University of Edinburgh
Division of Informatics

Formalizing Proofs in Isabelle/HOL of Equational Properties for the λ-Calculus using One-Sorted Variable Names

4th Year Dissertation
Computer Science and Mathematics

James Brotherston

June 3, 2001

Abstract: We present the Isabelle/HOL formalisation of some key equational properties of the untyped λ-calculus with one-sorted variable names. Existing machine formalisations of λ-calculus proofs typically rely on alternative representations and/or proof principles to facilitate mechanization and we briefly account for these works. Our own development remains faithful to the standard textbook presentation and the usual pen-and-paper proof practices; we reason purely inductively over the standard first-order syntax of the calculus, using only primitive proof principles of the syntax and the reduction relations under consideration. We prove the confluence property of the λ-calculus at the raw syntactic level and derive confluence of the real λ-calculus (the structural collapse onto equivalence classes of the raw calculus) via a general result about abstract rewrite systems which we also formalise. We then show a technical property of the *residual theory* of the calculus which suggests the general applicability of the method to other equational properties of the calculus. Finally, we make some proof-technical observations pertaining to the extent to which our Isabelle proofs may be automated.

Acknowledgements

My sincerest thanks are due in the first instance to my supervisor, René Vestergaard, for innumerable pieces of advice (by turns both practical and cryptic), for technical and moral support, for important references, and not least for feeding me for a week during my visit to Jean-Yves Girard's group in Marseilles in March this year (which he made possible).

Martin Hofmann (October '00 - April '01) and David Aspinall (April - May '01) at LFCS have acted as official supervisors for the project since René departed for Marseilles and I would like to take the opportunity to thank them both for all their advice and helpful suggestions particularly at the write-up stage.

Thanks are due also to: James Margetson, Larry Paulson, and Markus Wenzel for technical advice on using Isabelle/HOL; to LFCS for their assistance in the initial stages of the project; to the TMR-LINEAR group in Marseilles for kindly hosting me during my visit; to Ross Duncan for helpful discussions; and lastly to Josefin for all her support.

Contents

1 The λ-calculus and formalised reasoning **3**
- 1.1 The λ-calculus: a model of computation . . . 3
- 1.2 Presentations of the λ-calculus . . . 3
  - 1.2.1 First-order abstract syntax with one-sorted variable names 4
  - 1.2.2 First-order abstract syntax with two-sorted variable names 5
  - 1.2.3 Nameless terms: de Bruijn indices . . . 7
  - 1.2.4 Axiomatising α-equivalence . . . 8
- 1.3 λ^{var}: a first-order development . . . 10
  - 1.3.1 Reduction relations . . . 10
  - 1.3.2 Structural induction for λ-terms over $FOAS_{VN}$ . . . 11
  - 1.3.3 Substitution à la Curry . . . 11
  - 1.3.4 Defining substitution and reduction . . . 12
  - 1.3.5 (Connections with) Barendregt's Variable Convention . . 14
  - 1.3.6 Isabelle: a generic theorem prover . . . 15

2 Confluence of the raw and real λ-calculi **17**
- 2.1 Getting started in Isabelle . . . 17
- 2.2 Basics of λ^{var} . . . 18
  - 2.2.1 A First Proof . . . 19
- 2.3 Substitution lemmas . . . 20
- 2.4 Diamond property of Parallel β up to BCF-Initiality . . . 21
- 2.5 Weak α_0 / β commutativity . . . 28
- 2.6 Fresh-naming α-confluence with BCF-Finality . . . 32
- 2.7 Confluence of λ^{var} . . . 36
- 2.8 Confluence of λ^{real} . . . 38
- 2.9 Paying the cost to be the boss . . . 43

3 Barendregt-Style Reasoning is Correct, Sometimes **45**
- 3.1 The Marked λ^{var}-Calculus . . . 46
- 3.2 Non-Blocked β-Residual Theory . . . 47

4 Automatibility of Isabelle Proofs **53**
- 4.1 Safe vs. unsafe rules . . . 53
- 4.2 Explicit instantiations . . . 55
- 4.3 Side-conditions on rule application . . . 57
- 4.4 Inherent complexity . . . 58

5 Conclusion **61**

Bibliography **63**

Preface

As the title suggests, the aim of this project is proof formalisation in Isabelle/HOL. We work in the setting of the λ-*calculus*: a universal model of computation which in the past has been widely studied in its different formulations. The formalisation of proofs in this setting has been an area of study in itself and has attracted attention from many quarters. It is a surprising fact that the existing proof formalisations all employ alternatives to the standard representation of the calculus, which is used for conducting pen-and-paper proofs. This is because the names which are traditionally used to represent binding in a term can overlap when we manipulate terms in the calculus.

This particular formal impassé was recently overcome by René Vestergaard, who in 2000 developed a new presentation of the λ-calculus over the standard first-order abstract syntax and using names to represent binding. The innovation comes in the careful definition of operations within the calculus so as to avoid the traditional formal problems caused by variable name overlap. The use of first-order abstract syntax enables one to perform all reasoning in the calculus by purely inductive means; this makes the calculus particularly amenable to formalisation in a theorem prover such as Isabelle, where one is limited to using induction as a proof principle.

The proof of the *confluence* property in this setup was published by Vestergaard and the author at RTA'01 [17]. A full version of the article has been invited to be submitted to a special issue of Informartion and Computation with selected papers from RTA'01. A follow-up article [18], also by Vestergaard and the author, showing other equational properties of the calculus, has also been accepted for MERLIN'01. In both papers, the primary contribution of the author was the Isabelle/HOL formalisation of the results presented there and it is the details of these formalisations which we aim to present here. The proof methodology we use and the results we present are due to Vestergaard, unless otherwise stated; the formal Isabelle proofs are due to the author.

In this dissertation we aim, firstly, to give an overview of the issues involved in formalising λ-calculus proofs and, secondly, to provide an insight into the details of our own formalisation work. We also make some meta-level observations in Chapter 4 about some of the technical issues which arose out of the Isabelle formalisation.

1. The λ-calculus and formalised reasoning

1.1 The λ-calculus: a model of computation

The λ-calculus is a branch of mathematical logic, developed in the 1930s by Church, that is intended to capture the concept of a function. The pure λ-calculus is type-free and consists of λ-abstractions (functions), variables, and applications of one function to another. All structural manipulations in the calculus are made explicit by the use of *reduction relations*. A reduction relation is a set of pairs relating λ-terms to their associated term under the reduction. (We shall examine reduction relations in some detail in section 1.3.1.)

Church devised the λ-calculus [3] as part of an ultimately abortive attempt to provide a foundation for logic and parts of mathematics. This attempt failed largely because the resulting system admitted the well known Russell Paradox. Still, the pure λ-calculus embedded in the theory was a success and led to the formulation of Church's thesis: the effectively computable functions are exactly those that can be described using the λ-calculus.

The λ-calculus is closely linked with several areas of study in both mathematics and computer science including categorical logic, proof theory, type theory and the semantics of programming languages (for examples consult [1]). It is particularly useful in the study of functional programming languages, since such languages (Lisp, SML, Miranda) can be seen as extensions of the λ-calculus with constants and types.

1.2 Presentations of the λ-calculus

Several different presentations of the λ-calculus have been formulated over the years. The formulation which is used almost exclusively when conducting pen-and-paper proofs is first-order abstract syntax with one-sorted variables, which is our main area of attention. However, this formulation also presents some formal difficulties as we shall see shortly. To counter these problems, various alternative inceptions of the syntax have been proposed in order to facilitate machine formalisation and we also examine those formalisations which are most relevant to our own work.

1.2.1 First-order abstract syntax with one-sorted variable names

The standard first-order abstract syntax for the λ-calculus is generated by the following (inductive) definition:

$$\Lambda^{var} ::= x \mid \Lambda^{var}\Lambda^{var} \mid \lambda x.\Lambda^{var}$$

i.e. the set of all λ-terms, Λ^{var}, consists exactly of those terms which can be constructed from atomic elements of the set (variables) using the provided constructors (application respectively abstraction).

A term in the λ-calculus is thus finite and is either a variable, an application of one term to a another, or a functional abstraction of a variable over a term (which we call the *body* of the abstraction). When we write λ-terms we will assume that application 'binds tighter' than λ-abstraction.

We use the term *first-order* in the sense that only the abstract syntax tree is provided as a primitive constructor - there are no built-in notions of abstraction or functionality attached to objects constructed this way. We will refer to this syntax as the *raw syntax* of the λ-calculus, and we refer to terms constructed from the syntax as *raw terms*. (Later, we will consider the *real terms* of the calculus as equivalence classes of raw terms. The raw / real terminology was first introduced in Wells-Vestergaard [19] and used formally in Vestergaard-Brotherston ([17] and [18]), and we continue to use the notions here.)

We consider the set of variables in this syntax to be a countably infinite set of names with a decidable equality relation, =. 'Decidable equality' here means that we consider the set of variables to be constructed in some metalanguage in which equality is well-defined. Variables are *one-sorted* in the technical sense that all variables in the set have the same type (i.e. we cannot distinguish two variables other than by comparing their names). It will be necessary for us to distinguish between *free* and *bound* occurrences of variables within a given λ-term. Informally, an occurrence of a variable in a raw term is a bound occurrence if it falls within the scope of a λ-binder with the same name, and is a free occurrence otherwise. We make this notion precise with the following definition (by structural recursion on the set of λ-terms).

Definition 1 (Free and bound variables) *Define the (finite) set of free variables of a raw term,* $\mathrm{FV}(-)$*, by:*

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(e_1 e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\ \mathrm{FV}(\lambda x.e) &= \mathrm{FV}(e) \setminus \{x\} \end{aligned}$$

Similarly, define the (finite) set of bound variables of a raw term, $\mathrm{BV}(-)$*, by:*

$$\begin{aligned} \mathrm{BV}(x) &= \emptyset \\ \mathrm{BV}(e_1 e_2) &= \mathrm{BV}(e_1) \cup \mathrm{BV}(e_2) \\ \mathrm{BV}(\lambda x.e) &= \mathrm{BV}(e) \cup \{x\} \end{aligned}$$

Recall that terms in the λ-calculus are intended to represent parameterised functions. Therefore, we informally consider two terms in the raw λ-calculus to be the same (real) term if they differ only in the variable names they use to express the binding structure of the term. Later, we will derive a formal relation $=_\alpha$ for expressing such equalities; for now, we content ourselves with an example:

$$\lambda y.(\lambda x.xy) =_\alpha \lambda x.(\lambda z.zx)$$

Note that two terms which differ by a renaming of *free* variables will *not* be said to be equal in the above sense, so, for example, $\lambda x.y \neq_\alpha \lambda x.z$.

One operation on λ-terms which we will want to allow is the *substitution* of a term for a free variable. We will use the syntax $e_1[x := e_2]$ to denote the substitution of e_2 for the free variable x in the term e_1. However, note that naive syntactic replacement of e_2 for x is *not* a correct definition of substitution, as shown by the following example:

Example 1 *Suppose $\lambda y.(add\ x)y$ is the function on the natural numbers which takes one parameter — the bound variable y — and adds it to the free variable x. (Here the function symbol* add *is represented by a free variable.) Consider replacing x by the term consisting of the singleton free variable z into this term:*

$$(\lambda y.(add\ x)y)[x := z] \quad = \quad \lambda y.(add\ z)y$$

as one would expect. However, consider now replacing x by the term consisting of the free variable y:

$$(\lambda y.(add\ x)y)[x := y] \quad = \quad \lambda y.(add\ y)y$$

We see that 'substitution' now yields the function taking a parameter y and returning $2y$!

Clearly this was not the intended meaning of the function - what has gone wrong? In making the substitution $(\lambda y.(add\ x)y)[x := y]$, the free variable y is *captured* by the binder λy and becomes bound. (In terms of functional programming, this corresponds to confusion of variable scopes.) It is this clash between free and bound variable names which we wish to avoid in order to preserve correctness of substitution. We will now examine some alternative presentations of the λ-calculus which prevent this difficulty from arising.

1.2.2 First-order abstract syntax with two-sorted variable names

In their formalisation [12] of a Pure Type System which includes the λ-calculus, McKinna and Pollack take a novel approach to the issue of avoiding free variable capture by imposing a strict dichotomy on variable names, which in this set-up are *two-sorted*; every name is either a free variable or a *parameter* (i.e. a

bound variable), and the two sorts have different types. There is therefore no possibility of a syntactic confusion between the two sorts. The λ-calculus part of the McKinna-Pollack development can thus be defined inductively as follows:

$$\Lambda^{MP} ::= p \mid v \mid \Lambda^{MP}\Lambda^{MP} \mid \lambda\{p : \Lambda^{MP}\}.\Lambda^{MP}$$

where p and v range over the set of parameters and the set of free variables respectively. Note that the variable for a λ-abstraction has an explicit type which must itself be given by a term in the system.

McKinna-Pollack formalise a number of equational properties of their Pure Type System (an extension of the above system) using the LEGO theorem prover, including β-confluence (also known as the Church-Rosser Theorem) which we prove ourselves in chapter 2. (We give a formal definition of confluence there; informally, it means that if a term reduces in two different ways then these two divergences can be reduced to the same term.) One important facet of the McKinna-Pollack development is that the employed proof principles are different to the usual first-order principles. A predicate `Vclosed` on terms is defined corresponding to the notion of a *closed term*, and only terms which are `Vclosed` are considered to be well-defined for the purposes of reduction in the system. The definition of this predicate in (the λ-calculus part of) their system is as follows, where A,B range over the set of terms:

Vclosed(p)
Vclosed(A) & Vclosed($B[v := p]$) → Vclosed($\lambda\{n : A\}.B$)
Vclosed(A) & Vclosed(B) → Vclosed(AB)

Note that for a term to be `Vclosed` is equivalent to it having no free variables (and this is indeed proven to be the case). As only `Vclosed` terms are considered to be well-formed for the purposes of reasoning within the system, `Vclosed` is treated as an induction principle over well-formed terms. All of McKinna-Pollack's results are thus stated in a similar manner to the following example:

```
Goal DiamondProperty Vclosed par_red1
```

where `par_red1` is a reduction relation with the desired property. On the other hand, definition by structural recursion in the system is well-defined on all terms; the structural induction/recursion principle `Trec` is given in LEGO as follows, where `Trm` denotes the type of terms in the system:

```
[Trec:{C:Trm->Prop}
      {TVAR:{n:VV\}C (var n)}
      {TPAR:{n:PP\}C (par n)}
      {TLDA:{n:VV}{A,B:Trm}(C A)->(C B)->C (lda n A B)}
      {TAPP:{M,N:Trm}(C M)->(C N)->C (app M N)}
        {t:Trm}C t];
```

The correct reading of the above code is as follows. The first line tells us that the proposition C under consideration must have type `trm->Prop`. The next four lines give the premises for the induction principle, i.e. that C should hold of any parameter or variable, and that if C holds for the subterms of an application or

abstraction, then C should hold of the term itself. The conclusion then is that C is true of any term t. Definition by structural recursion is thus well-founded over all terms and not just those that are `Vclosed`.

We do not consider this approach further here but observe that the imposition of the syntactic distinction between parameters and variables is somewhat removed from standard pen-and-paper proof practices as witnessed by, e.g., the use of the `Vclosed` predicate. Variables in pen-and-paper proofs are still considered to be one-sorted, which is one reason why we will choose to work in the standard set-up despite its additional complication.

1.2.3 Nameless terms: de Bruijn indices

De Bruijn's contribution to the field was to define the λ-calculus using a set of *nameless terms* such that any two α-equivalent terms in the standard λ-calculus correspond to the same nameless term in the de Bruijn λ-calculus (λ^{dB}). (In fact, the presentation was originally developed by de Bruijn in order to facilitate a λ-calculus formalisation in his AutoMath system [5].) The set of nameless terms, Λ^* is defined as follows:

$$\Lambda^* ::= n \mid \Lambda^*\Lambda^* \mid \lambda\Lambda^*$$

where $n \in \mathcal{N} \setminus \{0\}$. Hence, variables in λ^{dB} are given as natural numbers. Given a raw λ-term in the standard syntax, we can construct its de Bruijn representation by replacing each bound variable occurrence by an index referring to the number of λs between it and its binding λ, so, for example:

$$\lambda x.x(\lambda y.y(\lambda z.zx)x) \mapsto_{dB} \lambda 1(\lambda 1(\lambda 13)2)$$

The free variable v_n is referred to by the natural number $n + k + 1$, where k is the λ-nesting level at which the number occurs. Note that this number will always be greater than the index given to any bound variable at the same λ-nesting level; therefore, the problem of bound / free variable overlap never occurs. Note also that when we perform substitution on λ^{dB} we generally need to update the indices given to all of the variables in order to account for their new level of λ-nesting after the substitution.

Probably the most notable λ-calculus formalisation work in a de Bruijn setting is an Isabelle/HOL proof development due to Tobias Nipkow. In [13] he defines a de Bruijn representation of the terms of the standard λ-calculus as given above, and gives the following definition of substitution:

$$\begin{aligned} i[k := s] &= \begin{cases} i-1 & \text{if } k < i \\ s & \text{if } k = i \\ i & \text{otherwise} \end{cases} \\ tu[k := s] &= t[k := s]u[k := s] \\ \lambda t[k := s] &= \lambda(t[k+1 := \text{lift } s0]) \end{aligned}$$

where lift s 0 increments all free variables in s. Observe that substitution as defined here is quite complex and may involve updating *all* of the variables in a term. Nipkow goes on to prove confluence of β-reduction, confluence of η-reduction and confluence of the conjoint $\beta \cup \eta$-reduction. The intricate way in which substitution operates means that the bulk of Nipkow's work goes into proving subtle lemmas involving the behaviour of the set of free variables under the lifting and substitution operations. However, working in this system enables Nipkow to use the standard (first-order) inductive proof principles of first-order abstract syntax. Much of his foundational work also employs diagrammatic reasoning about general abstract rewrite systems and we actually reuse some of his Isabelle work in this area for our own development. Huet [11] has also formalised some properties of the de Bruijn λ-calculus in the Gallina specification language used by the Coq theorem prover. He works in the *residual theory* of the λ-calculus, which we discuss in chapter 3 and derives β-confluence via the so-called Prism Theorem.

The use of de Bruijn indices as opposed to one-sorted variable names is the *de facto* standard for machine implementations of λ-calculus, since the nameless terms can be manipulated cleanly and efficiently using first-order methods, and no recourse is needed to the issue of variable-renaming with which we will concern ourselves. However, to the human reader it presents several intuitive problems (apart from the obvious illegibility of the representation). Firstly, when performing a substitution, potentially all of the indices in the term must be updated, and therefore parts of a term which have nothing to do with the actual target of the substitution are affected. Secondly, substitution enjoys some highly non-intuitive properties; for example, Nipkow [13] presents the following property of substitution in the de Bruijn λ-calculus, which is used in proving confluence of $\beta \cup \eta$:

$$e[i := i] = e[i + 1 := i]$$

(This property is due to Nipkow's definition of de Bruijn substitution; the subtle manner in which it updates indices throughout a term ensures the result of the substitution is the same on both sides. For a more detailed discussion of why this holds the reader is advised to consult [13].) Lastly, and perhaps most seriously, when examining de Bruijn representations of more complicated languages (e.g. a functional programming language) it is not at all obvious that the representation is correct. In other words, it can be hard to convince oneself, both formally and informally, that the language being represented is in fact the language under consideration. De Bruijn languages are thus not ideal for human consumption.

1.2.4 Axiomatising α-equivalence

All the approaches we have considered so far employ first order abstract syntax to represent λ-terms. In other words, we work with explicitly defined sets of variables and in order to define substitution and reduction in the calculus (which

we will do shortly) we need to reason about these *object-level* variable names. However, much recent effort has been devoted to formalising the behaviour of substitution and reduction directly at the level of α-equivalence classes (which we refer to as the real λ-calculus or λ^{real}). Working at the real level entails the study of suitable proof methods for the equivalence classes since the usual induction principles no longer come 'for free' once one abandons the first-order abstract syntax of the calculus. In fact, Hofmann [9] reports that adding an axiom of full recursion to higher-order abstract syntax results in an inconsistent theory.

One of the most accessible works of this kind is the PVS formalisation of the Church-Rosser theorem due to Ford-Mason [6]. Their approach uses the standard named-variable presentation used in textbooks and as defined in section 1.2.1, but explicitly formalises and makes use of the notion of α-equivalence. β-reduction is then defined (non-inductively!) on the α-equivalence classes and reasoning about reduction is carried out at the level of this quotient space. Naturally, the usual first-order induction principles cannot be used in this setting; all reasoning is carried out by induction on the *rank* of λ-terms, defined by structural recursion as follows:

$$\begin{aligned} rk(e) &= 1 \\ rk(\lambda x.e) &= 1 + rk(e) \\ rk(e_1 e_2) &= 1 + rk(e_1) + rk(e_2) \end{aligned}$$

In order for this definition to provide a well-founded induction principle, one must then show that the rank of a term is strictly larger than the ranks of any of its subterms. Ford-Mason define substitution with the help of a fresh-naming function which renames any offending binders; they then show that substitution is preserved over α-equivalence and from then on work at the level of the quotient space (at which induction over the rank of an expression is still possible) in order to prove Church-Rosser. A similar approach is taken by Homeier in a proof of CR using the HOL theorem prover [10], except that the involved proof principles are different; properties are proved both by strong rule induction (which we will discuss in detail in chapter 2) and by induction on the *height* of a term.

Proof developments which axiomatise the behaviour of λ-terms under α-equivalence in this manner (including the above-mentioned contributions but see also Gordon-Melham [7] and Gordon [8]) have the advantage that the treatment of variable renaming (i.e. α-reduction) is typically built into the proof principles which one employs and hence need not come into consideration when conducting reasoning. On the other hand, working with such higher-order proof principles does not lend much formal support to the standard, first-order proof practices to which we aim to stay faithful. In addition, the wide variety of subtly different proof principles involved — which are often tailored to the particular theorem prover or system under consideration — can make such proofs difficult to follow or standardise.

All machine-checked proofs in the λ-calculus have, to the best of our knowledge, used one of the alternatives presented in the last three sections (or a hybrid)

to the first order abstract syntax presentation with one-sorted variables. The reason for this is that this presentation is widely believed to present too many problems with renaming issues to facilitate fully formal proof developments. The main contribution of this dissertation will be to show that this is not, in fact, the case — the naive presentation as used in hand proofs is powerful enough to enable one to prove equational properties of the calculus (in a fully formal, machine-checked manner) that until now have resisted formalisation in this setup.

1.3 λ^{var}: a first-order development

The aim of this project will be to formalise the proof of some important equational properties of the λ-calculus using the first-order abstract syntax presentation of the calculus with one-sorted variables ($FOAS_{VN}$) as discussed in section 1.2.1. We will now look at how one may present the λ-calculus over $FOAS_{VN}$ in such a way that it is possible to deal with the variable-renaming issues we have identified. To do so, we first examine the notions of *reduction* and *structural induction* in the calculus and then define its operations in such a way as to ensure the compatibility of these notions.

1.3.1 Reduction relations

As mentioned in section 1.1, all structural manipulations of λ-terms must be made explicitly in the form of reductions. A term is said to reduce to another term if they are related by a reduction relation and we denote this reduction by an infix arrow $\rightarrow$. The main reduction relation we will consider is called β-reduction, given schematically in the traditional λ-calculus as follows:

$$(\lambda x.e_1)e_2 \mapsto_\beta e_1[x := e_2]$$

We consider the *contextual closure* of this relation; any suitable sub-term of a given term may be β-reduced providing it is of the correct form. We will formalise this notion later by giving a formal inductive definition of β-reduction. The sub-term upon which a particular reduction step acts is called the *redex* and is said to be *contracted* under the reduction step.

Notice that the above formulation of β-reduction involves the operation of substitution which we discussed earlier. Thus in order to define β-reduction correctly we must first ensure that the substitution invoked by the reduction is itself correct, i.e. does not permit the capture of free variables.

1.3.2 Structural induction for λ-terms over $FOAS_{VN}$

The principle of structural induction for terms over the first-order abstract syntax of the λ-calculus can be stated as follows:

$$\begin{array}{ll}\forall x, y, e_1, e_2. & P(x) \wedge (P(e_1) \wedge P(e_2) \rightarrow P(e_1 e_2)) \wedge (P(e_1) \rightarrow P(\lambda y.e_1)) \\ & \Downarrow \\ & \forall e.\, P(e)\end{array}$$

Informally, to prove a property P of an arbitrary term e in a first-order abstract syntax entails showing that P holds for any atomic construct in the syntax and that if P holds for the subterms of a composite term in the syntax, then it holds for the composite term itself. For the λ-calculus, the atomic terms are the variables and the composite terms are built using the application and abstraction constructors. This proof technique comes 'for free' with any first-order abstract syntax (see Burstall [2]) without recourse to semantical interpretations or other derived proof methods. Since we wish to conduct our proofs at the level of FOAS, this proof principle will turn out to be of vital importance.

1.3.3 Substitution à la Curry

We have already seen that to define the substitution operator $-[- := -]$ by naive syntactic replacement is not sufficient to ensure correctness of the operation. In 1958, Curry [4] made the following formal, inductive definition of substitution for the λ-calculus using first order abstract syntax and one-sorted variables, assuming a linear order on the set of variable names:

$$\begin{array}{rcl} x[y := e]_{Cu} & = & \left\{ \begin{array}{ll} e & \text{if } x = y \\ x & \text{otherwise} \end{array} \right. \\ e_1 e_2[y := e]_{\mathrm{Cu}} & = & e_1[y := e]_{\mathrm{Cu}} e_2[y := e]_{\mathrm{Cu}} \\ (\lambda x.e')[y := e]_{\mathrm{Cu}} & = & \left\{ \begin{array}{ll} \lambda x.e' & \text{if } x = y \\ \lambda x.(e'[y := e]_{\mathrm{Cu}}) & \text{if } x \neq y \wedge (x \notin \mathrm{FV}(e) \vee y \notin \mathrm{FV}(e')) \\ \lambda z.(e'[x := z]_{\mathrm{Cu}}[y := e]_{\mathrm{Cu}}) & \text{otherwise — first z not in } e \text{ or } e' \end{array} \right. \end{array}$$

Observe that this notion of substitution is intuitively correct; free variables are never unintentionally captured by λ-binders of the same name, due to the final clause of the definition which renames any offending binders.

However, the Curry λ-calculus (λ^{Curry}) suffers from at least one serious problem when we attempt to perform first-order equational reasoning with it — it will not admit proof by structural induction if the proof involves the final clause of substitution for λ-abstractions. This is because making the substitution $e'[x := z]_{\mathrm{Cu}}$ destroys the sub-term property, i.e. $e'[x := z]_{\mathrm{Cu}}$ is not (in general) a sub-term of $\lambda y.e'$. This makes it impossible to apply the induction hypothesis.

Example 2 *Consider the proof of the (trivial) lemma* $e[y := y]_{\mathrm{Cu}} = e$ *by structural induction on* e.

- *Case* $e = x$*:*
  - *Subcase* $x = y$*:* $x[y := y]_{\mathrm{Cu}} = y[y := y]_{\mathrm{Cu}} = y = x$.
  - *Subcase* $x \neq y$*:* $x[y := y]_{\mathrm{Cu}} = x$ *by definition.*
- *Case* $e = e_1 e_2$*: We have* $(e_1 e_2)[y := y]_{\mathrm{Cu}} = e_1[y := y]_{\mathrm{Cu}} e_2[y := y]_{\mathrm{Cu}}$ *and so the result holds by applying the induction hypothesis to* e_1 *and* e_2.
- *Case* $e = \lambda x.e'$*: There are three subcases to consider.*
  - *Subcase* $x = y$*: Then* $(\lambda y.e')[y := y]_{\mathrm{Cu}} = \lambda y.e'$ *by the first clause of Curry's definition.*
  - *Subcase* $x \neq y \wedge y \notin \mathrm{FV}(e')$*: Then* $(\lambda x.e')[y := y]_{\mathrm{Cu}} = \lambda x.(e'[y := y]_{\mathrm{Cu}}) = \lambda x.e'$ *by induction hypothesis.*
  - *Subcase* $x \neq y \wedge y \in \mathrm{FV}(e')$*: Then by the third clause of Curry's definition,* $(\lambda x.e')[y := y]_{\mathrm{Cu}} = \lambda z.e'[x := z]_{\mathrm{Cu}}[y := y]_{\mathrm{Cu}}$*, where* z *is a 'fresh variable'. But now we cannot proceed; our intended usage of the induction hypothesis* $e'[x := x] = e'$ *is blocked because in general* $e'[y := z] \neq e'$.

Hence the principle of structural induction is broken when we work in λ^{Curry}. However, we would like to be able to perform reasoning over the first order abstract syntax of λ^{var} using standard equational (rewriting) principles and structural induction as given in section 1.3.1. We will now give a definition of λ^{var} that enables us to do so.

1.3.4 Defining substitution and reduction

Our approach to the situation outlined above is to ensure that when we reason about substitutions, the offending clause of Curry's definition is never invoked. By doing so, we ensure the integrity of the sub-term property and so enable the use of structural induction on λ-terms. We do so by making a key alteration to Curry's formulation of substitution.

Definition 2 (Substitution) *The substitution operator,* $-[- := -]$*, on* λ^{var} *is defined as follows:*

$$
\begin{array}{rcl}
x[y := e] & = & \left\{ \begin{array}{ll} e & \textit{if } x = y \\ x & \textit{otherwise} \end{array} \right. \\
e_1 e_2[y := e] & = & e_1[y := e] e_2[y := e] \\
(\lambda x.e')[y := e] & = & \left\{ \begin{array}{ll} \lambda x.e'[y := e] & \textit{if } x \neq y \wedge x \notin \mathrm{FV}(e) \\ \lambda x.e' & \textit{otherwise} \end{array} \right.
\end{array}
$$

Note that this definition, like Curry's, is *total*: this is crucial from our formal viewpoint since all function definitions in Isabelle must be total on the given domain. If substitution were not total, we would then have had to define a notion of well-founded term (à la McKinna-Pollack) and conduct reasoning

over the domain of well-founded terms only. Notice also that this definition is only *partially* correct; in the case where Curry would perform a renaming step to ensure correctness, we simply discard the offending substitution. It appears as though there are more differences between the two — however, this is not so. To see this, observe that our definition coincides with Curry's when $y = x$, and also when $y \notin \mathrm{FV}(e')$, since it should be obvious that the following property holds of substitution:

$$y \notin \mathrm{FV}(e') \to e'[y := e] = e'$$

(We define the final clause the way we do to preserve totality of the substitution function and in order to prove certain 'renaming sanity' properties of substitution such as the above. We will present the formal proof of the above result and other similar propositions in chapter 2.) The partial correctness of our substitution is not an issue as we will ensure that β-reduction can only be performed when the resulting substitution behaves correctly. In order to do so, we need to first define the notion of the *capturing variables* of free occurrences of a variable in a term:

Definition 3 (Capturing Variables) *Define the capturing variables of free occurrences of x, $\mathrm{Capt}_x(-)$, in a given term as:*

$$\begin{array}{rcl} \mathrm{Capt}_x(y) & = & \emptyset \\ \mathrm{Capt}_x(e_1 e_2) & = & \mathrm{Capt}_x(e_1) \cup \mathrm{Capt}_x(e_2) \\ \mathrm{Capt}_x(\lambda y.e) & = & \begin{cases} \{y\} \cup \mathrm{Capt}_x(e) & \textit{if } x \neq y \wedge x \in \mathrm{FV}(e) \\ \emptyset & \textit{otherwise} \end{cases} \end{array}$$

Notice that if we have the condition $\mathrm{Capt}_x(e_1) \cap \mathrm{FV}(e_2) = \emptyset$, then the associated substitution $e_1[x := e_2]$ behaves correctly, i.e. the 'incorrect' clause of substitution as we define it is never applicable. (In fact, this condition is the weakest predicate ensuring correctness of substitution.) With this definition, in place, we can now define β-reduction in such a way as to avoid variable capture:

Definition 4 (β-reduction) *β-reduction on raw λ-terms is defined inductively thus:*

$$\frac{\mathrm{Capt}_x(e_1) \cap \mathrm{FV}(e_2) = \emptyset}{(\lambda x.e_1)e_2 \dashrightarrow_\beta e_1[x := e_2]}\,(\beta) \qquad \frac{e \dashrightarrow_\beta e'}{\lambda x.e \dashrightarrow_\beta \lambda x.e'}\,(Abs_\beta)$$

$$\frac{e_1 \dashrightarrow_\beta e_1'}{e_1 e_2 \dashrightarrow_\beta e_1' e_2}\,(AppL_\beta) \qquad \frac{e_2 \dashrightarrow_\beta e_2'}{e_1 e_2 \dashrightarrow_\beta e_1 e_2'}\,(AppR_\beta)$$

As our definition of substitution does not permit the renaming of λ-binders á la Curry when invoked, we will sometimes need to explicitly perform a renaming in order to enable the β-reduction of some term. We call this operation α-reduction (or α-renaming) and define it as a reduction relation:

Definition 5 (α-reduction) *Indexed α-reduction on raw λ-terms is defined inductively thus:*

$$\frac{y \notin \mathrm{Capt}_x(e) \cup \mathrm{FV}(e)}{\lambda x.e \overset{y}{\dashrightarrow}_{i\alpha} \lambda y.e[x := y]}\,(\alpha) \qquad \frac{e \overset{y}{\dashrightarrow}_{i\alpha} e'}{\lambda x.e \overset{y}{\dashrightarrow}_{i\alpha} \lambda x.e'}\,(Abs_\alpha)$$

$$\frac{e_1 \overset{y}{\dashrightarrow}_{i\alpha} e_1'}{e_1 e_2 \overset{y}{\dashrightarrow}_{i\alpha} e_1' e_2}\,(AppL_\alpha) \qquad \frac{e_2 \overset{y}{\dashrightarrow}_{i\alpha} e_2'}{e_1 e_2 \overset{y}{\dashrightarrow}_{i\alpha} e_1 e_2'}\,(AppR_\alpha)$$

Ordinary α-reduction is then given by as the union of the $\overset{y}{\dashrightarrow}_{i\alpha}$ and we denote this relation by $\dashrightarrow_\alpha$.

We define α-equality on λ-terms as the reflexive, symmetric, transitive closure of $\dashrightarrow_\alpha$:

$$=\!=_\alpha \;=\; (\dashrightarrow_\alpha \cup \dashleftarrow_\alpha)^*$$

Notice that our definition of $=\!=_\alpha$ coincides with our informal notion of α-equality introduced in section 1.2.1. Also note that the α-contraction rule has a side-condition similar to that on the β-contraction rule in order to ensure correctness of the corresponding substitution, and to ensure that no 'extra' variables are captured by changing the binder λx to λy.

As one might expect, the introduction of side conditions on our β-reduction relation, and on α-renaming, necessarily entails the introduction of similar conditions on the lemmas in our proof development which considerably complicate the proofs. It seems that the implicit assumption of other authors who have undertaken automated proof developments in λ-calculus [Nipkow] is that automated reasoning with such heavily conditioned rules and proof goals is not practically feasible. One objective of this paper will be to show that this is not, in fact, the case — the proof of equational properties in this calculus, while indeed substantially more involved, is nevertheless manageable.

1.3.5 (Connections with) Barendregt's Variable Convention

In his book *The Lambda Calculus: Its Syntax and Semantics* [1], the standard reference for the λ-calculus, Barendregt makes the following remarks—somewhat notorious within the literature—which we refer to collectively as the **Barendregt Variable Convention** (BVC):

> "**2.1.12. CONVENTION.** Terms that are α-congruent are identified. So now we write $\lambda x.x \equiv \lambda y.y$, etcetera.
>
> **2.1.13. VARIABLE CONVENTION.** If $M_1, ..., M_n$ occur in a certain mathematical context (e.g. definition, proof) then in these terms all bound variables are chosen to be different from the free

variables.

2.1.14. MORAL. Using conventions 2.1.12 and 2.1.13 one can work with λ-terms in the naive way."

Said differently, Barendregt's stance is that the problem of avoiding free variable capture when performing substitution can always be avoided provided one can rename terms 'for free' as necessary at any point in the proof. Hence in his pen-and-paper proofs, substitution and β-reduction on terms are performed under the assumption that such problems never occur. However, from a formal perspective this is clearly inadequate; it essentially amounts to ignoring the issues with which we are concerned.

However, we will show in the course of our proof development that the assumptions made by Barendregt in the above are actually correct; the operation of β-reducing any term can be largely separated from the issue of variable-renaming (given by α-reduction) and can proceed as one would expect in a pen-and-paper proof. (This statement needs considerable justification and will be expounded on in chapters 2 and 3.)

1.3.6 Isabelle: a generic theorem prover

Isabelle is a popular generic theorem prover developed at the University of Cambridge and TU Munich. It can be instantiated with a number of object-logics of which HOL (higher-order logic) is probably the most widely used. Theorem-proving in Isabelle is *tactic-based*: theorems are stated by the user as proof goals (possibly involving one or more *premises*), which are then refined or broken down into smaller subgoals by successively invoking tactics which apply the rules of the logic. Such rules are either supplied 'built in' by the definition of the logic or derived from lemmas proven earlier by the user. Once the subgoals in the proof become trivial they may be solved outright.

Isabelle also places a number of automatic tactics at the disposal of the user which use either simplification (i.e. equational rewriting) or classical reasoning, or a combination of both, to search for a proof or refinement of a goal. Such tactics are particularly useful when dealing with lemmas which have lengthy but reasonably trivial proofs, as such goals can often be proved entirely automatically.

We use Isabelle to implement our proofs because it provides a high degree of automation (via the simplifier and classical reasoner), but also because the recursive style of definition (á la ML) is particularly suitable for our setup, and because Isabelle has an extensive library of built-in theories and tactics.

2. Confluence of the raw and real λ-calculi

We now turn to a detailed examination of the Isabelle proof of a substantial result in the equational theory of the λ-calculus — *confluence*:

Definition 6 (Confluence) *A relation $\rightarrow_R$ is said to have the diamond ($\diamond$) property if:*

$$x \rightarrow_R y \,\&\, x \rightarrow_R y' \text{ implies } \exists z.\ y \rightarrow_R z \,\&\, y' \rightarrow_R z$$

If the transitive-reflexive closure of $\rightarrow_R$ (denoted by $\twoheadrightarrow_R$) has the diamond property then $\rightarrow_R$ is said to be confluent.

The primary goal of our this chapter will be the proof of confluence of $\dashrightarrow_{\alpha\cup\beta}$ in λ^{var}. (We will then use this property to derive the usual confluence property of the $\rightarrow_\beta$ relation in λ^{real}.)

2.1 Getting started in Isabelle

There are several possible interfaces through which one can use Isabelle but the one we use is ProofGeneral on XEmacs (available from http://zermelo.dcs.ed.ac.uk/~ proofgen/) for its ease of use and proof script management facilities. Having installed Isabelle and ProofGeneral on your system (for a guide refer to [20]), both programs are invoked automatically when the user opens a file in XEmacs with the appropriate filename extension.

Isabelle distinguishes between *theory files* (denoted by the extension .thy), which contain definitions of types, constants, relations *et cetera*, and *proof script files* (denoted by the extension .ML), which contain the actual tactic scripts for proving the various lemmas of our proof development. A theory file has at most one associated proof script file which must share the same name so, e.g., the proof scripts in MyProof.ML would use the definitions contained in MyProof.thy. However, a theory may itself depend on several subtheories so, e.g., we can make all the definitions and lemmas proved in the theory MyProof available to the new theory MyNextProof simply by adding an appropriate dependency to MyNextProof.thy.

In accordance with this bottom-up philosophy to building proofs, we start in Isabelle by defining the basics of λ^{var} as given in chapter 1 and by proving increasingly complex lemmas in order to prove our main result. However, note that determining what these lemmas should be does require some deconstruction of the original goal. Thus a combination of bottom-up and top-down methodologies is necessary for any large proof development.

For the interested reader, Appendix A contains code listings for all the theory files used during our proof development as well as statements of all of the individual lemmas proved. The full tactic scripts are not included here but are available from the author's homepage at http://www.dcs.ed.ac.uk/jjb/.

2.2 Basics of λ^{var}

We start by creating a new file Lambda.thy which will contain our definitions of the λ^{var}-calculus and associated notions (free variables of a term, for example). Initially we import the standard Isabelle/HOL theory library 'Main' as follows:

```
Lambda = Main + Variables +
```

The theory 'Variables' contains an implementation of the set of one-sorted variables as an isomorphic copy of the natural numbers. In fact, the main reason for using this isomorphism rather than using the natural numbers directly is that the syntax we use for substitution `-[-:=-]` already has a definition on the natural numbers in Isabelle and so using a new datatype generated from the naturals avoids the problems caused by overloading operators. In any case importing 'Variables' gives us a type of one-sorted variables which is called `var`.

The inductive definition of a λ-term can then be implemented as an ML-style datatype `lterm`:

```
datatype lterm = Var var | ``$'' lterm lterm (infixl 200)
   | Abs var lterm
```

Note that we use the infix symbol `$` for application; the use of the `infixl` indicates that application associates to the left, and the number 200 is the precedence of the operator. We can then define the free variables of an `lterm` by structural recursion on the datatype exactly as given in chapter 1:

```
consts
FV ::  lterm => var set
primrec
FV_Var ``FV(Var x) = {x}''
FV_App ``FV(e1 $ e2) = (FV(e1) Un FV(e2))''
FV_Abs ``FV(Abs x e) = (FV(e) - {x})''
```

Note that before we can define the function `FV(-)` we must first declare its type in the `consts` section of the theory file. In the case of `FV(-)` we need a function which takes a λ-term and returns the set of free variables which occur in it, so its type is naturally `lterm => var set`. Then the keyword `primrec` indicates the declaration of a so-called *primitive recursive* function which is defined by structural recursion on the constructors of a datatype. Each rule is given an identifier on the left hand side and the rules themselves are contained in quotation marks. We then go on to define BV(−), $\mathrm{Capt}_x(-)$, and substitution in a manner very similar to FV(−) (not given here, but see Appendix B).

In the corresponding proof script file Lambda.ML, we can now start to prove some basic properties of the λ^{var}-calculus.

2.2.1 A First Proof

For our first proof in Isabelle we return to our example of section 1.3.3; the proof of the 'Proposition of Renaming Sanity': $e[x := x] = e$. We state this as a goal for proof in Isabelle as follows:

```
Goal ``e[x:=Var x] = e'';
```

Isabelle's response is to make the goal the top level of the proof state:

```
Level 0 (1 subgoal)
e[x:=Var x] = e
1.  e[x:=Var x] = e
```

We can refine a subgoal in Isabelle by deploying one of its many *tactics*, which may incorporate some degree of automation. We saw in chapter 1 that the obvious way to proceed is by structural induction on the λ-term e. Isabelle provides a tactic `induct_tac` for just this purpose and we invoke it with the command `by()`, stating the term over which the induction is to take place and also the subgoal upon which the tactic is to operate:

```
by(induct_tac ``e'' 1);

Level 1 (3 subgoals)
e[x:=Var x] = e
1.  !!var.  Var var[x:=Var x] = Var var
2.  !!lterm1 lterm2.
[| lterm1[x:=Var x] = lterm1; lterm2[x:=Var x] = lterm2 |]
==> (lterm1 $ lterm2)[x:=Var x] = lterm1 $ lterm2
3.  !!var lterm.
lterm[x:=Var x] = lterm
==> Abs var lterm[x:=Var x] = Abs var lterm
```

Let us examine the new Isabelle proof state. The *level* of the state is simply the number of tactic invocations used so far. The numbered subgoals are listed after the statement of the main goal — as one would expect, there are 3 subgoals corresponding to the 3 cases of the structural induction. Note that the second and third subgoals contain *premises* (separated from the required conclusion by `==>`) which are just the inductive hypotheses for the induction. Multiple premises are separated by semicolons. Note also that the tactic also introduces (meta-level) universal quantification over the terms in the subgoals, denoted in Isabelle's logical framework by `!!`.

Each of these subgoals is fairly trivial as they can be solved simply by expanding the definition of substitution. To do this we can either use a *simplification* or rewriting tactic such as `Asm_simp_tac` to each subgoal in turn, or we can apply `Auto_tac`, which applies the simplifier to all of the subgoals of the proof state:

```
by(Auto_tac);

Level 2
e[x:=Var x] = e
No subgoals!
```

As there are no remaining subgoals, this proof is now finished and we can store it for future use using the `qed` command:

```
qed ``renaming_sanity_1'';
```

We can now use the proposition `e[x:=Var x] = e` in future proofs by referring to it explicitly by name, or we can add it to the set of default simplification rules with the command `Addsimps[renaming_sanity_1]`.

Hopefully the reader will have gained some insight by now into how one might go about conducting a proof in Isabelle/HOL. Although the proof we have just considered required only 2 tactic invocations and was almost entirely automatic, the proofs of later results are very involved and can require many dozens of tactic invocations. We will therefore only concentrate on the interesting points of the Isabelle development from now on and refer the interested reader to the complete development available from the author's homepage for full details.

We can prove many propositions similar to the one above by employing the same technique of structural induction followed by automatic simplification. We call such results propositions of 'renaming sanity' because they correspond to properties of terms (or sets of variables) under substitution which we would intuitively expect to hold. A selection of these are listed below. (Note that Isabelle uses `:` for set membership and `~:` for non-membership.)

Proposition 1 (Renaming Sanity) *For all raw λ-terms* `e, e'`*:*

- `x ~: FV e ==> e[x:=e'] = e`
- `y ~: FV e ==> e[x:=Var y][y:=Var x] = e`
- `Capt x e <= BV e`
- `x : FV (e[y:=e']) ==> x : FV e' | x : FV e`
- `[| x : FV e; x ~= y |] ==> x : FV (e[y:=e'])`
- `y ~: FV e ==> Capt y e = {}`
- `[| y ~: FV e; y ~= z |] ==> y ~: FV (e[x:=Var z])`

Proof All automatic by structural induction on `e`. □

2.3 Substitution lemmas

Armed with our low-level propositions, our first major proof target is a version of Barendregt's Substitution Lemma. Recall however that our version of substitution is only partially correct and so we must impose side-conditions on our corresponding lemma to ensure correctness of all the involved substitutions.

Lemma 2 (Substitution) *For any λ-terms `e1, e2, e3` and variables `x,y` we have:*

$$\begin{array}{l} y \notin \mathrm{FV}(e_2) \wedge x \neq y \wedge \\ (\mathrm{Capt}_x(e_3) \cap \mathrm{FV}(e_2) = \emptyset) \wedge (\mathrm{Capt}_y(e_1) \cap \mathrm{FV}(e_3) = \emptyset) \wedge \\ (\mathrm{Capt}_x(e_1) \cap \mathrm{FV}(e_2) = \emptyset) \wedge (\mathrm{Capt}_x(e_1[y := e_3]) \cap \mathrm{FV}(e_2) = \emptyset) \\ \Downarrow \\ e_1[y := e_3][x := e_2] = e_1[x := e_2][y := e_3[x := e_2]] \end{array}$$

The Isabelle formulation of this result is as follows:

```
[| Capt x e3 Int FV e2 = {}; Capt y e1 Int FV e3 = {};
   Capt x e1 Int FV e2 = {}; Capt x (e1[y:=e3]) Int FV e2 = {};
   y ~: FV e2; y ~= x |] ==>
e1[x:=e2][y:=e3[x:=e2]] = e1[y:=e3][x:=e2]
```

Furthermore, if we have that $x \notin \mathrm{FV}(e1)$ then the following alternative result holds (we omit the mathematical notation):

```
[| Capt x e3 Int FV e2 = {};  Capt x (e1[y:=e3]) Int FV e2 = {};
   Capt y e1 Int FV e3 = {};  Capt y e1 Int FV (e3[x:=e2]) = {};
   x ~: FV e1; y ~= x |] ==>
e1[y:=e3[x:=e2]] = e1[y:=e3][x:=e2]
```

Proof Both results follow by structural induction in the term `e1`. The variable and application cases are straightforward and the abstraction case follows by a painstaking case-splitting on variable names and the subcases for correctness of substitution. □

2.4 Diamond property of Parallel β up to BCF-Initiality

We will now move on to consider the notion of reduction in our calculus. Although our eventual goal is confluence of $-\!\!\twoheadrightarrow_{\alpha\cup\beta}$, we will start by considering β-reduction separately and first prove a diamond property of a so-called *parallel reduction relation* $-\!\!\mathrel{\#}\!\!\twoheadrightarrow_\beta$ which can contract any number of disjoint β-redexes in parallel:

Definition 7 (Parallel β-reduction) *Parallel β-reduction on raw λ-terms is defined inductively thus:*

$$\frac{}{x \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} x} \qquad \frac{e \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e'}{\lambda x.e \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} \lambda x.e'} \qquad \frac{e_1 \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e_1' \quad e_2 \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e_2'}{e_1 e_2 \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e_1' e_2'}$$

$$\frac{e_1 \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e_1' \quad e_2 \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e_2' \quad \mathrm{Capt}_x(e_1') \cap \mathrm{FV}(e_2') = \emptyset}{(\lambda x.e_1) e_2 \mathrel{-\!\!\#\!\!\twoheadrightarrow_\beta} e_1'[x := e_2']}$$

The Isabelle formulation of this relation is as follows:

```
inductive par_beta
intrs
 var   "Var x -|>B Var x"
 abs   "s -|>B t ==> Abs x s -|>B Abs x t"
 app   "[| s -|>B s'; t -|>B t' |] ==> s $ t -|>B s' $ t'"
 beta  "[| s -|>B s'; t -|>B t'; (Capt x s') Int FV(t') = {}|]
          ==> (Abs x s) $ t -|>B s'[x:=t']"
```

To define a relation inductively in Isabelle we use the keyword `inductive` followed by the relation identifier; then the (named) introduction rules for the relation follow the keyword `intrs`. The type for the relation (in this case `(lterm * lterm) set`) must also be defined in the `consts` section of the theory file and appropriate translations for the syntax `-|>B` added. Once a relation has been defined in this way the introduction rules may be added to Isabelle's classical reasoner using a command such as `AddSIs par_beta.intrs`.

We now proceed to prove some useful properties of parallel β-reduction.

Lemma 3 (Variable Monotonicity Under $-\!\!\twoheadrightarrow_\beta$) *For any λ-term t:*

- `t -|>B t' ==> FV t' <= FV t`
- `t -|>B t' ==> BV t' <= BV t`

Proof First note there is nothing surprising about this result as, clearly, `-|>B` cannot create new bound or free variable names. Both proofs can be conducted either by structural induction in `t` or, more efficiently, by *rule induction* over the contraction `t -|>B t'`. The rule induction principle (`par_beta.induct`), which is supplied automatically by Isabelle, is complex to state but essentially involves showing that the desired property is preserved over applications of the introduction rules. We illustrate this technique with an example of its application for the first monotonicity result above:

```
Goal ``t -|>B t' ==> FV t' <= FV t''
by(etac par_beta.induct 1);

Level 1 (4 subgoals)
t -|>B t' ==> FV t' <= FV t
 1. !!x. FV (Var x) <= FV (Var x)
 2. !!s t x. [| s -|>B t; FV t <= FV s |]
             ==> FV (Abs x t) <= FV (Abs x s)
 3. !!s s' t t'.
      [| s -|>B s'; FV s' <= FV s; t -|>B t'; FV t' <= FV t |]
      ==> FV (s' $ t') <= FV (s $ t)
 4. !!s s' t t' x.
      [| s -|>B s'; FV s' <= FV s; t -|>B t'; FV t' <= FV t;
         Capt x s' Int FV t' = {} |]
      ==> FV (s'[x:=t']) <= FV (Abs x s $ t)
```

Since the induction principle is supplied as an *elimination rule* by Isabelle we use the tactic `etac` to perform elimination resolution with the subgoal.

As expected, the induction yields one subgoal for each introduction rule of the relation. Premises occurring in an introduction rule are translated into appropriate induction hypotheses in the premises of the corresponding subgoals.

It should be obvious that the first three subgoals are all immediate by induction and indeed these can be solved automatically by Isabelle. The fourth goal follows by the induction hypothesis and two of our earlier 'renaming sanity' propositions concerning the behaviour of FV(−) under substitution. □

Lemma 4 (Substitutivity of Parallel-β) *For any λ-terms s, t:*

```
[| s -|>B s'; t -|>B t'; Capt x s Int FV t = {};
   Capt x s' Int FV t' = {} |]
==> s[x:=t] -|>B s'[x:=t']
```

Proof The most efficient approach is by rule induction in `s -|>B s'` although structural induction on `s` is also possible. (The proof burden is actually the same in either case but the case splitting is a little kinder when rule induction is used.) In order to ensure that the induction takes place over all of the premises the goal must first be stated in Isabelle as:

```
s -|>B s' ==> t -|>B t' --> Capt x s Int FV t = {} -->
Capt x s' Int FV t' = {} --> s[x:=t] -|>B s'[x:=t']
```

which, once proved, can easily be manipulated to have the form in the statement of the lemma above.

Rule induction again yields four subgoals of which the first three follow fairly straightforwardly by induction. Some work however is needed to show that the premises of the lemma are also preserved over the induction. For the final subgoal, we need to substantiate:

```
Level 25 (1 subgoal)
1. !!s s' ta t'a xa.
      [| s -|>B s';
          t -|>B t' -->
          Capt x s Int FV t = {} -->
          Capt x s' Int FV t' = {} --> s[x:=t] -|>B s'[x:=t'];
          ta -|>B t'a;
          t -|>B t' -->
          Capt x ta Int FV t = {} -->
          Capt x t'a Int FV t' = {} --> ta[x:=t] -|>B t'a[x:=t'];
          Capt xa s' Int FV t'a = {}; t -|>B t';
          Capt x (Abs xa s $ ta) Int FV t = {};
          Capt x (s'[xa:=t'a]) Int FV t' = {} |]
       ==> (Abs xa s $ ta)[x:=t] -|>B s'[xa:=t'a][x:=t']
```

which needs considerable work to prove. There are 3 subcases to consider: $xa = x$, $xa \neq x \wedge xa \in \mathrm{FV}(t)$, and $xa \neq x \wedge xa \notin \mathrm{FV}(t)$. This is achieved in

Isabelle by using the tactic `case_tac` with an appropriate predicate, e.g. for the first subcase we use:

```
by(case_tac‘‘xa = x’’ 1);
```

Each subcase splits into further subcases and uses a Substitution Lemma. The number of case-splits and level of detail required to prove substitutivity of $-\!\!\twoheadrightarrow_\beta$ makes this one of the most challenging technical results of the project. □

In order to prove the diamond property of $-\!\!\twoheadrightarrow_\beta$, we follow a proof technique due to Takahashi [16] which employs the *complete development* β-relation, $--\!\!\twoheadrightarrow_\beta$, so-called because it attempts to contract *all* of the redexes in a term:

Definition 8 (Complete β-development) *Complete β-development on raw λ-terms is defined inductively thus:*

$$\frac{}{x \dashrightarrow_\beta x} \qquad \frac{e \dashrightarrow_\beta e'}{\lambda x.e \dashrightarrow_\beta \lambda x.e'} \qquad \frac{e \dashrightarrow_\beta e'}{xe \dashrightarrow_\beta xe'} \qquad \frac{e_1e_2 \dashrightarrow_\beta e' \quad e_3 \dashrightarrow_\beta e'_3}{(e_1e_2)e_3 \dashrightarrow_\beta e'e'_3}$$

$$\frac{e_1 \dashrightarrow_\beta e'_1 \quad e_2 \dashrightarrow_\beta e'_2 \quad \mathrm{Capt}_x(e'_1) \cap \mathrm{FV}(e'_2) = \emptyset}{(\lambda x.e_1)e_2 \dashrightarrow_\beta e'_1[x := e'_2]}$$

and the corresponding Isabelle definition is:

```
inductive comp_dev
intrs
 var   "Var x ->CD Var x"
 abs   "s ->CD t ==> Abs x s ->CD Abs x t"
 appV  "s ->CD t ==> Var x $ s ->CD Var x $ t"
 appA  "[| (t1 $ t2) ->CD t'; s->CD s' |]
        ==> ((t1 $ t2) $ s) ->CD (t' $ s')"
 beta  "[| s ->CD s'; t ->CD t'; (Capt x s') Int FV(t') = {} |]
        ==> (Abs x s) $ t ->CD s'[x:=t']"
```

It is not too difficult to see that $--\!\!\twoheadrightarrow_\beta$ in fact is only defined for those λ-terms in which all (potential) redexes satisfy the side condition on the β-contraction rule. We can show that any term has a complete development provided it is of a form which, in line with earlier work ([17], [18]), we refer to as *Barendregt Conventional Form* (BCF):

Definition 9 (Barendregt Conventional Form) *A raw λ-term is said to be a BCF term if all of the bound variables occurring in it are different and furthermore are different from all of the free variables.*

In Isabelle, we define BCF via an auxiliary predicate $UB(-)$ on terms which is true iff all of the bound variables in the term are different:

```
primrec
 UB_Var "UB(Var x) = True"
```

```
 UB_App "UB(e1 $ e2) = (UB(e1) & UB(e2) & (BV(e1) Int BV(e2) = {}))"
 UB_Abs "UB(Abs x e) = (UB(e) & x~:BV(e))"

defs
 BCF "BCF(e) == (UB(e) & (FV(e) Int BV(e) = {}))"
```

It should hopefully be clear that renaming a term to an equivalent BCF term before reducing it is the 'natural' thing to do when working under the Barendregt Variable Convention. Later, we will show formally that it is always possible to do so via a sequence of α-reductions. For now, we prove two important lemmas concerning $-\!\!\twoheadrightarrow_\beta$ and $-\!\!\!\!\twoheadrightarrow_\beta$:

Lemma 5 (BCF Enables $-\!\!\twoheadrightarrow_\beta$) *For all raw λ-terms s:*

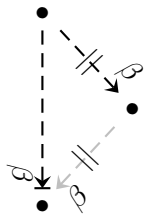
```
BCF(s) ==> (EX t. s ->CD t)
```

(Note that `EX` *in Isabelle/HOL stands for existential quantification.)*

Proof By structural induction in `s`. When `s` is an application, `s = s1 $ s2`, we need to case-split on the constructors of `s1` which can be done using the command `by(case_tac ''s1'' 1);`. (In this case `case_tac` automatically recognises `s1` as a term rather than a predicate as in earlier proofs.) The only case which does not follow straightforwardly by induction is the case when `s1` is an abstraction (and hence `s` is a redex), in which case we use some simple properties about disjointness of variable names in BCF terms and a variable monotonicity result for `->CD` similar to that for `-|>B` in Lemma 2. (In fact, we can use the same lemma and just prove that `->CD <= -|>B`, which can be proved automatically after rule induction on `->CD`.) □

Our next result is best understood in our diagram notation(taken from [17]) which should be read as follows: assume the universal quantification of the terms given as solid circles and reductions given as black arrows, and conclude the existence of the terms given as empty circles and reductions given as shaded arrows.

Lemma 6 (CD / Parallel-β Triangle)




```
s ->CD t' ==> ALL t. s -|>B t --> t -|>B t'
```

(Note that `ALL` *in Isabelle/HOL stands for universal quantification. Also note that* `-->` *stands for implication; this should not be confused with* `==>` *which separates the premises and conclusion of an Isabelle goal.)*

Proof

```
by(etac comp_dev.induct 1);
by(ALLGOALS strip_tac);
by(ALLGOALS (blast_tac (claset() addSIs[par_beta_subst])));
```

□

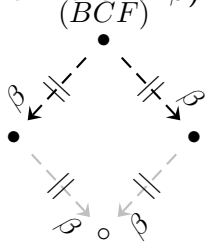
The Isabelle proof as given above is very short but contains some interesting features. The first tactic is just the application of rule induction for `->CD` as we have seen before. The second line uses the special HOL tactic `strip_tac` which removes all object-level implications and universal quantifications and moves them to the level of the logical framework, allowing easier manipulation of the subgoal. For example, applying `strip_tac` to the main goal above would yield:

```
[| s ->CD t'; s -|>B t |] ==> t -|>B t'
```

The *tactical* ALLGOALS applies the specified single-goal tactic to all of the subgoals in the current proof state. Tacticals are functions for combining tactics; Isabelle provides many built-in tacticals but new ones may also be constructed by the user. (All of the proofs presented here use only the standard tactics and tacticals, however.)

Lastly, we call Isabelle's *classical reasoner* on all of the subgoals using the tactic `blast_tac`. The proof does not work however unless the reasoner 'knows about' the substitutivity lemma for `-|>B` which has the Isabelle identifier `par_beta_subst` (Lemma 3). Therefore we add the lemma as an introduction rule to the set of rules available to the reasoner (called the *claset*). We do this within the tactic invocation and not as a separate command because we wish to add `par_beta_subst` to the claset for the duration of this tactic only. Therefore we do not permanently alter the claset for future proofs. This is the safest approach when conducting a large development, as it is not always possible to determine what the effect on the classical reasoner will be (if any) when adding new rules to its claset. In the worst case, adding new rules can lead to non-termination in the automatic tactics employed by the reasoner. It should never lead to unsoundness since only rules built-in to the logic or derived from previously proven lemmas may be added to the claset.

Lemma 7 (Diamond property of Parallel-β)




```
[| BCF s; s -|>B t; s -|>B t' |]
==> EX s'. t -|>B s' & t' -|>B s'
```

Proof The result follows in an aesthetically pleasing manner from the two previous lemmas. By Lemma 4, since `s` is a BCF it has a complete development,

i.e. for some term `u`, `s ->CD u`. By two applications of Lemma 5, we then have that `t -|>B u` and `t' -|>B u` thereby giving the required existential witness for the proof. The Isabelle proof is similarly concise:

```
by(dtac BCF_implies_exists_CD 1);
by(etac exE 1);
by(dtac par_beta_CD_triangle 1);
by(Blast_tac 1);
```

The tactic `dtac` performs *destruct resolution* with the subgoal; it matches the first premise of the specified rule with one of the subgoal premises and replaces it with the conclusion of the rule. In this case the subgoal premise `BCF(s)` is replaced with the conclusion `EX u.  s ->CD u` of Lemma 4 (bound to the identifier `BCF_implies_exists_CD`). The elimination rule `exE` is then applied to remove the object-level quantifier. To apply Lemma 5 we then perform destruct-resolution again with the new subgoal premise `s ->CD u` to give the new subgoal:

```
[| s -|>B t; s -|>B t'; ALL t. s -|>B t --> t -|>B ta |]
 ==> EX s'. t -|>B s' & t' -|>B s'
```

which can now be solved automatically by an invocation of the classical reasoner (`Blast_tac`) with the default claset. □

One interesting point about our use of Takahashi's proof methodology employing the complete development 'trick' is that it provides a considerable practical simplification over a direct proof of the diamond property of $\twoheadrightarrow_\beta$ as well as being more aesthetically pleasing. The reason for this is that knowing that a term has a complete development immediately entails, by the induction hypothesis, that no β-contraction on the term can ever be blocked (i.e. for no redex $(\mathrm{L}xe_1)e_2$ do we have $\mathrm{Capt}_x(e_1) \cap \mathrm{FV}(e_2) \neq \emptyset$). Therefore the bulk of the proof burden is to show that BCF terms have a complete development, which as we have seen is straightforward by structural induction. A direct proof of $\diamond(\twoheadrightarrow_\beta)$, on the other hand, would require us to justify the enabling of the reductions needed to close the diagram by reasoning about the behaviour of BCF terms under reduction. This entails substantial complication in our setup.

In [13], Nipkow proves the diamond property of a parallel-β relation for the λ-calculus using de Bruijn indices, both directly and via Takahashi's method. He postulates:

> "[The question is whether] anything has been gained by the formalization of Takahashi's proof. The answer seems to be no ... the ingenuity of her approach is wasted in the presence of mindless search procedures, aka tactics."

Although this is true for Nipkow's development — his use of de Bruijn indices allow the many case distinctions in the direct proof to be dealt with automatically — we feel, for the reasons given above, that this is *not* true in the case

of our own development. This is mainly because our use of conditional rules causes difficulties for Isabelle's automatic tactics. (We will return to this topic later in chapter 4.)

2.5 Weak α_0 / β commutativity

The aim of this section is to develop our theory of α-reduction in λ^{var} in order to arrive at a result showing that parallel-β reduction commutes with α-renaming under suitable conditions. We start by translating our definition of α-reduction (Definition 5) into an appropriate Isabelle relation.

```
inductive sq_alpha
intrs
  alpha  "[|x~=y; y~:(FV(e) Un Capt x e)|] ==>
          ((Abs x e),y) ->sA (Abs y (e[x:=Var y]))"

inductive i_alpha
intrs
 index  "(s,y) ->sA t ==> (s,y) ->iA t"
 aappL  "(s,y) ->iA t ==> (s$u,y) ->iA t$u"
 aappR  "(s,y) ->iA t ==> (u$s,y) ->iA u$t"
 aabs   "(s,y) ->iA t ==> ((Abs x s),y) ->iA (Abs x t)"
```

Note that we have used two inductive relations here in order to separate out the actual α-renaming step from the contextual closure of the relation but this is, as such, not necessary for conducting proofs and we could equally have used a single inductive definition as for `-|>B`. The non-indexed α-relation and its inverse are defined inductively via `->iA` in the obvious way as `alpha` and `rev_alpha` respectively (with the infix syntax `->A` and `A<-`). We can thus easily define α-equality, $==_\alpha$ and the reflexive-transitive closure of α-reduction, $-\!\!\twoheadrightarrow_\alpha$, via the following syntax translation:

```
translations
"s ->>A t" == "(s,t) : alpha^*"
"s =A= t"  == "(s,t) : (alpha Un rev_alpha)^*"
```

where `^*` in Isabelle denotes the Kleene-closure operator.

With these definitions in place we can start to prove some important properties about α-reduction:

Lemma 8 ($-\!\!\rightarrow_\alpha$-symmetry)

For all raw λ-terms `e,e'`:

```
e ->A e' ==> e' ->A e
```

Proof We prove the equivalent symmetry property for `->iA`; symmetry of `->A` follows automatically from it.

```
Goal "((e,y) ->iA e') ==> (EX x. ((e',x) ->iA e))";
by(etac i_alpha.induct 1);
```

All cases of the rule induction can be solved automatically apart from the case `(e,y) ->sA e'`, i.e. when `e` is contracted. To generate the proper inductive hypotheses we must invoke a 'generation lemma' which we create via the `mk_cases` function as follows:

```
val sq_alpha_E = sq_alpha.mk_cases "(e,y) ->sA e'";
```

This generation lemma, which is supplied as an elimination rule, can then be applied to the premise `(e,y) ->sA e'` using `etac sq_alpha_E` in the subgoal to yield the appropriate induction hypotheses in the subgoal:

```
[| x ~= y; e = Abs x ea; e' = Abs y (ea[x:=Var y]); y ~: FV ea;
   y ~: Capt x ea |]
 ==> EX x. (e', x) ->iA e
```

The remaining proof burden is then non-trivial but can be resolved using simplification and a number of our "renaming sanity" propositions showing that side-conditions enabling the "reverse" α-reduction are enabled. □

Lemma 9 ($==_\alpha$ is Directable) *For all raw λ-terms `e, e'`:*

```
e =A= e' ==> e' ->>A e
```

Proof This property can easily be seen by symmetry of $\dashrightarrow_\alpha$ since any path of α-reductions can be made unidirectional by a finite number of applications of symmetry. A blackboard proof involving a sketch of a path from `e` to `e'` would be almost direct. In Isabelle, however, we need to prove the property by induction in the reflexive-transitive generation of `=A=` (recall that although `=A=` is symmetric as well as reflexive and transitive, it is defined as the reflexive-transitive closure of `(->A Un A<-)`, so we need to induct over this first and then use the symmetric definition). This is supplied as a built-in induction principle `rtrancl_induct` in Isabelle:

```
Goal "(e =A= e') ==> (e' ->>A e)";
by(etac rtrancl_induct 1);
by(Full_simp_tac 2);

Level 2 (2 subgoals)
e =A= e' ==> e' ->>A e
 1. e ->>A e
 2. !!y z. [| e =A= y; y ->A z | y A<- z; y ->>A e |]
             ==> z ->>A e
```

When we perform this kind of induction there are always two cases; the reflexive (or base) case and the transitive (or inductive) case. Since `->>A` is reflexive by definition the first subgoal is trivial. For the transitive case the inductive hypothesis is `y ->>A e`; we need to substantiate that in fact `z ->>A e` given the disjunction in the subgoal premises. To proceed, we automatically solve the first subgoal and perform *disjunction elimination* on the second:

```
by(Fast_tac 1);
by(etac disjE 1);

Level 4 (2 subgoals)
e =A= e' ==> e' ->>A e
 1. !!y z. [| e =A= y; y ->>A e; y ->A z |] ==> z ->>A e
 2. !!y z. [| e =A= y; y ->>A e; y A<- z |] ==> z ->>A e
```

For the first subgoal we have that `z ->A y` by Lemma 8; for the second we also have that `z ->A y` by definition of `A<-`. The conclusion follows in both cases by applying the induction hypothesis. □

Lemma 10 (Substitutivity of (indexed) $\dashrightarrow_\alpha$) *For all raw λ-terms `s, t:`*

- `[| (s,y) ->iA t; Capt x s Int FV(u) = {};`
  `Capt x t Int FV(u) = {}; x~=y |]`
  `==> (s[x:=u],y) ->iA t[x:=u]`
- `[|(s,y) ->iA t; x~=y|] ==> (e[x:=s],y) ->iA e[x:=t]`

Proof Both properties follow by rule induction on `(s,y) ->iA t`. The difficult case for both results is when `s` is a redex which is contracted, in which case the conclusion follows by some complex but uninteresting reasoning about variable names showing the satisfaction of the side-conditions necessary for the appropriate contractions. For the first result (which is much harder than the second) a Substitution Lemma is also required. □

We are almost ready now to prove the commutativity result which we seek. However, we are not *quite* ready; observe that if we make a bad choice for the new bound variable name when we α-reduce a term, there is a possibility of invalidating a β-redex (and hence the two reductions cannot commute):

$$\begin{array}{lll}(\lambda x.(\lambda y.x))z & \dashrightarrow_\beta & \lambda y.z \\ (\lambda x.(\lambda y.x))z & \dashrightarrow_\alpha & (\lambda x.(\lambda z.x))z\end{array}$$

and note that $(\lambda x.(\lambda z.x))z$ does *not* β-reduce because the side condition is no longer satisfied. However, this clearly cannot happen if the name chosen for the *alpha*-reduction does not appear anywhere in the term. Thus we introduce the notion of *fresh-naming α-reduction*, $\dashrightarrow_{\alpha_0}$:

Definition 10 (α_0-reduction) *Fresh-naming α-reduction, $\dashrightarrow_{\alpha_0}$, is defined on raw λ-terms as follows:*

$$e \dashrightarrow_{\alpha_0} e' \Leftrightarrow^{def} \exists z.e' \overset{e}{\dashrightarrow}_{i\alpha_0} z \wedge z \notin \mathrm{FV}(e) \cup \mathrm{BV}(e)$$

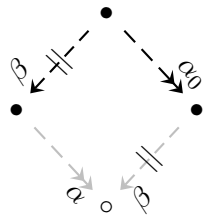
We define this as a relation `->A0` in Isabelle via an indexed version of $\dashrightarrow_{\alpha_0}$, `->iA0`:

```
inductive i_alpha0
intrs
 ialpha0 "[|(s,y) ->iA t; y~:(FV(s) Un BV(s))|]
           ==> (s,y) ->iA0 t"
inductive alpha0
intrs
 strip   "(s,y) ->iA0 t ==> s ->A0 t"
```

The transitive-reflexive closure of `->A0` is denoted `->>A0`.

We are now ready to prove the main result of this section.

Lemma 11 (Weak α_0 / Parallel-β Commutativity)




```
[| s ->A0 t; s -|>B t' |] ==> (EX s'. t -|>B s' & t' ->>A s')
```

Proof By induction in the reflexive-transitive generation of `->>A0`. The reflexive step is trivial. For the inductive step we need to show that the lemma holds with `->A0` in place of `->>A0` and the case then goes through easily by induction. To do so, we first prove the following lemma:

```
(s,y) ->iA0 t ==> ALL t'. s -|>B t' -->
(EX s'. t -|>B s' & (t',y) ->>iA s')
```

which once proved can easily be 'lifted' to the non-indexed versions `->A0` and `->>A` and put into the required form above. The lemma follows by rule induction on `(s,y) ->iA0 t` and then an involved case-splitting on `s -|>B t'`; the details are substantial (over 100 tactic invocations) and are omitted here. The proof relies crucially upon the careful definition of `->>iA`, which is not the true reflexive-transitive closure of `->iA` as it can rename many redexes but is limited to always using the *same* name each time. Hence it is indexed by a single variable rather than a vector of variables. This allows both the substitutivity results of Lemma 10 to be easily lifted from `->iA` to `->>iA`; these results have crucial applications in the proof. (NB. it is straightforward to show that `->>iA` is a subset of `->>A` so that the lemma holds for `->>A` as required.)

The interested (or concerned!) reader is advised to study the relevant theories Alpha, AlphaZero, and WeakABComm of the complete Isabelle development at the author's homepage for further details. □

2.6 Fresh-naming α-confluence with BCF-Finality

So far we have proved that $-\!\!\twoheadrightarrow_\beta$ has the diamond property (up to BCF-initiality) and commutes with $--\!\!\twoheadrightarrow_{\alpha_0}$. Our next objective is to show a diamond property of $--\!\!\twoheadrightarrow_\alpha$ itself, i.e. to show that any two α-divergences from a raw λ-term can be α-renamed to the same term. In fact, we will show that α_0-reduction is enough to resolve the divergence and that furthermore the resulting term can be chosen to be a BCF.

NB. In the following lemmas and diagrams the notation $--\circ_{\alpha_0}$ is used to denote the reflexive (*not* reflexive-transitive) closure of $--\!\!\to_{\alpha_0}$, and the notation $\vec{z_i}$ to denote a vector of variable names.

Definition 11 (Reflexive α_0-reduction) *The reflexively closed fresh-naming α relation $--\circ_{\alpha_0}$ is defined in Isabelle as follows:*

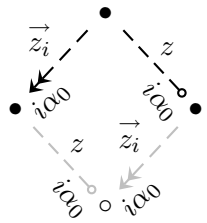
```
inductive i_alpha1 (* reflexive version of i_alpha0 *)
intrs
  var    "x~=y ==> (Var x,y) ->iA1 Var x"
  contr  "y~:(BV(Abs x e) Un FV(Abs x e)) ==>
          (Abs x e,y) ->iA1 Abs y (e[x:=Var y])"
  abs    "[|(e,y) ->iA1 e'; x~=y|] ==>
          (Abs x e,y) ->iA1 Abs x e'"
  appL   "[|(e1,y) ->iA1 e1'; y~:(BV(e2) Un FV(e2))|] ==>
   (e1 $ e2,y) ->iA1 e1' $ e2"
  appR   "[|(e2,y) ->iA1 e2'; y~:(BV(e1) Un FV(e1))|] ==>
   (e1 $ e2,y) ->iA1 e1 $ e2'"

inductive cl_ialpha1 (* equivalent to (alpha0)^* *)
intrs
  refl   "(e,[]) ->>iA1 e"
  trans  "[|(e1,xs) ->>iA1 e2; (e2,x) ->iA1 e3|]
           ==> (e1,x#xs) ->>iA1 e3"
```

Note that the reflexive-transitive closure `->>iA1` is indexed by a *list* of variables (as it can choose a different fresh name for each reduction it performs). Hence it cannot be defined simply as the Kleene closure of `->iA1`. It can be proved however (straightforwardly by rule induction) that `->>iA1` is exactly equivalent to the non-indexed `->>A0`. Thus although we work with the reflexive `->iA1` and `->>iA1`, all the results we prove for `->>iA1` automatically hold for `->>A0` so this need not be a source of concern. The reason for using `->iA1` and not `->iA0` in

the following lemmas is that some cases of the results require the single-step α_0 relation to be reflexively closed in order to go through.

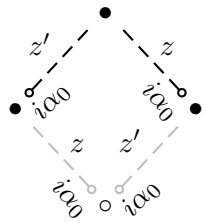
Lemma 12 (Semi-confluence of $\dashrightarrow\!\!\!\!\!\rightarrow_{\alpha_0}$) *For any z and $\vec{z_i}$ such that $z \notin \{z_i\}$:*




```
(e1,zs) ->>iA1 e2 ==> ALL e3. (e1,z) ->iA1 e3 -->
~(z mem zs) --> (EX e4. (e2,z) ->iA1 e4 & (e3,zs) ->>iA1 e4)
```

(Note that mem in Isabelle is a predicate for list membership.)

Proof By induction in the reflexive-transitive generation of `->>iA1`. The reflexive case is trivial. The induction step is the following, for any $z \neq z'$:



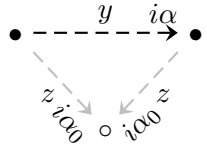

```
(e1,z1) ->iA1 e2 ==> ALL e3. (e1,z2) ->iA1 e3 -->
z1~=z2 --> (EX e4. (e2,z2) ->iA1 e4 & (e3,z1) ->iA1 e4)
```

This property is proved by rule induction in `(e1,z1) ->iA1 e2`. Every case except the `Var` rule case is non-trivial; reflexivity is needed for the case where the divergence is caused by two α_0-steps on the same abstraction. The proof uses a large number of sub-lemmas including a special version of substitutivity (Lemma 10) in which the substituted term is a variable:

```
[| (e,z) ->iA1 e'; y~=z; y~:Capt x e |]
 ==> (e[x:=Var y],z) ->iA1 e'[x:=Var y]
```

Special cases of the Substitution Lemma (Lemma 2) are also needed. The proof involves more than 100 tactic invocations and so further details are omitted here. □

Lemma 13 (Triangle Property of Indexed $\dashrightarrow_\alpha$) *Where $z \neq y$ and z is fresh with respect to the initial λ-term:*



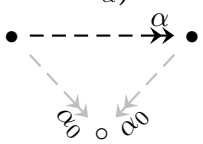
```
(e1,y) ->iA e2 ==> z~:(FV(e1) Un BV(e1) Un {y}) -->
(EX e3. (e1,z) ->iA1 e3 & (e2,z) ->iA1 e3)
```

Proof By rule induction in `(e1,y) ->ia e2`. The proof is reasonably concise and uses the first version of the Substitution Lemma (Lemma 2) and the following two 'renaming sanity' propositions:

- `[| y ~: FV e; y ~= z |] ==> y ~: FV (e[x:=Var z])`
- `y ~: BV e ==> y ~: BV (e[x:=Var z])`

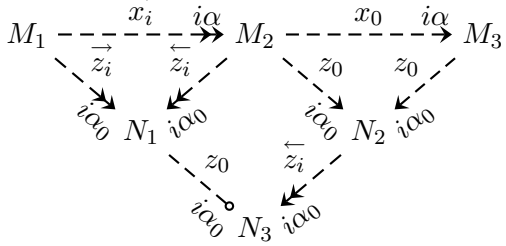
□

Lemma 14 (Triangle Property of $\dashrightarrow\!\!\!\!\twoheadrightarrow_\alpha$)




```
e1 ->>A e2 ==> EX e3. e1 ->>A0 e3 & e2 ->>A0 e3
```

Proof By rule induction in the $\overset{\vec{x_i}}{\dashrightarrow\!\!\!\!\twoheadrightarrow}_{i\alpha}$-relation underlying $\dashrightarrow\!\!\!\!\twoheadrightarrow_\alpha$. The reflexive case is trivial. The induction step reads as follows:



The quantification of the variables goes as follows: for any x_i's, (M_i's,) and z_i's, if there as many z_i's as x_i's, if they are all unique and different from the x_i's and $\mathrm{FV}(M_i) \cup \mathrm{BV}(M_i)$, the property holds. Notice how $\overrightarrow{z_i}$ gets reversed in the induction hypothesis and how that matches the uses of z_0.

For the Isabelle proof, we define the variable list-indexed reflexive-transitive closure of `->iA`, `->>ciA` in a manner analogous to the definition of the fresh-naming `->>iA1` and prove the Isabelle goal:

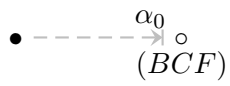
```
(e1,xs) ->>ciA e2 ==> ALL zs. length(zs) = length(xs)
--> uniqlist(zs) --> (ALL z.(z mem zs -->
~(z mem xs) & z~:(BV(e1) Un FV(e1)))) -->
(EX e3. (e1,zs) ->>iA1 e3 & (e2,rev zs) ->>iA1 e3)
```

which can be refined to the result above using our results concerning equivalence of `->>iA1` and `->>A0`, and of `->>ciA` and `->>A`. The built-in function on lists `length` returns the number of items in a list and the predicate `uniqlist` (which we define) returns true iff each variable in the list is different.

The proof of the induction step follows from (1) the induction hypothesis in the top left corner, (2) by Lemma 14 in the top right corner, and (3) by a

simple corollary to Lemma 13 in the centre. The proof itself is complicated hugely by the necessity of showing that the required property of the list `zs` is preserved over the induction, requiring us to employ a number of auxiliary lemmas concerning induction over the corresponding predicate. □

Lemma 15 (Existence of α_0-Renaming Sequence to BCF)




```
EX e2. e1 ->>A0 e2 & BCF(e2)
```

Proof The key result in order to show this lemma is the following — we write $\#_\lambda(e)$ for the number of λ-abstractions in e and $||\ \vec{x_i}\ ||$ for the number of elements in the vector $\vec{x_i}$:

$\forall\ \vec{x_i}, e_1.\ ||\ \vec{x_i}\ || = \#_\lambda(e_1)\ \wedge\ \{x_i\}$ all different $\wedge\ (\{x_i\} \cap (\mathrm{FV}(e_1) \cup \mathrm{BV}(e_1)) = \emptyset)$
$\Downarrow$
$\exists e_2. e_1 \overset{\vec{x_i}}{-\!\!\twoheadrightarrow}_{i\alpha} e_2 \wedge \mathrm{BCF}(e_2) \wedge \{x_i\} = \mathrm{BV}(e_2)$

The Isabelle translation of this property is the following goal:

```
ALL xs. lambdas e1 = length xs --> uniqlist xs -->
(ALL x. x mem xs --> x ~: FV e1 Un BV e1) -->
(EX e2. (e1, rev xs) ->>iA1 e2 & BCF e2 & BV e2 = set xs)
```

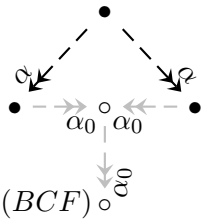
where `lambdas` is a function returning the number of abstractions in a given `lterm`.

The reason we prove such a complicated goal (compared to the main result) is that because we use structural induction the proof needs to be fully constructive — that is to say, the induction goal itself needs to reflect how we build the BCF term. Consider the case of a naive proof by structural induction where `e1` is an application, `e1 = e' $ e''`; knowing that `e' ->>A0 e1', e'' ->>A0 e1''` and `BCF(e1') & BCF(e1'')` does *not* enable us to conclude `BCF(e1' $ e1'')` because we have no way of knowing whether there is a variable overlap between `e1'` and `e1''`. We therefore specify the desired properties of the list of variable names to be used for the renaming (`uniqlist xs`) and its relationship to the target BCF term (`BV(e2) = set xs`)in order to enable the induction to go through correctly. (Note that `set xs` returns a set representation of the list `xs` in order to enable the comparison with `BV(e2)`.)

The proof itself is very involved but uninteresting and uses a variant of substitutivity (Lemma 10) as well as a host of auxiliary results concerning induction principles over the involved predicates. The variable case is easy; both the application and abstraction cases involve first extracting the induction hypotheses, then applying the appropriate α_0-rewrite rule and showing that the resulting

term is a BCF, that the bound variables of the term are exactly those in the list of variables and that the side conditions for the rewrite rule are in fact satisfied. □

Lemma 16 (Fresh-naming α-confluence with BCF-Finality)




```
[|e ->>A e1; e ->>A e2|] ==>
 EX e3. e1 ->>A0 e3 & e2 ->>A0 e3 & BCF(e3)
```

Proof Since $\text{--}\twoheadrightarrow_\alpha \subseteq \texttt{==}_\alpha$, we have that the two divergent terms are α-equal, i.e. `e1 =A= e2`. By directability of $\text{--}\twoheadrightarrow_\alpha$ (Lemma 9), we thus have `e1 ->>A e2` whence we can first resolve the divergence by Lemma 14, and then rename the resulting term to a BCF by Lemma 15. □

2.7 Confluence of λ^{var}

We are now ready to prove the confluence property of $\text{--}\to_{\alpha\cup\beta}$. To do so we first present (without proof) the following well-known classical result for proving confluence due to Tait and Martin-Löf. The Isabelle proof of the result is due to Nipkow [13] and by importing his theory we can re-use it directly in our own development:

Theorem 17 (Tait/Martin-Löf)

$$\begin{array}{l}\exists \to_2 \,.\, \to_1 \subseteq \to_2 \subseteq \twoheadrightarrow_1 \;\wedge\; \diamond(\to_2)\\ \Downarrow\\ \mathrm{Confl}(\to_1)\end{array}$$

```
[| diamond(R); T <= R; R <= T^* |] ==> confluent(T)
```

where `diamond` and `confluent` have the appropriate translations in Isabelle.

Thus in order to prove confluence of $\text{--}\to_{\alpha\cup\beta}$ it suffices to prove the diamond property of a relation $\to_2$ such that $\text{--}\to_{\alpha\cup\beta} \subseteq \to_2 \subseteq \text{--}\twoheadrightarrow_{\alpha\cup\beta}$. We choose the composite relation $(\text{--}\twoheadrightarrow_\alpha ; \text{-}\!\!\shortmid\!\!\to_\beta)$ as our $\to_2$ and define it in Isabelle as follows:

```
inductive relation2
intrs
  cons "[|e1 ->>A e2; e2 -|>B e3|] ==> e1 ->2 e3"
```

We now go about proving that `->2` has the desired properties.

Lemma 18 (α / β Hierarchy)

$$\text{--}\!\twoheadrightarrow_{\alpha\cup\beta} \subseteq (\text{--}\!\twoheadrightarrow_{\alpha}; \text{-}\!\!\twoheadrightarrow_{\beta}) \subseteq \text{--}\!\twoheadrightarrow_{\alpha\cup\beta}$$

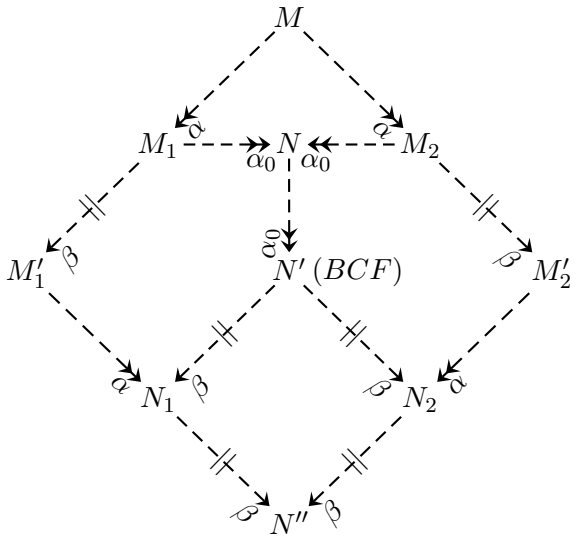
- `alpha Un beta <= relation2`
- `relation2 <= (alpha Un beta)^*`

Proof First note that although we have not in fact given the Isabelle definition of $\text{--}\!\!\twoheadrightarrow_{\beta}$ (`beta`), it is just the obvious translation of Definition 4 from the first chapter. For the first result, we can prove the inclusion `beta <= par_beta` automatically by rule induction on `beta`; the result follows easily from this and the fact that `-|>B` and `->>A` are reflexive. Similarly, the second result follows straightforwardly from the inclusion `par_beta <= beta^*`, provable by rule induction on `par_beta`, and a number of small auxiliary lemmas showing the contextual closure of `->>B`. □

Lemma 19 (The α / β Diamond) *The relation $\text{--}\!\twoheadrightarrow_{\alpha}; \text{-}\!\!\twoheadrightarrow_{\beta}$) has the diamond property.*

`diamond(relation2)`

Proof



For the M's given in the above diagram we resolve the divergence by constructing: N' by Lemma 16; N_1 and N_2 by two applications of Lemma 11; and finally N'' by Lemma 7, observing that it can be applied since N' is a BCF by construction. The Isabelle proof exactly mirrors this construction, using `dtac` to apply each of the lemmas in turn and `etac` with the rules `exE` and `conjE` to remove object-level existential quantifiers and conjunctions as necessary. □

Hence all the preconditions for the application of Theorem 17 are satisfied, allowing us to finally pull the sword from the stone:

Theorem 20 (Confluence of the λ^{var}-Calculus)

$$Confl(\dashrightarrow_{\alpha\cup\beta})$$

```
confluent(alpha Un beta)
```

Proof We instantiate Theorem 17 with the results from Lemmas 18 and 19 as follows:

```
Goal "confluent(alpha Un beta)";
by(rtac diamond_to_confluence 1);

Level 1 (3 subgoals)
confluent (alpha Un beta)
 1. diamond ?R
 2. alpha Un beta <= ?R
 3. ?R <= (alpha Un beta)^*
```

The tactic `rtac` performs resolution between the subgoal and the conclusion of the named rule (Nipkow's `diamond_to_confluence`) so that the new subgoals are the premises of the rule. Note that the meta-level object variable `?R` represents an argument which has not yet been instantiated. Unknowns may be instantiated when we perform resolution. For example, to discharge the new subgoals we perform resolution again with the rules we generated in the proofs of Lemmas 18 and 19:

```
by(rtac relation2_subset_alphabeta 3);
by(rtac alphabeta_subset_relation2 2);
by(rtac diamond_relation2 1);
```

The first of these resolutions forces the instantiation of `?R` to `relation2`. We then have no choice over the value of `?R` when resolving the remaining subgoals, so it is important when working with unknonws in a proof state not to accidentally instantiate values to 'bad' values which then invalidate other subgoals. (As `relation2` is the correct choice here, however, we have done the right thing.) □

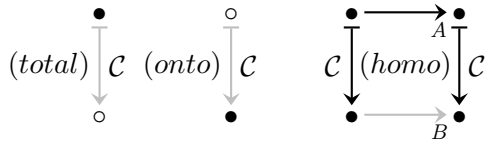
2.8 Confluence of λ^{real}

We have proved, at expenditure of great effort, the confluence property of $\dashrightarrow_{\alpha\cup\beta}$ in the raw λ-calculus (λ^{var}). Recall however that we consider the raw calculus to be in some sense representative of the *real* λ-calculus in which α-equal terms are identified and for which reduction takes place over the α-equivalence classes of terms. It would be desirable therefore to prove formally that the confluence property is not some sort of syntactic 'miracle' due to the

particulars of our presentation and still holds when one considers β-reduction on the α-equivalence classes of λ^{var}.

The relationship between the raw and real λ-calculi we consider is characterised by the notion of *structural collapse*:

Definition 12 (Structural Collapse) *Assume two relations with explicit carrier sets: $\rightarrow_A \subseteq A \times A$ and $\rightarrow_B \subseteq B \times B$. A mapping, $\mathcal{C} : A \longrightarrow B$, will be said to be a* structural collapse *if it is total, onto and a homomorphism between $\rightarrow_A$ and $\rightarrow_B$:*

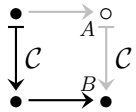
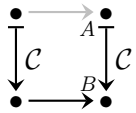


In Isabelle, totality of functions is automatic since all functions must be totally defined on their declared domain. We formalise the notion of structural collapse as follows. (We include the type definitions as well in order to highlight the fact that our definition is independent of all the definitions of λ^{var}.)

```
consts
  onto :: "['a => 'b, 'a set, 'b set] => bool"
  homo :: "['a => 'b, 'a set, 'b set, ('a * 'a) set,
            ('b * 'b) set] => bool"
  struct_coll :: "['a => 'b, 'a set, 'b set, ('a * 'a) set,
                   ('b * 'b) set] => bool"
defs
  onto_def "onto F A B == ALL y. y:B --> (EX x. x:A & F(x)=y)"
  homo_def "homo F A B RelA RelB == RelA <= A<*>A &
            RelB <= (B<*>B) & (ALL x x' y y'. ((x,y):RelA -->
            F(x)=x' --> F(y)=y' --> (x',y'):RelB))"
  struct_coll_def "struct_coll F A B RelA RelB == onto F A B &
                   homo F A B RelA RelB"
```

We now present a new result about abstract rewrite systems (ARS) under structural collapse, due to Vestergaard in [17].

Theorem 21 (Preservation / Reflection of Diamond) *Given a structural collapse, $\mathcal{C}$, from $\rightarrow_A$ to $\rightarrow_B$, we have the following implications:*

1.  $\Rightarrow$ $Diamond(\rightarrow_A) \rightarrow Diamond(\rightarrow_B)$
2.  $\Rightarrow$ $Diamond(\rightarrow_A) \leftrightarrow Diamond(\rightarrow_B)$

We prove both results in Isabelle (see theory `AbstractRewrites`*); the formulation of the second result is as follows:*

```
[|struct_coll F A B RelA RelB & (ALL x y x' y'. (x',y'):RelB &
```

```
 F(x)=x' & F(y)=y' --> (x,y):RelA)|]
==> diamond(RelB) = diamond(RelA)
```

Proof Both directions of the biimplication are straightforward to establish by a simple back-and-forth style argument. For example, assuming $Diamond(\rightarrow_A)$ and given a $\rightarrow_B$-divergence in B, we use onto-ness of F to find corresponding terms in A which are related by $\rightarrow_A$ by the case premise. This divergence can then be resolved by $Diamond(\rightarrow_A)$ and so we can close the $\rightarrow_B$-divergence by the homomorphism property of F. The Isabelle proof exactly mirrors this argument. □

NB. One might reasonably ask whether the conditions above are necessary (although clearly they are sufficient) for the preservation and reflection of the diamond property to hold between relations. In [17] it is shown that if we weaken the premise then there are counter-examples to both directions of the biimplication. The conditions given above may therefore be thought of as weakly necessary.

The next aim of our proof development should now be clear; to define β-reduction, $\rightarrow_\beta$, on α-equivalence classes of λ^{var} and show that the α-collapse function satisfies the conditions for Theorem 21 to hold between $\dashrightarrow_{\alpha\cup\beta}$ and $\rightarrow_\beta$. To define real β-reduction on α-equivalence classes, we first define a function returning the α-equivalence class of an `lterm` and then use it to define a new type `acls` of α-equivalence classes by set comprehension:

```
defs
  alphaclass "alphaclass(e) == {e'. e' =A= e}"

typedef acls = "{y. EX (x::lterm). y = alphaclass x}"
```

Two conversion functions `Rep_acls` and `Abs_acls` are provided by Isabelle automatically from the type definition above. They can be thought of as the functions which explicitly convert an `acls` to the corresponding `lterm set`, and vice versa respectively. With this in mind we can define the real β-reduction relation as well as a membership function for α-equivalence classes and the composite (raw) relation $==_\alpha; \dashrightarrow_\beta; ==_\alpha$ as shown below. We include the type declarations as well to highlight the different domains involved.

```
consts
 real_beta      :: "(acls * acls) set"
 beta_mod_alpha :: "(lterm * lterm) set"
 mem_ac :: "lterm => acls => bool"

defs mem_ac_def "mem_ac e A == (Rep_acls A = alphaclass e)"
inductive beta_mod_alpha
intrs
 cons "[|s =A= s'; s' ->B t'; t' =A= t|] ==> s -ABA-> t"
inductive real_beta
```

```
intrs
  lift "s ->B t ==>
        Abs_acls(alphaclass s) =>B Abs_acls(alphaclass t)"
```

and we define the transitive-reflexive closures of `-ABA->` and `=>B` with double-headed arrows in the usual way.

The main result relating real β-reduction and reduction in the raw λ^{var}-calculus is then the following, where $\lfloor e \rfloor$ is used to denote the α-equivalence class generated by the raw term e:

Proposition 22 (Raw Characterisation of Real β)

$$\lfloor e \rfloor \twoheadrightarrow_\beta \lfloor e' \rfloor \;\Leftrightarrow\; e\ (==_\alpha; \dashrightarrow_\beta; ==_\alpha)^\star\ e' \ \vee\ e ==_\alpha e'$$

```
Abs_acls(alphaclass(e)) =>>B Abs_acls(alphaclass(e')) =
(e -ABA->> e' | e =A= e')
```

Proof The reverse implication is straightforward; the case `e =A= e'` is trivial and the case `e -ABA->> e'` is dealt with by induction on the reflexive-transitive generation of `-ABA->>` whence the implication follows by the induction hypothesis and the definition of `alphaclass`.

For the forward implication, we prove the following result, where `e in ac1` is syntactic sugar for `mem_ac e ac1`:

```
ac1 =>>B ac2 ==> ALL e e'. (e in ac1 --> e' in ac2 -->
(e -ABA->> e' | e =A= e'))
```

which once proved can be manipulated to have the required form. The property is proved by reflexive-transitive induction on `ac1 =>>B ac2`. The proof is fairly uninteresting and relies on careful manipulation of the involved sets of α-equivalent terms. □

Proposition 23 (Raw Characterisation Equivalence)

$$(==_\alpha; \dashrightarrow_\beta; ==_\alpha)^\star \cup\ ==_\alpha = \dashrightarrow\!\!\!\!\!\twoheadrightarrow_{\alpha\cup\beta}$$

```
(beta_mod_alpha)^* Un (alpha Un rev_alpha)^* = (alpha Un beta)^*
```

Proof On applying the rules for proving inclusions and applying the simplifier, we see that the proposition amounts to showing:

```
Level 5 (3 subgoals)
beta_mod_alpha^* Un (alpha Un rev_alpha)^* = (alpha Un beta)^*
 1. !!a b. a -ABA->> b ==> a ->>AB b
 2. !!a b. a =A= b ==> a ->>AB b
 3. !!a b. a ->>AB b ==> a -ABA->> b | a =A= b
```

(Note that `->>AB` is the Isabelle translation of $\dashrightarrow\!\!\!\!\!\twoheadrightarrow_{\alpha\cup\beta}$.) Each of the subgoals is proved by a separate induction on the reflexive-transitive generation of the

relation appearing in the subgoal premise. The first two subgoals are very straightforward and rely heavily on directability of `=A=` (Lemma 9). The third is longer to prove but the proof itself involves no special effort. □

Proposition 24 (Second Raw Characterisation of Real β)

$$\lfloor e \rfloor \twoheadrightarrow_\beta \lfloor e' \rfloor \;\Leftrightarrow\; e \dashrightarrow_{\alpha\cup\beta} e'$$

```
Abs_acls(alphaclass(e)) =>>B Abs_acls(alphaclass(e')) = e ->>AB e'
```

Proof Immediate from the previous two propositions. □

It is now straightforward to show that the function `alphaclass` is indeed a structural collapse:

Lemma 25 ($\lfloor - \rfloor$ is a Structural Collapse) *$\lfloor - \rfloor$ is a structural collapse from λ^{var} to λ^{real} with respect to the rewrite relations $\dashrightarrow_{\alpha\cup\beta}$ and $\rightarrow_\beta$. Furthermore, $\lfloor - \rfloor$ satisfies the condition for case 2 of Theorem 21 to hold.*

- ```
  struct_coll (Abs_acls o alphaclass) (UNIV::lterm set)
  (UNIV::acls set) ((alpha Un beta)^*) (real_beta^*)
  ```
- ```
  ALL x y x' y'. (x',y'):(real_beta^*) &
  (Abs_acls o alphaclass)(x)=x' & (Abs_acls o alphaclass)(y)=y'
  --> (x,y):((alpha Un beta)^*)
  ```

Proof The proofs of the two results are very similar and we will only consider the first here. First note that texttttAbs_acls o alphaclass is just the composition of the two functions, i.e. the function which given an `lterm`, computes the `lterm set` of α-equivalent terms and casts it to the desired type `acls`. The keyword `UNIV` in the HOL framework is used to denote the universal set; we consider a mapping from the set of all raw terms to the set of all α-equivalence classes so the two sets A and B in Theorem 21 must be given the appropriate types, which we do using the casting operator `::`.

In order to simplify the first goal we wish to expand out the definition of `struct_coll` and any definitions upon which it relies. One way to do so is to use the following rewriting tactic:

```
by(rewrite_goals_tac [struct_coll_def, onto_def]);
```

which expands all occurrences of the definitions `struct_coll` and `onto` throughout all the goals of the proof state. If we wish to restrict the rewriting to just one of the subgoals we can use the `SELECT_GOAL` tactical as follows:

```
by(SELECT_GOAL (rewrite_goals_tac [struct_coll_def]) 1);
```

After expanding the definitions appropriately and applying the simplifier, we find that the proof burden of the lemma boils down to:

```
Level 1 (2 subgoals)
struct_coll (Abs_acls o alphaclass) UNIV UNIV
((alpha Un beta)^*) (real_beta^*)
 1. !!y. EX x. Abs_acls (alphaclass x) = y
 2. !!x y. x ->>AB y ==>
           Abs_acls (alphaclass x) =>>B Abs_acls (alphaclass y)
```

whence the second subgoal follows by Proposition 23 and the first follows from the rules automatically generated from the type definition of `acls`. (These rules are rather technical and the interested reader is advised to consult *FIXME:reference* for details.) □

In conclusion, we therefore have:

Theorem 26 (Confluence of the Real λ-Calculus)

$$Confl(\rightarrow_\beta)$$

```
confluent(real_beta)
```

Proof By Lemma 25 applied to Theorem 21. □

2.9 Paying the cost to be the boss

We have chosen to concentrate on the main points of the proof structure here while simultaneously attempting to provide some insight into the specifics of the Isabelle/HOL formalisation. However, for reasons of space and perspicuity it has been necessary to omit many of the fine details and auxiliary results of the confluence proof. The full Isabelle proof development contains the proofs of over 200 individual lemmas and is approximately 4000 lines of code in size (including comments).

It is worth noting that Nipkow provides a proof of confluence in the λ-calculus using a de Bruijn-style presentation [13], for which the Isabelle proof of β-confluence is of the order of 500 lines of code. This gulf between the sizes of the two proofs is probably due in part to Nipkow's expertise with the Isabelle system, but is mainly a result of the more complicated setup in which we choose to work. Because the concept of α-reduction (and thus the raw / real calculi distinction) is made redundant by the use of de Bruijn indices, it is sufficient to prove the diamond property of $-\!\!\twoheadrightarrow_\beta$ and then apply the Tait/Martin-Löf theorem directly. In contrast, we must not only prove the diamond property of $-\!\!\twoheadrightarrow_\beta$ with our more complex conditional rules, but then prove several equally substantial results showing commutativity of α-renaming with itself and with β-reduction. Furthermore, in order to show that confluence in this calculus is meaningful we must then explicitly show that the confluence property maps

onto the α-collapse of the raw calculus. However, our proof methodology has substantial advantages too. We have shown, as a first, that it is possible to conduct fully formal reasoning with the λ-calculus at the level of $FOAS_{VN}$ (as one would do by hand) — and feasibly so, although the cost of the proof is reasonably high. Furthermore, we showed in Lemma 7 the crucial role played by our formal variant of the Barendregt Variable Convention (the BCF predicate), thus providing a so-called *rational reconstruction* of the BVC. We would also add that much of our proof effort goes into results which collectively suffice to prove most equational properties of β (for example, commutativity of α / β) and so the cost we have paid can be considered a 'one-time expense'.

3. Barendregt-Style Reasoning is Correct, Sometimes

In chapter 2 we showed that it is possible to prove at least one substantial equational result (confluence) for the λ-calculus over $FOAS_{VN}$ in full formality and including a proper treatment of variable-renaming issues. To do so, we needed to show the 'traditional' key lemma for the result (diamond property of $-\!\!\twoheadrightarrow_\beta$) and then to construct lemmas showing that α-renaming commutes with itself and with the involved 'computational' relation (β-reduction) in order to a construct a similar key lemma for a relation incorporating α-reduction (in our case $(-\!\!\twoheadrightarrow_\alpha; -\!\!\twoheadrightarrow_\beta)$). In other words, there is apparently no reason why a similar methodology would work for any other equational result in λ-calculus theory. The aim of this chapter is to prove a result in Isabelle which suggests that our method is indeed applicable to other equational results of the calculus.

Before moving on to the specifics of the result and its proof, let us informally consider what it is that we would like to be able to prove. Suppose it could be shown that, for a large class of (raw) λ-terms, no β-reduction can ever block the β-reduction of all of the other redexes in the term. This would immediately imply that a host of properties of the λ-calculus involving β-reduction can also be proved for λ^{var} using our first-order proof technology, since no α-renaming is then required for the property to hold. To establish such a property for the real λ-calculus would still require the relevant α-commutativity lemmas but these can be thought of as forming an intermediate proof layer between the key lemma and the abstract reasoning required to lift the result to the real level — the point is that α-renaming is *not* required for the key lemma to hold and so can be proved (and formalised) directly in our λ^{var}.

Of course, for the "renaming freeness" result to be meaningful, we would need to prove it for a class of λ-terms such that the proofs of most equational properties do not move 'outside' of the class. For this reason, we now work in the *residual theory* of the λ-calculus, in which (pseudo-)redexes may be *marked* and, furthermore, only these marked redexes may be contracted. A marked λ-term is said to be a *residual* of another term if they are related by a number of residual β-contractions (which by definition reduce only marked redexes). It is well-known that the proofs of most equational results in the λ-calculus respect the residual theory in the sense that they do not need to 'move outside it', i.e. no redexes which do not appear in the original term need be contracted. Therefore, working with residuals is the approach we take towards establishing our desired result.

As before, we start by defining a marked variant of the λ^{var}-calculus we worked with in Chapters 1 and 2, and then move towards the renaming freeness result we seek by building a theory of intermediate lemmas about the behaviour of the calculus under substitution and reduction.

3.1 The Marked λ^{var}-Calculus

Definition 13 (The Raw, Marked λ-Calculus) *The terms of the raw, marked λ^{var}-calculus are defined inductively as follows:*

$$\Lambda^{\text{var}}_{\text{res}} ::= x \mid \Lambda^{\text{var}}_{\text{res}}\Lambda^{\text{var}}_{\text{res}} \mid \lambda x.\Lambda^{\text{var}}_{\text{res}} \mid (\lambda x.\Lambda^{\text{var}}_{\text{res}}) \mathbin{@} \Lambda^{\text{var}}_{\text{res}}$$

We refer to the final clause of this definition as a marked redex. *When we define (residual) β-reduction later we will allow only marked redexes to be contracted.*

In Isabelle we may reuse our theory of one-sorted variables but otherwise may not reuse any code from Chapter 2 as all predicates and relations must now be defined over a new inductive datatype, `mlterm`, for raw marked λ-terms. We therefore start by defining a new theory as follows:

```
MarkedLambda = Variables

 datatype mlterm = Var var
                 | "$" mlterm mlterm (infixl 200)
                 | Abs var mlterm
                 | Redex var mlterm mlterm
                   ("<Abs _ _> @ _ " [200,0,0] 200)
```

Note that the definition is exactly like our earlier definition of `lterm` except for the new constructor `Redex` for marked redexes. The expression following the definition of `Redex` gives a syntactic translation; instead of `Redex x e e'` we may instead write `<Abs x e> @ e'`.

We proceed to define $\mathrm{FV}(-)$, $\mathrm{BV}(-)$, $\mathrm{Capt}_x(-)$, substitution $-[- := -]$, and the BCF predicate exactly as for ordinary `lterm`s. For the marked redex case, we define each of these functions to have the same value as it would for an ordinary unmarked redex, so, for example, $\mathrm{FV}((\lambda x.e) \mathbin{@} e') = \mathrm{FV}((\lambda x.e)e')$. Below is the Isabelle definition of $\mathrm{FV}(-)$ on `mlterm`s (the other definitions are made in similar fashion):

```
primrec
  FV_Var "FV(Var x) = {x}"
  FV_App "FV(e1 $ e2) = (FV(e1) Un FV(e2))"
  FV_Abs "FV(Abs x e) = (FV(e) - {x})"
  FV_Red "FV(<Abs x e1> @ e2) = (FV(e1) - {x}) Un FV(e2)"
```

With our definition of the marked λ^{var}-calculus, all of the auxiliary results (not involving reduction) of the ordinary λ^{var} from our previous development can be re-proven in the new marked case by straightforward structural induction, including Proposition 1(Renaming Sanity) and Lemma 2 (Substitution) from Chapter 2. However, the proofs are a little more difficult because we are now forced to consider an extra inductive case when the term is a marked redex; typically the proof is much like the abstraction case but with a more complex inductive hypothesis. We must also be more careful when using automatic tactics such as `Auto_tac` which are applied to all subgoals by default, as the

complicated function definitions for the redex case sometimes prove too much for Isabelle and lead to non-termination.

3.2 Non-Blocked β-Residual Theory

Once the legwork of tediously re-proving various facts about the marked version of the λ^{var} calculus is complete, we turn our attention to the more interesting concept of reduction in this calculus. We consider two β-reduction relations here: a single-step residual-β reduction relation (which may only contract marked redexes) and the completely developing residual-β reduction relation, which contracts *all* of the marked redexes in a term.

Definition 14 (Residual-β reduction) *Single-step residual-β reduction on raw, marked λ-terms is defined inductively thus:*

$$\frac{\mathrm{Capt}_x(e_1) \cap \mathrm{FV}(e_2) = \emptyset}{(\lambda x.e_1) \mathbin{@} e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_1[x := e_2]}\ (\beta^{\mathrm{res}})$$

$$\frac{e_1 \dashrightarrow_{\beta^{\mathrm{res}}} e_1'}{e_1 e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_1' e_2} \qquad \frac{e_1 \dashrightarrow_{\beta^{\mathrm{res}}} e_1'}{(\lambda x.e_1) \mathbin{@} e_2 \dashrightarrow_{\beta^{\mathrm{res}}} (\lambda x.e_1') \mathbin{@} e_2}$$

$$\frac{e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_2'}{e_1 e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_1 e_2'} \qquad \frac{e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_2'}{(\lambda x.e_1) \mathbin{@} e_2 \dashrightarrow_{\beta^{\mathrm{res}}} (\lambda x.e_1) \mathbin{@} e_2'}$$

$$\frac{e \dashrightarrow_{\beta^{\mathrm{res}}} e'}{\lambda x.e \dashrightarrow_{\beta^{\mathrm{res}}} \lambda x.e'}$$

In our Isabelle development this translates to:

```
inductive res_beta
intrs
  rbeta   "(Capt x s) Int FV(t) = {} ==>
           (<Abs x s> @ t ->RB s[x:=t])"
  rbappL  "s ->RB t ==> s$u ->RB t$u"
  rbappR  "s ->RB t ==> u$s ->RB u$t"
  rbabs   "s ->RB t ==> Abs x s ->RB Abs x t"
  rbredL  "s ->RB t ==> <Abs x s> @ u ->RB <Abs x t> @ u"
  rbredR  "s ->RB t ==> <Abs x u> @ s ->RB <Abs x u> @ t"
```

and as usual we also define `->>RB` to be the reflexive-transitive closure of `->RB`.

Notice that this relation is essentially no different from the single-step β-reduction defined in Chapter 1 (except that only marked redexes may be contracted). The greater number of introduction rules is necessary to enforce the contextual closure over the set of marked terms (which has more constructors than the set of unmarked terms).

In contrast, the completely developing residual-β relation defined below has at least one substantial difference from the complete development relation introduced in Section 2.4:

Definition 15 (Completely Developing Residual-*beta* Reduction) *The completely developing residual-beta reduction relation on raw, marked λ-terms is defined inductively as follows:*

$$\frac{e_1 \dashrightarrow_{\beta^{\mathrm{res}}} e_1' \quad e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_2' \quad \mathrm{Capt}_x(e_1') \cap \mathrm{FV}(e_2') = \emptyset \quad x \in \mathrm{FV}(e_1')}{(\lambda x.e_1) \mathbin{@} e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_1'[x := e_2']} \; (\beta^{\mathrm{res}\,*})$$

$$\frac{e_1 \dashrightarrow_{\beta^{\mathrm{res}}} e_1' \qquad x \notin \mathrm{FV}(e_1')}{(\lambda x.e_1) \mathbin{@} e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_1'} \; (\beta^{\mathrm{res}\,*}_{\mathrm{lazy}})$$

$$\frac{}{x \dashrightarrow_{\beta^{\mathrm{res}}} x} \; (\mathrm{Var}^{\beta^{\mathrm{res}}}_{*}) \qquad \frac{e \dashrightarrow_{\beta^{\mathrm{res}}} e'}{\lambda x.e \dashrightarrow_{\beta^{\mathrm{res}}} \lambda x.e'} \; (\mathrm{L}^{\beta^{\mathrm{res}}}_{*})$$

$$\frac{e_1 \dashrightarrow_{\beta^{\mathrm{res}}} e_1' \qquad e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_2'}{e_1 e_2 \dashrightarrow_{\beta^{\mathrm{res}}} e_1' e_2'} \; (\mathrm{A}^{\beta^{\mathrm{res}}}_{*})$$

The Isabelle translation is as follows:

```
inductive comp_res_beta
intrs
  crbeta1  "[|s ->|RB s'; t ->|RB t';
            (Capt x s') Int FV(t') = {};  x:FV(s')|]
              ==> <Abs x s> @ t ->|RB s'[x:=t']"
  crbeta2  "[|s ->|RB s'; x~:FV(s')|] ==> <Abs x s> @ t ->|RB s'"
  crbvar   "Var x ->|RB Var x"
  crbapp   "[|s ->|RB s'; t ->|RB t'|] ==> s$t ->|RB s'$t'"
  crbabs   "s ->|RB t ==> Abs x s ->|RB Abs x t"
```

Notice that although the complete development relation $\dashrightarrow_{\beta^{\mathrm{res}}}$ contracts all marked redexes, it does so using one of two possible rules, depending on whether the abstracted variable x occurs free in the contraction term e_1'. If it does, the usual contraction side-conditions apply; if not, we may simply discard the substitution (and the related side-conditions) since in this case we have $e_1'[x := e_2'] = e_1'$. Neither do we require that e_2 actually contracts, as it will be discarded in any case. The reason for introducing this rule is rather technical, having to do with the proof of substitutivity for $\dashrightarrow_{\beta^{\mathrm{res}}}$, and we will return to it shortly.

Meanwhile, we first prove some other important properties of $\dashrightarrow_{\beta^{\mathrm{res}}}$ which should be familiar from Section 2.4:

Lemma 27 (Variable Monotonicity Under $\dashrightarrow_{\beta^{\mathrm{res}}}$) *For any marked λ-term t:*

- `t ->|RB t' ==> FV t' <= FV t`

- `t ->|RB t' ==> BV t' <= BV t`

Proof By rule induction in `t ->|RB t'`. The proof is exactly like that of the equivalent property of `-|>B` (Lemma 3). □

Lemma 28 (Residual-Completion of BCF's)

$$(BCF) \bullet \xrightarrow[\beta^{\mathrm{res}}]{} \circ$$

```
BCF(s) ==> (EX t. s ->|RB t)
```

Proof By structural induction in `s`. The proof is like that of Chapter 2's Lemma 5 except that we need not perform case-splitting when `s` is an application. Instead, we perform a case-splitting on the appropriate variable condition when `s` is a marked redex case in order to determine which of the two contraction rules should be used. The case for the lazy contraction rule is straightforward by induction and for the non-lazy contraction rule we use Lemma 27 combined with the BCF variable disjointness property to justify the satisfaction of the necessary side-conditions. □

Lemma 29 (Substitutivity of C.D. Residual-β) *For any marked λ-terms `s,t`:*

- ```
  [| s1 ->|RB t1; s2 ->|RB t2; Capt x s1 Int FV s2 = {};
   Capt x t1 Int FV t2 = {} |] ==> s1[x:=s2] ->|RB t1[x:=t2]
  ```

- ```
  [| s1 ->|RB t1; x ~: FV t1; Capt x s1 Int FV t = {} |]
   ==> s1[x:=t] ->|RB t1
  ```

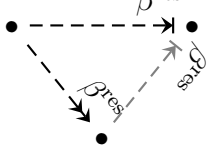
Proof Both by rule induction on `s -|>RB s'`. The proof of the first result is very similar to that for the substitutivity property of `-|>B` and uses all the same auxiliary lemmas, including the corresponding Substitution Lemmas (Lemma 2). However, the proof cannot work in *exactly* the same way since some of the (degenerate) cases for the `-|>B` result rely upon the reflexivity of that relation, and the complete development `->|RB` cannot be reflexive since by definition it must contract all of the redexes in a term. This is why we need the lazy contraction rule — it allows the contraction of a redex to go ahead in degenerate cases when the resulting substitution is discarded and as a result we cannot possibly justify the side-conditions on the normal contraction rule. An exposition of even the interesting cases of the proof is not possible here due to their complexity, but for the interested reader the proofs of all the results in this chapter are available at http://www.dcs.ed.ac.uk/~ jjb/mech_Barendregt_VB/ .

Also, as we now have two contraction rules it is necessary to prove a the second substitutivity lemma above which pertains to the lazy contraction rule. The proof is much easier than that of the first result; all of the inductive cases are straightforward except that for the normal contraction rule, which follows from a case-splitting on variable names and does not require any major auxiliaries (such as Substitution Lemmas).

N.B. Notice that as the first substitutivity result pertains to the normal contraction rule for `->|RB` we will only ever need to apply it when `x:FV t1`. Adding this fact as a premise to the first result would almost certainly eliminate some of the subcases of the proof, making it shorter and more efficient. However, in practice we choose to prove it in the above form because we can reuse the structure (and large parts of the tactic sequence) of the Isabelle proof of Lemma 4. Altering the premises of a complicated goal can have substantial repercussions with respect to the structure of the proof and the exact formulation of the required auxiliary lemmas, which is why we have chosen not to do so in this instance. □

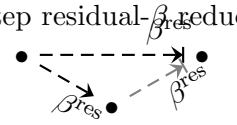
As was the case when we investigated parallel-β reduction, we find that substitutivity is the main proof burden of a crucial 'triangle lemma' for $\dashrightarrow_{\beta^{\mathrm{res}}}$ and $\dashrightarrow\!\!\!\!\!\rightarrow_{\beta^{\mathrm{res}}}$:

Lemma 30 (Completion-Absorption of Residual β)




```
s ->>RB t ==> s ->|RB u --> t ->|RB u
```

Proof By induction in the reflexive-transitive generation of `s ->>RB t`. The reflexive case is trivial; for the transitive case, by induction it suffices to prove the equivalent property for one-step residual-β reduction:




```
s ->|RB u ==> ALL t. s ->RB t --> t ->|RB u
```

The proof is by rule induction in `s ->|RB u`. The interesting cases are those for the two contraction rules, each of which follows immediately by application of the appropriate Substitutivity Lemma (Lemma 29). □

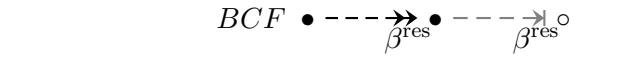
Let us informally summarise the meaning of Lemmas 28 and 30. Between them, they say that:

- for any raw, marked λ-term which is a BCF, one may contract all of the marked redexes in a term using the complete development relation $\dashrightarrow_{\beta^{\mathrm{res}}}$, and:

- if any raw, marked λ-term has a complete development, then for any number of residual-β reductions performed on that term, the complete development relation is still enabled on the resulting term.

From this it follows that the complete development residual-β reduction relation $\dashrightarrow_{\beta^{\mathrm{res}}}$ can never be blocked by any number of residual-β contractions on BCF terms. We make this precise with the following theorem:

Theorem 31 (Non-Blocked β-Residual Theory)




```
[| BCF(s); s ->>RB t |] ==> EX u. t ->|RB u
```

Proof Immediate by Lemmas 28 and 30. □

At the start of this chapter we stated our intention to prove a result (in the β-residual theory) which would suggest the general applicability of our proof methodology employed in chapter 2. To what extent has this been achieved by our Theorem 31? We already know that it is always possible to construct BCF terms from any raw λ-term (Lemma 15). Theorem 31 says that for any residual of a BCF term you can contract all of the marked redexes (except, possibly, for those which are discarded by the lazy contraction rule) without disabling the complete development, i.e. without causing any variable clash. In other words, the β-residual theory of the λ^{var}-calculus is renaming-free up to BCF-initiality. This implies that a host of equational results which 'respect' the residual theory can be proved (up to BCF-initiality) by first-order means and thus that our proof methodology can indeed be applied more generally. In fact, Vestergaard has used the result in [18] to show both the confluence property of β and the strong (finite) development property in a non-standard, yet very simple manner.

NB. We conjecture that Theorem 31 would still hold if the lazy contraction rule were not used in the definition of $\dashrightarrow_{\beta^{\mathrm{res}}}$. In that case we could conclude that we could contract *all* of the marked redexes without causing a variable clash. However, as we have seen when discussing the proof of substitutivity of $\dashrightarrow_{\beta^{\mathrm{res}}}$ (Lemma 29), the absence of the lazy contraction rule would have severe repercussions for the proof of this lemma as the discharging of certain subcases would only be achievable by building a BCF predicate or similar condition into the premises of the lemma. We would then have to show that the condition was preserved over the reflexive-transitive generation of $\dashrightarrow_{\beta^{\mathrm{res}}}$ in the proof of Lemma 30 in order to apply substitutivity, greatly complicating the induction. Thus we leave the conjecture as a question to be (possibly) answered in the future.

4. Automatibility of Isabelle Proofs

In the two previous chapters detailing our Isabelle developments, we saw some situations in which we made use of Isabelle's automatic tactics in order to reduce (or even eliminate totally) the effort we needed to expend on the proof (cf. Lemmas 1, 3, 6, 7, 18). However, most of the time we use sequences of manual tactics (although we frequently call upon the simplifier to perform equational rewriting automatically) and apply the classical reasoner only when we have reduced the goal to trivial subgoals. In some cases dozens or even hundreds of tactic invocations are necessary in order to solve particular goals (cf. Lemmas 4, 11, 12, 15). A natural question, then, is how one might determine whether or not a particular proof is amenable to automation in Isabelle.

Developing new techniques and heuristics for proof search strategies and for making proofs more amenable to automation is an ongoing area of research. Our focus during this project has been to achieve the Isabelle formalisation of the results we have presented and not to achieve any particular level of efficiency in doing so. There are many reasons why you might want to use a theorem prover; our main motivation for doing so is to add formal weight to the λ-calculus results we present. If they had not been machine-proved in full formality, it would be difficult for a reader to convince themselves that some obscure subcase of one of the (many) lemmata had not been overlooked or incorrectly dealt with. Providing one is prepared to trust the theorem prover we work with, the mechanisation of the proofs goes a long way towards assuaging this doubt.

However, our adventures in Isabelle have given some insight as to why certain proofs will not yield to the system's automatic tactics and in this chapter we attempt to summarise our findings. Although to the best of our knowledge there are no hard and fast rules as to whether or not a particular proof can be mechanised, we discuss some factors that can increase the difficulty of doing so.

4.1 Safe vs. unsafe rules

We have already seen that Isabelle incorporates a component called the *classical reasoner* which can be invoked by the user to search for a proof of a subgoal. This invocation occurs in the form of a call to one of several automatic tactics, each of which uses a slightly different search strategy (e.g. depth-first, best-first, iterative deepening) and may in turn appeal to the simplifier, which performs left-to-right equational rewriting. Whichever tactic is chosen, the corresponding proof search takes place over a collection of logical rules which we call the *claset*. The default claset contains just the standard (basic and derived) rules of HOL [14], but the user may freely add rules to, or delete rules from, the claset to suit their own needs.

Rules in the claset fall into two categories: *safe* rules and *unsafe* rules. Generally speaking, a rule is considered to be safe if it never reduces a provable goal to unprovable subgoals. For example, the rule for conjunction elimination (&-E) in first-order propositional logic is safe since if a goal is provable using the premise $A\&B$, then certainly it is also provable using the separate premises A, B. When searching for a proof, safe rules may therefore be applied more or less indiscriminately.

The classification of unsafe rules is somewhat less clear (as some rules are far more unsafe than others!) but there are certain properties of inference rules which allow them to be classified as unsafe. Any rule is unsafe if it leads to a loss of information in the proof state. For example, the rule for disjunction introduction (v-I1) in first-order propositional logic is unsafe because it reduces a goal of the form $A|B$ to the new goal A. A rule can also be considered unsafe if it can be applied infinitely many times (and so cause the search engine to loop). An example of such a rule is the basic HOL identity-symmetry rule `sym: s = t ==> t = s`, which can clearly be applied repeatedly to any subgoal premise of the form `x = y`. These are the two most common types of unsafe rules but there are other cases. For example, any rule which instantiates an unknown in the proof state (e.g. the standard resolution-with-instantiation tactic `res_inst_tac`) is unsafe as it can falsify subgoals (as we discussed in the proof of Theorem 20 in Chapter 2). Any rule which can lead to an explosion of the proof state space, such as a large number of applications of (v-E), can also be considered unsafe as it can make the proof intractable. Finally, such special rules as those for case splitting and induction are particularly unsafe as it is generally impossible for the classical reasoner to 'guess' the variables over which the induction or case-splitting should take place.

Clearly, however, the classical reasoner cannot be expected to provide much help if it is not permitted to use any unsafe rules! The way the reasoner works is rather complicated (for a more detailed exposition consult Chapter 11 of [15]) and depends on the tactic / search strategy invoked but basically comes down to the following: rules in the claset are explicitly defined as safe or unsafe, and the reasoner allows each search to use a (usually unlimited) number of safe rule applications combined with a bounded number of unsafe rule applications. Safe rules are usually given priority over those which instantiate unknowns, which in turn have priority over loss-of-information rules; the safe / unsafe distinction, which in any case is blurred, may thus be viewed simply as a way of assigning precedences to the inference rules. Many of the automatic tactics allow the user to specify the number of unsafe applications that are permitted in order to speed up the proof process. Some tactics also employ *iterative deepening* whereby the number of unsafe rule applications increases after each unsuccessful proof attempt.

What is the conclusion then with regards to our own Isabelle developments? Simply put, if one of our proofs requires the use of many unsafe rules (perhaps in a very specific order) then the chances of the classical reasoner finding a proof is smaller because it will always prefer applying safe rules to unsafe ones.

We could always try to classify a needed unsafe rule as safe but then the danger would be that the rule is applied too early and falsifies the subgoal. On the other hand, we could try to manually apply as many of the necessary unsafe rules as possible before invoking the classical reasoner. Note that this in fact happens in virtually all of our proofs to a small extent; most of our reasoning is carried out inductively and we always apply the appropriate induction tactics *before* we attempt any further steps (manual or automatic). It often happens then that the cases which follow straightforwardly by induction can then be solved automatically by Isabelle. However, in the non-trivial inductive cases we typically require careful reasoning involving unsafe rules which are non-trivial to automate.

4.2 Explicit instantiations

When conducting an Isabelle proof, unknowns often arise in the proof state, either quantified at the object level — using the Isabelle/HOL syntax `EX x. P(x)` where P is a predicate — or at the meta-level of the logical framework. In the latter case the unknown is denoted by a question mark, possibly followed by a number of variables indicating the allowed form of the unknown. For example, the unknown expression `?s.x e e` is used to denote an unknown expression `?s` involving (at most) the variables `x, e` and `e'`. Although there are several technical differences between the two types, we consider them to be essentially the same here since we can always convert one into the other using the HOL rules. The most important difference is that meta-level unknowns are permitted to be instantiated when we perform resolution on a subgoal, whereas object-level, existentially quantified unknowns must be instantiated explicitly by the user (although sometimes the automatic tactics will attempt to do this, usually in trivial cases).

Unknowns may be explicitly stated in proof goals (cf. Lemmas 7, 11, 16) as existentially quantified variables or, more often, may be introduced in a proof state at the meta-level as the result of a rule application (c.f. proof of Theorem 20). When this happens, the unknowns will almost certainly have to be instantiated at some point during the proof. Sometimes Isabelle can 'guess' the right instantiation for the proof; for example the instantation for a goal like:

```
Level 29 (1 subgoal)
BCF s --> (EX t. s ->CD t)
 1. !!var lterm t.
    [| BCF(Abs var lterm); lterm ->CD s |] ==>
     EX t. Abs var lterm ->CD t
```

which arises as one of the inductive cases of Lemma 5, is not difficult to guess! On the other hand, if we wanted to explicitly give the required witness for the proof then we could use the resolution-with-instantiation tactic with the rule for existential introduction:

```
by(res_inst_tac [("x","Abs var t")] exI 1);

Level 30 (1 subgoal)
BCF s --> (EX t. s ->CD t)
 1. !!var lterm t.
    [| BCF(Abs var lterm); lterm ->CD s |] ==>
     Abs var lterm ->CD Abs var s
```

Although in the above example the result follows immediately from the definition of the complete development relation `->CD` (and so can be solved by `Blast_tac` either before or after the instantiation), in general we might have considerable work to do even after we provide the witness for a proof. In our development, often we will prove that a reduction exists (to close a diagram, for example) by specifying the witness for the proof halfway through and then working backwards to show that the claimed reduction is enabled, which can be non-trivial. In such situations, the classical reasoner tends to perform badly.

Even worse is the situation which often arises in complex proofs where crucial lemmas must be introduced into the premises of a subgoal at specific stages of the proof. Typically the lemma will take the form of a destruction rule which contains more unknowns than the premise with which it resolves; the 'extra' unknowns are introduced into the proof state and must usually be explicitly instantiated to facilitate the intended usage of the lemma. Here is a simple example. Consider the 'renaming sanity' lemma `x : FV e ==> e[x:=e'] = e` and a proof goal of the form:

```
Level 0 (1 subgoal)
x ~: FV s ==> P
 1. x ~: FV s ==> P
```

where P is some formula to be proved. Suppose that we intend to replace the premise of the goal with the new premise `s[x:=t] = s`, which follows by our renaming lemma. We try performing ordinary destruct-resolution with the lemma on the subgoal as follows:

```
by(dtac renaming_sanity_2 1);

Level 1 (1 subgoal)
x ~: FV s ==> P
1. s[x:=?e'] = s ==> P
```

We see that one unknown has been introduced by the rule application. In trivial cases Isabelle can usually find the right instantiation but, if the goal contains several substitution expressions, the unknown `?e'` may be bound to the wrong identifier. To ensure that this cannot happen, we may use the corresponding instantiation tactic instead:

```
by(dres_inst_tac [("e'","t")] renaming_sanity_2 1);

Level 1 (1 subgoal)
x ~: FV s ==> P
```

```
1. s[x:=t] = s ==> P
```

which gives us the premise we need to complete the proof. The key point is that the classical reasoner (clearly) cannot apply this sort of step because there are simply too many choices for the expression which may be instantiated. Furthermore, in some proofs we will 'cut in' a tautology to the premises of a subgoal. For example, we might decide to add an instance of reflexivity of `-|>B` as follows:

```
by(cut_inst_tac [("e","s")] par_beta_refl 1);

Level 2 (1 subgoal)
x ~: FV s ==> P
 1. [| s[x:=?e'] = s; s -|>B s |] ==> P
```

Although the classical reasoner may well use the rule `par_beta_refl` to resolve a particular subgoal, it will never simply decide to add it to the premises with a particular instantiation this way. For the sake of evenhandedness, we should point out that this kind of forward reasoning is not strictly necessary to conduct proofs; ordinary (backward) resolution suffices. However, using a strictly top-down proof strategy can greatly complicate the proof and lead to an explosion of the search space. Therefore, if a proof incorporates uses of these 'explicit instantiation' tactics then in our experience this is a strong indication that the classical reasoner will encounter difficulties in solving the goal under consideration.

4.3 Side-conditions on rule application

When we originally defined the λ^{var}-calculus in Chapter 1, we observed that the cost of working with one-sorted variable names (as opposed to de Bruijn indices or some other variant) was that we were forced to impose side-conditions on the α- and β-contraction rules ensuring the correctness of the involved substitutions. Unsurprisingly, we then saw that virtually all of our major results involved inductive cases with premises ensuring the appropriate substitutions or reductions were enabled. Apart from the complication of the proofs themselves, the involvement of these premises has a number of immediate consequences for our Isabelle proof developments.

Firstly, performing induction over a goal with a number of premises means that the premises themselves form part of the induction hypothesis. Usually it is a non-trivial matter to show that the premises are conserved over the induction. Furthermore, in many of our proofs we perform case-splits within particular inductive cases, increasing the number of assumptions in the corresponding subgoals. When we have a complicated subgoal to prove, there may be a dozen or more assumptions involved and it is often the case that more than one assumption will match the premise of an inference rule we wish to apply. Since resolution by elimination or destruction always operates on the

leftmost matching assumption of the subgoal, in such cases we are forced to explicitly rotate the assumptions in order to apply the rule correctly by using `rotate_tac` or a similar 'subgoal management' tactic. For manual proofs this is nothing more than an inconvenient detail (although it does add to the 'brittleness' of a proof). For the automatic tactics, however, it can lead to a factorial blowup in the search space because for each rule application there are several premises to which it may 'sensibly' be applied. Furthermore, the number of rules themselves which may be applied is greater because at each stage the classical reasoner may operate on any of several premises, each of which may enable many inference rule applications.

One might ask, given that rules are applied to the leftmost applicable assumption, whether it would be possible to preorder the assumptions before invoking the reasoner such that the automatic tactics always operate on the 'right' assumption before trying the other possibilities. In our experience, this can work in some cases; putting a particular assumption first in the subgoal can improve the performance of `Blast_tac` and other tactics. However, it should be clear that it simply is not practical in general because assumptions are constantly created and destroyed by the application of inference rules.

4.4 Inherent complexity

We have given some 'rules of thumb' for determining whether or not the fully automatic tactics of Isabelle are likely to succeed. We remark that in order to ascertain whether these heuristics are applicable, one does need to know something about the structure of the proof itself (and hence whether or not many premises are involved, whether explicit instantiations are heavily required and so on). We also note that, in one sense, the heuristics we have given are essentially measures of the "inherent complexity" of a proof. In other words, a proof which is inherently 'hard' (for humans) is more likely to involve many unsafe rule applications, to contain many assumptions and to require the usage of more auxiliary lemmas which may need to be instantiated appropriately, than a proof which is inherently 'easy'.

The type of proofs at which Isabelle excels are those in which there are many inductive cases but in which each case is relatively straightforward, which would be laborious by hand but is easily solved by the classical reasoner. In contrast, we often work with proofs which are quite deep (in the literal as well as metaphorical sense!) and in the non-trivial inductive cases require many manual tactic applications to prove. We feel that it is unreasonable to expect such proofs to be amenable to "mindless search procedures (aka tactics)" (Nipkow). On the other hand, one should note that Isabelle does provide many *semi-automatic* tactics, which apply both the classical reasoner and the simplifier in a limited manner to perform all safe logical inferences and / or equational rewrites, and we make liberal use of these throughout our development. Additionally, one may create customised automatic or semi-automatic strategies in

Isabelle either by writing ones own tactics from scratch or by combining tactics using (built-in or user-defined) tacticals. We have not explored this possibility during the project and leave it as a possible direction for further study.

5. Conclusion

In chapter 1 we introduced the untyped λ-calculus as a model of computation particularly useful for characterising the notion of a function. We then examined a few alternative presentations of the λ-calculus with reference to formalisation of machine proofs. Notably, well-known equational properties of the simplest presentation of the calculus — first order abstract syntax with one-sorted variable names — have, to the best of our knowledge, resisted machine formalisation in a setup relying only on the standard (inductive) proof principles of the calculus and its defined reduction relations. We explained the major issues pertaining to formalisation in this setting and defined the raw λ^{var}-calculus with which we conduct our proofs.

The main contribution of this dissertation is an account of the Isabelle formalisation of the first proof of β-confluence in the λ-calculus using one-sorted variable names, which was first published by Rene Vestergaard and the author in [17]. In chapter 2 we presented the main steps of the proof and accounted for some of the specifics of their corresponding Isabelle mechanisations, introducing new syntax and concepts as we proceeded. In particular, we saw in section 2.8 how we deduced confluence of the real λ-calculus by forming the equivalence classes of terms modulo α and applying a completely abstract result concerning preservation and reflection of confluence across a structural collapse between two rewrite systems.

The proof methodology we use is fairly complex and not obviously applicable to other results in the literature; in chapter 3 we presented a technical property of the residual theory of λ^{var} (again published by Vestergaard and the author in [18]) showing that no α-renaming steps are required on BCF terms in proofs which respect the residual theory. This shows that other equational properties of the calculus are amenable to formalisation along the lines of our confluence proof.

Finally, noting that our Isabelle/HOL implementation of the results of Chapters 2 & 3 is rather large and complex, we asked why our proof goals were not always amenable to the automatic proof tactics provided by Isabelle. Although we were not able to provide hard and fast answers, we highlighted some characteristics of many of our proofs and attempted to explain on a technical level why these posed problems for Isabelle's classical reasoning engine. We also suggested that these characteristics could be taken as a measure of the inherent difficulty of a proof (which naturally only makes sense when discussing proofs in Isabelle). We did not fully account for this conjecture and for now we leave it for others to investigate.

One potential direction for future research would be to attempt the Isabelle formalisation of other equational properties in the λ-calculus, using our own theory files as a basis for the development. Such properties could include strong normalisation, confluence of $\beta \cup \eta$, and the η-over-β Postponement Theo-

rem. Another possibility might be the development of custom tactics and proof strategies to increase the extent to which automation of our proof development is feasible.

Bibliography

[1] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics (Revised Edition)*. North-Holland, 1984.

[2] Rod Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12, 1967.

[3] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[4] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[5] 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.*, 34:381–392, 1972.

[6] J. Ford and I. A. Mason. Operational Techniques in PVS – A Preliminary Evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*, 2001.

[7] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. Von Wright, J. Grundy, and J. Harrison, editors, *(TPHOL'96)*, volume 1125 of *LNCS*. Springer Verlag, 1996.

[8] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Proceedings of HUG'93*, Springer Lecture Notes in Computer Science. Springer-Verlag, 1994.

[9] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proceedings of LICS'99*, pages 204–213, 1999.

[10] Peter V. Homeier. A proof of the Church-Rosser Theorem for the lambda calculus in higher order logic. Technical report, US Department of Defense, 2001. Available from http://www.cis.upenn.edu/ homeier.

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

[12] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.

[13] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In *Proceedings of CADE-13*, 1996. LNCS 1104.

[14] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. Isabelle's Logics: HOL. Technical report, Computer Laboratory, University of Cambridge, October 1999. Available from http://www.cl.cam.ac.uk/Research/HVG/Isabelle/cambridge.htm.

[15] Lawrence C Paulson. The Isabelle Reference Manual. Technical report, Computer Laboratory, University of Cambridge, October 1999. Available from http://www.cl.cam.ac.uk/Research/HVG/Isabelle/cambridge.htm.

[16] Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118:120–127, 1995.

[17] René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names (*Barendregt was right after all ... almost*). In Aart Middeldorp, editor, *Proceedings of RTA-12*, volume 2051 of *LNCS*. Springer-Verlag, 2001.

[18] René Vestergaard and James Brotherston. The mechanisation of Barendregt-style equational proofs (the residual perspective). In *Proceedings of MERLIN-1*, 2001. Collocated with IJCAR'01.

[19] Joe Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In Gert Smolka, editor, *Proceedings of ESOP-9*, volume 1782 of *LNCS*. Springer Verlag, 2000.

[20] Markus Wenzel. The Isabelle System Manual. Technical report, TU München, February 2001. Available from http://www.cl.cam.ac.uk/Research/HVG/Isabelle/cambridge.htm.

Appendix A

The following appendix contains the transcripts of theories comprising the complete Isabelle development of the results presented in the dissertation. For each theory of the development, the associated transcript lists the contents of the theory files (i.e. the type, function and relation definitions) and the statements of the lemmas proved. The full tactic scripts are *not* included here but are available online from http://www.dcs.ed.ac.uk/~jjb/.