More Church-Rosser Proofs (in Isabelle/HOL)

Tobias Nipkow*

Technische Universität München**

Abstract. The proofs of the Church-Rosser theorems for β, η and $\beta \cup \eta$ reduction in untyped λ-calculus are formalized in Isabelle/HOL, an implementation of Higher Order Logic in the generic theorem prover Isabelle. For β-reduction, both the standard proof and the variation by Takahashi are given and compared. All proofs are based on a general theory of commutating relations which supports an almost geometric style of confluence proofs.

1 Introduction

The Church-Rosser theorem for β-reduction is one of the basic properties of λ-calculus. The de facto standard proof can be found in [1]. It has been machine-checked a number of times already [13, 9, 6, 12]. A recently published alternative proof technique [14] prompted me to apply Isabelle/HOL to this problem. The resulting formalization has the following distinctive features:

η-Reduction Not just β but also, for the first time, η and $\beta \cup \eta$ are treated.

Proof techniques In the case of β, two different proof techniques are compared: the standard one and the one due to Takahashi.

Substitution Two different notions of substitution are formalized and related.

Abstractness Based on a theory of binary relations, including transitive and reflexive closure, an abstract theory of commuting relations is developed which supports an almost geometric style of confluence proofs.

Automation With the exception of the low level details of substitution, most proofs are almost automatic, using a few general purpose tactics.

Although the paper provides a complete formalization of the Church-Rosser proofs, it is not meant as an introduction to the underlying intuitions, which the reader is assumed to be familiar with. The purpose of the paper is to contrast textbook mathematics with its formalization in a theorem prover, and to contrast this latest exercise with other published formalizations. The overall structure of the proofs follows Chapter 3 of [1], except that all abstract relational reasoning is concentrated in one section.

The complete formalization (including proofs) is available on the web via `http://www4.informatik.tu-muenchen.de/~nipkow/isabelle/HOL/Lambda/`.

* Research supported by ESPRIT BRA 6453, *TYPES*.

** Institut für Informatik, TU München, 80290 München, Germany.
`http://www4.informatik.tu-muenchen.de/~nipkow/`

2 Isabelle/HOL

Isabelle is an interactive theorem prover which can be instantiated with different object-logics. One particularly well-developed instantiation is Isabelle/HOL, which supports Church's formulation of Higher Order Logic and is very close to Gordon's HOL system [4]. In the remainder of the paper HOL is short for Isabelle/HOL.

We present no proofs but merely definitions and theorems. Hence it suffices to introduce HOL's surface syntax. A detailed introduction to Isabelle and HOL can be found elsewhere [8]. We have used Regensburger's LaTeX-converter to improve readability.

Formulae The syntax is standard, except that there are two universal quantifiers ($\forall$ and $\bigwedge$), two implications ($\longrightarrow$ and $\Longrightarrow$) and two equalities ($=$ and $\equiv$) which stem from the object and meta-logic, respectively. The distinction can be ignored while reading this paper. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ is short for the nested implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$.

Types follow the syntax for ML-types, except that the function arrow is $\Rightarrow$ rather than $\rightarrow$.

Theories introduce constants with the keyword `consts`, non-recursive definitions with `defs`, and primitive recursive definitions with `primrec`. Further constructs are explained as we encounter them.

Although we do not present any of the proofs, we usually indicate their complexity. If we state that some proof is automatic, it means that it was either solved by rewriting or by the "classical reasoner", `fast_tac` in Isabelle parlance. The latter provides a reasonable degree of automation for predicate calculus proofs. Note, however, that its success depends on the right selection of lemmas supplied as parameters.

3 Abstract Reduction Systems

3.1 Relations

The whole development is based on the HOL theory of binary relations, which are simply sets of pairs, i.e. of type `(`$\tau \times \tau$`)set`. In particular there are two polymorphic operations * and $^=$ for the transitive-reflexive and the reflexive closure, respectively. They come with a number of theorems, in particular induction for *

```
⟦ (a,b) ∈ R*;
  P a;
  ⋀y z. ⟦ (a,y) ∈ R*; (y,z) ∈ R;  P y ⟧ ⟹ P z
⟧ ⟹ P b
```

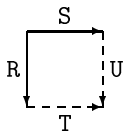
and some equational laws like `(R*)* = R*` and `(R=)* = R*`.

3.2 Squares

The rewriting literature is full of confluence-like notions which are defined formally on the predicate calculus level but are accompanied by diagrams in the form of squares. Proofs are almost exclusively performed at the level of diagrams. We mimic this approach by introducing the notion of a `square`:

```
consts square :: (α×α)set ⇒ (α×α)set ⇒ (α×α)set ⇒ (α×α)set
                 ⇒ bool
defs   square R S T U ≡ ∀x y. (x,y) ∈ R ⟶ (∀z. (x,z) ∈ S ⟶
                                (∃ u. (y,u) ∈ T ∧ (z,u) ∈ U))
```

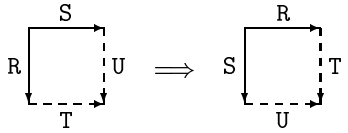
where α is a type variable, i.e. `square` is polymorphic. Thus `square R S T U` corresponds directly to the diagram



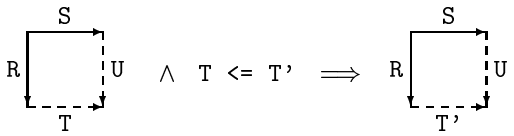
Although `square` is simply an abbreviation and all reasoning could be conducted directly on the predicate calculus level, there are distinct advantages in conducting arguments at the level of squares: for the human, proofs are much easier to find and explain on this level than on the level of quantifiers. In the context of automated deduction, there is another reason: induction, which many automatic systems cannot handle, can be encapsulated as a lemma about squares, which is then amenable to automation (see the Strip Lemma below).

For the convenience of the reader, occurrences of squares in formulae are often replaced by the corresponding diagram.

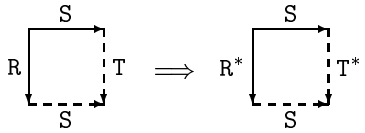
First we need a few basic lemmas about `square`:



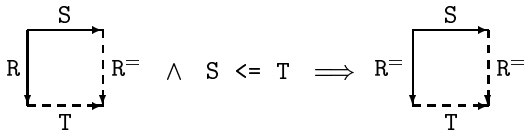
(1)



(2)



(3)



$$\texttt{S <= T} \implies \text{(4)}$$

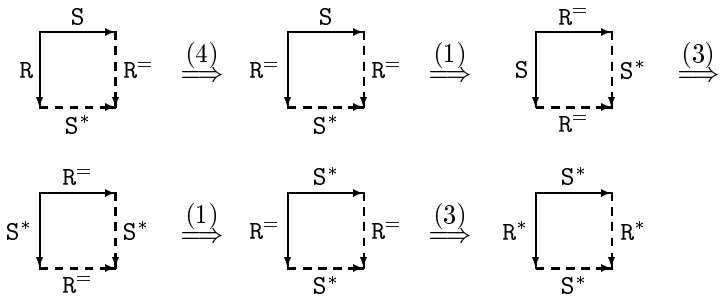
where <= is overloaded (Isabelle has type classes) and means $\subseteq$ on sets.

Lemma (3) is usually called the *Strip Lemma*. It is easily proved by induction on $\texttt{R}^*$ and it also encapsulates induction: all of the subsequent lemmas about square which require induction are proved by means of the Strip Lemma instead.

As an example of the diagrammatic proof method afforded by square we examine the proof of

$$\texttt{square R S S}^* \texttt{ R}^= \implies \texttt{square R}^* \texttt{ S}^* \texttt{ S}^* \texttt{ R}^* \tag{5}$$

which is the key lemma for proving commutation of β and η later on:



Trivial as this proofs may look, it should be kept in mind that it contains two hidden inductions. We refrain from showing any further such chains, most of which fast_tac finds automatically.

3.3 Confluence

The notion of a square is sufficient to define the usual confluence-like concepts:

```
consts commute :: (α × α)set ⇒ (α × α)set ⇒ bool
       confluent, diamond :: (α × α)set ⇒ bool

defs commute R S ≡ square R S S R
     diamond R   ≡ commute R R
     confluent R ≡ diamond(R^*)
```

We simply list the key lemmas, which form the basis for the confluence proofs of β, η and $\beta \cup \eta$, respectively:

$$[\![\texttt{diamond R; S <= R; R <= S}^*]\!] \implies \texttt{confluent S} \tag{6}$$

follows essentially from the Strip Lemma and the fact that $R^* = S^*$.

$$\texttt{square R R } R^{=} \; R^{=} \Longrightarrow \texttt{confluent R} \tag{7}$$

follows directly from (5) and (2).

$$[\![\texttt{confluent R; confluent S; commute}(R^*)(S^*)]\!] \Longrightarrow \texttt{confluent}(R \cup S) \tag{8}$$

is often called the lemma of *Hindley and Rosen*.

Finally, we come to the Church-Rosser property itself

```
consts Church_Rosser :: (α × α)set ⇒ bool

defs Church_Rosser R ≡ ∀x y. (x,y) ∈ (R ∪ converse(R))* ⟶
                             (∃z. (x,z) ∈ R* ∧ (y,z) ∈ R*)
```

(converse R ≡ {(y,x) . (x,y)∈R}) and prove it coincides with confluence:

```
confluent R = Church_Rosser R
```

The proof, although short, requires a certain amount of guidance.

4 Lambda terms and reductions

Lambda terms are represented in de Bruijn notation [3] which is conveniently expressed as an inductive data type with three constructors for variables, application and abstraction:

```
datatype db = Var nat | @ db db (infixl 200) | Fun db
```

Note that application is written infix. Thus the term $\lambda x.(xx)$ is represented as `Fun(Var 0 @ Var 0)`.

4.1 Lifting and substitution

In order to describe β-reduction for de Bruijn terms we need the notions of "lifting" and substitution. The starting point of our formalization is the report by Rasmussen [12] who ported Huet's theory of residuals [6] to Isabelle/ZF, Isabelle's set theory. Initially we benefit a lot from Rasmussen's work but we soon diverge because (a) we take a direct route to the Church-Rosser theorem instead of going via residuals and (b) we also include η-reduction.

```
consts  lift   :: db ⇒ nat ⇒ db
        subst  :: db ⇒ db ⇒ nat ⇒ db   (_[_/_] [300,0,0] 300)

primrec lift db
  lift (Var i) k  =  (if i < k then Var i else Var(Suc i))
  lift (s @ t) k  =  (lift s k) @ (lift t k)
  lift (Fun s) k  =  Fun(lift s (Suc k))
```

```
primrec subst db
  (Var i)[s/k]  =  (if k < i then Var(pred i)
                    else if i = k then s else Var i)
  (t @ u)[s/k]  =  t[s/k] @ u[s/k]
  (Fun t)[s/k]  =  Fun (t[lift s 0 / Suc k])
```

`t[s/i]` is syntactic sugar for `subst t s i` and mimics the traditional $t[s/x]$.

Most of the work went into proving the following lemmas, of which the last one is the crucial one.

```
i < Suc k ⟶ lift (lift t i) (Suc k) = lift (lift t k) i

j < Suc i ⟶ lift (t[s/j]) i = (lift t (Suc i))[lift s i/j]

i < Suc j ⟶ lift (t[s/j]) i = (lift t i)[lift s i/Suc j]

(lift t k)[s/k] = t

i < Suc j ⟶ t[lift v i/Suc j][u[v/j]/i] = t[u/i][v/j]
```

All lemmas are proved by structural induction on `t`, where the application and abstraction cases go through by simplification. Only the variable cases require painful case distinctions and explicit reasoning about inequalities between natural numbers.

Initially I tried to find and prove these lemmas from scratch but soon decided to steal instead: Rasmussen's ZF proofs carried over almost verbatim to HOL; I only had to prove five simple lemmas about `<` to create the same arithmetic basis. Without this productivity boost, the whole development would have taken much longer than it did.

4.2 β-Reduction

Once we have substitution, β-reduction is easy:

```
consts  beta :: (db × db) set

syntax  →β :: db ⇒ db ⇒ bool  (infixl 50)

translations  s →β t  ≡  (s,t) ∈ beta

inductive beta
  (Fun s) @ t →β s[t/0]
  s →β t ⟹ s@u →β t@u
  s →β t ⟹ u@s →β u@t
  s →β t ⟹ Fun s →β Fun t
```

Above we introduce `beta` as a relation on de Bruijn terms and `s` $\to_\beta$ `t` as syntactic sugar for `(s,t)` $\in$ `beta`. Beta-reduction is defined as an inductive relation. Except for the syntactic sugar and the trick of using a generic *-operator, this is very much standard.

We also introduce infix syntax for `beta`$^=$ and `beta`*,

```
syntax  →β=, ↠β :: db ⇒ db ⇒ bool  (infixl 50)

translations   s →β= t  ≡  (s,t) ∈ beta=
               s ↠β t  ≡  (s,t) ∈ beta*
```

two concepts needed later on. It helps to know that $\twoheadrightarrow_\beta$ is also a congruence,

```
s ↠β s' ⟹ Fun s ↠β Fun s'
s ↠β s' ⟹ s @ t ↠β s' @ t
t ↠β t' ⟹ s @ t ↠β s @ t'
⟦ s ↠β s'; t ↠β t' ⟧ ⟹ s @ t ↠β s' @ t'
```

which follows almost automatically using a few basic lemmas about *, in particular induction.

4.3 η-Reduction

At this point I had to improvise because I was not aware of previous formalizations of η in the context of de Bruijn notation (see Section 8.5). The following approach seemed fairly obvious:

```
consts free :: db ⇒ nat ⇒ bool
       decr :: db ⇒ nat ⇒ db
       eta  :: (db × db) set

syntax  →η :: db ⇒ db ⇒ bool (infixl 50)

translations   s →η t  ≡  (s,t) ∈ eta

primrec free db
  free (Var j) i = (j=i)
  free (s @ t) i = (free s i | free t i)
  free (Fun s) i = free s (Suc i)

inductive eta
   ¬free s 0 ⟹ Fun(s @ Var 0) →η decr s 0
   s →η t ⟹ s@u →η t@u
   s →η t ⟹ u@s →η u@t
   s →η t ⟹ Fun s →η Fun t
```

`free t i` determines if `i` is free in `t`, `decr t i` is supposed to decrement all free variables in `t` greater than `i`, and the first clause of the inductive definition of `eta` formalizes $\lambda x.(s\ x) \to_\eta s$ if x is not free in s.

Initially I defined `decr` as a primitive recursive function:

```
decr (Var j) i = (if j<=i then Var j else Var(pred j))
decr (s @ t) i = (decr s i) @ (decr t i)
decr (Fun s) i = Fun (decr s (Suc i))
```

where `pred` is the predecessor on `nat`.

This was a recipe for disaster because I now had to relate `decr` to both `subst` and `lift`. This lead to a collection of unsavoury lemmas similar to those relating `subst` and `lift`. Fortunately, it then dawned on me that `decr` is not a new concept but already inherent in `subst`:

```
defs  decr t i  ≡  t[Var i/i]
```

This definition disables the substitution aspect of `subst` by replacing `Var i` by itself, thus reducing `subst` to decrementation. The alert reader may have noticed that in the context of η-reduction it does not matter what we substitute for `i`, because `decr t i` is only called if `i` is not free in `t`. This is the gist of the following lemmas:

```
¬free s i ⟶ s[t/i] = s[u/i]
```

```
¬free s i ⟹ s[t/i] = decr s i
```

The first is proved automatically by induction on `s`, the second is a trivial consequence.

The main point of reducing `decr` to `subst` is that the number of auxiliary lemmas required later on boils down to a mere handful:

```
free (lift s k) i = (i < k ∧ free s i ∨ k < i ∧ free s (pred i))
```

```
free (s[t/k]) i = (free s k ∧ free t i ∨
                   free s (if i<k then i else Suc i))
```

```
s →η t ⟹ free t i = free s i
```

```
s →η t ⟹ s[u/i] →η t[u/i]
```

```
s →η t ⟹ decr s i →η decr t i
```

The first two are de Bruijn specific, quite tricky to come up with, and are proved by induction on `s` and painful arithmetic reasoning. The next two are of a general nature, quite intuitive, and automatic by induction on `s` $\to_\eta$ `t`. The last one is a trivial consequence of the one before.

We also introduce infix syntax for `eta`$^=$ and `eta`*,

```
syntax  →η=, →→η :: db ⇒ db ⇒ bool  (infixl 50)

translations   s →η= t  ≡  (s,t) ∈ eta=
               s →→η t  ≡  (s,t) ∈ eta*
```

two concepts needed later on. It helps to know that $\twoheadrightarrow_\eta$ is also a congruence,

```
s →↠η s' ⟹ Fun s →↠η Fun s'
s →↠η s' ⟹ s @ t →↠η s' @ t
t →↠η t' ⟹ s @ t →↠η s @ t'
⟦ s →↠η s'; t →↠η t' ⟧ ⟹ s @ t →↠η s' @ t'
```

which follows almost automatically using a few basic lemmas about *, in particular induction.

5 Confluence of β

The key idea is to use lemma (6) to reduce confluence of $\rightarrow_\beta$ to the diamond property of a new reduction, say $\Rightarrow_\beta$, such that $\Rightarrow_\beta$ lies in between $\rightarrow_\beta$ and $\twoheadrightarrow_\beta$.

5.1 Parallel β-reduction

It is well known that the following form of parallel (and nested) β-reduction has the desired properties:

```
consts  par_beta :: (db × db) set

syntax  ⇒β :: db ⇒ db ⇒ bool  (infixl 50)

translations   s ⇒β t  ≡  (s,t) ∈ par_beta

inductive par_beta
  Var n ⇒β Var n
  s ⇒β t ⟹ Fun s ⇒β Fun t
  ⟦ s ⇒β s'; t ⇒β t' ⟧ ⟹ s @ t ⇒β s' @ t'
  ⟦ s ⇒β s'; t ⇒β t' ⟧ ⟹ (Fun s) @ t ⇒β s'[t'/0]
```

It is a simple inductive consequence that $\Rightarrow_\beta$ is reflexive.

The proofs of `beta <= par_beta` and `par_beta <= beta`* are again, modulo the application of induction, automatic. Using lemma (6), it only remains to prove the diamond property of $\Rightarrow_\beta$.

However, before we get there, we need two lemmas, namely that $\Rightarrow_\beta$ is compatible with lifting and substitution:

```
t ⇒β t' ⟶ lift t k ⇒β lift t' k
s ⇒β s' ⟶ t ⇒β t' ⟶ t[s/k] ⇒β t'[s'/k]
```

Both lemmas are proved by induction on `t`. The second is the key to the diamond property proof, two different versions of which we now present.

5.2 Takahashi's proof

Takahashi [14] needs a mere page to sketch the complete confluence proof, which is probably the best one can do on paper. This succinctness is achieved by using the notion of *complete development* of a term:

```
consts  cd  :: db ⇒ db
primrec cd db
  cd(Var n)  =  Var n
  cd(s @ t)  =  (case s of
                   Var n => s @ (cd t)
                 | s1 @ s2 => (cd s) @ (cd t)
                 | Fun u => (cd u)[cd t/0])
  cd(Fun s)  =  Fun(cd s)
```

Unfortunately we cannot define `cd` exactly as above using the current version of `primrec`: in the second clause, only the recursive calls `cd s` and `cd t` are allowed because `s` and `t` are the only *direct* subterms of the argument `s @ t`. Fortunately we do not have to resort to well-founded recursion but can patch the problem: instead of `cd u` we use `deFun(cd s)`, where `deFun` is a new function defined by `deFun(Fun s) = s`. The reader is invited to convince herself that this achieves the desired result.

It is now straightforward to prove the *completion theorem* for `cd`:

`s` $\Rightarrow_\beta$ `t` $\longrightarrow$ `t` $\Rightarrow_\beta$ `cd s`

Apart from the induction on `s` and a case distinction, the proof is automatic. This is the result of combined simplification and logical reasoning.

The proof of `diamond par_beta`, i.e. the diamond property of $\Rightarrow_\beta$, is now completely automatic: the existential variable `u` in the definition of `diamond`, i.e. `square`, is instantiated by `cd x` with the help of the completion theorem and unification.

5.3 The standard proof

Traditionally [1], the diamond property is proved directly. This is less slick than Takahashi's proof because it involves a laborious case distinction on the relative positions of the two redexes contracted at the root of the diamond, and each case leads to a different completion of the diamond. This, however, does not bother Isabelle: `diamond par_beta` is proved automatically, modulo the initial application of induction on `par_beta`. This was a little surprising to me, because of the required existential witnesses and the lack of an obvious candidate `cd x`.

This raises the question if anything has been gained by the formalization of Takahashi's proof. The answer seems to be no: Takahashi requires an additional function `cd` and the proof of the completion theorem for `cd` is in fact a little more complicated than the direct proof of the diamond property of $\Rightarrow_\beta$. The point is that the ingenuity of her approach is wasted in the presence of mindless search procedures, aka tactics.

6 Confluence of $\beta \cup \eta$

Confluence of η on its own is an anticlimax: no auxiliary concepts are required,

```
square eta eta (eta^=) (eta^=)
```

is proved automatically by induction on `eta`, and `confluent eta` follows directly by lemma (7).

The real challenge is the confluence of $\beta \cup \eta$. We proceed the way of Hindley and Rosen by combining confluence of β and η (which we already have) with the commutation of $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_\eta$ (which remains to be proved). The latter requires some further lemmas

```
s →β t ⟹ free t i ⟶ free s i
```

```
s →β t ⟹ decr s i →β decr t i
```

which are automatic by induction on `s` $\rightarrow_\beta$ `t`. We also need the recursion equations of the primitive recursive version of `decr` shown above; they follow directly by simplification. This enables us to prove the following very de Bruijn specific lemma

```
¬free t (Suc i) ⟹ decr t i = decr t (Suc i)
```

by induction on `t`. Finally there are two more lemmas on the interaction of $\rightarrow_\eta$ with lifting and substitution

```
s →η t ⟹ lift s i →η lift t i
```

```
s →η t ⟶ u[s/i] ↠η u[t/i]
```

proved by induction on `s` $\rightarrow_\eta$ `t` and `u` respectively. The proof of the key commutation lemma

```
square beta eta (eta*) (beta^=)
```  (9)

is by induction on the definition of `beta`. Although the resulting subgoals could in principle be solved automatically, `fast_tac` hits a complexity barrier. Explicit case analysis of the `eta`-component of the square yields a total of 6 subgoals, each of which is solved automatically. Using lemmas (5) and (8) together with the confluence of β and η and lemma (9),

```
confluent (beta ∪ eta)
```

follows automatically.

7 An optimization

The code for lifting and substitution given above is used in most theoretical investigations of de Bruijn terms because it only requires successor and predecessor on natural numbers. However, most implementations use an optimized form of substitution and lifting, which replaces n 1-step liftings by 1 n-step lifting:

```
consts  substn :: db ⇒ db ⇒ nat ⇒ db
        liftn  :: nat ⇒ db ⇒ nat ⇒ db

primrec liftn db
  liftn n (Var i) k  = (if i < k then Var i else Var(i+n))
  liftn n (s @ t) k  = (liftn n s k) @ (liftn n t k)
  liftn n (Fun s) k  = Fun(liftn n s (Suc k))

primrec substn db
  substn (Var i) s k = (if k < i then Var(pred i)
                        else if i = k then liftn k s 0 else Var i)
  substn (t @ u) s k = (substn t s k) @ (substn u s k)
  substn (Fun t) s k = Fun (substn t s (Suc k))
```

An exception is Huet, who uses the above versions in his proofs, which complicate them severely. On top of the lemmas in section 4.1, he needs a number of further lemmas dealing with addition, among them the following beauty:

```
substn (substn t s p) u (p+n) =
substn (subst t u (Suc(p+n))) (substn s u n) p
```

We show now that you can have your cake and eat it too: do your proofs with the simple form of substitution and lifting, and prove the optimized versions equivalent later on. This turns out to be much simpler than working with the optimized code directly. The following sequence of lemmas does the trick:

```
liftn 0 t k = t
liftn (Suc n) t k = lift (liftn n t k) k
substn t s k = t[liftn k s 0 / k]
```

All three are proved by induction on `t` followed by simplification. As a corollary to the last lemma we obtain `substn t s 0 = t[s/0]`. Thus we can safely replace `(Fun t) @ s` $\rightarrow_\beta$ `t[s/0]` by `(Fun t) @ s` $\rightarrow_\beta$ `substn t s 0` in the definition of $\rightarrow_\beta$.

8 Comparisons

A comparison with the treatments by Shankar [13], Pfenning [9], Huet [6] and Rasmussen [12] is difficult because the frameworks and aims differ significantly: Shankar uses a much weaker logic than the others (no explicit quantifiers), and Pfenning uses higher-order abstract syntax, thus bypassing de Bruijn terms in favour of the built-in λ-abstraction of ELF. Huet and Rasmussen develop a

general theory of residuals, which yields the Church-Rosser theorem as a by-product. Hence we only highlight how our approach differs from the others.

There is also the work by McKinna and Pollack [7, 10] who have proved the Church-Rosser theorem in LEGO using Takahashi's approach. What sets their work apart is that they formalize λ-terms with named variables rather than de Bruijn indices. However, they do not provide many details, since for them the Church-Rosser theorem is only a small part of a much larger development.

8.1 Proof techniques

We formalized both Takahashi's and the standard proof of the diamond property. Although Takahashi's approach is intellectually appealing, the standard direct proof turned out to be simpler and shorter, since its many case distinctions disappear through automation. For systems with a low degree of automation, for example LEGO and Coq, Takahashi's method may be preferable. It may also be preferable for the Boyer-Moore prover because it yields a simple witness function, which their quantifier-free logic requires.

Of course, Takahashi's approach may well turn out to be significantly better than the traditional methods when we try to formalize more advanced theorems like standardization [14].

8.2 Substitution

We have followed Shankar and Rasmussen in using the 1 step lifting function `lift` to simplify the proofs of the basic substitution lemmas. This paid off because these lemmas required about a third of the work (including the failed attempts before porting Rasmussen's proofs) and would have been *much* harder had we used Huet's optimized versions `liftn` and `substn`. Fortunately we could prove both forms of lifting and substitution equivalent quite easily. This is in accordance with the accepted wisdom that correctness proofs should always start from the simplest version of a system and that optimizations should come later.

8.3 Abstractness

All of section 3 is independent of λ-calculus. It is part of a general theory of relations and can be reused in many other contexts. Although it has probably not shortened our development by much, this abstractness has definitely made the proofs more transparent and is an investment for the future. The other authors conduct all arguments in terms of the specific relations $\rightarrow_\beta$, $\twoheadrightarrow_\beta$ and $\Rightarrow_\beta$, and even define the transitive-reflexive closure of $\rightarrow_\beta$ via an explicit inductive definition instead of using a generic operation like *.

8.4 Automation

With the exception of the low level details of substitution, most of the Isabelle proofs are almost automatic, using a few general purpose tactics. Due to the

relatively high degree of automation, some of our proofs require a fair bit of search and take up to 20 seconds. Loading the complete development takes 160 seconds on a SPARCstation 20/712. If Isabelle provided more automation for arithmetic (as the Boyer-Moore prover does), the substitution lemmas should be automatic as well. Another area where automation seemed to break down was transitivity: a few times it had to be invoked explicitly to avoid an explosion of the search space.

In terms of lemmas required, Shankar's proof has a similar granularity as ours, but each individual lemma is proved automatically. Due to the absence of inductively defined relations and existential quantifiers, the formalization required by the Boyer-Moore prover is not exactly natural. Pfenning's proofs are the opposite: they follow the traditional mathematical argument quite closely, but provide no automation at all. With respect to automation, Huet's proofs are closer to Pfenning's, Rasmussen's are closer to ours.

8.5 η-Reduction

I am not aware of any machine-checked proofs for η or $\beta \cup \eta$. However, after I had finished my own development I found out about the following two formalizations of η in λ-calculi with explicit substitutions.

Hardin [5] briefly considers, in our notation, the following definition of $\rightarrow_\eta$:

$$\texttt{Fun(lift t 0 @ Var 0)} \rightarrow_\eta \texttt{t} \tag{10}$$

She rejects it because the subterm `lift t 0` gives rise to critical pairs with the rules for `lift`. We conjecture that (10) works in our setting because the equations for `lift` are on a different level than (10). Hardin settles for a conditional variant of (10) which leads to quite involved confluence proofs.

Briaud [2] follows a very similar train of thought as I did to arrive at the conclusion that `decr` should be expressed in terms of `subst`. In order to avoid a conditional rule he introduces a fictitious term $\perp$ and defines

$$\texttt{Fun(t @ Var 0)} \rightarrow_\eta \texttt{t[}\perp\texttt{/0]}$$

Although he avoids tricky lemmas about the interaction of `free` and `subst`/`lift`, he pays for it in terms of the complications $\perp$ causes.

8.6 Resources — not a comparison

The complete proof effort took 9 days, of which 3 days were taken up by the ground work including confluence of β, and a further 6 days to develop the theory of squares, η and $\beta \cup \eta$. Of those 6 days, almost 2 were spent developing the de Bruijn theory of η on paper, and one was wasted with the initial primitive recursive definition of `decr`. The size (in number of lines) of the final theories, including all comments, definitions and proofs, is shown in the following table:

Relations	Terms, Substitution, β	Confluence β	η, Confluence $\beta \cup \eta$
135	255	140	255

9 Conclusion

It should be obvious from the above comparisons that the field as a whole is making progress: formalizations have become more natural and shorter, and the degree of automation is increasing. We are also beginning to reuse other people's work (as in the case of Rasmussen's proofs). Yet each system still has painful shortcomings, for example arithmetic in the case of Isabelle. More work on the integration of decision procedures is urgently needed.

Acknowledgments. Thanks are due to James McKinna for detailed discussions, to Randy Pollack for his short cut in the abstract confluence proofs [11], and to Konrad Slind for his excellent suggestions for improvements.

References

1. H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North Holland, 2nd edition, 1984.
2. D. Briaud. An explicit Eta rewrite rule. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lect. Notes in Comp. Sci.*, pages 94–108. Springer-Verlag, 1995.
3. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
4. M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.
5. T. Hardin. Eta-conversion for the language of explicit substitutions. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming*, volume 632 of *Lect. Notes in Comp. Sci.*, pages 306–321. Springer-Verlag, 1992.
6. G. Huet. Residual theory in λ-calculus: A formal development. *J. Functional Programming*, 4:371–394, 1994.
7. J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Comp. Sci.*, pages 289–305. Springer-Verlag, 1993.
8. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
9. F. Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *J. Automated Reasoning*, 199? To appear.
10. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.
11. R. Pollack. Polishing up the Tait–Martin-Löf proof of the Church–Rosser theorem. Unpublished manuscript, Jan. 1995.
12. O. Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, May 1995.
13. N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.
14. M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118:120–127, 1995.

This article was processed using the LaTeX macro package with LLNCS style