patterns of `A` and `B`:

  ```
  Ssub.mk_cases redexes.con_defs "A <== B";
  ```

  and add these with `addSEs` to the rule set. The ability to create specific versions of the elimination rules using `mk_cases` turned out to be extremely useful.

If a proof requires both simplification and elimination the following extension to the simplification set combining the simplifier and the classical reasoner was very effective:

```
X_ss setloop (SELECT_GOAL (safe_tac X_cs))
```

Simplifying with this set will first simplify as much as possible and then apply the tactic specified by the *looper*. This process continues until no further changes occur or (hopefully) the goal is solved. The looper in this case is `safe_tac` which only applies rules included as safe rules and performs some equality instantiation. The main use of this tactic in connection with the simplifier is to apply the various elimination rules created by `mk_cases`. The proof of the `prism` theorem showed in Section 4.5 is a good example of the use of this simplification set.

A general problem in relation to automation is rules like transitivity because they have a tendency to loop due to the introduced unbound variable. In this development we encountered that proofs got quite manual in connection with transitivity of less-than and compatibility. Another problem regarding automation is universal-quantifiers in the assumptions, usually originating from induction hypotheses. The tactic `fast_tac` does not handle these very well and will usually run out of store. The simplifier, however, handles all-quantification better and can usually find the right induction hypothesis.

Version	Run time	Lines	Commands
Coq	14'	1936	2033
Initial ZF	33'	1879	760
Final ZF	15'	1378	519

Figure 19: Measurements of Coq and Isabelle proofs

6 Analysis

We begin this section by presenting some statistics to compare the Coq development with the initial and final version of the Isabelle development. Figure 19 shows the result but the measurements need some comments. The proofs in Isabelle are more verbose than in Coq since they are performed directly in the SML environment and all rule-sets have to be referred to explicitly. Coq has a simple parser built on top of ML so the commands are relatively short and rule-sets are used implicitly. Therefore the number of lines is not a good measure of proof sizes. Instead we have estimated the number of commands used by the various versions by counting '.' and ';' in Coq and ';' at the end of a line in Isabelle. Not all '.' end proof commands in Coq nor do all ';' at end of line ends proofs commands in Isabelle and the method slightly favours Isabelle. It should, however, give a rough estimate of the command usage.

All run times are measured on a Sparc2 workstation. The removing of superfluous type information in Isabelle alone accounted for the biggest decrease in run time from 33' to 22'. There are various reasons why Isabelle is slower than Coq. ZF is built on top of the Isabelle meta logic which gives an extra layer, and the inductive definitions are derived in ZF, i.e. all returned rules are proved by the inductive definition package. This is in contrast to Coq, where one reasons directly in the meta logic and the inductive definitions are "hardwired" into the logic. Finally the greater automation of the Isabelle proofs requires a larger search space. However, the run times for all versions are at a level where they do not constitute any practical problem in the proof development. Proofs are developed interactively and none of the individual proof commands takes more than 5-10 seconds.

Changing the definition of substitution decreased the number of commands from 284 to 147, still leaving substitution as the class of proofs needing the most commands. The same was the case in Coq where the substitution proofs required 854 commands. What complicates automation of these proofs is that most places require reasoning with transitivity of less-than, which unfortunately makes the simplifier loop. The functional formulation made the prism-related proofs decrease from 119 to 30 commands.

The inductive package for ZF works extremely well and inductive definitions can be formulated very close to conventional mathematics. Although every inductive definition is proved in ZF this is all done automatically and constitutes no problem. The returned rules provide a good platform for automation as shown above. In Coq, Huet needed manually to define a recursive function related to every inductive predicate, as the so called "inversion principle". This is not needed in Isabelle as `mk_cases` will produce all needed elimination rules. In contrast to inductive definitions, recursive definitions still seem to be very complicated in Isabelle and better tools are needed. All our equality rules in connection with recursive definitions were proved in one step by the simplifier, which suggests that automatic tools could be constructed.

7 Conclusion

In this paper we have shown how a relatively large piece of mathematics can be ported from Coq and be naturally formalised in Isabelle ZF. The current formulation is much closer to conventional mathematics than previous formulations due mainly to the use of partial functions and the proofs are much shorter due to automation.

Porting a proof saves considerable time compared to inventing and requires much less knowledge of the proof domain. We spent less than 2 weeks in porting the proof and had only a little knowledge about the residual theory of the λ-calculus though a good knowledge of Isabelle. At the outset we would not have been able to invent the proof from scratch, but Huet's proof "paved the way". In contrast Huet is a highly respected specialist of both Coq and the λ-calculus and spent 1–3 months developing his proof from an existing pre-formal ML version. We conclude that porting is fairly easy but still expensive in human time. It was essential for easy porting that both systems provide similar tools to define datatypes, inductive predicates and recursive functions. However, constructing the changes in the proofs required much more knowledge of the domain and was rather time consuming.

Both constructing and changing the substitution proofs was lengthy and tiresome but can be seen as a difficulty of reasoning with de Bruijn indexes. Huet mention that it took him a month just to get his substitution proofs right. In spite of the relative low complexity of the proofs they occupy a large part of the total proof script. It is annoying to spend so much time on uninteresting details. Arithmetic decision procedures would be a great relief and would reduce the size of these proofs drastically.

Isabelle ZF proved to form a good basis for doing general mathematical proofs. The inductive definition package worked very well and the returned rules provided a good platform for automation. The flexible parser and pretty printer mechanisms allowing any mixfix notation enabled us to use a notation very close to conventional mathematics and thus greatly increased the overview in the interactive proof process. Recursive functions, however, are still defined at a very low abstraction level and better tools to allow SML like definitions are needed. This work suggests that automation possibilities exist. Finally the automation tools provided can be employed very effectively by carefully defining appropriate rule sets. They solved a large number of proofs automatically in this development which is also reflected in the relatively short proof script.

8 Acknowledgements

This work has been done during a visit to the Computer Laboratory at Cambridge University and I would like to thank everyone there for providing a most pleasant atmosphere and interesting discussions. Larry Paulson especially has contributed many interesting comments and suggestions and has always been willing to discuss and help me with Isabelle. Robin Sharp made a number a interesting comments on this paper. Finally I thank Gérard Huet for showing interest in this project and for answering my questions.

References

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[CM92] J. Camilleri and T. F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge, Computer Laboratory, August 1992.

[D+93] Gilles Dowek et al. The Coq proof assistant user's guide. Technical Report 154, INRIA-Rocquencourt, 1993.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Hue92] Gérard Huet. Constructive computation theory, part I. Course notes, DEA Informatique, Mathématique et Applications, Paris, 1992.

[Hue94] Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[Pau94a] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 148–161, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[Pau94b] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[Pfe92] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, Carnegie Mellon University, September 1992.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, July 1988.