Technical Report

UCAM-CL-TR-177
ISSN 1476-2986

Number 177

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Experimenting with Isabelle in ZF Set Theory

P.A.J. Noel

September 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Experimenting with Isabelle in ZF set theory

P.A.J. Noel
Computer Laboratory
University of Cambridge

7 September 1989

Abstract

The theorem prover Isabelle has been used to axiomatise ZF set theory with natural deduction and to prove a number of theorems concerning functions. In particular, the axioms and inference rules of four theories have been derived in the form of theorems of set theory. The four theories are:

- $\lambda_{\beta\eta}$, a form of typed lambda calculus with equality,
- $\mathcal{Q}_0$, a form of simple type theory,
- an intuitionistic first order theory with propositions interpreted as the type of their proofs,
- PPλ, the underlying theory of LCF.

Most of the theorems have been derived using backward proofs, with a small amount of automation.

Contents

1 Introduction

Although various forms of set theory have been used in attempting to formalise the foundation of mathematics, set theory is often considered too clumsy to use for reasoning about functions. Some higher order formal systems are generally considered more suitable. However, because of the intuitive aspect of set theory and the acquired knowledge about its properties, attempts are often made to express the semantics of the higher-order formal systems in terms of set theory (see [3] for the case of type theory). For the same reasons, new and more expressive set theories are also considered: for instance, the theory of non-well-founded-sets [1] allows a set to belong to itself (taking 'belong' to be a transitive relation), and, consequently, allows self-application of set-theoretic functions, as well as the definition of a type of types.

The aim of my research was twofold: firstly to develop, within ZF set theory, a collection of useful theorems concerning functions; secondly to prove, within the theory, the axioms and inference rules of some commonly used theories. The theories in question are:

- typed lambda calculus with equality,
- simple type theory $\mathcal{Q}_0$, as formalised in [2],
- a first order theory, with propositions interpreted as the type of their proofs,
- PPλ, the underlying theory of LCF.

The set theoretic concept of function is adequate for each of these theories. It is worth noting here that, when defining a function in set theory, one has the choice to use either a function of predicate logic, or a set theoretic function. The application of a function of the first kind, say $f1$, to the term a is expressed by $f1(a)$. The application of a function of the second kind, say $f2$, to a will be expressed by $f2`a$. When the type of a is known, it is preferable to use the second form in order to be able to reason about the function. The function $f1$ is too big to be an object of set theory, since its domain is not a set. However, it is possible to define the restriction of $f1$ to a set A. In the following, such a restriction will be named $lam(A, f1)$.

Despite its limitations, the concept of set theoretic function is in fact more expressive than the concept of function in simple type theory. The basic set in a model of simple type theory is a countable set. In set theory, the set of natural numbers, ω, is modelled by such a set. Thus, the functions of various orders which may be expressed in simple type theory are also expressible in set theory. Many more functions may be defined in set theory since there exists sets, such as the transfinite ordinals, which may not be constructed in simple type theory.

Isabelle is a theorem prover well suited for the task of deriving axioms and inference rules, since, in their general form, Isabelle's theorems are inference rules. Furthermore, higher order unification, which is provided in Isabelle, is necessary

when using the schematic axioms and rules of set theory, or any schematic theorem which may be derived in it. I have extended the intuitionistic first order theory, already set up in Isabelle, first to classical first order logic, then to ZF set theory. The other theories have been derived within set theory.

The main part of the paper is divided in three sections: section 2 presents what has been proved, and the definitions used in the process; section 3 is concerned with some of the issues relating to the proofs themselves; section 4 contains a brief description of some related work.

2 ZF set theory and the derived theorems

The axioms and rules of inference which have been chosen for the set theory are presented first. Then a set of definitions relevant to functions is introduced, and some of the theorems concerning them are discussed. The axioms of $\lambda_{\beta\eta}$, in their explicitly typed form, are included in these theorems. Finally, the axioms and rules of inference for the other theories mentioned in the introduction are presented, expressed in their set theoretic form.

2.1 ZF set theory

Set theory has been constructed on top of the first order intuitionistic logic already defined in Isabelle. The basic axioms of this logic are expressed in a natural deduction style ([14],[16]). The logic is first extended by two inference rules:

- the substitution of equals by equals, 'eq-elim', which is required for a logic with equality:

$$\frac{a = b \qquad P(b)}{P(a)}$$

  The axioms concerning the reflexivity, symmetry and transitivity of equality are already in the basic theory. In fact, the symmetry and transitivity axioms may be derived from the above axiom and the reflexivity axiom (which may be seen as an 'equality introduction' rule)

- an axiom used to extend intuitionistic logic to classical logic, the refutation axiom:

$$\frac{\begin{array}{c}[Not(A)] \\ A\end{array}}{A}$$

  with the following definition of Not:

$$Not(A) \leftrightarrow (A \longrightarrow False)$$

The theory is then extended to include the ZF axioms displayed on page 6. Here are some remarks concerning the notations: the symbol ':' stands for the membership predicate $\in$; the term $Collect(A, P)$ stands for $\{x \in A \mid P(x)\}$; the term $Replace(f, B)$ stands for $\{f(x) \mid x \in B\}$. An alternative notation for $Collect(A, P)$ is $[x \| x : A, P(x)]$. An alternative notation for $Replace(f, B)$ is $[f(x) \| x : B]$. The double square brackets convert object-level formulae into meta-level propositions. '$\Longrightarrow$' is the Isabelle symbol for the meta-level implication; it may be understood as representing an object-level inference. The theorems of Isabelle are generally in the form of derived inference rules. The constant symbols of set theory have been defined with appropriate types within Isabelle. The symbols which have not been defined are taken to be universally quantified meta-level variables. In ZF6, for instance, x and A are universally quantified over the terms of the theory, and P over the functions from terms to formulae. From a syntactic point of view, the variables quantified over terms may be seen as standing for free variables, and the other quantified variables for schematic variables (i.e. uninstantiated constants) of various orders. Meta-level variables within the scope of different quantifiers stand for different object level variables. When a theorem (or an axiom) is used in a proof, the meta-level variables for which the scope of the quantification is the whole theorem are interpreted as schematic variables and may thus become instantiated through unification during the proof.

In [17] and [8], ZF1 is axiomatized as $A \subseteq B \ \& \ B \subseteq A \longrightarrow A = B$. However, the bi-implication is easily obtained by using the substitution rule of equality (eq_elim). Note that ZF1 may also be introduced as an extension of classical logic without equality. In [9] for instance, equality is defined by the extension axiom of page 6, and the extension axiom becomes the axiom schema:

$$\forall x. \forall y \,.\, x = y \longrightarrow (P(x) \leftrightarrow P(y))$$

From these, the properties of equality are derivable.

Usually, the axioms ZF2 to ZF8 are given in the form of existentially quantified statements: e.g. 'there is a set with only elements a and b' (pairing). In each case, the uniqueness of the set may be proved, thus allowing the definition of a new term to represent it. For ZF3 to ZF7, these definitions are used as axioms, in place of the usual existential ones. This is the same form of axiomatisation as the one for the sequent form of ZF in [14].

There are many ways of formalizing the 'null' axiom:

- as an existential statement about the null set
- as an expression of the defining property of the null set, if the null set is taken as a primitive symbol
- as a definition of the null set

The axiom may even be omitted completely, since the existence and uniqueness of a null set may be derived from the other axioms. The formalisation which has been chosen here is the second one above: 0 is taken as a primitive symbol.

ZF axiomatisation

```
ZF 1 - extension    [| A = B <-> A <= B & B <= A |]

ZF 2 - null:        [| Not(a:0) |]

ZF 3 - pairing:     [| x : {a,b} <-> x=a | x=b |]

ZF 4 - Union:       [| A : Union(C) <-> EXISTS B. A:B & B:C |]

ZF 5 - Power:       [| A : Pow(B) <-> A <= B |]

ZF 6 - Collect:     [| x : Collect(A,P) <-> x : A &  P(x) |]

ZF 7 - Replace:     [| x : Replace(f,B) <-> EXISTS a. a : B & x=f(a) |]

ZF 8 - INF_0:       [| 0:INF |]
       INF_succ:    [| n:INF |] ==> [| succ(n):INF |]

ZF 9 - foundation: [| Not(A=0) |] ==> [| EXISTS u. u:A & u Int A = 0 |]
```

Useful Definitions

```
subset:         [| A <= B <-> ALL x. x:A --> x:B |]

strict_subset: [| A << B <-> A <= B & Not(A=B) |]

Un:             [| A Un B = Union({A,B}) |]

succ:           [| succ(n) = n Un {n} |]

Inter:          [| Inter(C) = [ x || x:Union(C), ALL y. y:C --> x:y] |]

Int:            [| A Int B = Inter({A,B})   |]

Dif:            [| A - B = [ y || y: A, Not(y:B)] |]

singleton:      [| {a} = {a,a} |]

Pair:           [| <a,b> = {{a},{a,b}} |]

Hd:             [| Hd(A) = Union(Inter(A)) |]

Tl:             [| Tl(A) = Union([ X || X:Union(A),
                                        Not(X:Inter(A)) | Union(A) = Inter(A)])|]
Product:        [| A*B = [ x || x:Pow(Pow(A Un B)),
                              EXISTS a. EXISTS b. a:A & b:B & x= <a,b>]|]
```

ZF8 ensures the existence of an infinite set. In the chosen axiomatisation, the primitive symbol INF represents one such set, and its defining properties are given by INF_0 and INF_succ. Note that INF has been introduced for convenience only. It is possible to define the set of natural numbers, ω, directly from the existential version of ZF8: it is the intersection of the sets which satisfy the properties specified by the axiom. It can then be shown that ω itself satisfies these properties.

The foundation axiom is given in its usual form. From it will be derived the theorems:

$$\neg(a \in a) \qquad \textit{and} \qquad \neg(a \in b \;\&\; b \in a)$$

Using the definition of Hd and Tl, the following theorems may be derived:

- $Hd(A)$ is the first element of A if A is an ordered pair
- $Tl(A)$ is the second element of A if A is an ordered pair

2.2 Relations and functions

2.2.1 Definitions

New concepts are introduced in a theory through definitions. If the form of the definitions satisfies adequate criteria, adding definitions does not alter the basic theory, but merely provides syntactic variants for some of the expressions of the theory. For this to be the case, the definition of terms must satisfy the following requirements (see [17], chapter 2):

- non-creativity criterion: a definition satisfies this criterion if the only sentences not containing the new symbol which are provable in the theory with the new definition are the ones which were provable in the theory without it. Adding definitions satisfying this criterion to a given theory results in what is sometimes called a 'conservative extension' of the theory.
- eliminability criterion: a definition satisfies this criterion if every sentence which contains the new symbol is equivalent to one without the new symbol.

Often, conditional definitions seem appropriate. For instance, the application of a term to another term is definable only if the first term is a function, and the second term is from an appropriate set. Conditional definitions, however, satisfy the criterion of non-creativity, but not the criterion of eliminability. If function application, for instance, is defined by a conditional definition, then any sentence involving $f\hat{}a$ could not be eliminated in favour of a sentence not involving $\hat{}$ when a does not belong to the domain of f or f is not a function. On the other hand, the use of an unconditional definition forces us to give a meaning to $f\hat{}a$, even when f is not a function or a is not in the domain of f, creating the possibility of proving non-sensical theorems.

The choice which has been made here was to stick to unconditional definitions, most of them of the simple form:

$$new\_symbol(x_1, x_2, ..., x_n) = exp(x_1, x_2, ..., x_n)$$

where $\{x_1,x_2,...\}$ is a set of variables, possibly empty, and $exp(x_1, x_2, ...)$ is a term not involving the new symbol (recursive definitions are thus disallowed) and having $x_1,x_2,...$ as the only free variables. Since the actual domain of application of the definitions may be greater than the intended one, care must be taken that the theorems involving the defined terms include the appropriate typing information.

When defining new symbols in this way, the existence and uniqueness of the defining term is obviously ensured. However, to ensure that the meaning of the new symbol is the desired one, some conditions of existence and uniqueness must normally be satisfied. For instance, reasoning about *the* least fixed point is appropriate, only if the set of least fixed points is a singleton set. Checking that these conditions are satisfied constitutes a significant part of the work involved in proving theorems within set theory. Such checks may not be ignored without losing preciseness in the meaning of the defined terms.

Two of the definitions on pages 9-10 have a form which differs slightly from the simple form above. However, the terms concerned are only syntactic variant of well formed terms of set theory. First, the definition of lambda abstraction involves a function variable: in $lam(A, E)$, E is a function variable. The expression defining $lam(A, E)$ is a first order term only if E is not quantified. This condition is enforced automatically in Isabelle by specifying in the definition of *lam* that the second argument is a function from terms to terms: since quantification in first order logic is restricted to be over terms, the second argument cannot be quantified. The same kind of remark applies to $Collect(A, P)$, in which P stands for a function from terms to formulae. Second, the definition of *Curry* involves a particular representation of abstractions: one using %, which is available in Isabelle. Its use within *lam* is acceptable since $lam(A, \%(x)E(x))$ may always be replaced by its well formed definiens $\{\langle x, E(x)\rangle \| x : A\}$.

The remainder of this section consists of comments concerning the definitions of pages 9-10.

The definitions of *Domain*, *Range* and *Image* apply to any set, not just relations.

$Partial\_order(D)$ is the set of partial orders defined on the underlying set D. Given a partial order, say R, its underlying set is completely determined, and may be refered to either by $Domain(R)$ or $Range(R)$. If D is the underlying set of a partial order R, and Y is a subset of D, $Lubs(R, Y)$ is the set of least upper bounds of Y in D. It has been proved that, if this set is not empty, it is a singleton, thus justifying the definition of the least upper bound $Lub(R, Y)$ as $\bigcup(Lubs(R, Y))$. Note, however, that if $Lubs(R, Y)$ is empty, $Lub(R, Y)$ is defined as 0, even though there is no least upper bound.

Definitions concerning relations

```
*** General definitions concerning relations ***

     Reflexive(D) = [ R || R:Pow(D*D), ALL x. x : D --> <x,x> : R]

Anti_symmetric(D) = [ R || R:Pow(D*D),
                           ALL x. ALL y. <x,y> : R & <y,x> : R --> x=y]

    Transitive(D) = [ R || R:Pow(D*D), ALL x. ALL y. ALL z.
                           <x,y> : R & <y,z> : R --> <x,z> : R]

        Domain(C) = [ a || a:Union(Union(C)), EXISTS b. <a,b> : C]

         Range(C) = [ b || b:Union(Union(C)), EXISTS a. <a,b> : C]

       Image(R,X) = [ y || y : Range(R), EXISTS x. x:X & <x,y> : R]

*** Transitive closure ***

T_clos(R) = Inter([ S || S:Pow(Domain(R)*Range(R)), R<=S &
                    ALL x. ALL y. ALL z. <x,y> : S & <y,z> : R --> <x,z> : S])

Init(R,x) = [ y || y:Domain(R), <y,x> :T_clos(R) ]

*** Orders ***

Partial_order(D) = [ R || R:Pow(D*D), R : Reflexive(D) &
                          R : Anti_symmetric(D) & R : Transitive(D)]

  Total_order(D) = [ R || R:Partial_order(D),
                          ALL x. ALL y. x:D & y:D --> <x,y> : R | <y,x> : R]

*** Upper bounds and least upper bound ***

 Ubs(R,Y) = [ y || y : Domain(R), R : Partial_order(X) &
                   Y <= X & ALL x. x:Y --> <x,y> : R]

Lubs(R,Y) = [ y || y : Ubs(R,Y), ALL z. z : Ubs(R,Y) --> <x,z> : R]

 Lub(R,Y) = Union(Lubs(R,Y))

*** Complete lattice ***

Complete_lattice(X) = [ R || R:Partial_order(X),
                             ALL Y. Y <= X --> Not(Lubs(R,Y) = 0)]

*** Well-founded relation ***

Well_founded(X) = [ R || R:Pow(X*X), ALL Y. Y <= X & Not(Y=0)
                         --> EXISTS x. x:Y & Not(EXISTS y. y:Y & <y,x> : R)]
```

Definitions concerning functions

```
*** Functions ***

Function(A,B) = [ F || F:Pow(A*B),
                       ALL x. ALL y. ALL z. <x,y> : F & <x,z> : F --> y=z]

*** Total functions ***

        A->B = [ F || F:Function(A,B), ALL x. x : A --> EXISTS y. <x,y> : F]

*** Application and lambda abstraction ***

          F`a = Union(Image(F,{a}))
     lam(A,E) = [ <x,E(x)> || x:A]

*** Composition  ***

        f @ g = [ X || X: Domain(f)*Range(g),
                       X=<x,y> & EXISTS z.  <x,z>:g & <z,y>:f ]

*** Currying function ***

     Curry(f) = lam(First(Domain(f)),%(x)lam(Second(Domain(F)),%(y)f`<x,y>))

*** Monotone functions ***

Monotone(RA,RB) = [ F || F:Domain(RA)->Domain(RB),
                         ALL x. ALL y. <x,y> : RA --> <F`x,F`y> : RB ]

*** Fixed points ***

      Fix(F) = [ x || x : Domain(F), F`x = x ]
 Lfixs (F,R) = [ x || x : Fix(F), ALL y. y:Fix(F) --> <y,x> : R ]
  Lfix (F,R) = Union(Lfixs(F,R))

*** Definitions concerning the natural numbers ***

successor_set(A) = [ X || X:Pow(A), 0:X & ALL x. x:X --> succ(x):X ]
           Omega = Inter(successor_set(INF))

*** Function defined by simple recursion: recs(f,a)  ***

   recs(f,a) = Inter([ R || R:Pow(Omega*Domain(f)), <0,a> : R &
                            ALL n. ALL y. <n,y> : R --> <succ(n), f`y> : R ])

*** Function defined by well-founded recursion: wrec(X,Y,R,f)   ***

 Restrict(f,R,x) = [ <y,f`y> || y:Domain(f), y : Init(R,x) ] |]

 wrec_s(X,Y,R,f) = [ F || F : Function(X,Y), ALL x. x : Domain(F)
                                  --> F`x = f`x`Restrict(F,R,x)
                                      & ALL y. y:Init(R,x) --> y : Domain(F) ]

  wrec(X,Y,R,f) = Union(wrec_s(X,Y,R,f))
```

A well-founded relation has been defined here as a relation in which every subset has a minimal element. The well-founded induction may then be shown to apply to any set on which a well-founded relation is constructed. However, this result would not be true in intuitionistic logic. An alternative definition of a well-founded relation is simply that it is a relation satisfying the well-founded induction.

The definition of function application is justified by a theorem stating that, when f is a function and the image of a under f exists, then that image is unique. Care must be taken, however, when using application: the result of the application has always a value. Normally, most theorems concerning an application $f\hat{}a$ should be subject to the conditions that f is a function, and a is in the domain of f.

Composition is defined for any set f and g, not just functions. Similarly, $recs(f,a)$ and $wrec(X,Y,R,f)$ are defined for any set f, even though the intended meaning of the definitions is restricted to the case where f is a function. Also, $Curry(f)$ is defined for any set f and its intended meaning is restricted to the case where f is a function, the domain of which is a product.

It has been proved that, if f is a total function from D to D and R is a partial order over D, then the set of least fixed points of f, $Lfixs(f,R)$, is either the empty set or a singleton set. Theorems refering to the least fixed point, $Lfix(f,R)$, should be subject to the condition that $Lfixs(f,R)$ is not empty.

Suppes [17] defines natural numbers, i.e. the members of ω, as the ordinals which are well-ordered by the inverse of the membership relation (defined on them). An ordinal is a complete set (or transitive set, i.e. such that every member is also a subset of the set), connected by the membership relation. Because it expresses more explicitly the properties of the membership relation over the natural numbers and the ordinals, such a definition could result in simpler proofs than the ones resulting from the definition of page 10. For example, the theorem stating that, for any two ordinals m amd n, either $m \in n$ or $n \in m$ or $n = m$ is a direct consequence of the property of connectedness in Suppes' definition.

The simple recursion theorem states that, if f is a function of type $A \rightarrow A$ and a is an element of A, then there exists a unique function g of type $\omega \rightarrow A$ with the following properties:

$g\hat{}0 = a$

$\forall n \,.\, n \in \omega \longrightarrow g\hat{}succ(n) = f\hat{}(g\hat{}n)$

The definiens in the definition of the term $recs(f,a)$, on page 10, represents one possible expression of this function. The proof that $recs(f,a)$ is actually a function follows the informal proof of [8]. An alternative definition is

$$recs(f,a) = \bigcup(\{recs\_s(f,a,n) \mid n \in \omega\})$$

with

$$recs\_s(f,a,n) = \{g \in n \rightarrow A \mid g\hat{}0 = a \;\&\; \forall\, i \,.\, i \in n \longrightarrow g\hat{}succ(i) = f\hat{}(g\hat{}i)\}$$

A proof that $recs(f,a)$, as defined here, is a function is given in [17].

Finally, notice that the term $lam(A, \%(x)f\hat{}x)$ may not be used to describe the restriction of a partial function, f, to the set A. This is because $f\hat{}x$ is defined for every value of x, and thus $lam(A, \%(x)f\hat{}x)$ is a total function over A, whereas the restriction of f to A need not be total on A. *Restrict* has been defined to remedy this problem: every element of $Restrict(f, R, x)$ is an element of f.

2.2.2 Some theorems concerning functions

Some theorems concerning functions, proved within set theory, are listed on pages 13-14. First, an explanation concerning the notations may be useful. The symbol '!', which occurs in several theorems, is the Isabelle notation for the meta-level universal quantifier. The typing rule for abstractions, for instance, should be understood as

$$(\bigwedge x\,.\,[\![x : A]\!] \Longrightarrow [\![E(x) : B]\!]) \Longrightarrow [\![lam(A, E) : A \to B]\!]$$

where $\bigwedge$ is the meta-level universal quantifier. From a syntactic point of view, the variable x within the scope of the quantifier may be taken to represent a free variable, chosen in such a way that there is no occurence of it outside the scope of the quantifier.

Among the theorems listed, there is a set of axioms for a typed λ calculus with equality. The main difference between this form of λ calculus and the more standard forms ($\lambda_{\beta\eta}$ with equality in [10] pages 162-165, for instance) lies in the representation of typing. In set theory, typing must be explicit; it is not part of the syntax. Accordingly, there are two rules related to typing inferences, one concerning application, the other abstraction. The β, η and ξ rules correspond to the rules of the same name in $\lambda_{\beta\eta}$. The other rules of $\lambda_{\beta\eta}$ are simply instances of the substitution rule. Note that the rules which have been proved within set theory are more general than the rules of $\lambda_{\beta\eta}$, since $E(x)$ and $F(x)$ are not restricted to be in functional form.

Next, some general properties of functions are listed. The theorems concerning the least upper bound are used here in the proof of Tarski's fixed point theorem, and later, in the proof of theorems concerning domain theory. For the sake of clarity, only the most significant of the theorems which have been proved are displayed on page 13. Among the more complicated theorems which are not listed, there is the following one, required to prove that the function space induced by two cpo's is itself a cpo:

$$\frac{\forall m\,.\,m \in Y \longrightarrow \neg(Lubs(R, \{P(m,n) \mid n \in X\}) = 0)}{Lubs(R, \{Lub(R, \{P(m,n) \mid n \in X\}) \mid m \in Y\}) = Lubs(R, \bigcup(\{\{P(m,n) \mid n \in X\} \mid m \in Y\}))}$$

i.e. given a partition of a set, if every subset in this partition has a lub, then the lub of the set of lub's of the subsets is the lub of the given set.

General theorems concerning relations and functions

```
*** Uniqueness of the image of a singleton under a function ***
    (this theorem justifies the definition of application)

[| f : Function(A,B) |] ==> [| a : Image(f,{x}) |] ==> [| b : Image(f,{x}) |]
   ==> [| a=b |]

***  Typed lamda calculus ***

  1. typing rules

application type  [| f : A->B |] ==> [| x: A |] ==> [| f`x : B |]
abstraction type  (!(x)[| x:A |]==>[| E(x):B |]) ==> [| lam(A,E) : A->B |]

  2.  main rules

beta  conversion  [| a : A |] ==> [| lam(A,E) ` a = E(a) |]
eta conversion    [| f : A->B |] ==> [| lam(A,%(x)f`x) = f |]
xi rule       (!(x)[| x:A |]==>[| E(x) = F(x) |]) ==> [| lam(A,E) = lam(A,F)|]

***  General properties of functions  ***

extensionality [| f : A->B ] ==> [| g : A->B |]
                  ==> [| f=g <-> ALL x. x:A --> f`x = g`x |]
total functions with empty domain  [| 0->B = {0} |]
total functions with empty range   [| Not(A=0) |] ==> [| A->0 = 0 |]
Composition (1) [| f : B->C |] ==> [| g : A->B |] ==> [| f @ g : A->C |]
            (2) [| f : B->C |] ==> [| g : A->B |] ==> [| a : A |]
                  ==> [| (f @ g) ` a = f ` (g`a) |]
currying (1) [| f : (A*B)->C |] ==> [| Not(B=0) |] ==> [| Curry(f) : A->B->C|]
         (2) [| f : (A*B)->C |] ==> [| x : A |] ==> [| y : B |]
               ==> [| Curry(f)`x`y = f` <x,y> |]

*** Properties of Lub's  ***

Uniqueness    [| a : Lubs(R,Y) |]  ==> [| b : Lubs(R,Y) |] ==> [| a=b |]
Lub is an ub  [| Not(Lubs(R,Y) = 0) |] ==> [| x:Y |] ==> [| <x,Lub(R,Y)> : R|]
Lub is least  [| Not(Lubs(R,Y)=0)  |] ==> [| x : Ubs(R,Y)  |]
                ==> [| <Lub(R,Y),x> : R |]
Lubs of pairs (1) [| Not(Lubs(RA,{x,y})=0) |] ==> [| Lub(RA,{x,y})=y |]
                    ==> [| <x,y> : RA |]
              (2) [| RA : Partial_order(A) |] ==> [| <x,y> : RA |]
                    ==> [| y : Lubs(RA,{x,y}) |]
```

Fixed point and recursion theorems

```
*** Fixed point ***

basic properties
[| R : Partial_order(A) |] ==> [| x : Lfixs(F,R) |] ==> [| y : Lfixs(F,R) |]
   ==> [| x = y |]
[| R : Partial_order(A) |] ==> [| Not(Lfixs(f,R) = 0) |]
   ==> [| f^Lfix(f,R) = Lfix(f,R) |]
[| R : Partial_order(A) |] ==> [| Not(Lfixs(f,R) = 0) |] ==> [| x : Fix(f) |]
   ==> [| <Lfix(f,R),x> : R |]

Properties of inverse relations (used in the fixed point theorem)
[| R : Complete_lattice(A) |] ==> [| Inv(R) : Complete_lattice(A) |]
[| R : Partial_order(A) |] ==> [| f : Monotone(R,R) |]
   ==> [| f : Monotone(Inv(R),Inv(R)) |]

Tarski's fixed point theorem
[| f : Monotone(Lat, Lat) |] ==> [| Lat : Complete_lattice(A) |]
     ==> [| Glb(Lat, [ u || u : A, <f^u,u> : Lat ]) : Lfixs(f,Lat) |]
[| f : Monotone(Lat, Lat) |] ==> [| Lat : Complete_lattice(A) |]
     ==> [| Lfix(f,Lat) = Glb(Lat, [ u || u : A, <f^u,u> : Lat ]) |]

*** Simple recursion ***

Omega          [| 0 : Omega |]
               [| n : Omega |] ==> [| succ(n) : Omega |]
               [| n : Omega |] ==> [| m : n  |] ==> [| m <= n |]
               [| m : Omega |] ==>[| n : Omega |] ==> [| m:n | n:m | m=n |]
               [| n : Omega <-> n=0 | EXISTS m. n=succ(m) & m:Omega |]

Mathematical (1) [| X <= Omega  |] ==> [| X : successor_set(A) |]
induction           ==> [| X = Omega |]
             (2) [| a : Omega  |] ==> [| P(0) |]
                    ==> [| ALL n. n: Omega --> P(n) --> P(succ(n)) |]
                    ==> [| P(a) |]

simple       (1) [| f : A->A |] ==> [| a : A |] ==> [| recs(f,a) : Omega->A |]
recursion    (2) [| f : A->A |] ==> [| a : A |] ==> [| recs(f,a)^0 = a  |]
             (3) [| f : A->A |] ==> [| a : A |] ==> [| n : Omega |]
                    ==> [| recs(f,a)^succ(n) = f^recs(f,a)^n |]

*** well-founded recursion ***

well-founded   [| a : X |] ==> [| R : Well_founded(X) |]
induction         ==> [| ALL x. x:X --> (ALL y. <y,x> : R --> P(y) --> P(x) |]
                  ==> [| P(a) |]

well-founded (1) [| R : Well_founded(X) |] ==> [| f : X->Pow(X*Y)->Y |]
recursion           ==> [| wrec(X,Y,R,f) : X->Y |]
             (2) [| R : Well_founded(X) |] ==> [| f : X->Pow(X*Y)->Y |]
                    ==> [| x : X |]
                    ==> [| wrec(X,Y,R,f)^x = f^x^Restrict(wrec(X,Y,R,f),R,x)|]
```

The greatest lower bound of a partial order, which is used in Tarski's theorem, is simply the lub of the inverse partial order. The properties of inverse relations relevant to Tarski's theorem have been proved. In particular, the theorem concerning the inverse of a complete lattice justifies the definition of a complete lattice as simply a complete upper semi lattice: the theorem states that a complete upper semi-lattice is also a complete lower semi-lattice, and is therefore a complete lattice. Two theorems are required to express Tarski's fixed point theorem. The two are necessary: if the glb in the second theorem is the empty set, the theorem asserts that $Lfix(f, Lat)$ is also the empty set. However, this could mean either that the least fixed point is the empty set, or that the least fixed point does not exist. It is the first theorem which asserts the existence of the least fixed point. Note that the second theorem would be sufficient to express Tarski's theorem if the definition of the least fixed point was a conditional definition, restricted to the case where the set of least fixed points is not empty.

The basic properties of ω lead to the two forms of mathematical induction, which are themselves used in the proof of the simple recursion theorem. A proof of the recursion theorem for primitive recursion may be derived from the proof for simple recursion. First, notice that the definition of $recs(f, a)$ includes the case where a and $f\hat{}b$ (for $b : A$) are functions of the same type. In the case of curried functions, the conditions satisfied by *recs* may then be rewritten:

$$recs(f, a)\hat{}0\hat{}x_1\hat{}...x_m = a\hat{}x_1\hat{}...x_m$$

$$\forall n \,.\, n \in \omega \longrightarrow recs(f, a)\hat{}succ(n)\hat{}x_1\hat{}...x_m = f\hat{}(recs(f, a)\hat{}n\hat{}x_1\hat{}...x_m)\hat{}x_1\hat{}...x_m$$

(for any instance of x_1, $...x_m$ satisfying the type restriction on a and f).

Thus, to obtain primitive recursion, it remains only to allow the given function, f, to depend on the iteration step, n (i.e. $f = h\hat{}n$). This could be achieved by redefining *recs* as a function of h rather than f, and proving the corresponding properties. Alternatively, a primitive recursive function based on the function h, of type $\omega \to A \to A$, and the set a, of type A, could be derived from simple recursion in the following way:

1. define a function f of type $\omega * A \to A$ by $f(x) = \langle succ(Hd(x)), h\hat{}Hd(x)\hat{}Tl(x)\rangle$
2. define a function g by simple recursion: $g = recs(f, \langle 0, a\rangle)$
3. define the required primitive recursive function as

$$precs(h, a) = \lambda n \in \omega \,.\, Tl(g\hat{}n)$$

The function $precs(h, a)$ is such that:

$$precs(h, a)\hat{}0 = a$$

$$precs(h, a)\hat{}succ(n) = h\hat{}Hd(g\hat{}n)\hat{}Tl(g\hat{}n)$$

in which $Tl(g`n)$ may be seen, from the definition of *precs*, to be equal to $precs(h, a)`n$, and $Hd(g`n)$ may be shown by induction to be equal to n.

The function constructed by well-founded recursion could be expressed by a simpler term: as in the case of simple recursion, the function *wrec* need not depend on the sets X and Y, since X is simply the domain of f and the range of the union of the range of f may be substituted for Y. Such a simplification is desirable, if only to make the expressions containing recursive functions less cumbersome.

It is worth noting that ordinal recursion may be seen as a special case of well-founded recursion: all that is required, is to prove that the membership relation $mem(\alpha)$ defined on any ordinal α is a well-founded relation. f is a function which must accept as an argument the restriction of the recursive function at the current ordinal, but need not be a function of the ordinal itself in order to obtain the full generality of ordinal recursion. The form of ordinal recursion which may be obtained from the well-founded recursion theorem is as follows:

$$\frac{ordinal(\alpha) \qquad f \in Pow(\alpha * Y) \rightarrow Y \qquad x \in \alpha}{orec(\alpha, f)`x = f`Restrict(orec(\alpha, f), mem(\alpha), x)}$$

where the recursive function is (using a simplified form of *wrec*):

$$orec(\alpha, f) = wrec(mem(\alpha), lam(\alpha, \%(x)f)$$

$lam(\alpha, \%(x)f)$ is the function which returns f for every element of α.

In each of the recursion theorems, the hypothesis specifying the type of f could be omitted, since application, domain and range are defined for any set, not just functions (one of the formulations of ordinal recursion in [17] is given in this form). With such a formulation, however, the typing of the corresponding recursive functions could not be specified by the existing simple typing theorems.

Notice, finally, that in both, simple recursion and well-founded recursion, the set of theorems includes first a specification of the type of the recursive function, then the basic properties of the function. It should be straightforward to prove by induction that the function satisfying these basic properties is unique.

2.2.3 Example of the use of recursion: lists

This section provides only an outline of some of the work that will be required to define and use lists. No proof has actually been performed.

Lists may be represented by sequences, the type of which may be defined in the following way:

$$Seq(type) = \bigcup(\{n \rightarrow type \mid n \in \omega\})$$

The abstract data type for lists is then obtained by defining the constructors *hd*, *tl*, and *cons*.

Alternatively, a list of a given type, *type*, may be represented by ordered pairs, the first element of the pairs being of type *type*, the second element being either a

list of type *type* or a null element. We may define the null element by $Nil = type$, and thus ensure, by the foundation axiom, that it is not an element of *type*. The type of such a list may be defined recursively: $\{Nil\}$ is the type of lists of length 0; if x is the type of lists of length n, then $type * x$ is the type of lists of length $n+1$. Unfortunately, simple recursion is not adequate for the task: it is not possible to define a set-theoretic function to represent $\lambda x \,.\, type * x$, since such a function must have a defined set as a domain, and no such set has yet been proved to exist — one such set is the set we are attempting to define. What is required is the generalised theorem for simple recursion (see [8] page 143), which states that if a formula $F(x,y)$ is such that y has a unique value for every x, then there exists a function f such that $f\hat{}0$ is any set a and $F(f\hat{}n, f\hat{}succ(n))$ holds for each n in ω. The formula $y = type * x$ satisfies the appropriate uniqueness condition. Thus there is a recursive function $f(type)$ such that $f(type)\hat{}0 = \{Nil\}$ and $f(type)\hat{}succ(n) = type * f(type)\hat{}n$. Using the more meaningful $type^n$ in place of $f(type)\hat{}n$, the type of list becomes:

$$List(type) = \bigcup(\{type^n \mid n \in \omega\})$$

A similar definition may be obtained for binary trees:

$$Tree(type) = \bigcup(\{ft(type)\hat{}n \mid n \in \omega\})$$

in which ft is specified by

$$ft(type)\hat{}0 = type$$

$$ft(type)\hat{}succ(n) = type * ft(type)\hat{}n \cup type * (ft(type)\hat{}n * ft(type)\hat{}n)$$

In order to process lists by well-founded recursion, a well-founded relation must be found. The relation of immediate sublist may be used for this purpose:

$$sublist(type) = \bigcup(\{\{\langle x, \langle y, x\rangle\rangle \mid y \in type\} \mid x \in List(type)\})$$

Insertion sort may be used to illustrate the use of well-founded recursion. The insertion sort must satisfy the following properties:

$$\begin{aligned} isort(type,r)\hat{}Nil &= Nil \\ isort(type,r)\hat{}\langle y,z\rangle &= ins(type,r)\hat{}y\hat{}(isort(type,r)\hat{}z) \end{aligned}$$

$$\begin{aligned} ins(type,r)\hat{}x\hat{}Nil &= \langle x, Nil\rangle \\ ins(type,r)\hat{}x\hat{}\langle y,z\rangle &= When(\langle x,y\rangle, r, \langle x,\langle y,z\rangle\rangle, \langle y, ins(type,r)\hat{}x\hat{}z\rangle) \end{aligned}$$

In these equations, r is the ordering relation. The logical function $When$, defined in section 2.4, is such that:

$$\frac{a \in S}{When(a,S,A,B) = A} \qquad \frac{\neg(a \in S)}{When(a,S,A,B) = B}$$

The functions *ins* and *isort* which satisfy these properties may be defined by well-founded recursion. The basic function for *ins* is:

$$\begin{aligned} f\_ins(type, r, x) \;=\; & lam(List(type), \%(u)lam(Function(List(type), List(type)), \\ & \%(v)When(u, \{Nil\}, \langle x, Nil\rangle, \\ & When(\langle x, Hd(u)\rangle, r, \langle x, u\rangle, \langle Hd(u), v\hat{}Tl(u)\rangle))))) \end{aligned}$$

Using the simplified form of *wrec*, the function *ins* is simply the function:

$$ins(type, r) = lam(type, \%(x)wrec(sublist(type), f\_ins(type, r, x)))$$

Finally, *isort* is defined by:

$$\begin{aligned} f\_isort(type, r) \;=\; & lam(List(type), \%(u)lam(Function(List(type), List(type)), \\ & \%(v)When(u, \{Nil\}, Nil, ins(type, r)\hat{}Hd(u)\hat{}(v\hat{}Tl(u)))) \end{aligned}$$

$$isort(type, r) \;=\; wrec(sublist(type), f\_isort(type, r))$$

2.3 Simple type theory

The name of simple type theory seems to have been given to many different formal systems. Some of them are simplified forms of Russell's type theory (see [18] or [9]). In these theories there is a unique basic type i of individuals. All the other types are types of relations or functions of various orders built on this basic type. In some other type theories (such as the ones refered to in [2]), there are two basic types: the type i of individuals and the type o of propositions. In this form of the theory, functions and predicates may be defined over propositions. In particular, the connectives may be defined as predicates.

It has been shown that ZF set theory is more expressive that simple type theory. An argument concerning the first kind of type theory ([9]) runs as follows. Simple type theory may be formalised in monadic form without functions. The basic set of any model of simple type theory is an infinite countable set; if T_0 is such a set, every term of the monadic form of the theory is represented by an element of $Pow^n(T_0)$ for some natural number n. However, many more terms, with different interpretations, are available in set theory: for instance, taking ω as a representation of T_0, $\bigcup(Pow^n(\omega) \mid n \in \omega)$ is a term of set theory, and so are the powers of its power set, and the union of all the previously defined terms ... By using another set to represent the type of propositions, the argument may be extended to include both kinds of type theory. Thus simple type theory may be represented within ZF set theory.

In the first kind of type theory, it is possible to translate directly the terms and formulae of the type theory into ZF set theory. To every term of type theory corresponds a relation or a function of some order constructed on ω (or any other infinite countable set) in set theory. The translation of formulae (see [9] for more

details) consists of

- rewriting the atomic formulae $P(x_1, x_2, ...x_n)$ into monadic form $Q(\langle x_1, x_2, ...x_n\rangle)$
- replacing the atomic formulae in monadic form $Q(x)$ by $x : Q$
- making explicit the typing requirements:
  $\forall x : A \,.\, E(x)$ becomes $\forall x \,.\, x : A \longrightarrow E(x)$
  and
  $\exists x : A \,.\, E(x)$ becomes $\exists x \,.\, x : A \;\&\; E(x)$.

In the second kind of type theory, another type is used: the type of propositions. The main problem in converting such a theory to set theory is that some expressions of type theory are both terms and formulae. The syntactic rules of set theory forbid this. However, it is possible to model type theory *within* set theory by representing both terms and formulae of type theory by terms of set theory. The set ω may be used to model the individuals. As may be verified on page 20, where Andrews' formulation (theory $\mathcal{Q}_0$ in [2], pages 163-164) of type theory has been translated into set theory, the translation requires only a two-value set to represent the type of propositions. This set, T, may be understood as a set of truth-values. In the axioms of $\mathcal{Q}_0$, listed below, the types are specified by the subscripts:

1. $g_{o\to o}\mathbf{t} \;\&\; g_{o\to o}\mathbf{f} = \forall x_o \,.\, g_{o\to o}x_o$
2. $x_\alpha = y_\alpha \longrightarrow h_{\alpha\to o}x_\alpha = h_{\alpha\to o}y_\alpha$
3. $(f_{\alpha\to\beta} = g_{\alpha\to\beta}) = (\forall x_\beta \,.\, f_{\alpha\to\beta}x_\beta = g_{\alpha\to\beta}x_\beta)$
4. $(\lambda x_\alpha . f_\beta(x_\alpha))y_\alpha = f_\beta(y_\alpha)$
5. $\iota_{(i\to o)\to i}(\lambda x_i . y_i = x_i) = y_i$

Andrews' approach is interesting because of its minimalist aspect: the connectives and the truth values are defined simply using the λ symbol and a set of symbols for typed equality. The approach makes clear how type theory may be seen as an extension of a typed λ calculus with equality. The fourth axiom is the β rule of λ calculus. Although Andrews uses a set of five primitive axioms in place of this axiom, he points out that the two formulations are equivalent. Apart from the typed equality, the only other logical symbol of the theory is the typed description symbol ι, the properties of which may be derived from the fifth axiom. The single rule of inference is simply the rule of substitution of equals by equals (through typed equality).

The same definitions as the ones given in [2] are used in the translation into set theory, except for the truth values: in Andrews' formulation, they are defined

Simple type theory

```
 *** Definitions ***

        tr = {0}
       fls = 0
         T = {tr,fls}
     Eq(A) = [ X || X:(A*A)*T, EXISTS x. EXISTS y.
                        X = < <x,x>,tr> | (Not(x=y) & X = < <x,y>,fls> ) ]
       And = lam(T,%(x)lam(T,%(y)Eq((T->T->T)->T)
                 ^ <lam(T->T->T,%(g)g^tr^tr),lam(T->T->T,%(g)g^x^y)>))
     Imply = lam(T,%(x)lam(T,%(y)Eq(T)^ <x,(And^x^y)>))
       Neg = Eq(T)^fls
        Or = lam(T,%(x)lam(T,%(y)Neg^(And^(Neg^x)^(Neg^y))))
    All(A) = lam(A->T,%(P)Eq(A->T)^ <lam(A,%(x)tr),P>)
  Exist(A) = lam(A->T,%(P)Neg^(All(A)^lam(A,%(x)Neg^(P^x))))
   Desc(A) = lam(A->T,%(f)Union([x||x:A,f^x=tr]))

*** Typing rules  (already derived) ***

[| F : A->B |] ==> [| x : A |] ==> [| F^x : B |]
(!(x)([| x : A |] ==> [| P(x) : B |])) ==> [| lam(A, P) : A->B |]

*** Theorems concerning the typed equality ***

[| Eq(S) : (S*S)->T |]
[| a:S |] ==> [| b:S |] ==> [| a=b <-> Eq(S)^ <a,b> = tr|]
[| a:S |] ==> [| b:S |] ==> [| Not(a=b) <-> Eq(S)^ <a,b> = fls|]

*** Theorems recovering Andrews' definition of tr and fls ***

[| tr = Eq((T*T)->T)^ <Eq(T),Eq(T)> |]
[| fls = Eq(T->T)^ <lam(T,%(x)tr),lam(T,%(x)x)> |]

*** Theorems expressing the axioms of simple type theory ***

* Axiom 1 *
[| g : T->T |] ==> [| Eq(T)^ <And^(g^tr)^(g^fls),All(T)^g)> = tr |]
[| g : T->T |] ==> [| And^(g^tr)^(g^fls) = All(T)^g |]
* Axiom 2  *
[| h : A->T |] ==> [| x:A |] ==> [| y:A |]
   ==> [| Imply^(Eq(A)^ <x,y>)^(Eq(T)^ <h^x,h^y>) = tr |]
* Axiom 3  *
[| f : A->B |] ==> [| g : A->B |]
   ==> [| Eq(T)^ <Eq(A->B)^ <f,g>,All(A)^lam(A,%(x) Eq(B)^ <f^x,g^x>)> = tr |]
[| f : A->B |] ==> [| g : A->B |]
   ==> [| Eq(A->B)^ <f,g> = All(A)^lam(A,%(x) Eq(B)^ <f^x,g^x>) |]
* Axiom 4 (beta conversion) *
[| a : A |] ==> [| Eq(T)^ <L(A, P) ^ a,P(a)> = tr |]
[| a : A |] ==> [| L(A, P) ^ a = P(a) |]
* Axiom 5  *
[| x : A |] ==> [| Eq(T)^ <Desc(A)^lam(A,%(y)Eq(A)^ <x,y>),x> = tr |]
[| x : A |] ==> [| Desc(A)^lam(A,%(y)Eq(A)^ <x,y>) = x |]
```

in terms of the basic symbols; in the translation, t and f are the predefined sets tr and fls (ideally different from the other used sets, but for simplicity taken here as $\{0\}$ and 0), and the set of truth values, T, is defined as {tr,fls}. Andrews' definitions for t and f are however recovered as theorems of set theory.

The relation between the typed equality and the equality of set theory is given by the theorems concerning typed equality on page 20. Note that the left implication in the sentence

$$a : S \;\&\; a = b \leftrightarrow Eq(S)\hat{}\langle a, b\rangle = tr$$

is provable if $Not(bot = tr)$ may be proved, in which bot is the value resulting from the application of a function to a term of incorrect type. This is the case here, since $bot = 0$ and $tr = \{0\}$. However, the weaker theorem of page 20 is adequate for the subsequent proofs.

The first version of the axioms 1 to 5 on page 20 differ from the corresponding axioms in [2] only in the fact that the typing of variables is expressed explicitly as hypotheses and that every axiom of the form A in [2] becomes $A = tr$ in set theory. A simplified version of some of the axioms, which uses the set-theoretic equality instead of typed equality, is given. It is through the transformation from typed equality to set-theoretic equality that the inference of Andrews' systems can be carried over to set theory.

Axiom 5 and the definition of the description operator require some explanation. In the definition of page 20, the description operator is the function which, when applied to a truth-valued function f, returns the inverse image of $\{tr\}$ under f, and its type may be proved to be $(A \to T) \to Pow(\bigcup(A))$. An operator of type $(A \to T) \to A$ satisfying axiom 5 could be obtained if there was a function which, when applied to a truth-valued function f, returns an element a of A such that $f\hat{}a = tr$ when such an element exists. But such a function may not be defined without the axiom of choice. The chosen definition is however adequate since it allows the derivation of axiom 5. The axiom itself is more general than the corresponding axiom in Andrews' formulation. If ω is taken as the basic set of type theory, Andrews' axiom is simply the specialisation of axiom 5 with A replaced by ω.

It should be clear that, to every proof in Andrews' formal system, corresponds a proof of set theory derived from the theorems of page 20, and the corresponding definitions.

2.4 Intuitionistic logic with proof objects

In the previous section, a model of simple type theory, in which formulae are interpreted as truth values, was constructed within set theory. In this section a model of a first order intuitionistic logic with quantification over types, in which formulae are interpreted as sets of proofs, is constructed. The construction follows the general idea behind the concept of 'propositions as types'. The connectives are

defined on page 23. The meaning of the definitions may be taken to be as follows:

- The set of proofs of $A\ Or\ B$ is the disjoint union of the set of proofs of A and the set of proofs of B. To a proof a of A corresponds a proof $\langle a, 0\rangle$ of $A\ Or\ B$; to a proof b of B corresponds a proof $\langle b, \{0\}\rangle$ of $A\ Or\ B$.
- The set of proofs of $A\ And\ B$ is the cartesian product of the set of proofs of A and the set of proofs of B.
- The set of proofs of $All(A, P)$ is a dependent function space: the set of total functions f over A such that, if $x \in A$, then $f\hat{}x$ is a proof of $P(x)$.
- The set of proofs of $Exist(A, P)$ is a dependent product: the set of pairs $\langle x, y\rangle$ such that $x \in A$ and y is a proof of $P(x)$.

The set of proofs of the implication $A \to B$ is the set of total functions from A to B, i.e. the term $A \to B$, as already defined. It is a particular case of the definition of *All*: $A \to B = All(A, \%(x)B)$. Note that, similarly, the definition of *And* is a particular instance of the definition of *Exist*: $A\ And\ B = Exist(A, \%(x)B)$.

The introduction and elimination rules, listed on page 23, are theorems of set theory. The rules concerning the implication are simply the typing rules of $\lambda_{\beta\eta}$.

Two examples of theorems, which have been proved using these rules in a backward style, are given below. The variables with a name starting with the symbol '?' are schematic variables which become instantiated through unification during the proofs.

An attempt to prove

$$?x : ((P\ And\ Q)\ Or\ R) \to ((P\ Or\ R)\ And\ (Q\ Or\ R))$$

produces

$$\begin{array}{l} lam(\ (P\ And\ Q)\ Or\ R, \\ \quad \%(ka)\langle When(Hd(ka), \quad P\ And\ Q, lam(P\ And\ Q, \%(kb)\langle Hd(kb), 0\rangle), \\ \qquad\qquad\qquad\quad lam(R, \%(kb)\langle kb, 0\rangle))\hat{}Hd(ka), \\ \qquad\qquad When(Hd(ka), \quad P\ And\ Q, lam(P\ And\ Q, \%(kb)\langle Tl(kb), 0\rangle), \\ \qquad\qquad\qquad\quad lam(R, \%(kb)\langle kb, \{0\}\rangle))\hat{}Hd(ka)\rangle) \\ \quad : ((P\ And\ Q)\ Or\ R) \to ((P\ Or\ R)\ And\ Q\ Or\ R) \end{array}$$

An attempt to prove

$$\frac{a : A}{?x : All(A, Q) \to Exist(A, Q)}$$

produces

$$lam(All(A, Q), \%(ka)\langle a, ka\hat{}a\rangle) : All(A, Q) \to Exist(A, Q)$$

Intuitionistic logic with proof objects

```
*** Definitions ***

     (A Or B) = (A*{0}) Un (B*{{0}})
    (A And B) =  A*B
     All(A,P) = [ F || F : A->Union([P(y)||y:A]), ALL x. x:A --> F^x : P(x) ]
   Exist(A,P) = Union ([ {x} * P(x) || x : A ])
          Fls = 0

When(x,S,A,B) = Union([X || X: {A} Un {B}, (x:S & X=A) | (Not(x:S) & X=B)])

*** Theorems concerning the conditional term ***

     [| x : S |] ==> [| When(x,S,A,B) = A |]
[| Not(x : S) |] ==> [| When(x,S,A,B) = B |]

*** Derived rules ***

And_intr   [| a : A |] ==> [| b : B |] ==> [| <a,b> : A And B |]
And_elim1  [| x : A And B  |] ==> [| Hd(x) : A |]
And_elim2  [| x : A And B  |] ==> [| Tl(x) : B |]
Or_intr1   [| x : A |] ==> [| <x,0> : A Or B |]
Or_intr2   [| x : B |] ==> [| <x,{0}> : A Or B |]
Or_elim    [| x : A Or B |]
           ==> (!(y)[| y : A |]  ==> [| f(y) : Z |])
           ==> (!(y)[| y : B |]  ==> [| g(y) : Z |])
           ==> [| When(Tl(x),{0},lam(A,f),lam(B,g)) ^ Hd(x) : Z |]
Imply_intr (!(x)[| x : A |] ==> [| f(x) : B |]) ==> [| lam(A,f) : A->B |]
Imply_elim [| f : A->B |] ==> [| x : A |] ==> [| f^x : B |]
All_intr   (!(x)[| x:A |] ==> [| f(x):P(x) |]) ==> [| lam(A,f) : All(A,P) |]
All_elim   [| f : All(A,P) |] ==> [| x : A |] ==> [| f^x : P(x) |]
Exist_intr [| x : A |]==> [| y : P(x) |] ==> [| <x,y> : Exist(A,P) |]
Exist_elim [| p : Exist(A,P) |]
           ==> (!(x)!(y)[| x : A |] ==> [| y : P(x) |] ==> [| f(y) : Z |])
           ==> [| f(Tl(p)) : Z   |]
Fls_elim [| x : Fls |] ==> [| y : A |]

*** Sequent style rules ***

And_el [| x : A And B |]
       ==> (!(x)!(y)[| x:A |] ==> [| y:B |] ==> [| f(x,y):Z |])
       ==> [| f(Hd(x),Tl(x)) : Z |]
Imp_el  [| x : A -> B |] ==> [| y : A |]
       ==> (!(x)[| x : B |] ==> [| f(x) : Z |])
       ==> [| f(x^y) : Z |]
All_el  [| x : All(A,P) |] ==> [| y : A |]
       ==> (!(u)!(v)[| u:A |] ==> [| v:P(u) |] ==> [| f(v):Z |])
       ==> [| f(x^y) : Z |]
```

In the second example, the hypothesis ensures that the type A is not empty. If A was empty, it would be possible to prove $All(A,Q)$, but not $Exist(A,Q)$; thus there would be no proof of $All(A,Q) \rightarrow Exist(A,Q)$. Of course, this is a consequence of the use of quantification over types. The untyped quantification of first order intuitionistic logic can easily be modelled by using a countably infinite set such as ω in place of the set A. The theorem

$$?x : All(\omega,Q) \rightarrow Exist(\omega,Q)$$

may be proved without hypothesis.

2.5 Domain theory in $PP\lambda$

This section is concerned with the proof of further properties of functions within set theory. More specifically, it is concerned with continuous functions, and in particular the axioms of $PP\lambda$ relating to domain theory, as specified in [13] or [7].

The relevant definitions are given on page 25. The definition of *bottom* as the union of the set of bottoms elements is justified by the theorem which states that the bottom element of a partial order is unique if it exists.

The axioms and rules of inference of $PP\lambda$ relating to domain theory are listed on page 26 in a form in which functions are explicitly typed as continuous functions, and relations as cpos (in $PP\lambda$, the untyped symbol $\ll$ is used to express the implied ordering relation, whatever the underlying set) . The theorems concerning the least fixed points may be compared to Tarski's fixed point theorem: here, the type of the relation is more general (a cpo rather than a complete lattice), and the type of the function is more specific (continuous rather than monotone).

The remainder of $PP\lambda$ consists of the axiomatisation of first order logic with equality and the β and η conversion rules. Note that $PP\lambda$ also includes a formalism which induces an ordering relation on products and disjoint unions. The construction of the corresponding cpos has not been developed here, but is expected to be simpler than in the case of function spaces.

2.6 Reasoning about types

The development of theories such as simple type theory and PPλ within set theory shows that set theory is suitable to reason about functions. Despite the restriction imposed by the foundation axiom, set theory seems also to be well suited to reason about types. Given some basic types, such as ω or T, one may construct the standard types such as cartesian product, disjoint union, dependent product and dependent function space (as in section 2.4), etc. In fact every set denoted by the terms of set theory may be considered as representing a type. Thus, types need not be disjoint. Subtypes are easily defined: for instance the type of continuous functions over the cpo RA, defined over the set A, may be seen as a subtype of $A \rightarrow A$, itself a subtype of the set of partial functions over A, $Function(A,A)$.

Definitions concerning $PP\lambda$

```
*** Definition of a cpo ***

 Bottoms(R) = [ b || b : Domain(R), ALL y. y : Range(R) --> <b,y> : R]

  bottom(R) = Union(Bottoms(R))

directed(R) = [ Z || Z : Pow(Domain(R)), Not(Z=0) &
                     ALL x. ALL y. x:Z & y:Z
                                 --> EXISTS z. z:Z & <x,z> : R & <y,z> : R]

     cpo(X) = [ Z || Z : Partial_order(X), Not(Bottoms(Z) = 0) &
                     ALL Y. Y : directed(Z) --> Not(Lubs(Z,Y)=0)]

*** cpo of natural numbers  (taking Omega as the bottom element) ***

        Nat = [ <Omega,n> || n:Omega ] Un [ <n,n> || n:succ(Omega) ]

*** Function space and induced order ***

Continuous(RA,RB) = [ K || K : Domain(RA)->Domain(RB),
                           RA : cpo(A) & RB : cpo(B) &
                           ALL Y. Y : directed(RA) --> Not(Lubs(RA,Y)=0) &
                                                       Not(Lubs(RB,[K^x||x:Y])=0) &
                                                       Lub(RB,[K^x||x:Y]) = K^Lub(RA,Y)]

      Func(RA,RB) = [ X || X : Continuous(RA,RB)*Continuous(RA,RB),
                           ALL f. ALL g. ALL x. X = <f,g>  & x:Domain(RA)
                                                  --> <f^x,g^x> : RB]


*** Definitions concerning the fixed point induction ***

          succ_rel = [ <n,succ(n)> || n:Omega ]

  Infinite_chain(R) = [ [f^n||n:Omega] || f : Monotone(succ_rel,R)]

Chain_complete(X,R) = [ Z || Z : Pow(X),
                             ALL Y. Y <= Z & Y : Infinite_chain(R)
                                    --> Lub(R,Y) : Z]
```

Domain theory in $PP\lambda$

```
*** Continuous functions ***

 [| F : Continuous(RA,RB) |] ==> [| F : Monotone(RA,RB) |]
 [| F : Continuous(RA,RB) |] ==> [|  X : directed(RA)  |]
    ==> [|  Lub(RB,Image(F,X)) = F^Lub(RA,X) |]

*** Uniqueness of bottom element ***

 [| R : Partial_order(X) |] ==> [| a : Bottoms(R) |] ==> [| b : Bottoms(R) |]
==> [| a=b |]

*** Constructions of some cpos **

cpo of natural numbers
 [| Nat : cpo(succ(Omega)) |]
 [| bottom(Nat) = Omega |]

cpo induced on a function space
 [|RA:cpo(A)|] ==> [|RB:cpo(B)|] ==> [| Func(RA,RB) : cpo(Continuous(RA,RB))|]
 [|RA:cpo(A)|] ==> [|RB:cpo(B)|] ==> [| bottom(Func(RA,RB)) = L(A,%(x)bottom(RB))|]

*** Domain theory ***

Extensionality
 [| f : Continuous(RA,RB) |] ==> [|  g : Continuous(RA,RB) |]
    ==> [|  RA : cpo(A)|] ==> [|  RB : cpo(B) |]
    ==> [|  ALL x. x : A --> <f^x,g^x> : RB |]
    ==> [|  <f,g> : Func(RA,RB) |]

Monotonicity
 [|  <f,g> : Func(RA,RB) |] ==> [|  <x,y> : RA |] ==> [|  <f^x,g^y> : RB |]

Minimality of bottom element
 [| R : cpo(X) |] ==> [| x : X |] ==> [| <bottom(R),x> : R |]

Least fixed point in cpo
 [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]
    ==> [|  Lub(RA,[recs(f,bottom(RA))^n || n:Omega]) : Lfixs(f,RA) |]
 [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]
    ==> [|  Lfix(f,RA) = Lub(RA,[recs(f,bottom(RA))^n || n:Omega])|]

Properties of least fixed points in cpos
 [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]
    ==> [|  f^Lfix(f,RA) = Lfix(f,RA)|]
 [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]  ==> [| x : Fix(f) |]
    ==> [|  <Lfix(f,RA),x> : RA |]

Fixed point induction
 [| RA : cpo(A) |] ==> [|  [ x || x:A, P(x)] : Chain_complete(A,RA) |]
    ==> [|  f : Continuous(RA,RA) |] ==> [|  P(bottom(RA)) |]
    ==> [|  ALL x. x:A --> P(x) --> P(f^x) |]
    ==> [|  P(Lfix(f,RA)) |]
```

Recursive types may be constructed in the same way that the type of lists was constructed in section 2.3. Taking as an example the simple case in which there is a unique type constructor, the total function symbol, and the chosen form of recursion is simple recursion, one may define the function f such that:

$$f\hat{}0 = A$$

$$f\hat{}succ(n) = f\hat{}n \cup \{Hd(x) \rightarrow Tl(x) \mid x \in f\hat{}n * f\hat{}n\}$$

where A is a set of basic type, e.g. {Omega,T}. This function defines a hierarchy of types. At any given level of the hierarchy, every new type has the form $A \rightarrow B$, in which A and B are types of the previous level. The set of all the types constructed in this way is $type_0 = \bigcup(\{f\hat{}n \mid n \in \omega\})$.

The more complicated case in which the type constructor is the dependent function space may be treated similarly. The recursive function f should now be specified by:

$$f\hat{}0 = A$$

$$f\hat{}succ(n) = f\hat{}n \cup \{\ X \in Pow(Function(\bigcup(f\hat{}n), \bigcup(f\hat{}n)) \mid \exists T\,.\,\exists F\,.\,X = All(T, \%(x)F\hat{}x)\ \&\ T \in f\hat{}n\ \&\ F \in T \rightarrow f\hat{}n\}$$

$All(T, \%(x)F\hat{}x)$ is the dependent function space defined in section 2.4. It is a new type at a given level of the hierarchy iff T is a type of the previous level, and F is a function which associates to every element of T a type of the previous level. As previously, the set of types, $type_0$, is the union of all the sets in the hierarchy.

If the types are polymorphic, i.e. if they are allowed to depend on types and not just on elements of types, the set of types must itself be considered as a type. To this end, following [3], one could specify a second hierarchy, in which the first level is $type_0$ and the level $n+1$ consists of all the types constructed recursively from the types in the level n and the level n itself. The set of all the polymorphic types constructed in this way is $Type = \bigcup(\{type_n \mid n \in \omega\})$.

The method illustrated above may be applied to define recursive types whenever the collection of basic types forms a set and the number of type constructors is finite.

3 Comments concerning the proofs

The main purpose of the research described in this report was to produce a set-theoretic equivalent of several theories. The emphasis has been to obtain a reasonably simple formulation of the theories, without much regard to the form of the proofs themselves. The proofs are generally long and cumbersome, and may certainly be improved greatly. This section gives a brief outline of the approach taken in the development of the proofs, together with the reasons for taking this approach and some comments and criticisms concerning it. The last part of the section describes an attempt at producing natural deduction proofs in forward style.

3.1 Choice of system of deduction

Before developing set theory with natural deduction, I was using the set theory in sequent calculus style already set up in Isabelle. There were two main reasons for switching from sequent calculus to natural deduction. First, the sequent calculus contains an extra formalism to express implication which is not necessary and may be confusing: the object level and meta-level implications are sufficient. The hypotheses of the meta level implications may easily be interpreted as assertions of object-level assumptions, and thus the assumptions of natural deduction fit naturally in the meta-level logic. Second, more general rules may be used with natural deduction in Isabelle. For instance, the tactic 'eresolve_tac [eq_elim] n', where 'eq_elim' is

$$\frac{a = b \qquad P(b)}{P(a)}$$

applied to the n^{th} goal, of the form

$$\frac{c = d \qquad \ldots}{Q}$$

restricts the possible unifiers of $P(a)$ and Q to the ones allowing a unification of $a = b$ and $c = d$. In effect, this ensures that Q is interpreted as a function of c, $q(c)$, during its unification with $P(a)$.

The rules of sequent calculus are well suited to backward proofs because they consist only of introduction rules. However, derived rules may be obtained in the natural deduction system which simulate the sequent rules, thus ensuring that natural deduction is also suitable for backward proofs.

3.2 Level of automation

Most proofs have been carried out backwards because such a style allows some form of automation through the use of tactics. An attempt was initially made to produce complete tactics, i.e. tactics which prove any theorem, for first order logic. These were 'dumb' procedures (breadth first and depth first with limited depth). Although most proofs took only a few seconds cpu on the Sun 3/75, two out of the fourty four first problems of Pelletier ([15]) had to be abandoned after an elapsed time of more than one hour. Among the others, the longest problem took 200 secs cpu. It was felt that much work would be required to produce a reasonably efficient and complete automatic tactic for set theory. One problem is to provide an ordering of the introduction and elimination rules (or right and left rules in sequent calculus) which results in an efficient proof. This is the general problem of finding an appropriate level to which definitions must be folded or unfolded to allow the resolution of goals by assumptions. A more tricky problem is due to the occurence of non-variable terms in set theory: the proof of an existentially quantified sentence often requires the construction of a term for which the sentence is provable; it seems very difficult to generate such a term

automatically. The problems are made more complex by the definition of the new symbols which are required in order to produce a simple formulation of theories within set theory. For the above reasons, the only form of automation which has been kept for the proofs in set theory is the one using the rules of classical logic without equality.

3.3 Tactics

The tactic which attempts to solve a goal automatically using the rules of classical logic is 'REPEAT(step_tac n)'. The tactic is applied to n^{th} goal. If the goal is solved, goal $n+1$ becomes goal n, and an attempt is made to solve this goal. The tactic usually results in the instantiation of schematic variables. These instantiations depend on the order in which the goals are solved, and the assumptions selected to solve a particular goal. Thus the tactic is not always applicable.

When appropriate the derived rules are expressed in the form

$$\frac{h_1 \quad h_2 \quad \ldots}{term_1 = term_2}$$

where the h_i's are hypothesis, which may or may not be present, and the $term_i$'s are terms. A simple ML function, *bimp*, has been written to convert this rule into

$$\frac{h_1 \quad h_2 \quad \ldots}{x \in term_1 \leftrightarrow x \in term_2}$$

The point of this conversion is that tactics have been developed which use rules of the form

$$\frac{h_1 \quad h_2 \quad \ldots}{form_1 \leftrightarrow form_2}$$

to replace the formula $form_1$ by the formula $form_2$ in the conclusion of the selected goal, or the hypotheses, or both. The tactics are respectively, 'unfold_right [rules] n', 'unfold_left [rules] n' and 'unfold_all [rules] n'. It is also possible to unfold only the first relevant formula in the hypotheses. The tactic 'rewrite_right_1 [rule] n' uses the rule 'eq_elim' to perform a one-step rewriting of the conclusion of the goal n from a conditional equality. Unfortunately, no practical version of the tactic has been devised to rewrite the hypotheses. Although some form of recursive rewriting would be an advantage, such a facility has not been found necessary for the simple-minded way in which the proofs have been developed.

By expressing the derived rules, whenever possible, in the form of conditional equalities or conditional bi-implications, it is possible to keep the number of rules to a minimum. The ML functions *intr* and *elim* have been written to convert such rules to a more standard format. Below are some examples of the use of these and other functions on the theorem $A = B$ with ML name *th*:

$$\begin{aligned} &bimp\ th: && x \in A \leftrightarrow x \in B \\ &intr(bimp\ th): && \frac{x \in A}{x \in B} \end{aligned}$$

$$elim(bimp\ th): \quad \frac{x \in B \qquad [x \in A]\ \ Z}{Z}$$

$$symmeq\ th: \quad B = A$$

$$symm(bimp\ th): \quad x \in B \leftrightarrow x \in A$$

3.4 Examples of proof

An example of backward proof is given on page 31. The required theorem and the specified tactics appear after the symbol '>'. Isabelle response to the specification of a tactic is a list of new goals. The proof is initiated by specifying the theorem to be proved: here, the theorem is

$$\frac{F \in A \to B \qquad x \in A}{F`x \in B}$$

The hypotheses in the first goal are interpreted by Isabelle as meta-level assumptions (this may be a bit confusing, since meta-level hypotheses are normally used to represent object level assumptions). The first step consists of a tactic which discharges these assumptions. The resulting meta-level implication may be interpreted as the formula $F`x \in B$ with object level assumptions $F \in A \to B$ and $x \in A$. The second step unfolds the 'total function' and 'Collect' symbols, according to the definitions of pages 10 and 6. The third step performs a classical deduction on the current goal. In the next step, 'Apply' is the following theorem:

$$\frac{F \in Function(A,B) \qquad \langle x,y \rangle \in F}{F`x = y}$$

Isabelle has to lift the rule over the meta-level quantifier (see [12] for a description of the lifting process) in order to perform the required unification. In particular, it replaces the variables A, B, and y respectively by $?A1(ka)$, $?B1(ka)$ and $?b1(ka)$. Rewriting $F`x$ with this conditional equality produces 3 subgoals. Two of them can be solved immediately by assumptions. The process consisting of unfolding followed by a classical deduction continues until the remaining subgoal is solved. The theorem 'prod_iff1', which has not been mentioned previously, is

$$\langle a,b \rangle \in A * B \leftrightarrow a \in A\ \&\ b \in B$$

Although this proof is simple, one of the problems my proofs suffer may be highlighted here. The goal

$$\frac{x \in A \qquad F \in Function(A,B) \qquad \langle x, ka \rangle \in F}{ka \in B}$$

which appears somewhere near the middle of the proof may itself be a useful theorem. It should have been proved first, and then used in the main proof. The reason for not doing so was to minimise the number of theorems. However, this

Example of proof(1): application type

```
> val asm = goal ND_function_set_thy
  "[|  F : A->B |]  ==> [| x : A |]  ==> [| F^x : B |]";

> by (discharge_tac asm 1);
 1. [| F : A -> B |] ==> [| x : A |] ==> [| F ^ x : B |]

> by (unfold_left [bimp Totalfunc,Collect] 1);
 1. [| x : A |] ==>
    [| F : Function(A, B) & ALL x. x : A --> EXISTS y. <x, y> : F |] ==>
    [| F ^ x : B |]

> by (REPEAT(step_tac 1));
 1. [| x : A |] ==>
    [| F : Function(A, B) |] ==> !(ka)[| <x, ka> : F |] ==> [| F ^ x : B |]

> by (rewrite_right_1 [Apply] 1);
 1. [| x : A |] ==> [| F : Function(A, B) |] ==>
    !(ka)[| <x, ka> : F |] ==> [| F : Function(?A1(ka), ?B1(ka)) |]
 2. [| x : A |] ==> [| F : Function(A, B) |] ==>
    !(ka)[| <x, ka> : F |] ==> [| <x, ?b1(ka)> : F |]
 3. [| x : A |] ==> [| F : Function(A, B) |] ==>
    !(ka)[| <x, ka> : F |] ==> [| ?b1(ka) : B |]

> by (REPEAT(assume_tac 1));
 1. [| x : A |] ==>
    [| F : Function(A, B) |] ==> !(ka)[| <x, ka> : F |] ==> [| ka : B |]

> by (unfold_left [bimp Function,Collect] 1);
 1. [| x : A |] ==>
    !(ka)[| <x, ka> : F |] ==>
         [| F : Pow(A * B) &
            ALL x. ALL y. ALL z. <x, y> : F & <x, z> : F --> y = z |] ==>
         [| ka : B |]

> by (REPEAT(step_tac 1));
 1. [| x : A |] ==> !(ka)[| <x, ka> : F |] ==>
                    [| F : Pow(A * B) |] ==> [| ka = ka |] ==> [| ka : B |]

> by (unfold_left [Pow,subset] 1);
 1. [| x : A |] ==> !(ka)[| <x, ka> : F |] ==>  [| ka = ka |] ==>
                    [| ALL x. x : F --> x : A * B |] ==> [| ka : B |]

> by (REPEAT(step_tac 1));
 1. [| x : A |] ==> !(ka)[| <x, ka> : F |] ==>
                    [| ka = ka |] ==> [| <x, ka> : A * B |] ==> [| ka : B |]

> by (unfold_left [prod_iff1] 1);
 1. [| x : A |] ==> !(ka)[| <x, ka> : F |] ==>
                    [| ka = ka |] ==> [| x : A & ka : B |] ==> [| ka : B |]

> by (REPEAT(step_tac 1));
(proof complete)
```

Example of proof(2): decomposition of an ordered pair.

```
> val asm = goal ND_set_thy
    "[| x : A * B |] ==> [| x = <Hd(x),Tl(x)> |]";

> by (discharge_tac asm 1);
 1. [| x : A * B |] ==> [| x = <Hd(x), Tl(x)> |]

> by (unfold_left [bimp Product,Collect] 1);
 1. [| x : Pow(Pow(A Un B)) &
        EXISTS a. EXISTS b. a : A & b : B & x = <a, b> |] ==>
    [| x = <Hd(x), Tl(x)> |]

> by (REPEAT(step_tac 1));
 1. [| x : Pow(Pow(A Un B)) |] ==>
    !(ka,kb)[| ka : A |] ==>
            [| kb : B |] ==> [| x = <ka, kb> |] ==> [| x = <Hd(x), Tl(x)> |]

> by (eresolve_tac [eq_elim] 1);
 1. [| x : Pow(Pow(A Un B)) |] ==>
    !(ka,kb)[| ka : A |] ==>
            [| kb : B |] ==> [| <ka, kb> = <Hd(<ka, kb>), Tl(<ka, kb>)> |]

> by (rewrite_right_1 [Pair_eq3] 1);
 1. [| x : Pow(Pow(A Un B)) |] ==>
    !(ka,kb)[| ka : A |] ==>
            [| kb : B |] ==> [| <ka, kb> = <ka, Tl(<ka, kb>)> |]

> by (rewrite_right_1 [Pair_eq4] 1);
 1. [| x : Pow(Pow(A Un B)) |] ==>
    !(ka,kb)[| ka : A |] ==> [| kb : B |] ==> [| <ka, kb> = <ka, kb> |]

> by (resolve_tac [refl] 1);

(proof complete)
```

advantage is more than offset by the disadvantages of lacking basic theorems. To prove the above basic theorem in the middle of a large proof may require many more steps than is required here: in particular, a selective unfolding on the left may unfold the wrong formula, and a non selective one may perform unwanted unfolding; also, the automatic classical deductions used here, may produce unwanted results when performed within a large proof. It is worth noting that breaking down a proof in some smaller constituents is much more easy in forward style than in backward style since the result of every step of a forward proof is a theorem.

The example on page 32 illustrates the use of the standard resolution tactics 'resolve_tac [rules] n' and 'eresolve_tac [rules] n', and in particular the tactic performing a substitution from an equality in the hypothesis, 'eresolve_tac [eq_elim] n'. The theorems appearing in the proof, and not mentioned before are:

$$\begin{array}{ll} Pair\_eq3: & Hd(\langle x,y\rangle) = x \\ Pair\_eq4: & Tl(\langle x,y\rangle) = y \\ refl: & x = x \end{array}$$

where x and y are free variables.

The two previous examples of proof are very simple. Many of the theorems mentioned in part 2 have much more complex proofs. The well founded recursion theorem was the hardest to prove. It involves forty lemmas, some of them having proofs of more than a hundred steps. The size of the proofs is in part a consequence of my style, and there are many ways of reducing it. Obviously, for any given theorem, some proofs are better than others. Some of the other factors which affect the length of a proof are listed below:

- Choice of definition: several examples of alternative definitions have been suggested in the last section of the paper. For instance, the definition of natural numbers as particular ordinals simplifies some proofs, as well as making them more general.

- An appropriate partition of the proofs into lemmas and theorems: it seems worthwhile to produce as many lemmas and theorems as possible.

- The use of appropriate tactics, in which the relevants rules are stored in some appropriate sequence . One such tactic has been used to evaluate the type of expressions in the set-theoretic formulation of simple type theory.

3.5 Forward proofs

Although most of the proofs have been written in backward style, some experiments have been carried out in forward style. The main disadvantage of the forward approach is the lack of automation: the process involved is proof checking rather than theorem proving. On the other hand, forward proofs present several advantages:

- As previously mentioned, every step produces a theorem, any one of which is easily available as a lemma if required.

- The proofs which are not trivial must be planned before being carried out. Many proofs are derived from an informal description. Such a description is normally provided in a forward style.

- The proofs are processed more efficiently: in backward style, all the subgoals have to be carried over through the proof, even though only one subgoal is processed at a time; this is obviously not the case in forward style.

As an example of forward style, a natural deduction proof of the main part of Tarski's theorem is shown on page 36. The object level assumptions could be adequately represented by meta-level hypotheses. However, this formalism leads to complex procedures to eliminate duplicate assumptions and select the appropriate assumptions to be discharged. Thus, for practical reasons, the object level assumptions have been represented by meta-level assumptions. On page 36, the assumptions are listed in square brackets after each displayed theorem. The proof uses the assumptions and previously proved lemmas listed on page 35. It consists of a sequence of ML statements specifying the required introduction and elimination rules, in the form of ML functions, together with the appropriate theorems. Here are the rules which are used in the proof of Tarski's theorem:

$\rightarrow E$: `fun th1 ' th2 = mp MRES th1 MRES th2`

$\rightarrow I$: `fun imp_i p th = imp_intr MRES (implies_intr (rprop p) th)`

$\forall I$: `fun all_i s t th = all_intr MRES (forall_intr (rterm s)) th))`

The ML function MRES, which has been defined using more basic Isabelle functions, performs a meta-level resolution, unifying the first hypothesis from its first argument with the conclusion of its second argument. The resulting unifier is the first one generated by Isabelle. The axioms *mp*, *imp_intr*, and *all_intr* are some of the elimination rules and inroduction rules described in [14]:

mp: $[\![P \longrightarrow Q]\!] \Longrightarrow [\![P]\!] \Longrightarrow [\![Q]\!]$

imp_intr: $([\![P]\!] \Longrightarrow [\![Q]\!]) \Longrightarrow [\![P \longrightarrow Q]\!]$

all_intr: $(\bigwedge y\,.\,[\![P(y)]\!]) \Longrightarrow [\![\forall\,.\,P(x)]\!]$

The ML functions *implies_intr* and *forall_intr* are meta-level introduction rules provided in Isabelle. The expression (*rprop p*) converts the string p to a valid proposition; the expression (*rterm s*) converts the string s to a valid term. If, for instance, *th* is the theorem $[\![Q]\!]$ under the assumptions $[\![P_1]\!]$, $[\![P_2]\!]$, ..., and p is the string "$[\![P_i]\!]$", then (*implies_intr*(*rprop p*)*th*) is the theorem $[\![P_i \longrightarrow Q]\!]$ under the same assumptions with every instance of P_i removed. If *th* is the theorem $[\![P(x)]\!]$ under some asumptions and s is the string "x", then (*forall_intr*(*rterm s*)*th*) is the theorem $\bigwedge x\,.\,[\![P(x)]\!]$ under the same assumptions.

The format *th1*'*th2* is meant to help relate the resulting proofs to corresponding ones in other forward systems, such as Automath, in which $\rightarrow E$ is expressed by a function application.

Example of forward proof: Tarski's theorem.

```
*** Assumptions ***

as_mono: [| F : Monotone(L, L) |]
           [ [| F : Monotone(L, L) |] ]
as_cl:   [| L : Complete_lattice(A) |]
           [ [| L : Complete_lattice(A) |] ]
as_x:    [| x : [ x || x : A, <x, F ^ x> : L ] |]
           [ [| x : [ x || x : A, <x, F ^ x> : L ] |] ]

*** Lemmas ***

Trans:
[| <?x, ?y> : L --> <?y, ?z> : L --> <?x, ?z> : L |]
[ [| L : Complete_lattice(A) |] ]
Anti_sym:
[| <?x, ?y> : L --> <?y, ?x> : L --> ?x = ?y |]
[ [| L : Complete_lattice(A) |] ]
Mono :
[| <?a, ?b> : L --> <F ^ ?a, F ^ ?b> : L |]
[ [| F : Monotone(L, L) |] ]
Lub_ub:
[| ?x : [ x || x:A, ?P(x) ]  --> <?x, Lub(L, [ x || x:A, ?P(x) ])> : L |]
[ [| L : Complete_lattice(A) |] ]
Lub_least:
[| ?x : Ubs(L, [ x || x:A, ?P(x) ]) --> <Lub(L, [ x || x:A, ?P(x) ]), ?x> : L|]
[ [| L : Complete_lattice(A) |] ]
Ub
[| ?x : A --> ?B <= A --> (ALL y. y : ?B --> <y, ?x> : L) --> ?x : Ubs(L, ?B)|]
[ [| L : Complete_lattice(A) |] ]


type_fx:
[| ?x : A --> F ^ ?x : A |]
[ [| F : Monotone(L, L) |], [| L : Complete_lattice(A) |] ]
type_lub:
[| Lub(L, [ u || u : A, ?P(A, u) ]) : A |]
[ [| L : Complete_lattice(A) |] ]
type_flub:
[| F ^ Lub(L, [ u || u : A, ?P(A, u) ]) : A |]
[ [| F : Monotone(L, L) |], [| L : Complete_lattice(A) |] ]
subtype:
[| [ x || x:?A, ?P(x) ] <= ?A |]
prop_collect:
[| ?x : [ x || x:?A, ?P(x) ] --> ?P(?x) |]
def_collect:
[| ?x : ?A --> ?P(?x) --> ?x : [ x || x:?A, ?P(x) ] |]
```

```
*** Proof of main theorem ***

val fix1 =  Mono ' (Lub_ub ' as_x);
   [| <F ^ x, F ^ Lub(L, [ x || x : A, <x, F ^ x> : L ])> : L |]
   [ [| x : [ x || x : A, <x, F ^ x> : L ] |], [| L : Complete_lattice(A) |],
     [| F : Monotone(Lat, L) |] ]

val fix2 = Trans ' (prop_collect ' as_x) ' fix1;
   [| <x, F ^ Lub(L, [ x || x : A, <x, F ^ x> : L ])> : L |]
   [ [| x : [ x || x : A, <x, F ^ x> : L ] |], [| L : Complete_lattice(A) |],
     [| F : Monotone(L, L) |], [| x : [ x || x : A, <x, F ^ x> : L ] |],
     [| L : Complete_lattice(A) |] ]

val fix3 = imp_i "[| x : [ x || x : A, <x, F ^ x> : L ] |]" fix2;
   [| x : [ x || x : A, <x, F ^ x> : L ] -->
      <x, F ^ Lub(L, [ x || x : A, <x, F ^ x> : L ])> : L |]
   [ [| L : Complete_lattice(A) |], [| F : Monotone(L, L) |],
     [| L : Complete_lattice(A) |] ]

val fix4 = all_i "x" fix3;
  [| ALL x. x : [ x || x : A, <x, F ^ x> : L ] -->
            <x, F ^ Lub(L, [ x || x : A, <x, F ^ x> : L ])> : L |]
  [ [| L : Complete_lattice(A) |], [| F : Monotone(L, L) |],
    [| L : Complete_lattice(A) |] ]


val fix5 = Ub ' type_flub ' subtype ' fix4;
   [| F ^ Lub(L, [ u || u : A, <u, F ^ u> : L ]) :
      Ubs(L, [ u || u : A, <u, F ^ u> : L ]) |]
   [ ... ]

val semi_fix = Lub_least ' fix5;
   [| <Lub(L, [ x || x : A, <x, F ^ x> : L ]),
       F ^ Lub(L, [ u || u : A, <u, F ^ u> : L ])> : L |]
   [ ... ]

val fix6 = Lub_ub ' (def_collect ' (type_fx ' type_lub) ' (Mono ' semi_fix));
   [| <F ^ Lub(L, [ u || u : A, <u, F ^ u> : L ]),
       Lub(L, [ x || x : A, <x, F ^ x> : L ])> : L |]
   [ ... ]

val fix = Anti_sym ' fix6 ' semi_fix;
   [| F ^ Lub(L, [ u || u : A, <u, F ^ u> : L ]) =
      Lub(L, [ x || x : A, <x, F ^ x> : L ]) |]
   [ ... ]

*** Code to eliminate the duplicate assumptions ***

val fix7 = (imp_i "[| L : Complete_lattice(A) |]" fix) ' as_cl;
val lfix = (imp_i "[| F : Monotone(L,L) |]" fix7) ' as_mono;
   [| F ^ Lub(L, [ u || u : A, <u, F ^ u> : L ]) =
      Lub(L, [ x || x : A, <x, F ^ x> : L ]) |]
   [ [| F : Monotone(L, L) |], [| L : Complete_lattice(A) |] ]
```

The effect of the above functions may be seen on page 36, where the proof has been broken down into shorter proofs, and the result of each subproof has been displayed. The ellipses stand for multiple assumptions of the form $F : Monotone(L, L)$ or $L : CompleteLattice(A)$, which are the assumptions on which the final theorem depends. The style of the proof is similar to the style obtained in other forward proof systems, such as the calculus of constructions (a proof of Tarski's theorem in the calculus of constructions may be found in [11]).

Note that, although the proofs may contain schematic variables during their development (originating, for instance, from a previous theorem or from a $\forall E$ rule), these variables become normally instantiated in some unification before the end of the proof. The resulting proof is therefore a standard natural deduction proof.

4 Some related work

The concern of Boyer et al in [4] is to show that automatic proofs may be constructed in set theory using first-order resolution. To this aim, they use a finite axiomatisation of set theory — the vonNewmann-Bernay-Gödel axiomatisation — and write it in clausal form. Theorems are proved by refuting their negation, also in clausal form. Although their eventual aim is to to provide an automatic theorem prover, they recognise that automation is presently limited by the problems mentioned in section 3.2 (finding an appropriate level of expansion for the definitions, and dealing with a large number of lemmas and theorems) and that some form of heuristics will have to be used. The sample proof they provide (for the theorem stating that the composition of homomorphisms is a homomorphism) has been mechanically checked, but not generated automatically.

Corella ([5]) has developed ZF set theory within higher order logic. He shows that the resulting theory is a conservative extension of ZF set theory within first order logic. He argues that the higher order axiomatisation provides a means of defining schematic axioms which is not available in a first order formulation. However, a counter-argument has been provided in this paper: the availability of schematic variables in Isabelle has made possible a standard first-order formulation of set theory. Corella has developed a proof checker, 'Watson', which includes the higher order axiomatisation of ZF. As in LCF, the inference rules are defined by functions (or algorithms), and the theorems may not be interpreted as derived inference rules.

5 Conclusion

Several theories concerning functions have been developed within ZF set theory using the theorem prover Isabelle. One advantage of developing such theories within set theory, rather than defining them by sets of axioms and inference rules, is that the approach provides a uniform and consistent system for reasoning about

functions. Furthermore, although this has not been done here, it has been pointed out that functions may be defined from a richer collection of sets than in the case of simple type theory: for instance the ordinals greater than ω cannot be constructed in simple type theory. More functions may also be defined in set theory through the use of ordinal recursion. It has also been shown that set theory is a suitable theory to reason about types, including complex types such as polymorphic dependent function spaces.

Isabelle is well suited to the development of theories within theories: since its theorems are in the form of inference rules, it is as easy to use a theory defined by derived theorems, as it is to use one predefined by axioms and inference rules.

Although the formulation of the theories within set theory has a standard form, the proofs themselves are currently cumbersome. More work is now required in order to convert the existing proofs into shorter and more readable ones, and to increase the level of automation. One of the ways in which this can be done is through the development of appropriate tactics within Isabelle.

Acknowledgements: I am indebted to Larry Paulson for his numerous suggestions concerning both, the theoretical aspect of the research and the use of Isabelle. I would also like to thank Tobias Nipkow, Thomas Foster and Martin Coen for reading the draft of the paper and providing useful comments. The funding of the research was provided by the SERC grant GR/E0355.7.

References

[1] Peter Aczel. *Non-well-founded sets.* CSLI lecture notes 14, 1988.

[2] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[3] A. Borzyszkowski, R. Kubiak, J. Leszczylowski, and S. Sokolowski. *Towards a set-theoretic type theory.* Technical Report, Institute of Computer Science, Polish Academy of Sciences, September 1988.

[4] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos. Set theory in first-order logic: clauses for godel's axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.

[5] Francisco Corella. *Mechanising Set Theory.* Technical Report RC 14706 (*65927), IBM Research Division, 1989.

[6] D. Gabbay and F. Guenthner, editors. *Handbook of Philosophical Logic.* Reidel Publishing Company, 1983.

[7] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer-Verlag, 1979. LNCS 78.

[8] A. G. Hamilton. *Numbers, sets and axioms.* Pergamon Press, 1982.

[9] William S. Hatcher. *The Logical Foundations of Mathematics.* Pergammon Press, 1982.

[10] J. Roger Hindley and Jonathon P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[11] G. P. Huet. Induction principles formalised in the calculus of constructions. In *TAPSOFT 87*, pages 276–286, Springer-Verlag, 1987. LNCS 249.

[12] Lawrence C. Paulson. *The Foundation of a Generic Theorem Prover.* Technical Report 130, University of Cambridge Computer Laboratory, 1988.

[13] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[14] Lawrence C. Paulson. *A preliminary user's manual for Isabelle.* Technical Report 133, University of Cambridge Computer Laboratory, 1988.

[15] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata, JAR 4 (1988), 236–236.

[16] Goran Sundholm. Systems of deduction. In [6] Vol 1 133-188, 1983.

[17] Patrick Suppes. *Axiomatic Set Theory.* Dover, 1972.

[18] G. Takeuti. *Proof Theory.* North Holland, 2nd edition, 1987.