Journal of Automated Reasoning **10**: 15–58, 1993.

Experimenting with Isabelle in ZF Set Theory

P. A. J. NOEL
Department of Computer Science, University of Manchester, Manchester, M13 9PL, England

(Received: 18 January 1990; accepted: 3 July 1991)

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 well-founded recursion theorem has been derived, allowing the definition of functions over recursive types (such as the length and the append functions for lists). The theory of functions has been developed sufficiently within ZF to include PPλ, the theory of continuous functions forming the basis of LCF. Most of the theorems have been derived using backward proofs, with a small amount of automation.

Key words. Set theory, Isabelle, theorem proving.

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 experiment with the theorem prover Isabelle [13, 14], using different kinds of proof mechanisms: secondly to prove, within ZF set theory, a collection of useful theorems concerning functions. In particular, the well-founded recursion theorem has been derived, allowing the definition of functions over recursive types (such as the length and the append functions for lists). The theory of functions has been developed sufficiently within ZF to include PPλ, the theory of continuous functions forming the basis of LCF [7]. The developed theory may be used to define set theoretic semantics for other theories, and to derive the axiomatisation of the theories within ZF. This is illustrated in Appendix C, where an axiomatisation of simple type theory is derived, and in Appendix D, where an axiomatisation of intuitionistic first-order logic is derived from a semantics in which the formulae are interpreted as the set of their proofs. 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 $lam(A, f1)$.

Isabelle is a theorem prover that is well suited for the task of deriving axioms and inference rules, since, in their general form, Isabelle's theorems are inference rules. In particular higher order unification, which is provided in Isabelle, is required when using the schematic axioms and rules of set theory, or any schematic theorem which may be derived in it. ZF set theory has been defined as an extension of a first order theory already set up in Isabelle.

The main part of the paper is divided into four sections: Section 2 contains an overview of Isabelle; Section 3 presents some of the theorems which have been proved, and the definitions used in the process; Section 4 is concerned with some of the issues relating to the proofs themselves; Section 5 mentions some related work.

2. An Overview of Isabelle

Isabelle is a generic theorem prover both designed and written in ML by Larry Paulson [13–15]. Isabelle's pure theory (or meta-theory) is a fragment of higher order logic defined within a simply typed lambda calculus. The basic inference rules of the meta-theory are 'built' into Isabelle, together with some derived ones, and in particular a resolution rule. Other theories may be defined as extensions of the meta-theory, by specifying their signature (i.e. their types and typed constants) and their set of inference rules (including axioms, which are simply inference rules without antecedents). The following sections give a brief overview of Isabelle. Detailed descriptions may be found in the papers referenced above.[1]

2.1. ISABELLE'S META-THEORY

The signature of the meta-theory has one basic type: the type of propositions, *prop*. The main constants in the signature are the implication $\Rightarrow$, of type $prop \rightarrow prop \rightarrow prop$, and, for every type σ, a universal quantifier $\bigwedge_\sigma$ of type $(\sigma \rightarrow prop) \rightarrow prop$. Another notation for $\bigwedge_\sigma P$ is $\bigwedge x_\sigma.\ P(x_\sigma)$. The inference rules of meta-theory, expressed in natural deduction style, are given below.

Implication introduction and elimination rules:

$$\textit{implies\_intr}: \quad \frac{\begin{array}{c}[\Phi] \\ \Psi\end{array}}{\Phi \Rightarrow \Psi} \qquad \textit{implies\_elim}: \quad \frac{\Phi \Rightarrow \Psi \quad \Phi}{\Psi}$$

Quantifier introduction and elimination rules:

$$\textit{forall\_intr}: \quad \frac{\Phi(x)}{\bigwedge_\sigma \Phi} \qquad \textit{forall\_elim}: \quad \frac{\bigwedge_\sigma \Phi}{\Phi(a)}$$

where x is a variable of type σ not free in the assumptions, and a is an expression of type σ. The rule *implies_intr* allows one to use meta-level assumptions when specifying theorems. The quantifier introduction and elimination rules allow theorems to be expressed with or without outermost quantification. From the rule *implies_elim* and the quantifier rules, derived rules may be obtained which describe the principles underlying the various object-level inferences. One of the most useful of these rules is the resolution rule:

$$\frac{\beta_1 \Rightarrow \ldots \Rightarrow \beta_m \Rightarrow \Phi \qquad\qquad \Phi_1 \Rightarrow \ldots \Rightarrow \Phi_i \Rightarrow \ldots \Rightarrow \Phi_n \Rightarrow \Psi}{\Phi_1\theta \Rightarrow \ldots \Rightarrow \Phi_{i-1}\theta \Rightarrow \beta_1\theta \Rightarrow \ldots \Rightarrow \beta_m\theta \Rightarrow \Phi_{i+1}\theta \Rightarrow \ldots \Rightarrow \Phi_n\theta \Rightarrow \Psi\theta}$$

in which the iterated implications associate to the right. θ is some unifier resulting from a higher order unification of the propositions Φ_i and Φ, and $\Gamma\theta$ represents the result of applying the substitution θ to the proposition Γ. Not all of the free variables need to be instantiated during the unification. Variables of a new kind, schematic variables, are used to identify the free variables which are to be instantiated. Schematic variables are represented by symbols, the first character of which is a question mark. Free variables behave as constants in the resolution. Both kinds of variable are normally involved when the resolution rule is used in backward proofs. In this case, the right premise represents the state of the proof of the formula Ψ: it asserts that Ψ is provable if the propositions $\Phi_j (0 \leqslant j \leqslant n)$ are provable. The free variables of Ψ are interpreted as constants during the proof. The left premise is a derived inference rule which is used to replace Φ_i (a 'goal') by new propositions ('subgoals'). In order to perform the appropriate unification of Φ and Φ_i, all the free variables are replaced by schematic variables before the rule is used as a left premise in the resolution rule.

The resolution rule used in the basic tactic for backward proofs is one in which Φ has the simple form $[\![P]\!]$. However, Φ_i may be a complex proposition involving implication and quantification. Isabelle may apply two extra derived rules to the left premise in order to obtain a proposition with the same structure as Φ_i. If Φ_i is the implication $\Theta \Rightarrow \Phi_i'$, a left premise of the appropriate form is obtained by using implication lifting:

$$\frac{\beta_1 \Rightarrow \ldots \Rightarrow \beta_n \Rightarrow \Phi}{(\Theta \Rightarrow \beta_1) \Rightarrow \ldots \Rightarrow (\Theta \Rightarrow \beta_n) \Rightarrow (\Theta \Rightarrow \Phi)}$$

If Φ_i is in the form $\bigwedge u_\sigma . \Phi_i'(u_\sigma)$, a left premise of the appropriate form is obtained by using quantification lifting:

$$\frac{\beta_1 \Rightarrow \ldots \Rightarrow \beta_n \Rightarrow \Phi}{(\bigwedge u_\sigma \cdot \beta_1\theta) \Rightarrow \ldots \Rightarrow (\bigwedge u_\sigma \cdot \beta_n\theta) \Rightarrow (\bigwedge u_\sigma \cdot \Phi\theta)}$$

where θ represents a substitution in which schematic variables are 'lifted' to become functions of u_σ: any expression of the form $?x(v_1, v_2, \ldots, v_n)$ is substituted by $?x'(v_1, v_2, \ldots, v_n, u_\sigma)$. The point of the lifting is that $?x'(v_1, v_2, \ldots, v_n, u_\sigma)$ may unify with expressions involving the bound variable u_σ, while $?x(v_1, v_2, \ldots, v_n)$ may not. When Φ_i is in the form $\wedge u_\sigma \cdot (\Theta(u_\sigma) \Rightarrow \Phi_i'(u_\sigma))$, both kinds of lifting are used.

2.2. PROOFS IN ISABELLE

Resolution is the basic step of both forward proofs and backward proofs. Backward proofs are actually processed in a forward manner: given a proposition Ψ, a proof of it will be a forward proof starting with the theorem $\Psi \Rightarrow \Psi$, and producing Ψ after a repeated use of the resolution rule. At every stage of the proof an intermediary theorem (the current proof state) is derived. As mentioned in Section 2.1, the left premise of the resolution rule may be any previously proved theorem in which the outer quantification has been removed and schematic variables have been used in place of free variables. The right premise is the previous proof state. When the proof of only one instance of $\Psi(x_\sigma)$ is required, the right premise may contain schematic variables. In this case, the initial theorem to be used is $\Psi(?x_\sigma) \Rightarrow \Psi(?x_\sigma)$ in which $?x_\sigma$ is a schematic variable which may be instantiated during the proof.

The resolution rule exhibits clearly how a goal Φ_i produces a set of subgoals, $\beta_1\theta$, $\beta_2\theta$, etc. Isabelle provides a basic tactic, 'resolve_tac', to perform resolution: the tactic 'resolve_tac ths i' attempts to resolve the ith goal of the current proof state with one theorem from the specified sequence 'ths'. If more than one unifier exists, Isabelle chooses one resolvent and stores the others in a lazy list, to be retrieved if backtracking occurs. Tacticals are provided which may be used to define sequences of tactics, alternative tactics, backtracking, etc. The user may also use these predefined tactics and tacticals to specify his/her own ones in ML.

2.3. DEFINITION OF THEORIES

A new theory is created by completing the signature of another theory and adding new axioms to it, or by merging existing theories. Isabelle comes equipped with a few theories defined within the meta-theory. The set theory described in this paper has been built on top of an intuitionistic logic with natural deduction already present in Isabelle and described in Figure 1. The logic includes two basic types: the type of terms, *exp*, and the type of formulae, *form*. A constant of type *form* $\rightarrow$ *prop*, represented by the brackets $[\![$ and $]\!]$, associates to any formula the corresponding meta-level proposition: if f is a formula, $[\![f]\!]$ is the corresponding proposition. The Isabelle notation for the metal level constants and the brackets is as follows:

expression	*Isabelle notation*
$\Rightarrow$	$==>$
$\wedge x_\sigma \cdot \Phi(x_\sigma)$	$!(x_\sigma)\Phi(x_\sigma)$
$[\![\ldots]\!]$	$[\vert \ldots \vert]$

*** Constants ***

Internal notation:

symbol	meta-type	precedence	description
=	$(exp, exp) \rightarrow form$	left 6	equality
&	$(form, form) \rightarrow form$	right 5	conjunction
|	$(form, form) \rightarrow form$	right 4	disjunction
-->	$(form, form) \rightarrow form$	right 3	implication
<->	$(form, form) \rightarrow form$	right 3	bi-implication
False	$form$		false formula
Forall	$(exp \rightarrow form) \rightarrow form$		universal quantifier
Exist	$(exp \rightarrow form) \rightarrow form$		existential quantifier

Alternative notation for input/output:

```
ALL x. P(x)      stands for  Forall P
EXISTS x. P(x)   stands for  Exists P
```

```
*** Axioms ***

*Equality*

refl:          [| a=a |]
sym:           [| a=b |] ==> [| b=a |]
trans:         [| a=b |] ==> [| b=c |] ==> [| a=c |]

*Propositional logic*

conj_intr:     [| P |] ==> [| Q |] ==> [| P&Q |]
conjunct1:     [| P&Q |] ==> [| P |]
conjunct2      [| P&Q |] ==> [| Q |]
disj_intr1:    [| P |] ==> [| P|Q |]
disj_intr2:    [| Q |] ==> [| P|Q |]
disj_elim:     [| P|Q |] ==> ([| P |] ==> [| R |]) ==> ([| Q |] ==> [| R |])
               ==> [| R |]
imp_intr:      ([| P |] ==> [| Q |]) ==> [| P-->Q |]
mp:            [| P-->Q |] ==> [| P |] ==> [| Q |]
False_elim     [| False |] ==> [| P |]
iff_def:       P<->Q == (P-->Q) & (Q-->P)

*Quantifiers*

all_intr:      (!(y) [| P(y) |]) ==> [| ALL x.P(x) |]
spec:          [| ALL x.P(x) |] ==> [| P(a) |]
exists_intr:   [| P(a) |] ==> [| EXISTS x.P(x) |]
exists_elim:   [| EXISTS x.P(x) |] ==> (!(y)[| P(y) |] ==> [| R |]) ==> [| R |]
```

Fig. 1. First-order intuitionistic logic.

where x_σ is a variable of type σ. The axioms of Figure 1 are meta-level formulae defining the inference rules of first-order intuitionistic logic with natural deduction. For instance, the axiom

$$(\llbracket P \rrbracket \Rightarrow \llbracket Q \rrbracket) \Rightarrow \llbracket P \rightarrow Q \rrbracket$$

specifies the implication introduction rule:

$$\frac{\begin{array}{c}[P]\\ Q\end{array}}{P \to Q}$$

The meta-level implication may be used to represent a premise with assumption, as well as an inference. The quantifier introduction rule of the metal-level logic allows the above axiom to be expressed without an outermost meta-level quantification of P and Q. Some axioms, however, require the use of quantification. This is the case, for instance, for the axiom specifying the universal quantifier introduction rule:

$$(\wedge y \cdot [\![P(y)]\!]) \Rightarrow [\![\forall x \cdot P(x)]\!]$$

Here the scope of the quantification of y is the antecedent of the rule. Such an expression may resolve with theorems of the form $[\![Q]\!] \Rightarrow (\wedge y \cdot [\![P(y)]\!])$ but not of the form $[\![Q(y)]\!] \Rightarrow [\![P(y)]\!]$, thus embodying the condition under which the corresponding rule can be applied, namely that the free variable which is to be quantified must not be free in the assumptions. The bi-implication symbol is defined using meta-level equality $==$:

$$P \leftrightarrow Q == (P \to Q) \wedge (Q \to P)$$

Meta-level equality is simply a means of defining new symbols standing for more complex expressions. In the rest of the paper, object-level equality will be used for this purpose.

Note that there is no built-in facility to reason about theories. Should there be any such requirement, the meta-theory itself should be defined as a new logic in Isabelle.

3. Developing ZF Set Theory

The main aim of the work described in this paper was to develop enough of ZF set theory to be able to reason about functions, and in particular recursive functions. Set theory has been first defined on top of the first order intuitionistic logic mentioned earlier. Its axiomatisation consists of some variant of the standard axioms, together with a set of definitions for commonly used concepts such as subset, product etc. The other theories defined in this paper are simply extensions of ZF with non-recursive definitions, and are thus conservative extensions of ZF. Most of these definitions concern terms, rather than predicates: i.e. a deliberate choice has been made to define properties by terms, say P_i, with the corresponding predication to a term x represented by $x \in P_i$, rather than define predicates P_i' with the corresponding properties represented by $\{x \mid P_i'(x)\}$. This approach seems to be more in the spirit of set theory, and is well suited to reason about types.

The axioms and rules of inference which have been chosen for set theory are presented first. Then a set of definitions concerning functions is introduced and some

relevant theorems, including recursion theorems, are discussed. An axiomatisation of typed lambda calculus is included in these theorems.

3.1. ZF SET THEORY

Set theory has been constructed on top of the first-order intuitionistic logic mentioned earlier. 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:

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

  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 elimination' 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*:

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

The theory is then extended to include the new constants displayed in Figure 2 and the ZF axioms, displayed in Figure 3. Three new constants are required in ZF set theory: the membership predicate $\in$, the empty set 0, and a constant to define pairs (the constant ':', defining the insertion of a set into another, and used simply to insert two sets into the empty set). All other constants may be defined in terms of these basic ones. For convenience, however, the axiomatisation given in Figure 3 introduces other basic constants, such as 'Union', 'Pow', . . . (the justification for this approach will be given later in this section). When a defined constant is required by an axiom, its definition appears before the axiom in Figure 3. Other useful definitions are listed in Figure 4. The symbols other than the constants are taken to be universally quantified meta-level variables. In ZF6, for instance, x and A are universally quantified over the type *exp*, and P over $exp \rightarrow form$. From a syntactic point of view, the variables quantified over *exp* may be seen as standing for free variables, and the other quantified variables for schematic variables (i.e. uninstantiated constants) of various orders.

*** internal notation ***

Basic symbol	meta-type	precedence	description
:	$(exp, exp) \rightarrow form$	right 6	membership: $\in$
0	exp		empty set:
::	$(exp, exp) \rightarrow exp$	right 7	inclusion: $a :: b :: 0 = \{a, b\}$
Pow	$exp \rightarrow exp$		power set
Union	$exp \rightarrow exp$		union of a family of sets
Collect	$(exp, exp \rightarrow form) \rightarrow exp$		Collect$(A,P) = \{x \in A \mid P(x)\}$
Repl	$(exp \rightarrow exp \rightarrow form, exp) \rightarrow exp$		Repl$(R,B) = \{x \mid \exists y \,.\, y \in B \wedge R(y, x)\}$
INF	exp		infinite set

Defined symbol	meta-type	precedence	description
<=	$(exp, exp) \rightarrow form$	right 6	subset:$\subseteq$
<<	$(exp, exp) \rightarrow form$	right 6	strict subset: $\subset$
Inter	$exp \rightarrow exp$		intersection of a family of sets
Un	$(exp, exp) \rightarrow exp$	right 7	union of two sets
succ	$exp \rightarrow exp$		successor function
Int	$(exp, exp) \rightarrow exp$	right 8	intersection of two sets
Replace	$(exp \rightarrow exp, exp) \rightarrow exp$		Replace$(f,B) = \{f(x) \mid x \in B\}$
Pair	$(exp, exp) \rightarrow exp$		ordered pair
Hd	$exp \rightarrow exp$		Hd(Pair(a,b)) = a
Tl	$exp \rightarrow exp$		Tl(Pair(a,b)) = b
*	$(exp, exp) \rightarrow exp$	right 8	product of two sets
-	$(exp, exp) \rightarrow exp$	right 7	difference of two sets
Functional	$(exp \rightarrow exp \rightarrow form) \rightarrow form$		predication of a functional relation
Total	$(exp \rightarrow exp \rightarrow form) \rightarrow form$		total functional relation

*** Alternative notation for input/output ***

{a,b}	stands for	a :: b :: 0
{a}	stands for	{a,a}
<a,b>	stands for	Pair(a,b)
[x || x:A, P(x)]	stands for	Collect(A,P)
[f(x) || x:B]	stands for	Replace(f,B)

Fig. 2. ZF constants.

In [17] and [8], ZF1 is axiomatised as $A \subseteq B \wedge B \subseteq A \rightarrow A = B$. 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 Figure 3, and the extension axiom becomes the axiom schema:

$$\forall x \cdot \forall y \cdot x = y \rightarrow (P(x) \leftrightarrow P(y))$$

From these, the properties of equality are derivable.

Usually, the axioms ZF2 and 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, the defining formulae are used as axioms, in place of

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

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) |]

       Functional: [| Functional(R) <->
                      ALL x. ALL y. ALL z. R(x,y) & R(x,z) --> y=z |]

ZF 7 - Replacement: [| Functional(R) |] ==>
                    [| y : Repl(R,B) <-> EXISTS a. a : B & R(a,y) |]

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

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

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

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

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

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

Fig. 3. ZF axiomatisation.

```
Total:         [| Total(R) <-> Functional(R) & ALL x. EXISTS y. R(x,y) |]

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

Replace:       [| Replace(f,A) = Repl(%(x)%(y)(y=f(x)),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>]|]

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

Fig. 4. Simple definitions.

the usual existential ones. This is the same form of axiomatisation as the one for the sequent form of ZF in [13].

There are many ways of formalising 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.

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 has been derived the theorems:

$$\neg(a \in a) \quad \text{and} \quad \neg(a \in b \wedge b \in a)$$

One point to note in the definitions of Figure 4 is the use of the % symbol in place of the abstraction symbol λ. In the definition of *Replace*, `%(x)%(y)(y=f(x))` is an expression of type $exp \rightarrow exp \rightarrow form$, as required by the typing of *Repl*. The following theorem has been derived using the definition of *Replace*:

$$x \in Replace(f, B) \leftrightarrow \exists a \cdot a \in B \wedge x = f(x)$$

Although less general than *Repl*, the constant *Replace* has been used whenever possible because of its simpler expansion.

A number of other basic theorems have been proved, which are required in order to proceed with the development of a theory of relations and functions. Some of them, and their proof in sequent style, may be found in [15].

3.2. RELATIONS AND FUNCTIONS

Set theory has been developed with the particular aim of reasoning about functions. A new theory, consisting of a set of new constants together with a set of axioms defining them, has been built on top of ZF set theory. However, since the axioms are simply definitions of well formed expressions of ZF, the new theory is a conservative extension of ZF. The first part of this section consists of a general discussion about definitions and specific comments concerning some of the definitions which have been introduced to reason about relations and functions. Next, some of the theorems which have been proved are discussed, and in particular several recursion theorems. Finally, the two last subsections mention some possible applications of the work presented in this paper: reasoning about the set theoretic semantics of other theories and reasoning about types. For the sake of clarity, only the definitions and main theorems involved in the proof of the recursion theorems have been displayed in the main part of the

paper (further developments are described in appendices, while simple definitions which have not been required for these developments, such as the ones concerning currying, have been omitted).

3.2.1. *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 new constants must satisfy the following requirements (see [17], Ch. 2):

- a theory T to which a new definition D is added must form a conservative extension of T: i.e. the only sentences not containing the new symbol which are provable in $T \wedge D$ are the ones which are provable in T.
- any new definition must satisfy the eliminability criterion: i.e. every sentence which contains the defined symbol must be equivalent to one without it.

Sometimes, conditional definitions seem appropriate. For instance, the application of a term to another term is defined only if the first term is a function, and the second term is from an appropriate set. An unconditional definition may still be used if a meaning is given to $f\,\hat{}\,a$ when f is not a function or a is not in the domain of f. We then have to ensure that any theorem concerning function application includes the appropriate typing hypotheses. Although a conditional definition, in which the definition *itself* is subject to the typing constraints, would result in simpler theorems, it would not satisfy the criterion of eliminability: 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.

The choice which has been made in this paper was to use unconditional definitions whenever appropriate, most of them of the simple form $new\_symbol(x) = exp(x)$, where x is a set of variables, possibly empty, and $exp(x)$ is a term not involving the new symbol (recursive definitions are thus disallowed) and having the variables in x as the only free variables. The free variables may be higher-order variables, as long as the terms being defined are well-formed terms of set theory. This is the case, for instance, in the definition of lambda abstraction:

$$lam(A, E) = \{\langle x, E(x)\rangle \mid x \in A\}$$

where E is a variable of type $exp \rightarrow$ exp. Unconditional definitions have sometimes a domain of application which is more general than is required. For instance, *Domain*, *Range* and *Image* apply to any set, not just relations; the application $F\,\hat{}\,a$ is also defined for any sets F and a. Care must be taken when using such general definitions: for instance, most theorems concerning an application should be subject to the conditions that F is a function, and a is in the domain of F. When using an unconditional definition, the existence and uniqueness of the defining term are obviously

Some properties of 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]

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]

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

Some attributes of relations:

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

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

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

Composition:
Ra @ Rb           = [ X || X: Domain(Ra)*Range(Rb), EXISTS x. EXIST y.
                           X = <x,y> & EXISTS z. <x,z> : Rb & <z,y> : Ra]

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])

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

Fig. 5. Definitions concerning relations.

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, the set of least fixed points is defined in Figure 15, Appendix A. Defining *the* least fixed point as the union of this set makes sense only if the set is a singleton. Checking conditions of this kind constitutes a significant part of the work involved in proving theorems within set theory.

Despite the general use of unconditional definitions in this paper, there are cases where conditional definitions have been preferred. The function *grecs_s* which is used to define generalised simple recursion in Figure 6, for instance, is defined only when the predicate P in its arguments defines a total function. The reason for choosing a conditional definition is that the existence and the uniqueness of the defining expression may be guaranteed under the stated conditions.

Some properties of functions:

```
Partial 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]
```

Function application and lambda abstraction:

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

Restriction of a function to an initial segment:

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

Definitions of 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)        = 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 generalised simple recursion:

```
Function over n < Omega (conditional definition):
[| Total(P) |] ==> [| grecs_s(P,a,n) = f <-> EXISTS A. f : (n Un {n})->A & f`0 = a
                                              & ALL i. i:n --> P(f`i,f`succ(i))  |]

Total function over Omega:
grecs(P,a)       = lam(Omega,%(n)grecs_s(P,a,n)`n)

Special case where R(x,y) is  y=F(x):
gfrecs(F,a)      = grecs(%(x)%(y)(y=F(x)),a)
```

Function defined by well-founded recursion:

```
Set of functions over restrictions of f in R:
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) ]
Recursive function:
wrec(X,Y,R,f)    = Union(wrec_s(X,Y,R,f))
```

Fig. 6. Definitions concerning functions.

The definitions of Figures 5 and 6 are generally straightforward. The type of the new constants may easily be inferred from their definition. Amongst the constants, there are three infix operators: the composition symbol '@', the total function symbol '--\>' and the function application symbol '`'. Both '@' and '--\>' are right associative while '`' is left associative. All three symbols have precedence 8. The remainder of this section consists of specific comments concerning the definitions.

General properties of functions:

```
Extensionality:
(1)    [| f : A->B ] ==> [| g : A->B |] ==> [| f=g <-> ALL x. x:A --> f^x = g^x |]

Total functions with empty domain:
(2)    [| 0->B = {0} |]

Total functions with empty range:
(3)    [| Not(A=0) |] ==> [| A->0 = 0 |]

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

composition
(5)    [| f : B -> C |] ==> [| g : A -> B |] ==> [| f @ g : A -> C  |]
(6)    [| f : B -> C |] ==> [| g : A -> B |] ==> [| a : A |]
      ==> [| (f @ g) ^ a = f ^ (g^a)  |]
```

Typed Lambda Calculus:

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

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

Properties of the natural numbers:

```
(12)   [| 0 : Omega |]
(13)   [| n : Omega |] ==> [| succ(n) : Omega |]
(14)   [| n : Omega |] ==> [| m : n  |] ==> [| m <= n |]
(15)   [| m : Omega |] ==>[| n : Omega |] ==> [| m:n | n:m | m=n |]
(16)   [| n : Omega <-> n=0 | EXISTS m. n=succ(m) & m:Omega |]
```

Fig. 7. General theorems concerning relations and functions.

Properties have been defined by sets. Thus *Partial_order*(D) is a set of partial orders which may be constructed on the set D; *Function*(A, B) is the set of partial functions from the set A to the set B.

The definition of function application by $F^{\wedge}a = \mathit{Union}(\mathit{Image}(F, \{a\}))$ is justified by Theorem (4) in Figure 7 which states that, when F is a function and the image of a under F exists, then that image is unique.

The definition of natural numbers given in Figure 6 has proved cumbersome to use. 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. 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 Figure 6. For example the theorem stating that, for any two ordinals m and 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 $A \to A$ and a is an element of A, then there exists a unique function g of type $\omega \to A$ with the following properties:

$$g\hat{}0 = a$$

$$\forall n \cdot n \in \omega \to g\hat{}succ(n) = f\hat{}(g\hat{}n)$$

The definiens in the definition of the term *recs*(f, a) 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

$$recs\_s(f, a, n) = \{g \in n \cup \{n\} \to Domain(f) \mid g\hat{}0 = a \wedge \forall i \cdot i \in n \to g\hat{}succ(i) = f\hat{}(g\hat{}i)\}$$

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)$f\hat{}s$) 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.

3.2.2. *Some Theorems Concerning Functions*

Some theorems concerning functions, proved within set theory, are listed in Figures 7 and 8. The Theorems (7)–(11) form 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] pp. 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.

Simple recursion:

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

Simple recursion:
(3)    [| f : A->A |] ==> [| a : A |] ==> [| recs(f,a) : Omega->A |]
(4)    [| f : A->A |] ==> [| a : A |] ==> [| recs(f,a)^0 = a  |]
(5)    [| f : A->A |] ==> [| a : A |] ==> [| n : Omega |]
       ==> [| recs(f,a)^succ(n) = f^recs(f,a)^n |]
```

Generalised simple recursion:

```
(6)    [| Total(P) |] ==> [| grecs(P,a)^0 = a |]
(7)    [| Total(P) |] ==> [| n: Omega |]
       ==> [| P(grecs(P,a)^n,grecs(P,a)^succ(n)) |]

(8)    [| gfrecs(F,a)^0 = a |]
(9)    [| n: Omega |] ==> [| gfrecs(F,a)^succ(n) = F(gfrecs(F,a)^n) |]
```

Well-founded recursion:

```
Well-founded  induction
(11)   [| a : X |] ==> [| R : Well_founded(X) |]
       ==> [| ALL x. x:X --> (ALL y. <y,x> : R --> P(y) --> P(x) |]
       ==> [| P(a) |]
(12)   [| a : X |] ==> [| R : Well_founded(X) |]
       ==> [| ALL x. x:X --> (ALL y. y : Init(R,x) --> P(y)) --> P(x) |]
       ==> [| P(a) |]

Well-founded recursion
(13)   [| R : Well_founded(X) |] ==> [| x : X |]
       ==> (!(y) [| y:Init(R,x) |] ==> [| f^y^Restrict(wrec(X,Y,R,f),R,y) : Y |])
       ==> [| Restrict(wrec(X,Y,R,f),R,x) : Init(R,x)->Y |]
(14)   [| R : Well_founded(X) |] ==> [| x : X |]
       ==> (!(y) [| y:Init(R,x) |] ==> [| f^y^Restrict(wrec(X,Y,R,f),R,y) : Y |])
       ==> [| f^x^Restrict(wrec(X,Y,R,f),R,x) : Y  |]
       ==> [| wrec(X,Y,R,f)^x = f^x^Restrict(wrec(X,Y,R,f),R,x) |]

(15)   [| R : Well_founded(X) |]
       ==> (!(x) !(g) [| x : X |] ==> [| g : (Init(R,x)->Y) |] ==> [| f^x^g : Y |])
       ==> [| wrec(X,Y,R,f) : X->Y |]
(16)   [| R : Well_founded(X) |] ==> [| x : X |]
       ==> (!(x) !(g) [| x : X |] ==> [| g : (Init(R,x)->Y) |] ==> [| f^x^g : Y |])
       ==> [| wrec(X,Y,R,f)^x = f^x^Restrict(wrec(X,Y,R,f),R,x) |]
```

Fig. 8. Recursion theorems.

The basic properties of ω lead to the two forms of mathematical induction: (1) and (2) in Figure 8. Note that A in theorem (1) is a free variable of type *exp* (i.e. a term). Since free variables are required in ZF to represent higher order types (there is no way, for instance, of representing the replacement axiom of Figure 3 within the object language, and a free variable R has to be introduced to refer to the relations of the

language), it seems natural to allow their use in the representation of terms as well. The alternative form of theorem (1) in which A is existentially quantified within the second premise would be slightly more difficult to use in proofs. Theorems (1) and (2) are used in the proof of the simple recursion theorem, formalised in (3), (4) and (5). 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 $f^\wedge b$ (for $b : A$) are functions of the same type. In the case of curried functions, the conditions satisfied by *recs* may then be rewritten:

$$recs(f, a)^\wedge 0^\wedge x_1^\wedge \ldots x_m = a^\wedge x_1^\wedge \ldots x_m$$

$$\forall n \cdot n \in \omega \rightarrow recs(f, a)^\wedge succ(n)^\wedge x_1^\wedge \ldots x_m$$

$$= f^\wedge (recs(f, a)^\wedge n^\wedge x_1^\wedge \ldots x_m)^\wedge x_1^\wedge \ldots x_m$$

(for any instance of $x_1, \ldots x_m$ 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. $f = h^\wedge n$). 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 $\omega \rightarrow A \rightarrow A$, and the set a, of type A, could be derived from simple recursion in the following way:

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

$$precs(h, a) = \lambda n \in \omega \cdot Tl(g^\wedge n)$$

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

$$precs(h, a)^\wedge 0 = a$$

$$precs(h, a)^\wedge succ(n) = h^\wedge Hd(g^\wedge n)^\wedge Tl(g^\wedge n)$$

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

The generalised theorem for simple recursion (see [8] p. 143) states that if a formula $P(x, y)$ is such that y has a unique value for every x, then there exists a unique function F such that $F^\wedge 0$ is any set a and $P(F^\wedge n, F^\wedge succ(n))$ holds for each n in ω. The Theorems (6) and (7) in Figure 8 show that the function $grecs(P, a)$ possesses these properties. There is no typing theorem for the generalised simple recursion. However, from the definition it may be deduced that it is a total function with domain ω. The recursive function $gfrecs(F, a)$ is a weaker version of $grecs(P, a)$: one in which the functional expression $P(x, y)$ is of the form $y = F(x)$. Theorems (8) and (9) are the specialisation of (6) and (7) for this particular case.

The theorems concerning well-founded recursion are presented in two pairs. The second pair, consisting of the Theorems (15) and (16), has been derived from the first pair, (13) and (14), by using the well-founded induction formalised in (12). Although

(15) and (16) are easier to use, (13) and (14) have been retained because they are stronger: they do not require that $f\hat{}x\hat{}g \in Y$ be true for every $g \in (Init(R, x) \to Y)$, but only for restrictions of the recursive function.

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:

$$\frac{ordinal(\alpha) \quad \begin{matrix}[y \in \alpha \quad g \in y \to Y] \\ f\hat{}g \in Y\end{matrix} \quad x \in \alpha}{orec(\alpha, Y, f)\hat{}x = f\hat{}Restrict(orec(\alpha, Y, f), mem(\alpha), x)}$$

where the recursive function is:

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

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

Notice that for each kind of recursion, the set of theorems specifies the basic properties of the corresponding function. It should be straightforward to prove by induction that the function satisfying these basic properties is unique.

3.2.3. *Example of the Use of Recursion: Lists*

Lists may be represented by sequences. The type of finite sequences may be defined in the following way:

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

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 take the null list to be *type*, and thus ensure, by the foundation axiom, that it is not an element of *type*. The types of lists of length n and the type of lists of any length are defined in Figure 9, and the main theorems concerning them in Figure 10. Binary trees could be defined similarly:

$$tree(type, n) = gfrecs(\lambda x \cdot type * x \cup type * (x * x), \{type\})\hat{}n$$

$$Tree(type) = \bigcup(\{tree(type, n) \mid n \in \omega\})$$

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

```
Type of lists of length n:
list(type,n)   = gfrecs(%(x)type*x,{type})^n

Type of finite lists:
List(type)     = Union([list(type,n) || n:Omega ])

Immediate sublist relation:
Sublist(type)  = Union([ [<x,<y,x> > || y:type ] || x:List(type)])
```

An example of recursive function, the function 'length(type)':

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

f_length(type) = lam(List(type),%(u)lam(Function(List(type),Omega),%(v)
                        When(v,Init(Sublist(type),u)->Omega,
                             When(u,{type},0,succ(v^Tl(u))),0) ))

length(type)   = wrec(List(type),Omega,Sublist(type),f_length(type))
```

Fig. 9. Definitions concerning lists.

```
(1)    [| list(type,0) = {type} |]
(2)    [| n : Omega |] ==> [| list(type,succ(n)) = type * list(type,n) |]
(3)    [| type : List(type) |]
(4)    [| x : type |] ==> [| y : List(type) |] ==> [| <x,y> : List(type) |]
(5)    [| x : List(type) |] ==> [| Not(x : {type}) |]
       ==> [| x : type*List(type) |]
```

Theorems concerning the function length:

```
(6)    [| length(type) : List(type) -> Omega |]
(7)    [| length(type)^ type = 0 |]
(8)    [| y : type |] ==> [| x : List(type) |]
       ==> [| length(type)^ <y,x> = succ(length(type)^x) |]
```

Fig. 10. Theorems concerning lists.

The function *length(type)* illustrates the use of well-founded recursion in the definition of recursive functions. The function *When*, which is used in its definition is simply a conditional operator satisfying the following properties:

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

The definition of the basic function *f_length* is cumbersome. It is defined as a total function over *Function(List(type), ω)* and therefore must be such that for *all* functions g, not just restrictions of total functions, *f_length(type)*$\hat{}a\hat{}g$ is of type ω. This is the purpose of the first occurrence of *When*. All the recursive functions over lists may be defined in a similar way.

3.2.4. *Further Developments*

Further developments of ZF concerning functions are given in the appendices. Appendix A concerns Tarski's fixed point theorem; Appendix B concerns the part of domain theory which form the logical basis of LCF. Another line of work consists in defining the semantics of other theories within ZF set theory, and then proving the axioms of the theory. Appendix C includes a semantics for simple type theory, together with an axiomatisation of it derived within ZF. Appendix D includes a semantics for intuitionistic first-order logic in which formulae are interpreted as sets of proofs, together with a derived axiomatisation of the logic. The same approach has also been used to define a semantics for a first-order temporal logic. In this case, the formulae are interpreted as sets of sequences (or lists) of states, where states are themselves defined by sets of variables. This is an example where a means of reasoning about both functions and sets is required. The basic theorems of the temporal logic have been derived within ZF. A set of more specific theorems has also been derived, which, when sequenced by an appropriate tactic in a backward proof, generates automatically models of propositional temporal formulae.

3.3. REASONING ABOUT TYPES

The previously described developments show that set theory is suitable for reasoning about functions. Despite the restriction imposed by the foundation axiom, set theory seems also to be well suited for reasoning about types. Given some basic types, such as ω or the type of truth values *bool*, one may construct the standard types such as cartesian product, disjoint union, dependent product and dependent function space. The dependent product $D\_product(A, P)$ is the set of all pair $\langle x, y \rangle$ such that $x \in A$ and $Y \in P(x)$:

$$D\_product(A, P) = \bigcup(\{\{x\} * P(x) \mid x \in A\})$$

The dependent function space $D\_function(A, P)$ is the set of all total functions f over A such that, if $x \in A$ then $f\hat{}x \in P(x)$:

$$D\_function(A, P) = \{F \in A \to \bigcup(\{P(y) \mid y \in A\}) \mid \forall x \cdot x \in A \to F\hat{}x \in P(x)\}$$

In fact every set denoted by 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 $A \to A$, itself a subtype of the set of partial functions over A, $Function(A, A)$.

Recursive types may be constructed in the same way that the type of lists was constructed in Section 3.2.3. 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:

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

$$f^{\wedge}succ(n) = f^{\wedge}n \cup \{Hd(x) \to Tl(x) \mid x \in f^{\wedge}n * f^{\wedge}n\}$$

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

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:

$$f^{\wedge}0 = A$$

$$f^{\wedge}succ(n) = f^{\wedge}n \cup \{X \in Pow(Function(\bigcup(f^{\wedge}n), \bigcup(f^{\wedge}n)) \mid \exists T \cdot \exists F.$$

$$X = D\_function(T, \%(x)F^{\wedge}x)$$

$$\wedge\ T \in f^{\wedge}n\ \wedge\ F \in T \to f^{\wedge}n\}$$

The dependent function space defines a new type at a given level of the hierarchy for all types T of the previous level, and all functions F associating 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.

4. Comments Concerning the Proofs

The main purpose of the research described in this paper was to develop set theory sufficiently to be able to reason about functions. The emphasis has been to obtain a large number of theorems, formulated in a simple way, without much regard to the form of the proofs themselves and to the level of automation. 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. Most of the proofs have been carried out in backward style. The last part of the section describes an attempt at producing natural deduction proofs in forward style.

4.1. 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 (see [16], for instance, for a comparison between systems of deduction). The rules of sequent calculus are well suited

to backward proofs because they consist only of introduction rules. They can be used to systematically eliminate the constants in the goals. However, when compared with natural deduction, 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 antecedents of meta-level implications may be interpreted as object-level assumptions, and thus the assumptions of natural deduction fit naturally in the meta-level logic. Furthermore, 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. The introduction rules of natural deduction have the same form as the right rules of sequent calculus. However, the elimination rules have generally a form different from the left rules. In the case of intuitionistic first-order logic (Figure 1) three elimination rules are not in sequent style: the ones concerning conjunction, implication and universal quantification. Here are the three corresponding sequent-style rules:

$$\textit{conj\_elim}: \quad [\![P \wedge Q]\!] \Rightarrow ([\![P]\!] \Rightarrow [\![Q]\!] \Rightarrow [\![R]\!]) \Rightarrow [\![R]\!]$$

$$\textit{imp\_elim}: \quad [\![P \rightarrow Q]\!] \Rightarrow [\![P]\!] \Rightarrow ([\![Q]\!] \Rightarrow [\![R]\!]) \Rightarrow [\![R]\!]$$

$$\textit{all\_elim}: \quad [\![ALL\ x \cdot P(x)]\!] \Rightarrow ([\![P(a)]\!] \Rightarrow [\![R]\!]) \Rightarrow [\![R]\!]$$

For ease of use, the chosen derived rules are such that the connectives to be eliminated are in the leftmost position. It should be clear, however, that the rules above are equivalent to the corresponding left rules of sequent calculus. In the case of a logic with equality, one more pair of rules is required: the reflexivity rule, $[\![a = a]\!]$, which is an equality introduction rule, and the substitution rule $[\![a = b]\!] \Rightarrow [\![P(b)]\!] \Rightarrow [\![P(a)]\!]$, which is an equality elimination rule. The resulting set of rules in sequent style may be used to eliminate the constants on the right and on the left of any given goal. There are several problems concerning the 'exists_intr' and 'all_elim' rules. First, they introduce new schematic variables which are likely to be instantiated later on in the proof. Usually, only some of the possible instantiations lead to a proof, and backtracking is necessary when a wrong choice is made. Second, more than one instantiation may be required during the course of a proof. Thus the rules are not complete. A complete rule should be expressed in such a way that the quantification is carried through the proof. A complete 'all elimination' rule, for instance, could be expressed in the form:

$$[\![ALL\ x \cdot P(x)]\!] \Rightarrow ([\![P(a)]\!] \Rightarrow [\![ALL\ x \cdot P(x)]\!] \Rightarrow [\![R]\!]) \Rightarrow [\![R]\!]$$

However, such complete rules must be used with care in automatic tactics, as they may lead to circularities. Also, as discussed in [13], when several quantifier rules apply, they should be applied in such a way that the schematic variables which are introduced impose the least possible constraints on future unifications: the variables must be functionally dependent on the previously introduced universally quantified variables. Thus, 'all_intr' should precede 'exists_intr' and 'exits_elim' should precede 'all_elim'.

In the sequent-style rules discussed above, it is the rightmost meta-level implication which stands for the symbol $\vdash$ of the corresponding sequent rules. Thus the natural deduction formalism allows only a single formula on the right of $\vdash$. While this is adequate for intuitionistic logic, it is not for classical logic. The extra sequent rules for classical logic may be expressed as:

not_right: $[\![H, P \vdash R]\!] \Rightarrow [\![H \vdash R, \neg P]\!]$

not_left: $[\![H \vdash R, P]\!] \Rightarrow [\![H, \neg P \vdash R]\!]$

A possible natural deduction formulation of the extra rules is:

not_intr: $([\![P]\!] \Rightarrow [\![Q]\!]) \Rightarrow ([\![P]\!] \Rightarrow [\![\neg Q]\!]) \Rightarrow [\![\neg P]\!]$

not_elim: $[\![\neg\neg P]\!] \Rightarrow [\![P]\!]$

In backward proof, the elimination of negation by 'not_intr' reintroduces a new negation, leading to the possibility of circularities in automatic proofs. In Section 4.4 a natural deduction formulation of classical logic which is closer to the sequent formulation will be discussed.

The new constants of ZF on the right-hand-side of a membership symbol may be eliminated in the same way as the other constants of classical logic. The main problem concerns equality, for which ZF provides an extra axiom, the extension axiom. A method for dealing with equality in automated tactics will also be discussed in Section 4.4.

4.2. TACTICS

The basic tactics of Isabelle are described in [13] and [15]. The tactic 'resolve_tac ths i' resolves the ith goal of the current proof state with one of the theorems in the list ths. The tactic 'assume_tac i' attempts to find a unifier between the consequent of goal i and one of its antecedents. The goal is eliminated from the proof state if the unification is successful. Both the tactics just mentioned produce a new proof state, together with a lazy list of alternatives.

A third basic tactic is the resolution 'with elimination': the tactics 'eresolve_tac ths i' performs the same function as the sequence of tactics 'resolve_tac ths i THEN assume_tac i', except for the fact that the assumption which allows the elimination of the first subgoal is eliminated from the other subgoals. The tactic is generally used with elimination rules (i.e. left rules of sequent calculus). As an illustration of the way 'eresolve_tac' works, consider the tactic 'eresolve_tac [eq_elim] i', where 'eq_elim' is

$$\frac{?a = ?b \quad ?P(?b)}{?P(?a)}$$

and the ith goal of the current proof state is

$$\frac{c_1 = c_2 \quad H}{Q(c_3, c_1)}$$

The resolution step produces initially the new subgoals:

$$\frac{c_1 = c_2 \quad H}{c_3 = ?b} \qquad \frac{c_1 = c_2 \quad H}{Q(?b, c_1)}$$

The subsequent assumption step fails. Backtracking occurs and the next proof state generated by the resolution step is retrieved. The subgoals are now:

$$\frac{c_1 = c_2 \quad H}{c_1 = ?b} \qquad \frac{c_1 = c_2 \quad H}{Q(c_3, ?b)}$$

which are further reduced, after the assumption step and the elimation of the antecedent $c_1 = c_2$, to:

$$\frac{H}{Q(c_3, c_2)}$$

In effect, the tactic ensures that $Q(c_3, c_1)$ is interpreted as a function of c_1, during its unification with $P(a)$.

More complex tactics have been constructed in ML from these basic Isabelle tactics. 'REPEAT(step_tac i)' attempts to solve a goal automatically using the rules of classical logic and the assumption rule. The tactic is applied to the ith goal. If the goal is solved, goal $i + 1$ becomes goal i, 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 thus the tactic is not always suitable.

Some of Isabelle's tactics perform rewriting using the meta-level equality. However, the development of ZF described in this paper uses exclusively object-level definitions. Specific tactics for rewriting have been written using object-level equalities. Some tactics perform conditional rewriting, using rules of the form

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

or

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

where the h_is are hypotheses, which may or may not be present, the $term_i$s are terms, and the $form_i$s are formulae.

With rules of the first form, the tactic 'unfold_right [rules] i' replaces each formula in the conclusion of goal i which unifies with a formula $form_1$ by the corresponding formula $form_2$, while new goals are created from the corresponding hypotheses h_1, h_2, ... Similarly, the tactic 'unfold_left [rules] i' performs the same function in the hypotheses of the selected goal, and 'unfold_all [rules] i' in both the conclusion and the hypotheses. With a rule of the second form, the tactic 'rewrite_right_1 [rule] i' performs a one-step rewriting of the conclusion of the ith goal.

```
> 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)
```

Fig. 11. Example of proof (1): application type.

Simple ML functions have been written to perform conversions between various forms of the same rule. One such function, which is used in the examples of Figures 11 and 12 is the function *bimp*. This is a function which rewrites rules of the second form above into the first form:

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

```
> 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)
```

Fig. 12. Example of proof (2): decomposition of an ordered pair.

4.3. EXAMPLES OF PROOF

An example of backward proof is given in Figure 11. The required theorem and the specified tactics appear after the symbol '>'. The command specifying a tactic results in a list of new goals. The proof is initiated by specifying the theorem to be proved: here the theorem is

$$\frac{F \in A \to B \quad x \in A}{F\hat{}x \in B}$$

The object-level assumptions $F \in A \to B$ and x $\in A$ in the first goal are interpreted by Isabelle as meta-level assumptions (this may be a bit confusing since normally object-level assumptions are represented by meta-level implications). The first step consists of a tactic which discharges these assumptions. The second step unfolds the 'total function' and 'Collect' symbols, according to the definitions of Figures 6 and 3. The third step performs a classical deduction on the current goal. In the next step, 'Apply' is the following theorem:

$$\frac{F \in \mathit{Function}(A, B) \quad \langle x, y \rangle \in F}{F\hat{}x = y}$$

Isabelle has to lift the rule over the meta-level quantifier in order to perform the required unification. In particular it replaces the variables A, B, and y, respectively, by $?A1(ka)$, $?B1(ka)$ and $?b1(ka)$ (recall that variables preceded by a question mark are schematic variables). Rewriting $F\,\hat{}\,x$ with this conditional equality produces three 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

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

which appears somewhat 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 advantage is more than offset by the disadvantage 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 require a shift of the relevant formula to the leftmost position, and a non selective one may perform some unwanted unfolding; also, the automatic tactic for classical deduction is more likely to produce unwanted results when performed within a large proof. It is worth noting that breaking down a proof into 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 which may be used in the rest of the proof.

The example in Figure 12 illustrates the use of the standard resolution tactics 'resolve_tac [rules] i' and 'eresolve_tac [rules] i', and in particular the tactic performing a substitution from an equality in the hypothesis, 'eresolve_tac [eq_elim] i'. `refl` is the reflexivity axiom for equality, $x = x$. The theorems appearing in the proof, and not mentioned before are:

`Pair_eq3`: $\mathit{Hd}(\langle x, y\rangle) = x$

`Pair_eq4`: $\mathit{Tl}(\langle x, y\rangle) = y$

where x and y are free variables.

The two previous examples of proof are very simple. Many of the theorems mentioned in Section 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 general tactics, in which the relevant rules are stored in some appropriate sequence: i.e. an increase in the level of automation.

4.4. LEVEL OF AUTOMATION

Most of the theorems mentioned in this paper have been proved using the simple tactics mentioned in the previous section. However, some initial work has been done on automation and a tactic has been written with which a number of simple theorems have been proved automatically. This tactic uses a basic enumerative strategy. It eliminates the constants and performs all the possible substitutions until a goal may be solved by contradiction, or by an application of the 'null' axiom. The current implementation of the tactic is restricted in several ways. In particular, it applies only to formulae with predicate variables and function variables of arity at most 1, and it does not check whether the specified goal may be solved by the infinity axiom or the foundation axiom. Two problems had to be solved: how to avoid circularities when eliminating negation in natural deduction, and how to eliminate the equality symbols in a way which does not introduce incompleteness.

The tactic deals with the first problem by expressing every goal in the form

$$H \Rightarrow [\![False]\!]$$

This can be achieved by resolving the goal with the following rule:

$$([\![\neg P]\!] \Rightarrow [\![False]\!]) \Rightarrow [\![P]\!]$$

The resulting goal may be interpreted as a sequent in which all formulae preceded by a negation symbol appear to the right of the $\vdash$ symbol and all other formulae to the left. In this way, every sequent may be represented in natural deduction style. The introduction rules have to be converted into elimination rules. For instance, the conjunction introduction rule

$$[\![P]\!] \Rightarrow [\![Q]\!] \Rightarrow [\![P \wedge Q]\!]$$

becomes the elimination rule:

$$[\![\neg(P \wedge Q)]\!] \Rightarrow ([\![\neg P]\!] \Rightarrow [\![R]\!]) \Rightarrow ([\![\neg Q]\!] \Rightarrow [\![R]\!]) \Rightarrow [\![R]\!]$$

(note that the constant *False* could be used in place of the variable R). The negation rules of sequent calculus, which simply move the negation symbols to the right or to

the left of the $\vdash$ symbol are not required. However, a rule to eliminate double negation is necessary:

$$[\![\neg\neg P]\!] \Rightarrow ([\![P]\!] \Rightarrow [\![R]\!]) \Rightarrow [\![R]\!]$$

With respect to the equality symbol, its introduction and elimination rules need to be more general than in the case of classical logic. They are based on the extension axiom:

$$[\![a = b \leftrightarrow a \subseteq b \wedge b \subseteq a]\!]$$

However, to perform substitutions one must now take account of the fact that equality may be present without being explicit.

Automation is achieved by using the following tactic with a depth first strategy:

```
fun basic_tac r i = simp_tac i
                    ORELSE DETERM(eresolve_tac r i)
                    ORELSE (subs_tac i
                            APPEND eresolve_tac unsafe_r i)
```

where i specifies the goal to which the tactic applies and r specifies a list of elimination and introduction rules. The list of rules which has been used concerns the following constants: *Replace*, *Collect*, the pairing symbol, the symbols defining both kinds of intersection and union, the powerset, subset and equality symbols, and the classical connectives. It does not contain the 'all elimination' rule nor 'exists introduction' rule: these two rules are contained in the list *unsafe_r*. The 'ORELSE' and 'APPEND' tacticals specify a choice of tactics. They differ in the following way: if one of the specified tactics is successful, the other one will be attempted after backtracking in the case of 'APPEND', but will not in the case of 'ORELSE'. Here are the functions of the various part of the tactic:

- *simp_tac i* attempts to solve the goal by contradiction, or using the null axiom. To improve efficiency, it also attempts to solve the goal using equality reasoning (making use of reflection, symmetry, transitivity and congruence).
- *DETERM* (*eresolve_tac r i*) eliminates a constant using the first applicable rule of r. The tactical *DETERM* makes this choice deterministic: no alternative proof state is generated. This may be done because the order in which the rules of r are used does not affect the provability of a proposition.
- *subs_tac i* is the tactic which chooses one of the negated formulae in the goal as a target for substitution (it moves the formula to the right) and performs the possible substitutions. The general substitution rule may be expressed as:

$$[\![P(x)]\!] \Rightarrow [\![y = x]\!] \Rightarrow [\![P(y)]\!] \tag{1}$$

Used on its own, this rule may lead to circularities after $y = x$ has been expanded. Thus two more rules have been used, specialised to the case where $P(y)$ is of the form $y \in \_$:

$$[\![x \in F(u)]\!] \Rightarrow [\![v = u]\!] \Rightarrow [\![y = x]\!] \Rightarrow [\![y \in F(v)]\!] \tag{2}$$

$$[\![x \in A]\!] \Rightarrow [\![y = x]\!] \Rightarrow [\![y \in A]\!] \quad (3)$$

Rule (2) is applied only if rule (3) cannot be applied. Similarly, rule (1) is applied only if rule (2) cannot be applied. Also, rule (2) must not be applied to the case where F is the identity function, as this may lead to circularities. As the identity function is the one produced by the first unification, all that is required is to consider only the alternative unifications. Rule (1) deals with the case where P is a monadic predicate variable.

- *eresolve_tac unsafe_r i* is the tactic performing the 'all introduction' and 'exists elimination' rules. It uses the uncomplete form of the rules.

Here are examples of the theorems proved by the tactic:

$$\exists x \cdot x \in X \leftrightarrow \neg X = 0$$

$$x \in \{a\} \leftrightarrow x = a$$

$$\{a, b\} = \{b, a\}$$

$$x \subseteq A \rightarrow x \cap y \subseteq A$$

$$A \in C \rightarrow \bigcap C \subseteq A$$

$$\bigcup \{B\} = B$$

$$\{P(x) \mid x : \{Q(y) \mid y \in A\}\} = \{P(Q(y)) \mid y \in A\}$$

The proofs took between a few seconds to 1 minute on a Sun 4. However, other simple theorems took much longer: the proof of $x \cap y \subseteq y \cap x$ took 7 minutes. It is clear that an increase in the complexity of the formulae, and an increase in the set of constants to eliminate would result in very inefficient proofs. The problem could be alleviated by providing relevant lemmas and theorems to guide the proofs, and by making sure that the unfolding of definitions occur only when necessary. Much more work is required to pursue this line of research.

4.5. 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.

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 (previously proved):

```
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) ] |]
```

Fig. 13. Example of forward proof: Tarski's theorem.

- 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 in Figure 14. The assumptions could be adequately represented by meta-level implications. However, this formalism leads to complex

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) |] ]
```

Fig. 14. Example of forward proof: Tarski's theorem.

procedures to eliminate duplicate assumptions and select the appropriate assumptions to be discharged. Thus, for practical reasons, the assumptions have been represented by meta-level assumptions. In Figure 14, the assumptions are listed in square brackets after each displayed theorem. The proof uses the assumptions and previously proved

lemmas listed in Figure 13. 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 ML definitions of the rules which are used in the proof of Tarski's theorem:

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

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

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

The ML function MRES, which has been defined using more basic Isabelle functions, performs a meta-level resolution without lifting, 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 *imp_intr*, *all_intr* and *mp* are some of the elimination rules and introduction rules of the intuitionistic logic displayed in Figure 1:

imp_intr: $(\llbracket P \rrbracket \Rightarrow \llbracket Q \rrbracket) \Rightarrow \llbracket P \rightarrow Q \rrbracket$

all_intr: $(\wedge y \cdot \llbracket P(y) \rrbracket) \Rightarrow \llbracket \forall x \cdot P(x) \rrbracket$

mp: $\llbracket P \rightarrow Q \rrbracket \Rightarrow \llbracket P \rrbracket \Rightarrow \llbracket Q \rrbracket$

The ML functions *implies_intr* and *forall_intr* are meta-level introduction rules described in Section 2.1. 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 $\llbracket Q \rrbracket$ under the meta-level assumptions $\llbracket P_1 \rrbracket$, $\llbracket P_2 \rrbracket$, . . . , and p is the string '$\llbracket P_i \rrbracket$', then (*implies_intr* (*rprop p*) *th*) is the theorem $\llbracket P_i \rrbracket \Rightarrow \llbracket Q \rrbracket$ under the same assumptions with every instance of P_i removed. If *th* is the theorem $\llbracket P(x) \rrbracket$ under some assumptions and s is the string 'x', then (*forall_intr* (*rterm s*) *th*) is the theorem $\wedge x. \llbracket P(x) \rrbracket$ under the same assumptions. Thus, if *th* identifies a theorem $\llbracket T \rrbracket$ under the meta-level assumptions $p \cup A$, then *imp_i p th* identifies the theorem $\llbracket p \rightarrow T \rrbracket$ under the assumptions A. If *th* identifies a theorem $\llbracket T(x) \rrbracket$ under the assumptions A, then *all_i* 'x' *th* identifies the theorem $\wedge x. \llbracket T(x) \rrbracket$ under the assumptions A. If *th*1 identifies the theorem $[H \rightarrow T \rrbracket$ under the assumptions A and *th*2 identifies the theorem $\llbracket G \rrbracket$ under the assumptions B, and if the first unifier of H and G is θ, then *th*1 '*th*2 is the theorem $T\theta$ under the assumptions $A \cup B$. The format *th*1 '*th*2 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.

The effect of the above functions may be seen in Figure 14, 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: *Complete_lattice*(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.

5. 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 von Newmann–Gödel–Bernays axiomatisation – and write it in clausal form. Theorems are proved by refuting their negation, also in clausal form. Although their eventual aim is to provide an automatic theorem prover, they recognise that automation is presently limited by the problems mentioned in Section 4.4 (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.

6. Conclusion

A number of theorems concerning functions have been provided within ZF set theory using the theorem prover Isabelle. 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. Thus, the development of a theory of functions within set theory provides a uniform and consistent system for reasoning about functions over arbitrary types. The development has proved adequate for defining the semantics of other theories, and deriving their axiomatisation. Isabelle is well suited to the derivation of theories within theories: since 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.

The emphasis of the work has been to obtain theorems formulated in a clear and simple way, rather than easy to prove. 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.

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, Martin Coen, David Carlisle, Brian Monahan and Frances Townsend for reading various drafts of the paper and providing useful comments. The funding of the research was provided by the SERC grant GR/E0355.7.

Appendices

A. TARSKI'S FIXED POINT THEOREM

A number of theorems concerning fixed points have been proved. Tarski's fixed point theorem concerns complete lattices. Appendix B includes fixed point theorems concerning cpos. Both make use of the concept of least upper bound. The definitions, and some of the theorems required in the proof of Tarski's fixed point theorem are given in Figures 15 and 16. Given a partial order, say R, its underlying set is completely determined, and may be referred 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. Theorem (1) shows that, if this set is not empty, it is

Upper bounds and least upper bound:

```
Ubs(R,Y)            = [ y || y : Domain(R), EXISTS X. 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))

Inv(R)              = [ <Tl(x),Hd(x)> || x:R ]

Glb(R,Y)            = Lub(Inv(R),Y)
```

Complete lattice:

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

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))
```

Fig. 15. Definitions concerning fixed points.

Basic properties of upper bound:

```
Uniqueness:

(1)   [| a : Lubs(R,Y) |] ==> [| b : Lubs(R,Y) |] ==> [| a = b |]

Lub is an ub:
(2)   [| Not(Lubs(R,Y) = 0) |] ==> [| x:Y |] ==> [| <x,Lub(R,Y)> : R |]

Lub is least:
(3)   [| Not(Lubs(R,Y)=0) |] ==> [| x : Ubs(R,Y) |] ==> [| <Lub(R,Y),x> : R |]

Lubs of pairs:
(4)   [| Not(Lubs(RA,{x,y})=0) |] ==> [| Lub(RA,{x,y})=y |] ==> [| <x,y> : RA |]

(5)   [| RA : Partial_order(A) |] ==> [| <x,y> : RA |] ==> [| y : Lubs(RA,{x,y}) |]
```

Basic properties of fixed points:

```
(6)   [| R : Partial_order(A) |] ==> [| x : Lfixs(f,R) |] ==> [| y : Lfixs(f,R) |]
      ==> [| x = y |]

(7)   [| R : Partial_order(A) |] ==> [| Not(Lfixs(f,R) = 0) |]
      ==> [| f^Lfix(f,R) = Lfix(f,R) |]

(8)   [| R : Partial_order(A) |] ==> [| Not(Lfixs(f,R) = 0) |] ==> [| x : Fix(f) |]
      ==> [| <Lfix(f,R),x> : R |]
```

Properties of inverse relations:

```
(9)   [| R : Complete_lattice(A) |] ==> [| Inv(R) : Complete_lattice(A) |]

(10)  [| R : Partial_order(A) |] ==> [| f : Monotone(R,R) |]
      ==> [| f : Monotone(Inv(R),Inv(R)) |]
```

Tarski's fixed point theorem:

```
(11)  [| f : Monotone(Lat, Lat) |] ==> [| Lat : Complete_lattice(A) |]
      ==> [| Glb(Lat, [ u || u : A, <f^u,u> : Lat ]) : Lfixs(f,Lat) |]

(12)  [| f : Monotone(Lat, Lat) |] ==> [| Lat : Complete_lattice(A) |]
      ==> [| Lfix(f,Lat) = Glb(Lat, [ u || u : A, <f^u,u> : Lat ]) |]
```

Fig. 16. Theorems concerning fixed points.

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 0, even though there is no least upper bound. A similar remark applies to the least fixed point of a function f in a relation R. Theorem (6) states that, if R is a partial order and the set of least fixed points $Lfixs(f, R)$ is not empty, then it is a singleton. Under these conditions, $Lfix(f, R)$ is *the* least fixed point of f.

The greatest lower bound of a partial order, which is used in Tarski's theorem, is simply the least upper bound of the inverse partial order. The properties of inverse relations relevant to Tarski's theorem have been proved. In particular, Theorem (9), which states that a complete upper semi-lattice is also a complete lower semi-lattice, justifies the definition of a complete lattice as simply a complete upper semi-lattice.

Both Theorems (11) and (12) are required to express Tarski's fixed point theorem. The two are necessary: if the glb in (12) 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 Theorem (11) which asserts the existence of the least fixed point. Note that Theorem (12) 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.

B. DOMAIN THEORY IN *PPλ*

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 *PPλ* relating to domain theory, as specified in [12] or [7].

The relevant definitions are given in Figure 17, while the theorems are listed in Figure 18. The definition of *bottom* as the union of the set of bottom elements is

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:

```
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]
```

Fig. 17. Definitions concerning *PPλ*.

Continuous functions:

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

Uniqueness of bottom element:

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

Constructions of some cpos:

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

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

Domain theory:

```
Extensionality:
(8)  [| 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:
(9)  [|  <f,g> : Func(RA,RB) |] ==> [|  <x,y> : RA |] ==> [|  <f^x,g^y> : RB |]

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

Least fixed point in cpo:
(11) [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]
     ==> [|  Lub(RA,[recs(f,bottom(RA))^n || n:Omega]) : Lfixs(f,RA) |]
(12) [| 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:
(13) [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]
     ==> [|  f^Lfix(f,RA) = Lfix(f,RA)|]
(14) [| f : Continuous(RA,RA) |] ==> [|  RA : cpo(A) |]  ==> [| x : Fix(f) |]
     ==> [|  <Lfix(f,RA),x> : RA |]

Fixed point induction:
(15) [| 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)) |]
```

Fig. 18. Domain theory in $PP\lambda$.

justified by Theorem (3), 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 have 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 part of $PP\lambda$ which is not displayed in Figure 18 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.

C. 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 referred 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. 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 $P(x_1, x_2, \ldots, x_n)$ into monadic form $Q(\langle x_1, x_2, \ldots, x_n \rangle)$
- replacing the atomic formulae in monadic form $Q(x)$ by $x : Q$
- making explicit the typing requirements:

  $\forall x : A \cdot E(x)$ becomes $\forall x \cdot x : A \rightarrow E(x)$

  and

  $\exists x : A \cdot E(x)$ becomes $\exists x \cdot x : A \ \& \ E(x)$.

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 in Figure 19, where Andrews' formulation (theory Q_0 in [2], pp. 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 Q_0, listed below,

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):

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

Theorems concerning the typed equality:

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

Theorems recovering Andrews' definition of tr and fls:

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

Theorems expressing the axioms of simple type theory:

```
* Axiom 1 *
(8)  [| g : T->T |] ==> [| Eq(T)^ <And^(g^tr)^(g^fls),All(T)^g)> = tr |]
(9)  [| g : T->T |] ==> [| And^(g^tr)^(g^fls) = All(T)^g |]
* Axiom 2  *
(10) [| h : A->T |] ==> [| x:A |] ==> [| y:A |]
     ==> [| Imply^(Eq(A)^ <x,y>)^(Eq(T)^ <h^x,h^y>) = tr |]
* Axiom 3  *
(11) [| 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 |]
(12) [| 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) *
(13) [| a : A |] ==> [| Eq(T)^ <L(A, P) ^ a,P(a)> = tr |]
(14) [| a : A |] ==> [| L(A, P) ^ a = P(a) |]
* Axiom 5  *
(15) [| x : A |] ==> [| Eq(T)^ <Desc(A)^lam(A,%(y)Eq(A)^ <x,y>),x> = tr |]
(16) [| x : A |] ==> [| Desc(A)^lam(A,%(y)Eq(A)^ <x,y>) = x |]
```

Fig. 19. Simple type theory.

the types are specified by the subscripts:

1. $g_{o\to o}\mathbf{t} \;\&\; g_{o\to o}\mathbf{f} = \forall x_o \cdot g_{o\to o} x_o$
2. $x_\alpha = y_\alpha \to h_{\alpha\to o} x_\alpha = h_{\alpha\to o} y_\alpha$
3. $(f_{\alpha\to\beta} = g_{\alpha\to\beta}) = (\forall x_\beta \cdot f_{\alpha\to\beta} x_\beta = g_{\alpha\to\beta} x_\beta)$

4. $(\lambda x_\alpha \cdot f_\beta(x_\alpha))y_\alpha = f_\beta(y_\alpha)$

5. $\iota_{i\to o)\to i}(\lambda x_i \cdot y_i = x_i) = y_i$

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 $\mathcal{Q}_0$, they are defined in terms of the basic symbols; in the translation, **t** and **f** are the predefined sets *tr* and *fls* (ideally different from the other sets used, 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 (Theorems (6) and (7) in Figure 19).

The relation between the typed equality and the equality of set theory is given by the Theorems (3), (4) and (5). Note that the left implication in the sentence

$$a \in S \;\&\; a = b \leftrightarrow Eq(S)^\wedge\langle a, b\rangle = tr$$

is provable if $Not(\perp = tr)$ may be proved, in which $\perp$ 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 (4) is adequate for the subsequent proofs.

The first version of the Axioms 1–5 in Figure 19 differs 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 type equality, is given. It is through the transformation from typed equality to set-theoretic equality that the inferences of $\mathcal{Q}_0$ can be carried over to set theory.

Axiom 5 and the definition of the description operator require some explanation. In the definition of Figure 19, 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 $f^\wedge a = tr$ 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.

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
```

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 |]
```

Fig. 20. Intuitionistic logic with proof objects.

D. INTUITIONISTIC LOGIC WITH PROOF OBJECTS

A semantics for a first order intuitionistic logic with quantification over types, in which formulae are interpreted as sets of proofs, is defined in Figure 20. The definitions of the connectives follow the general idea behind the concept of 'propositions as types'. Their meaning may be interpreted 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 of the logic have been derived within 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

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

An attempt to prove

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

produces

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

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 $All(A, Q) \to Exist(A, Q)$. 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

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

may be proved without hypothesis.

Notes

*The work has been carried out at the Computer Laboratory of the University of Cambridge.

[1] The version of Isabelle discussed in this paper is the one described in [13]; the latest version, described in [15], differs from it by a small change concerning only the syntax of the meta-language.

References

1. Aczel, Peter, *Non-Well-Founded Sets*, CSLI Lecture Notes 14 (1988).
2. Andrews, Peter B., *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press (1986).
3. Borzyszkowski, A., Kubiak, R., Leszczylowski, J., and Sokolowski, S., 'Towards a set-theoretic type theory', Technical report, Institute of Computer Science, Polish Academy of Sciences (September 1988).
4. Boyer, Robert, Lusk, Ewing, McCune, William, Overbeek, Ross, Stickel, Mark, and Wos, Lawrence, 'Set theory in first-order logic: clauses for Gödel's axioms', *J. Automated Reasoning* **2,** 287–327 (1986).
5. Corella, Francisco, 'Mechanising set theory', Technical Report RC 14706 (*65927), IBM Research Division (1989).
6. Gabbay, D. and Guenthner, F. (Eds.), *Handbook of Philosophical Logic*, D. Reidel Publishing Company (1983).
7. Gordon, Michael J. C., Milner, Robin, and Wadsworth, Christopher P., *Edinburgh LCF: A Mechanised Logic of Computation*, Springer-Verlag (1979). LNCS 78.
8. Hamilton, A. G., *Numbers, Sets and Axioms*, Pergamon Press (1982).
9. Hatcher, William S., *The Logical Foundations of Mathematics*, Pergamon Press (1982).
10. Hindley, J. Roger and Seldin, Jonathon P., *Introduction to Combinators and λ-Calculus*, Cambridge, University Press (1986).
11. Huet, G. P., 'Induction principles formalised in the calculus of constructions', in: *TAPSOFT 87*, pp. 276–286, Springer-Verlag (1987). LNCS 249.
12. Paulson, Lawrence C., *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge University Press (1987).
13. Paulson, Lawrence C., 'A preliminary user's manual for Isabelle', Technical Report 133, University of Cambridge Computer Laboratory (1988).
14. Paulson, Lawrence C., 'The foundation of a generic theorem prover', *J. Automated Reasoning* **5,** 363–397 (1989).
15. Paulson, Lawrence C. and Nipkow, Tobias, 'Isabelle tutorial and user's manual', Technical Report 189, University of Cambridge Computer Laboratory (1990).
16. Sundholm, Goran, 'Systems of deduction', in [6], Vol. 1, pp. 133–188 (1983).
17. Suppes, Patrick, *Axiomatic Set Theory*, Dover (1972).
18. Takeuti, G., *Proof Theory*, North-Holland, 2nd edn. (1987).