

PARTIAL OBJECTS IN TYPE THEORY

A Thesis

Presented to the Faculty of the Graduate School
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Scott Fraser Smith

January 1989

© Scott Fraser Smith 1989
ALL RIGHTS RESERVED

Partial Objects in Type Theory

Scott Fraser Smith, Ph.D.
Cornell University 1989

Intuitionistic type theories, originally developed by Martin-Löf, provide a foundation for intuitionistic mathematics, much as set theory provides a foundation for mathematics. They are of interest to computer scientists because the objects typed are computations, making type theory an appropriate setting for reasoning about computation. Type theories such as Nuprl or the theories of Martin-Löf have types for objects that always terminate, but objects which may diverge are not directly typable. If type theory is to be a full-fledged theory for reasoning about computations, we need to be able to reason about potentially diverging objects.

In this thesis we show how potentially diverging computations, which we call *partial objects*, may be typed by extending type theory to *partial object type theory*. New *partial types* are added to type partial objects. These types are usable: partial objects written in natural program notation can easily be shown to lie in the types. In addition to being able to express partial objects, it is also important to be able to reason about them; for this purpose general principles are given for proving facts about partial objects via induction.

The resulting theory serves as a foundation for computational as well as mathematical reasoning. It also gives insights into abstract recursion theory, leads to a new method for constructive reasoning, and sheds light on inductive methods for reasoning about recursive computation.

Biographical Sketch

Scott Fraser Smith was born in 1961. In 1983, he was awarded a BS in Chemistry and Computer Science from Purdue University. Five years later, he completed his doctorate at Cornell University.

Acknowledgements

This is not really my thesis; it collectively belongs to a group of people. My advisor, Prof. Constable, did admirably what advisors are meant to do: he pointed the way, and then listened carefully. Other members of the PRL group also can claim part of this thesis: Stuart Allen's work is the foundation on which this thesis lies, and his critiques of my work have always been relavatory, helping to turn ill-stated and ill-thought-out ideas into whole ones. Nax Mendler's work has also been a model for mine. The PRL group as a whole provided much intellectual stimulation, friendship and comraderie.

My undergraduate years at Purdue were well spent, in large part because I had the luck to meet up with Bill Jorgensen and others in the chemistry department.

Except the small part that belongs to me, the rest of this thesis was given to me by family and friends.

Contents

1	Introduction	1
1.1	Overview of this thesis	3
2	Type theory: background	4
2.1	Type systems and type theories	4
2.2	Principles of type theory	5
2.2.1	Types	5
2.2.2	Equality	7
2.2.3	The propositions-as-types principle	8
2.2.4	Type theory as a foundation for mathematics	9
2.2.5	Computational and mathematical interpretations	10
2.3	Nuprl	10
2.3.1	The Nuprl theory	11
2.3.2	The Nuprl system	12
3	A partial object type theory	13
3.1	Partial objects in existing type theories	13
3.2	New principles for partial objects	14
3.2.1	The partial type constructor	14
3.2.2	Forming partial objects	15
3.2.3	Reasoning about partial objects	15
3.3	A partial object type theory	16
3.4	The language of expressions	17
3.5	Assertions and hypothetical assertions	18
3.6	Rules	20
3.6.1	Rule conventions	21
3.6.2	Universes	21

3.6.3	Is	22
3.6.4	Evaluation	23
3.6.5	Equivalence	25
3.6.6	Inducement	26
3.6.7	Termination	27
3.6.8	Membership	28
3.6.9	Expression	28
3.6.10	Natural numbers	29
3.6.11	Dependent function space	29
3.6.12	Dependent product space	30
3.6.13	Partial type	31
3.6.14	Fixed-points	32
3.6.15	Computational induction	33
3.6.16	Miscellaneous	33
3.7	Reasoning in the theory	34
3.8	Logical reasoning	34
3.9	Extract-style proof	35
3.9.1	An extract-style proof	36
3.10	Defining Unions	37
3.11	Using types and rules	37
3.11.1	Using partial types	38
3.11.2	Untyped reasoning	39
3.12	Reasoning by induction	39
3.12.1	An example of computational induction	40
3.12.2	Induction on natural numbers	42
3.13	Defining equalities	42
4	Semantics of partial object type theory	44
4.1	The expressions	45
4.2	The type-free assertions	45
4.3	Defining the types and their inhabitants	49
4.4	Admissibility	55
4.4.1	Non-admissible types	56
4.4.2	Computational lemmas	56
4.4.3	Proof of admissibility	64
4.5	Consistency of the rules	67
4.5.1	Conventions	67

4.5.2	Proof of consistency	67
5	Topics in partial object type theory	74
5.1	Type theory as a programming logic	74
5.1.1	What makes good programming logic?	75
5.1.2	LCF and type theory compared	76
5.2	Fixed-points and induction	77
5.2.1	A unified fixed-point principle	77
5.2.2	Computational induction reconsidered	79
5.2.3	Two principles compared	80
5.3	Partial propositions	81
5.4	Abstract computability theory	82
5.5	Building a partial object type theory	83
5.5.1	Expressing computational induction	83
5.5.2	Expressing the fixed-point principle	84
A	A partial object Nuprl	85
A.1	The rules	86
	Bibliography	93

Chapter 1

Introduction

Type theories such as Nuprl [CAB⁺86] or Martin-Löf's CMCP [Mar82] are foundational theories for constructive mathematics; Martin-Löf had philosophical motivations for developing his intuitionistic type theory. Nuprl incorporates many of the ideas of CMCP, but it is also designed to be of practical use as the logic in a powerful theorem-proving system and as a foundation for reasoning about computing.

Type theories have an extensive collection of types, far surpassing those found in programming languages. Like programming languages, members of types are computations, but unlike programming language types, the computational objects typed in type theory are *total objects*: they always have values. For instance, the type $A \rightarrow B$ is the type of all total functions from A to B . Programming language types, on the other hand, contain *partial objects*, and there functions in $A \rightarrow B$ are partial functions from A to B .

As computer scientists, we would like type theory to be a foundational theory of computation. Type theory is a powerful theory for reasoning about total objects, but since there are no types for partial objects it is difficult to define and reason about them. It is crucial for a theory of computation to have as a component a theory of partial computation.

In Nuprl, it is possible to type partial functions by using the subtype constructor $\{x:A \mid P(x)\}$ (the type of all elements of A with property P) to restrict the domain of the function to exactly the elements for which it converges [CM85, CAB⁺86]. So if $D(x)$ is true just when $f(x)$ converges, f may be given the total type $\{x:A \mid D(x)\} \rightarrow B$. Domains $D(x)$ can in fact be defined for many functions. Some problems with this approach are functions must be proven to converge before they can be applied, and the domains D are large, making reasoning cumbersome. Another solution would

be desired, where partial objects are interpreted as themselves, not as total objects with elaborate conditions attached.

In this thesis, we give a new approach for constructing and reasoning about partial objects which we call *partial object type theory*. We propose new *partial types* \overline{A} and principles for using these types which allow unbounded computations to be expressed, typed, and reasoned about; the resulting theory is comparable in strength to Edinburgh LCF [GMW79] in its ability to reason about computations. The semantic treatment of types defined by Allen [All87b] is extended to give a natural semantics for these new concepts, meaning type theories that incorporate them can be proven consistent.

This new theory also sheds a different and interesting light on some old and venerable concepts. The theory is based on an abstract open-ended notion of computation which does not presume Church's thesis, but it comes as a surprise that there still are problems which can be proven unsolvable in the theory. Standard results of recursion theory may then be proven, and as more properties about computations are assumed more theorems are provable [CS88].

A new method for reasoning constructively may be used in the theory. To reason in type theory propositions are interpreted as types. Propositions P are translated to types P^* such that P is true just when P^* is nonempty; if $p \in P^*$, p is a construction that validates P . Using partial types, it is possible to construct *partial propositions* like $\forall x. \overline{\exists y. P(x, y)}$: the constructions that validate partial propositions may not always converge. Partial propositions are thus a *logical* notion of partiality.

In partial object type theory a general fixed-point principle allows fixed-points of functions to be typed. Fixed-point induction, developed by deBakker and Scott, is a general principle for reasoning about recursive programs; it is the induction principle used in LCF. We show fixed-point induction to be a specific case of the fixed-point principle. Computational induction is a new form of inductive reasoning in partial object type theory. Although computational induction is founded on different principles than the fixed-point principle, the two of them prove many of the same facts. Computational induction is a more straightforward principle, but proofs are longer than corresponding proofs done with the fixed-point principle.

Partial object type theory emerges as a viable theory for reasoning about computations, and makes type theory much more convincing as a foundation for computational reasoning, and thus more viable as a practical tool for writing correct programs.

1.1 Overview of this thesis

Chapter two provides background into some of the important principles of type theory. These principles are relevant to the construction of partial object type theory, and will be referenced frequently in later chapters.

In chapter three, a partial object type theory is presented. This theory is loosely based on Nuprl and CMCP, but unlike these theories, types come without any notion of equality on their members, and it is possible to freely reason about untyped computations. This theory was designed to give as simple a presentation of partial objects as possible. After the rules of the theory are given, sample theorems are proven which show how the theory may be used to reason.

Chapter four consists of a semantic account of the theory of chapter three, based on the non-type-theoretic semantics of Allen. This shows the rules of the theory to be sound, meaning the concept of partial object is sound. The types are interpreted by defining them inductively. Most of the rules then are quickly justifiable, but the fixed-point rule requires extra work. Only certain *admissible* types have fixed-points, and to prove that a collection of types is admissible requires the computational behavior of fixed-points to be examined in detail.

Chapter five is a loose collection of consequences of partial objects. First we argue that the theory is a powerful programming logic by way of comparison with LCF. LCF has proven itself to be useful, so type theory should also be useful, hopefully more so. The relationship between the fixed-point principle, fixed-point induction, and computational induction is discussed; this sheds light on the nature of induction over computations. Partial propositions and abstract computability, mentioned earlier, are briefly discussed. To conclude, we examine what the difficulties are in constructing a partial object type theory. The concepts of computational induction and the fixed-point principle pose special problems to theory designers; the main problems and possible solutions are explored.

In Appendix A, an extension to the Nuprl theory is given which implements partial objects.

Chapter 2

Type theory: background

There is much to be said about type theory; here we offer a few words to put the issues in focus. The chapter should be taken as an aid for understanding why type theories are put together as they are: some important principles of type theories are discussed, and the Nuprl type theory is looked at in more detail, helping to illustrate how type theory may be used to actually do mathematics.

2.1 Type systems and type theories

A *type system* is a formal system for assigning types to computational expressions. The typed λ -calculus (TLC), Gödel’s functionals of finite type, and the second-order λ -calculus (λ^2) [Gir71, Rey74] are all type systems. One fact that serves to distinguish type systems from typed programming languages is that only converging computations are typed in a type system. *Intuitionistic type theories* (called *type theories* for short) are particularly rich type systems that can serve as a logical basis for doing mathematics, much as can set theory or category theory. Some examples of type theories are Martin-Löf’s theories [Mar73, Mar82, Mar80], the Nuprl theory [CAB⁺86], Stenlund’s Theory of Species (TS) [Ste72], the Calculus of Constructions (CC) [CH85, CH88], and Feferman’s theory T_0 [Fef75] and its successor PX [HN87]. Some other theories of historical interest are AUTOMATH [dB70, dB80] and Scott’s theory of constructive validity [Sco70].

Martin-Löf has written extensively on how type theory is a foundation for doing mathematics [Mar80, Mar82, Mar83], and the approach taken here owes much to his views. Thus, hereafter the expression “type theory” generally refers to a theory

interpreted in the style advocated by Martin-Löf. Nuprl is a Martin-Löf-style type theory. TS, CC, and PX are not Martin-Löf-style theories, but many of the concepts of this thesis have analogues in them. Some of the concepts may also be applicable to more basic type systems.

2.2 Principles of type theory

A type theory has as its language a collection of *expressions*. Some of these expressions represent types, others computations; *types* are collections of expressions. *Assertions* (called judgements by Martin-Löf) express the truths of type theory; the statement $a \in A$ asserts that a inhabits (or, is a member of) the type A . Other forms of assertion are possible. The rules characterize the meaning of the assertions; however, the collection of rules is not to be viewed as a formal system, for more rules may be added at some future date.

There are some general features of type theories that are worthy of study. Here we consider the possibilities for types and notions of equality, review the principle whereby propositions may be represented as types, and mention some of the different ways type theory may be interpreted. Many of these principles also apply to type systems.

2.2.1 Types

The expressiveness of type theory is largely due to the diversity of types that are definable. Here we survey the types used in a range of theories.

Atomic data types

\mathbf{N} is a type of natural numbers; there could instead be a type *int* of integers. Numbers may be represented as $0, 1, 2, \dots$, or as $0, S(0), S(S(0)), \dots$. There can be finite types $\mathbf{0}, \mathbf{1}, \mathbf{2}, \dots$ of zero, one, two, ... elements. It may be sensible to have a type \mathbf{E} of all expressions.

Atomic propositional types

These types represent atomic propositions via the principle of propositions as types (discussed in section 2.2.3). In Nuprl, $a = b$ in A is a type which represents the

assertion $a = b \in A$. Some theories such as CC have no need for these types; others, such as the theory defined in chapter 3, have many atomic propositional types.

Functions

The type $A \rightarrow B$ is the space of all functions from A to B . The function space may be generalized to a dependent function space $x:A \rightarrow B$ (x may occur freely in B): the range type B may depend on the value the function was applied on. These types are also called Π -types. An informal example is

$$f \in n:\mathbb{N} \rightarrow \overbrace{\mathbb{N} \times \dots \times \mathbb{N}}^n.$$

$f(m)$ is an m -ary tuple of natural numbers.

Products

The type $A \times B$ is the product of types A and B ; its members are pairs $\langle a, b \rangle$, where $a \in A$ and $b \in B$. The product type may be generalized to a dependent product $x:A \times B$: the type B depends on the inhabitant of the type A . Such types are also known as Σ -types. This introduces a left-to-right dependency in the product: first we find an element $a \in A$, and then we find $b \in B[a/x]$ ¹ to give $\langle a, b \rangle \in x:A \times B$.

Unions

The type $A + B$ is the disjoint union of types A and B . The members of the type are injected into the right or left side: $inl(a) \in A + B$ implies $a \in A$, and $inr(b) \in A + B$ implies $b \in B$.

Subtypes

$\{x:A \mid P_x\}$ denotes those elements of A which have property P . Since P can be an arbitrary proposition, this gives a rich notion of subtype.

¹this denotes substituting free occurrences of x in B with a

Recursive and infinite types

$\text{rec}(t.A_t)$ denotes the type of least membership which is a solution to the type equation $t = A_t$, provided a solution exists. Many more type structures are expressible in the presence of this constructor. Lists of elements of some type B are expressed as $\text{rec}(t.\mathbf{1} + B \times t)$; constructive ordinals may be expressed by the type $\text{rec}(t.\mathbf{1} + \mathbb{N} \rightarrow t)$. The W-type of CMCP may be expressed by the recursive type $\text{rec}(t.x:A \times B \rightarrow t)$.

$\text{inf}(t.A_t)$ denotes the greatest type which is a solution to the type equation $t = A_t$. For instance, $\text{inf}(t.\mathbb{N} \times t)$ represents a stream of natural numbers. Recursive and infinite types for type theory have been developed by Mendler [Men87, CM85, MPC86]; PX also has recursive types [HN87].

Large types

Large types have types as inhabitants. In Nuprl and CMCP there is an infinite hierarchy of large types U_1, U_2, \dots ; U_1 has as members all of those types closed under the type constructors of the sort like those mentioned above. U_2 is then constructed by closing over all of these types plus the large type U_1 . In this fashion a hierarchy is constructed. This hierarchy is *predicative* because new objects are always defined in terms of existing objects. In CC and TS, there are *impredicative* large types. In CC, there are two large types *Prop* and *Type*, with $\text{Prop} \in \text{Type}$. *Prop* is impredicative: $X:\text{Prop} \rightarrow X \in \text{Prop}$, so types in *Prop* may quantify over *Prop* itself. Because of this, it is impossible to inductively define those types that are in *Prop*.

2.2.2 Equality

Two notions of equality found in type theories are worth contrasting. The types may come with a notion of equality on their inhabitants, and there may be some global notion of equality of computations.

In Nuprl and CMCP, to understand what a type is means not only to know its inhabitants, but also to know when two inhabitants are equal. The assertion $a \in A$ is replaced with a partial equivalence relation² $a = a' \in A$, meaning a and a' are equal members of the type A . For example, function equality is defined to be extensional equality: given two functions $f, f' \in A \rightarrow B$,

$$f = f' \in A \rightarrow B \text{ if } (f(a) = f'(a) \in B \text{ for all } a \in A).$$

²Partial equivalence relations are symmetric and transitive, but not necessarily reflexive.

In other type theories types may not come with a notion of equality on their inhabitants; instead there may be a notion of equality on computations. Such equalities are here called *computational equalities*; we use the notation $a \cong b$ to mean a and b are computationally equal. In many theories, α - β -interconvertibility is the notion of computational equality used. Other notions of equality are then defined by logical means.

In Nuprl and CMCP there is also a notion of computational equality; however, it is not a relation that is defined in the theory because it is incorporated into the type equalities:

$$\text{if } a \cong a' \text{ and } a \in A \text{ and } a' \in A, \text{ then } a = a' \in A.$$

All reasoning about computational equivalence is thus carried out using typed equalities. In CMCP, computational equality is taken to be outermost irreducibility. However, this equality is hard to work with because it is not a congruence: if $b \cong b'$ and $a[b/x] \in A$, it doesn't necessarily follow that $a[b'/x] \in A$. In Nuprl, computational equality is *type-free*: we need not type computational expressions to show they are related. This is a distinguishing feature of Nuprl. Type-free reasoning is further discussed in section 2.2.5. In the current Nuprl theory, the computation rules define a rich class of computationally related expressions, the *closed subterm variants* [All87b]. This is a much more general notion of computational equivalence than that of CMCP. Howe has recently extended the computational equality to an even more general notion [How88b], which is also a congruence.

2.2.3 The propositions-as-types principle

Logic may be carried out in type theories by translating propositions into types. If P is a proposition, let P^* represent the type it is translated to; then, P is true just when the type P^* has any inhabitant. This principle has been used by Curry, Howard, and DeBruijn, but it was Scott [Sco70] who first formulated a theory where the connection was semantic and not proof-theoretic; in his theory no extra axioms were needed to carry out constructive logic.

For P to be constructively true is to have a *construction* which *validates* P . For example, to constructively prove

$$\forall n. \text{prime}(n) \vee \neg \text{prime}(n)$$

is to have a decision procedure for whether n is prime or not. Constructive logic arises naturally from the principle of propositions as types: given a proposition P , if

P^* is inhabited, the object is a construction to validate P . This is evident if we look at how the individual propositions are customarily translated into type theory.

The translation is defined as follows:

$$\begin{aligned}
(A \Rightarrow B)^* &\stackrel{\text{def}}{=} A^* \rightarrow B^* \\
(A \ \&\ B)^* &\stackrel{\text{def}}{=} A^* \times B^* \\
(A \vee B)^* &\stackrel{\text{def}}{=} A^* + B^* \\
(\forall x:A. B)^* &\stackrel{\text{def}}{=} x:A^* \rightarrow B^* \\
(\exists x:A. B)^* &\stackrel{\text{def}}{=} x:A^* \times B^* \\
\text{false}^* &\stackrel{\text{def}}{=} \mathbf{0}
\end{aligned}$$

We can sketch how the inhabiting object of the type P^* can act as the validation for P .

CASE $A \ \&\ B$ true: This means $\langle a, b \rangle \in A^* \times B^*$, so $a \in A^*$ and $b \in B^*$, meaning a and b are validations for A and B . Thus, a and b together validate $A \ \&\ B$.

CASE $A \vee B$ true: This means $A^* + B^*$ is inhabited, so either $\text{inl}(a) \in A^* + B^*$ or $\text{inr}(b) \in A^* + B^*$. In the first case, a validates A ; in the second case, b validates B . In both cases, $A \vee B$ is validated.

CASE $A \Rightarrow B$ true: This means $f \in A^* \rightarrow B^*$; so, given any validation a for A , $f(a)$ gives a validation for B . f thus validates $A \Rightarrow B$.

CASE $\exists x:A. P_x$ true: This means $\langle a, p \rangle \in x:A \times P_x^*$; a is thus a member of A , and $p \in P^*[a/x]$, so p validates $P[a/x]$; $\exists x:A. P_x$ is thus validated.

CASE $\forall x:A. P_x$ true: This means $f \in x:A \rightarrow P_x^*$; thus, for all $a \in A$, $f(a) \in P^*[a/x]$, meaning $f(a)$ validates $P[a/x]$; $\forall x:A. P_x$ is thus validated.

An atomic proposition is translated into an atomic type which is inhabited just when the proposition is true. For example, in Nuprl $a = b \in A$ is translated to the type $a = b$ in A ; this type is inhabited by the element *axiom* just when $a = b \in A$. Type universes can also be viewed as proposition universes. $A \rightarrow U_1$ is a type of predicates on A ; higher universes allow higher-order predicates to be defined.

2.2.4 Type theory as a foundation for mathematics

Type theory may be viewed as a foundation for mathematical and computational reasoning. The rules are truths of the theory, but the class of rules is not fixed, just as we can never accept that we have completely described what the truths are. We allow for the addition of new types and computations and new principles for reasoning as they are discovered, so the theory is open-ended.

2.2.5 Computational and mathematical interpretations

Type theories are commonly interpreted in two ways; we call them *computational* and *mathematical* interpretations. In a computational interpretation, type-free computation is taken as the starting point. There is a collection of expressions and a method for computing on them; computations are collected into types based on their computational behavior.

The types could also be taken as the starting point. A type is viewed as a collection of *mathematical* objects; computations are just different notations for the same object. If a theory has a mathematical interpretation, every expression has a mathematical meaning, and the meaning of larger objects is defined in terms of the meanings of smaller objects. This compositionality of meaning is a desirable property for a theory to have. However, because all computationally related expressions are interpreted as the same object, there can be no understanding of the actual computation *process*. Thus, theories that have mathematical interpretations must also have computational interpretations if the theory is to be viewed as saying anything about computation. TLC, CC, and TS have both computational and mathematical interpretations: the Tait computability method for strong normalization gives a computational interpretation [Ste72, CH88], and there are numerous mathematical interpretations using domains and categories such as [Gir86, Ehr88]. However, it is possible for a type theory to have no mathematical interpretation; Nuprl is one such theory. The Nuprl computation rules define a type-free notion of equality on computations, and type-free reasoning can only be carried out in a theory where the meaning of computation is basic. Let us call such theories *explicitly computational* theories. It is easier to reason about computations in such a setting because there is no necessity to first type the computations. PX is an explicitly computational theory, and the type theory which we present in chapter 3 is an explicitly computational theory which takes full advantage of type-free reasoning methods.

2.3 Nuprl

Nuprl is a type theory developed at Cornell which has been implemented as a theorem-proving environment. Here we give a brief overview of Nuprl; for a more complete treatment, see [CAB⁺86]. Nuprl is an interactive theorem prover for type theory. The *Nuprl theory* is the underlying type theory, and the *Nuprl system* is the computer system which implements the theory.

2.3.1 The Nuprl theory

The Nuprl theory was designed to be a usable version of Martin-Löf's CMCP. The collection of types is richer than those of CMCP, the rules more powerful, and the presentation more comprehensible. All of the types discussed in section 2.2.1 (except the type of all expressions E and impredicative type universes) are types in Nuprl. Subtypes, recursive types, and infinite types for type theory were originally developed for Nuprl. As mentioned in section 2.2.2, Nuprl has a type-free notion of equality over computations which cuts down on the proof burden of type-checking; this equality is also much richer than the computational equality of CMCP. The rules are presented in goal-directed or *refinement-style* fashion: a rule is applied to a goal, and this gives subgoals which when proven validate the goal. Proofs are thus trees with nodes being goals and children of a node being its subgoals. The leaves of the tree are goals with no subgoals.

Using the propositions-as-types principle, to prove P we need to find an inhabitant of the type P^* . However, we often don't care what the inhabitant is, just so it exists. To this end, the rules are implemented with *extract forms*: The inhabiting objects is automatically synthesized. For example, for independent product, the rule

$$\begin{aligned} >> \langle a, b \rangle \in A \times B \\ >> a \in A \\ >> b \in B \end{aligned}$$

has extract form

$$\begin{aligned} >> A \times B \text{ (extract } \langle a, b \rangle) \\ >> A \text{ (extract } a) \\ >> B \text{ (extract } b). \end{aligned}$$

Inductively, when the proofs of A and B are complete, a and b can be extracted from those proofs; we may then form the pair $\langle a, b \rangle$ to extract from $A \times B$. Recall that $A \times B$ is the type which represents conjunction; this is confirmed by the fact that the above extract rule is the same as the and-introduction rule.

Given a proposition P , an inhabitant p of the type P^* is a program which validates P , so using this form of rule extracts the constructive content from a proof. Proving a proposition thus produces a program, so programs can be viewed as proofs [BC85]. In the predecessor of Nuprl, λ -PRL, programs are directly extracted from proofs of statements in constructive first-order logic [Bat79]. The refinement-style proof paradigm was originally developed in λ -PRL.

2.3.2 The Nuprl system

The Nuprl system is a computer environment implemented on Symbolics Lisp Machines in Zetalisp and on Sun workstations in Common Lisp. The system has features which make it a powerful tool for doing formal proofs. Objects such as theorems and definitions are kept in a library. A window system simplifies interaction; it is easy to “walk” through a proof step-by-step and to work on different parts of the proof in any order. A definition facility allows new notation to be defined. An evaluator for Nuprl expressions allows the programs extracted from proofs to be executed.

Nuprl has an automated metalanguage, the programming language ML. ML procedures may be applied to goals to automatically prove them or to make progress toward proving them; these procedures are called *tactics*. The use of automated metalanguage is central to theorem proving on machines, and ML is a programming language particularly well-suited to this task. The basic strategies of automated theorem proving such as backchaining and equality reasoning via congruence closure can be programmed as tactics; Howe has developed an extensive library of general-purpose tactics for use with Nuprl [How88a]. It is also often desirable to have tactics tailored for a particular domain of reasoning, for each new mathematical theory has new fairly trivial operations that general strategies will miss; numerous such libraries now exist.

A fair chunk of mathematics has been developed in Nuprl, including libraries of number theory [How86] and automata theory [Kre86]. Saddleback search and quicksort have both been proven correct in Nuprl [How88a]. Howe used Nuprl to show Girard’s paradox leads to a looping combinator, a result which was too cumbersome to be done by hand [How87, How88a]. Cleaveland has implemented CCS in Nuprl [Cle87], and Basin has given a constructive proof of Ramsey’s theorem in Nuprl [Bas88a].

Chapter 3

A partial object type theory

In this chapter we present a type theory for reasoning about partial objects. To show that a new treatment of partiality is necessary, we review how partial objects may currently be represented in type theory, and point out the weaknesses of these approaches. The key constituents of partial object type theory are outlined, and the theory is then presented in complete detail. Since in some aspects the theory differs from existing theories, examples of how concepts may be represented and proofs performed are given.

3.1 Partial objects in existing type theories

The types of type theory are *total types*: $\mathbb{N} \rightarrow \mathbb{N}$ is the type of total functions on the natural numbers; $\mathbb{N} \times \mathbb{N}$ is the type of pairs of natural numbers, with no provision for undefined elements. It is necessary to have types for total objects, but it is also important to have types for partial objects, which we call *partial types*. It is unreasonable to expect all functions to be expressible as total functions, because some total functions can be extremely difficult to prove total. Also, many procedures such as unbounded searches or non-well-founded recursions are inherently partial; we would like to think of these procedures as typable functions which we can reason about, even though they are not total.

In CMCP and CC, unbounded computation cannot be expressed. In such a setting, even total functions are more difficult to express than in a setting with unbounded operators [Blu67]. Nuprl is an explicitly computational type theory (see section 2.2.5 for a discussion of this issue), so computation is type-free. It is then pos-

sible to express unbounded recursion with the Y-combinator by taking fixed-points.

We can consider typing partial functions from A to B in Nuprl by defining them to be total functions with their domains restricted to those elements for which the function converges. A function f is then typed as the total function $f \in \{x:A \mid D(x)\} \rightarrow B$, where $D(a)$ is true just when $f(a)$ converges. The domain D can in fact be expressed for numerous functions using recursive types [CM85, CAB⁺86] because the inductive structure of the recursive calls can be defined as a recursive type. However, many functions, in particular higher-order ones, cannot have their domains expressed by any type D in the existing Nuprl theory. This formulation is also difficult to use, because before any function is applied it must be shown to terminate; this extra proof obligation is an unnecessary burden.

3.2 New principles for partial objects

Here we outline new principles for expressing and reasoning about partial objects which are the core of the partial object type theory that follows.

3.2.1 The partial type constructor

First, we propose adding a new type constructor, the partial type or bar type constructor \overline{A} . We wish to extend the (total) type A to admit diverging elements:

for all types A , $a \in \overline{A}$ iff (if a terminates, then $a \in A$).

The bar constructor can be thought of classically as adding an element \perp to the underlying type. The type $\mathbb{N} \rightarrow \overline{\mathbb{N}}$ is a type of partial functions on the natural numbers: if $f \in \mathbb{N} \rightarrow \overline{\mathbb{N}}$ and $n \in \mathbb{N}$, then $f(n) \in \mathbb{N}$, *provided* $f(n)$ terminates. One important property of this definition is that we may show objects in partial types are in total types if they terminate.

In standard programming languages all type constructors are partial. In a partial object type theory, some type constructors are partial and others are total, so it is possible to have types which have both partial and total aspects. The (lazy) programming language type $\mathbb{N} \times \mathbb{N}$ has as elements \perp , $\langle 1, 2 \rangle$, and also has elements $\langle \perp, 2 \rangle$ and $\langle \perp, \perp \rangle$. With partial types, it is possible to specify varying degrees of partiality. \perp and $\langle 1, 2 \rangle \in \overline{\mathbb{N} \times \mathbb{N}}$, but $\langle \perp, 2 \rangle$ and $\langle \perp, \perp \rangle \notin \overline{\mathbb{N} \times \mathbb{N}}$. On the other hand, $\perp \notin \overline{\mathbb{N} \times \mathbb{N}}$, but $\langle \perp, 2 \rangle$ and $\langle \perp, \perp \rangle \in \overline{\mathbb{N} \times \mathbb{N}}$. $\overline{\mathbb{N} \times \mathbb{N}}$ corresponds to the product of \mathbb{N} and \mathbb{N} in a lazy programming language. The partial type constructor is thus a

simple, natural and general notion for extending the types of type theory to allow for diverging inhabitants.

3.2.2 Forming partial objects

Having types for partial objects is a first step, but it doesn't necessarily mean that interesting partial functions will be typable. The most general notation for expressing unbounded computation is recursion, which may be viewed as computing by taking fixed-points of functionals. We introduce a general *fixed-point principle* for partial types to allow recursive procedures to be typed:

$$\text{if } f \in \overline{A} \rightarrow \overline{A}, \text{ then } \text{fix}(f) \in \overline{A}.$$

Letting \overline{A} be $\overline{\mathbb{N} \rightarrow \mathbb{N}}$, this principle may be used to type partial functions over the natural numbers. Unfortunately, because the type system is so expressive, this principle is not valid for all types (an example where it is not valid is given in section 4.4.1). This is analogous to what happens with the fixed-point induction principle, where not all predicates are admissible for induction. In section 5.2 this analogy is made more direct by showing how fixed-point induction may be carried out using the fixed-point principle.

3.2.3 Reasoning about partial objects

In type theory we are not satisfied with being able to just denote partial objects; we also need to be able to reason about them.

To begin with, it should be possible to reason cogently about termination: termination links partial objects to total objects. The theory has a predicate $t \downarrow$ which expresses the fact that t terminates. It is easier to reason about termination if it is also possible to reason about the evaluation process: the assertion $a \mapsto b$, which means a evaluates to b , accomplishes this. The assertion $a \succ b$ (read “ a induces b ”) means that in the process of computing a , b was in turn computed. If we wish to prove a terminates and know that

$$f \stackrel{\text{def}}{=} \text{if } a = 0 \text{ then } 1 \text{ else } f(b)$$

terminates, f induces the computation of a (that is how conditionals are evaluated), so a must also terminate. If a terminates, then the collection of computations induced by it form a well-founded ordering, for an infinitely regressing path would mean a

diverged. It is thus possible to derive an induction principle, *computational induction*, for reasoning inductively about computations. Computational induction is a general principle for reasoning about programs, and is a viable alternative to the fixed-point induction principle that is used in LCF and in other theories. Unlike fixed-point induction, no admissibility restrictions need be placed on predicates proven with computational induction.

3.3 A partial object type theory

We now define a type theory which incorporates these concepts. The theory is relatively simple in comparison to Nuprl or CMCP, designed more to be studied than to be used. However, it is not lacking in proof-theoretic strength, for we want to be sure these new concepts are sensible in a full theory. In appendix A, we propose an extension to the Nuprl theory which also incorporates these new concepts, but that theory is too large to serve as a good illustration. The theory we give now still owes more to Nuprl than any other: it is predicative and explicitly computational, and most of its basic type constructors are also found in Nuprl.

The computation system is of the same style as Nuprl, although the actual expressions are not the same. One extra feature it has is the ability to sequence the order in which expressions are evaluated, via $seq(a; x.b)$. Evaluation is the same outermost evaluation used in Nuprl. The theory has most of the basic types of Nuprl or CMCP, including natural numbers \mathbb{N} , dependent function space $x:A \rightarrow B$, dependent product $x:A \times B$, and a universe hierarchy U_1, U_2, \dots . However, there is no notion of equality built into the theory; all equalities are defined by the user as predicates. There is not even an interesting computational equality; in this sense the theory is unique amongst type theories, and it considerably simplifies the presentation. Since the logic is rich, it is possible to define the desired equalities, but reasoning about equality is more cumbersome than in Nuprl. To represent partial objects, the theory includes bar types \bar{A} and has a fixed-point principle to type them. New atomic types express the propositions $a \downarrow$ (*a halts*), $a \mapsto b$ (*a eval b*), and $a \succ b$ (*a ind b*). There is also a type of all expressions E , and a type $a \text{ in } A$ which expresses the proposition $a \in A$.

We now give the full presentation of the theory. The open-ended expression language is defined first, the forms of assertion about expressions such as typehood, membership, and evaluation are given, and the rules of the theory are presented.

3.4 The language of expressions

The *expressions* are the objects of discourse for the theory. The collection of expressions is open-ended, so we may add more expressions at some later date. Expressions are formed inductively by building new expressions from existing ones using *constructors*, of which there are three syntactic sorts: *type constructors*, *computation constructors*, and *data constructors*. These are defined simultaneously, so type constructors may have computation constructors as components, computation constructors may have data constructors as components, *et cetera*. The intermeshing of these three sorts enriches the notions of computation, type, and data.

DEFINITION 1 *The expressions of the language include variables (given in roman font), and are closed under the following constructors:*

The type constructors

$$\begin{aligned} &E, N, \\ &a \text{ is } b, a \text{ eval } b, a \text{ ind } b, a \text{ halts}, a \text{ in } A, \\ &\overline{A}, x:A \rightarrow B, x:A \times B, \\ &V, U_1, U_2, \dots \end{aligned}$$

the data constructors

$$0, 1, 2, \dots, \langle a, b \rangle, \lambda x.a$$

and the computation constructors

$$\begin{aligned} &pred(n), succ(n), if_zero(n; a; b), \\ &p.1, p.2, f(a), fix(f), seq(a; x.b) \end{aligned}$$

where a, b, f, n, p, A , and B denote arbitrary expressions, and x denotes a variable.

We know little about what is not an expression because the collection of expressions is open-ended, but the objects entailed by the above collection of constructors must certainly be expressions. *Type, data, and computation expressions* are those expressions whose outermost form is a type, data, or computation constructor, respectively. *Values* are those expressions which are either type expressions or data expressions; they constitute the results of computation. For a constructor to be a computation constructor, it must be known how to evaluate expressions built using the constructor. Computation expressions are sometimes called *computations*, and

data expressions called *data*; however, type expressions and types are different notions, for type expressions need to satisfy certain well-formedness conditions before they may be considered to be types.

For our metavariables, A, B, \dots, T and a, b, \dots, v range over expressions and X, Y, Z and w, x, y, z range over variables. Capital letters usually denote type expressions, n and m are expressions which usually denote natural numbers, p usually denotes a pair, f usually denotes a function, and v usually denotes a value, but this is an informal convention. U_α represents an arbitrary universe (α is a metavariable ranging over $1, 2, \dots$).

Notions of *bound* and *free* variables are the usual ones:

$\lambda x.b$	binds free occurrences of x in b
$seq(a; x.b)$	binds free occurrences of x in b
$x:A \rightarrow B$	binds free occurrences of x in B
$x:A \times B$	binds free occurrences of x in B

x is free in the expression x , and x remains free in any inductively constructed expression unless it is explicitly bound. An expression is *closed* if it contains no free variables.

$a[b_1, \dots, b_n/x_1, \dots, x_n]$ denotes the simultaneous substitution of distinct variables x_1, \dots, x_n with expressions b_1, \dots, b_n , renaming bound variables in the instance of capture.

3.5 Assertions and hypothetical assertions

There are six forms of assertion used in the theory, with the following meanings:

$a \in A$	means A is a type with inhabitant (member) a
A type	means A is a type
$a =_\alpha b$	means a and b are identical modulo α -conversion
$a \mapsto b$	means a evaluates to b
$a \succ b$	means the evaluation of a induces the evaluation of b
$a \downarrow$	means a terminates.

The last four assertions are called *type-free assertions* because they are assertions which involve no notion of type. These assertions are defined only for *closed* expressions a , b , and A ; meanings for open expressions are defined in terms of the meanings of closed expressions.

There is an additional property which must hold of the assertion $a \in A$: it must respect computational equivalence. Two expressions are *computationally equivalent*, notated $a \cong b$, if they have the same value, if any. Formally,

DEFINITION 2 $a \cong b$ iff $(a \mapsto v \text{ iff } b \mapsto v)$.

$a \in A$ must respect computational equivalence, so $a \in A$ and $a \cong b$ implies $b \in A$, and $a \in A$ and $A \cong B$ implies $a \in B$. The assertions $a =_\alpha b$, $a \mapsto b$, and $a \succ b$ don't respect computational equivalence, because they assert intensional properties of computations (intensional reasoning is discussed further in section 3.11.2).

The principle of propositions-as-types, discussed in 2.2.3, will be used to carry out constructive reasoning in this theory, so P is shown to be true by showing the corresponding type P^* to be inhabited. To make atomic assertions in the theory, the six forms of assertion are reflected into the theory as *atomic propositional types*:

a in A inhabited	iff $a \in A$
A in U_α inhabited (for some α)	iff A type
a is b inhabited	iff $a =_\alpha b$
a eval b inhabited	iff $a \mapsto b$
a ind b inhabited	iff $a \succ b$
a halts inhabited	iff $a \downarrow$.

Atomic propositional types are inhabited just when the proposition they represent is true. It doesn't matter much what these types have as inhabitants, so we let them be inhabited by every expression if they are inhabited by any expression. Types are expressions which inhabit some universe, so this allows us to assert an expression is a type without having a type of all types. Since all assertions may be phrased in terms of some type being inhabited, we can use $a \in A$ as a universal form of assertion, and treat all of the other assertions as types. This means that in the rules below, we need only consider proving assertions of the form $a \in A$.

A *hypothetical assertion* is an assertion made under the assumption that some other assertions are true. We only need to express hypothetical assertions of the form $a \in A$. The sequent-style hypothetical assertion used in the rules is:

$$x_1:A_1, \dots, x_n:A_n \mid > b \in B.$$

Informally, this means the *consequent* $b \in B$ is true under the *assumptions* $x_1 \in A_1, \dots, x_n \in A_n$.

DEFINITION 3 A sequent is of the form

$$x_1:A_1, \dots, x_n:A_n \mid > b \in B,$$

where x_1, \dots, x_n are distinct, each A_i has at most $x_1 \dots x_{i-1}$ free, and b and B have at most x_1, \dots, x_n free.

DEFINITION 4 *The sequent*

$$x_1:A_1, \dots, x_n:A_n \mid > b \in B$$

is true just when

for arbitrary closed expressions a_1, \dots, a_n , if

$$a_1 \in A_1 \text{ and}$$

$$a_2 \in A_2[a_1/x_1] \text{ and}$$

$$\vdots$$

$$\text{and } a_n \in A_n[a_1, \dots, a_{n-1}/x_1, \dots, x_{n-1}],$$

then

$$b[a_1, \dots, a_n/x_1, \dots, x_n] \in B[a_1, \dots, a_n/x_1, \dots, x_n].$$

3.6 Rules

The rules show how true sequents may be inferred from other true sequents. Rules are written in the form

$$\begin{array}{l} \langle \text{rule name} \rangle \quad x_1:A_1, \dots, x_n:A_n \mid > d \in D \\ \quad \quad \quad x_1:A_1, \dots, x_n:A_n, y_1:B_1, \dots, y_n:B_{n'} \mid > d' \in D' \\ \quad \quad \quad x_1:A_1, \dots, x_n:A_n, z_1:C_1, \dots, z_n:C_{n''} \mid > d'' \in D'' \\ \quad \quad \quad \vdots \end{array}$$

Rules assert that the top sequent (the *goal*) is true if the sequents directly below it (the *subgoals*) are true. This style is a top-down or *refinement-style* presentation. The assumptions in the goal are also assumptions for the subgoals, and going from a goal to its subgoals the assumption list is monotonically increasing.

A *proof* is a finite-depth tree with sequents for nodes; for all nodes in the tree, there exists a rule which has as goal the node and as subgoals the children of the node; leaves of the tree are thus instances of rules with no subgoals. The sequent at the top of the proof tree is true, because inductively truth is inherited upward in the tree. *Incomplete proofs*, as opposed to complete proofs, are proofs with some leaves that are not proven.

3.6.1 Rule conventions

Sequents are presented in an abbreviated form. The only assumptions listed in a goal are those relevant to the rule at hand, and could in fact occur anywhere in the list of assumptions. It is implicitly assumed that subgoals inherit all of the assumptions of the goal, unless explicitly stated otherwise. Assumptions shown in subgoals are new assumptions which are placed at the end of the assumption list. The atomic propositional types are inhabited by all expressions when true, so there is little interest in what actually inhabits the type; their rules will thus be abbreviated. For example, the rule for evaluation of fixed-points,

$$\begin{array}{l} \text{[eval fix]} \quad | > c \in \text{fix}(f) \text{ eval } v \\ \quad \quad \quad | > c' \in f(\text{fix}(f)) \text{ eval } v \end{array}$$

is abbreviated

$$\begin{array}{l} \text{[eval fix]} \quad | > \text{fix}(f) \text{ eval } v \\ \quad \quad \quad | > f(\text{fix}(f)) \text{ eval } v, \end{array}$$

which means the inhabiting objects c and c' can implicitly be anything in the abbreviated version. When atomic propositional types appear as assumptions they will not be labeled with a variable. Metavariables in rules range over arbitrary expressions, as long as the sequents thus defined are well-formed. We now present the rules, grouped together by type.

3.6.2 Universes

There is an infinite hierarchy of cumulative type universes U_1, U_2, \dots . All basic types are collected in the large type U_1 . With U_1 now defined, U_2 may be defined as the large type which has as a member U_1 and types constructed from it, *et cetera*. To assert that an expression is a type, one asserts that it inhabits one of these universes, i.e. $A \in U_\alpha$, where α is a positive integer.

There is a single *admissible universe* V which is a subcollection of U_1 . The types in V are those U_1 types for which the fixed-point principle discussed above is true. Some types are not admissible, so these types cannot be used to form fixed-points and are not in V . This matter will be discussed in section 3.6.14, where the rules for forming inhabitants of V are also given. The rules for showing what types inhabit the U_α will be given with the individual types, for that is also how types are shown to be well-formed.

Each universe is a member of higher universes:

[U form] $|> U_\alpha \in U_\beta$
 where $\alpha < \beta$.

The hierarchy is cumulative:

[U cumulativity] $|> A \in U_\alpha$
 $|> A \in U_\beta$
 where $\alpha > \beta$.

3.6.3 Is

$a =_\alpha b$ asserts that a and b are α -variants. The type $a \text{ is } b$ expresses this assertion:

[is form] $|> a \text{ is } b \in U_\alpha$

is is reflexive, symmetric, and transitive:

[is reflex] $|> a \text{ is } a'$
 where a and a' are α -variants.

[is sym] $b \text{ is } a \mid > a \text{ is } b$

[is trans] $a \text{ is } b, b \text{ is } c \mid > a \text{ is } c$

If a and b are different yet we have a hypothesis $a \text{ is } b$, any fact follows:

[is contradiction] $a \text{ is } b \mid > c \in C$

where a is not an α -variant of b for any substitution of free variables in a and b .

Identical expressions may be substituted for one another:

[is subst] $a \text{ is } b \mid > c[a/x] \in C[a/x]$
 $|> c[b/x] \in C[b/x]$

For all a and b , $a =_\alpha b$ is either true or false by inspection:

[is decision] $|> c \in C$
 $a \text{ is } b \mid > c \in C$
 $(a \text{ is } b) \rightarrow (0 \text{ is } 1) \mid > c \in C$

3.6.4 Evaluation

All expressions may be evaluated; $a \mapsto v$ asserts that a has value v . Results of evaluation are always values (data or type expressions). Each expression constructor has as part of its description a method for evaluating expressions built from it; evaluation always proceeds by carrying out the evaluation method for the outermost constructor of the expression. Values evaluate to themselves, so they may thus be thought of as having an evaluation method which is the identity function.

The assertion $a \mapsto b$ is expressed by the type $a \text{ eval } v$.

[eval form] $\quad \quad \quad |> a \text{ eval } v \in U_\alpha$

Values evaluate to themselves:

[eval value] $\quad \quad \quad |> v \text{ eval } v$

where v is a value.

If a evaluates to v , then v must be a value:

[eval idemp] $\quad \quad \quad a \text{ eval } v \quad |> v \text{ eval } v$

There is at most one value for a given computation:

[eval unique] $\quad \quad \quad a \text{ eval } v, a \text{ eval } v' \quad |> v \text{ is } v'$

We now give the rules which describe how each computation constructor is evaluated. The [eval <constructor>] rules show how the computation is done, and the [eval <constructor> unique] rules confirm that this is the only way to carry it out. For function application,

[eval app] $\quad \quad \quad |> f(a) \text{ eval } v$
 $\quad \quad \quad \quad \quad \quad |> f \text{ eval } \lambda x.b$
 $\quad \quad \quad \quad \quad \quad |> b[a/x] \text{ eval } v$

[eval app unique] $\quad \quad \quad f(a) \text{ eval } v \quad |> b \in B$
 $\quad \quad \quad \quad \quad \quad v':E \rightarrow E, f \text{ eval } v', v'(a) \text{ eval } v \quad |> b \in B$

For projection of components from pairs,

$$\begin{array}{l}
[\text{eval proj left}] \quad |> p.1 \text{ eval } v \\
\quad \quad \quad |> p \text{ eval } \langle a, b \rangle \\
\quad \quad \quad |> a \text{ eval } v
\end{array}$$

There is a symmetrical rule [eval proj right] for the right projection $p.2$.

$$\begin{array}{l}
[\text{eval proj left unique}] \quad p.1 \text{ eval } v \quad |> b \in B \\
\quad \quad \quad x:\mathbb{E}, y:\mathbb{E}, p \text{ eval } \langle x, y \rangle, x \text{ eval } v \quad |> b \in B
\end{array}$$

There is a symmetrical rule [eval proj right unique] for the right projection $p.2$.

For the operations on natural numbers,

$$\begin{array}{l}
[\text{eval succ}] \quad |> \text{succ}(n) \text{ eval } m \\
\quad \quad \quad |> n \text{ eval } m'
\end{array}$$

where m and m' are natural numbers, and m is one more than m' . There is a similar rule [eval pred] for $\text{pred}(n)$.

$$\begin{array}{l}
[\text{eval succ unique}] \quad \text{succ}(n) \text{ eval } m \quad |> b \in B \\
\quad \quad \quad n \text{ eval } m' \quad |> b \in B
\end{array}$$

where m and m' are natural numbers, and m is one more than m' . There is a similar rule [eval pred unique] for $\text{pred}(n)$.

$$\begin{array}{l}
[\text{eval if_zero true}] \quad |> \text{if_zero}(n; a; b) \text{ eval } v \\
\quad \quad \quad |> n \text{ eval } 0 \\
\quad \quad \quad |> a \text{ eval } v
\end{array}$$

$$\begin{array}{l}
[\text{eval if_zero false}] \quad |> \text{if_zero}(n; a; b) \text{ eval } v \\
\quad \quad \quad |> n \in \mathbb{N} \\
\quad \quad \quad n \text{ eval } 0 \quad |> 0 \text{ is } 1 \\
\quad \quad \quad |> b \text{ eval } v
\end{array}$$

$$\begin{array}{l}
[\text{eval if_zero true unique}] \\
\quad \quad \quad \text{if_zero}(n; a; b) \text{ eval } v, n \text{ eval } 0 \quad |> b \in B \\
\quad \quad \quad a \text{ eval } v \quad |> b \in B
\end{array}$$

There is a symmetrical rule [eval if_zero false unique] for the case that n is nonzero.

For fixed-points,

$$\begin{array}{l} \text{[eval fix]} \\ |> \text{fix}(f) \text{ eval } v \\ |> f(\text{fix}(f)) \text{ eval } v \end{array}$$

$$\begin{array}{l} \text{[eval fix unique]} \\ \text{fix}(f) \text{ eval } v \ |> b \in B \\ f(\text{fix}(f)) \text{ eval } v \ |> b \in B \end{array}$$

For sequencing computations,

$$\begin{array}{l} \text{[eval seq]} \\ |> \text{seq}(a; x.b) \text{ eval } v \\ |> a \text{ eval } v' \\ |> b[v'/x] \text{ eval } v \end{array}$$

$$\begin{array}{l} \text{[eval seq unique]} \\ \text{seq}(a; x.c) \text{ eval } v \ |> b \in B \\ v':E, a \text{ eval } v', c[v'/x] \text{ eval } v \ |> b \in B \end{array}$$

3.6.5 Equivalence

Recall that $a \in A$ means a inhabits type A , and in addition that for all b computationally equivalent to a and B computationally equivalent to A , $b \in B$. The following rules reflect this fact:

$$\begin{array}{l} \text{[equiv exp]} \\ |> a \in A \\ |> a' \in A \\ v:E, a' \text{ eval } v \ |> a \text{ eval } v \\ v:E, a \text{ eval } v \ |> a' \text{ eval } v \end{array}$$

$$\begin{array}{l} \text{[equiv type]} \\ |> a \in A \\ |> a \in A' \\ V:E, A' \text{ eval } V \ |> A \text{ eval } V \\ V:E, A \text{ eval } V \ |> A' \text{ eval } V \end{array}$$

$$\begin{array}{l} \text{[equiv hyp]} \\ x_1:A_1, \dots, x_n:A_n \ |> b \in B \\ x_1:A_1, \dots, x_i:A'_i, \dots, x_n:A_n \ |> b \in B \\ V:E, A'_i \text{ eval } V \ |> A_i \text{ eval } V \\ V:E, A_i \text{ eval } V \ |> A'_i \text{ eval } V \end{array}$$

3.6.6 Inducement

$a \succ b$ asserts that the evaluation of a in turn entails the evaluation of b . For example, $(\lambda x. \text{if_zero}(x; 0; 1))(pred(1)) \succ pred(1)$, for $pred(1)$ must be computed to arrive at the result. $(\lambda x. 1)(pred(1)) \succ pred(1)$ is false, because the argument to the function is never evaluated. The assertion $a \succ b$ is sensible because, as can be seen from the rules for evaluation, we know how to evaluate expressions and we know that there is no other way it can be done. The most important use of inducement is if a computation terminates, all induced computations form a well-founded ordering which admits induction. The computational induction rule (see section 3.6.15) is such an induction scheme.

$a \succ b$ is expressed with the type $a \text{ ind } b$:

[ind form] $\quad |> a \text{ ind } b \in U_\alpha$

ind is transitive:

[ind trans] $\quad a \text{ ind } b, b \text{ ind } c \quad |> a \text{ ind } c$

The value of a computation is induced by the computation:

[ind eval] $\quad a \text{ eval } b \quad |> a \text{ ind } b$
 $\quad a \text{ is } b \quad |> 0 \text{ is } 1$

Rules are now given to show what is directly induced by each computation expression.

For function application,

[ind app] $\quad |> f(a) \text{ ind } b[a/x]$
 $\quad |> f \text{ eval } \lambda x. b$

[ind app arg] $\quad |> f(a) \text{ ind } f$

For projection of components from pairs,

[ind proj left] $\quad |> p.1 \text{ ind } a$
 $\quad |> p \text{ eval } \langle a, b \rangle$

There is a symmetrical rule [ind proj right] for the right projection $p.2$.

[ind proj left arg] $\quad |> p.1 \text{ ind } p$

There is a symmetrical rule [ind proj right arg] for the right projection $p.2$.

For natural numbers,

[ind succ] $\quad |> \text{succ}(n) \text{ ind } n$

There is a symmetrical rule [ind pred] for pred .

[ind if_zero true] $\quad |> \text{if_zero}(n; a; b) \text{ ind } a$
 $\quad |> n \text{ eval } 0$

[ind if_zero false] $\quad |> \text{if_zero}(n; a; b) \text{ ind } b$
 $\quad |> n \in \mathbb{N}$
 $\quad n \text{ eval } 0 \quad |> 0 \text{ is } 1$

[ind if_zero arg] $\quad |> \text{if_zero}(n; a; b) \text{ ind } n$

For fixed-points,

[ind fix] $\quad |> \text{fix}(f) \text{ ind } f(\text{fix}(f))$

For sequencing,

[ind seq] $\quad |> \text{seq}(a; x.b) \text{ ind } b[v'/x]$
 $\quad |> a \text{ eval } v'$

[ind seq arg] $\quad |> \text{seq}(a; x.b) \text{ ind } a$

3.6.7 Termination

A computation terminates, written $a \downarrow$, if it evaluates to some value. The type $a \text{ halts}$ expresses this proposition:

[terminate form] $\quad |> a \text{ halts} \in U_\alpha$

[terminate intro] $\quad |> a \text{ halts}$
 $\quad |> a \text{ eval } v$

[terminate elim] $\quad a \text{ halts} \quad |> b \in B$
 $\quad v:E, a \text{ eval } v \quad |> b \in B$

If a terminates, any computation induced by a also terminates:

$$\begin{array}{l} \text{[terminate ind]} \quad | > a \text{ halts} \\ \quad \quad \quad \quad | > a' \text{ ind } a \\ \quad \quad \quad \quad | > a' \text{ halts} \end{array}$$

For some types, all of their inhabitants must converge; such types are called *total types*.

$$\begin{array}{l} \text{[terminate total]} \quad | > a \text{ halts} \\ \quad \quad \quad \quad | > a \in A \end{array}$$

Where A is, by inspection, of the form \mathbb{N} or $x:B \rightarrow C$ or $x:B \times C$ or U_α or V .

3.6.8 Membership

The expression $a \text{ in } A$ is the type which expresses the assertion $a \in A$. For $a \text{ in } A$ to be a type, A must also be a type to guarantee the type has been sensibly formed.

$$\begin{array}{l} \text{[member form]} \quad | > (a \text{ in } A) \in U_\alpha \\ \quad \quad \quad \quad | > A \in U_\alpha \end{array}$$

$$\begin{array}{l} \text{[member intro]} \quad | > a \text{ in } A \\ \quad \quad \quad \quad | > a \in A \end{array}$$

$$\begin{array}{l} \text{[member elim]} \quad | > a \in A \\ \quad \quad \quad \quad | > a \text{ in } A \end{array}$$

3.6.9 Expression

E is the type of all (closed) expressions. The theory has as a basis a collection of expressions, and we may make the theory more usable by admitting it as a type, because quantifying over E allows abstract reasoning about untyped computations to be performed.

$$\text{[E form]} \quad | > E \in U_\alpha$$

All expressions a are in this type:

$$\text{[E intro]} \quad | > a \in E$$

3.6.10 Natural numbers

\mathbb{N} is the type of natural numbers:

[N form] $|> \mathbb{N} \in \mathbb{U}_\alpha$

[N intro] $|> n \in \mathbb{N}$

Where n is one of $0, 1, 2, \dots$

The basic functions on natural numbers are successor, predecessor, and a conditional check for zero.

[N succ] $|> \text{succ}(n) \in \mathbb{N}$
 $|> n \in \mathbb{N}$

There is a symmetric rule [N pred] for $\text{pred}(n)$.

[N if_zero] $|> \text{if_zero}(n; a; b) \in A$
 $|> n \in \mathbb{N}$
 $n \text{ eval } 0 \quad |> a \in A$
 $x:(n \text{ eval } 0 \rightarrow 0 \text{ is } 1) \quad |> b \in A$

The computational induction principle (section 3.6.15) may be used to prove inductive facts about natural numbers once we know a simple function is total:

[N ind func] $|> \text{fix}(\lambda f. \lambda x. \text{if_zero}(x; 0; f(\text{pred}(x)))) \in \mathbb{N} \rightarrow \mathbb{N}$

To prove a property $P(n)$ for some natural number n , perform computational induction on the above function applied to n . This procedure is described in detail in section 3.12.2.

3.6.11 Dependent function space

The function space of this theory is a dependent function space $x:A \rightarrow B$: the range type may depend on the value the function was applied on. This type is also known as a Π -type.

For $x:A \rightarrow B$ to be a type, all expressions which it is defined in terms of must be types, so A must be a type, and $B[a/x]$ must be a type for all expressions $a \in A$:

$$\begin{array}{l}
[\text{func form}] \quad |> \ x:A \rightarrow B \in U_\alpha \\
\quad \quad \quad |> \ A \in U_\alpha \\
\quad \quad \quad x:A \ |> \ B \in U_\alpha
\end{array}$$

The inhabitants of dependent function spaces are λ -expressions:

$$\begin{array}{l}
[\text{func intro}] \quad |> \ \lambda x.b \in x:A \rightarrow B \\
\quad \quad \quad x:A \ |> \ b \in B \\
\quad \quad \quad |> \ A \in U_\alpha
\end{array}$$

The second subgoal assures $x:A \rightarrow B$ is a type. We know $B[a/x]$ is a type for all $a \in A$ from the first subgoal, but the first subgoal does not insure that A will be a type, necessitating the addition of the second subgoal.

If we don't know a function is a λ -expression, we can't apply the above rule; but, the fact that it inhabits *any* function space means it is a function:

$$\begin{array}{l}
[\text{func lam}] \quad |> \ f \in x:A \rightarrow B \\
\quad \quad \quad |> \ f \in y:C \rightarrow D \\
\quad \quad \quad x:A \ |> \ f(x) \in B \\
\quad \quad \quad |> \ A \in U_\alpha
\end{array}$$

To apply functions,

$$\begin{array}{l}
[\text{func elim}] \quad |> \ c \in C \\
\quad \quad \quad |> \ a \in A \\
\quad \quad \quad |> \ f \in x:A \rightarrow B \\
\quad \quad \quad y:B[a/x], f(a) \text{ is } y \ |> \ c \in C
\end{array}$$

Assuming $x:A \rightarrow B$ is a type, it follows from [func intro] and [func elim] that

$$\lambda x.b \in x:A \rightarrow B \text{ iff for all } a \in A, b[a/x] \in B[a/x].$$

3.6.12 Dependent product space

The type of pairs is the dependent product space $x:A \times B$; the right type may depend on the inhabitant of the left type. Such types are also called Σ -types.

The formation rule is the same as for dependent functions:

$$\begin{array}{l}
[\text{prod form}] \quad |> \ x:A \times B \in U_\alpha \\
\quad \quad \quad |> \ A \in U_\alpha \\
\quad \quad \quad x:A \ |> \ B \in U_\alpha
\end{array}$$

The inhabitants of product types compute to products $\langle a, b \rangle$:

$$\begin{array}{l} \text{[prod intro]} \quad | > \langle a, b \rangle \in x:A \times B \\ \quad \quad \quad | > a \in A \\ \quad \quad \quad | > b \in B[a/x] \\ \quad \quad \quad x:A \quad | > B \in U_\alpha \end{array}$$

The third subgoal assures that $x:A \times B$ is a sensible type. The following rule allows products to be used:

$$\begin{array}{l} \text{[prod elim]} \quad | > c \in C \\ \quad \quad \quad | > p \in x:A \times B \\ \quad \quad \quad x:A, y:B, p \text{ eval } \langle x, y \rangle \quad | > c \in C \end{array}$$

From the preceding two rules we have that if $x:A \times B$ is a type,

$$\langle a, b \rangle \in x:A \times B \text{ iff } a \in A \text{ and } b \in B[a/x].$$

3.6.13 Partial type

The bar type \overline{A} extends a type to have diverging inhabitants. Since this type is a novel notion, its use is discussed in more detail in section 3.11.1. For \overline{A} to be a type, A must be a type under the assumption that A converges. So if A diverges, \overline{A} is a type.

$$\begin{array}{l} \text{[bar form]} \quad | > \overline{A} \in U_\alpha \\ \quad \quad \quad A \text{ halts} \quad | > A \in U_\alpha \end{array}$$

The following two rules define what it means to inhabit a bar type:

$$\begin{array}{l} \text{[bar intro]} \quad | > a \in \overline{A} \\ \quad \quad \quad a \text{ halts} \quad | > a \in A \\ \quad \quad \quad | > A \in \overline{U}_\alpha \end{array}$$

$$\begin{array}{l} \text{[bar elim]} \quad | > a \in A \\ \quad \quad \quad | > a \text{ halts} \\ \quad \quad \quad | > a \in \overline{A} \end{array}$$

From these two rules, we can see that if \overline{A} is a type,

$$a \in \overline{A} \text{ iff } (a \downarrow \text{ implies } a \in A).$$

3.6.14 Fixed-points

Recursion may be expressed by taking fixed-points, so by typing fixed-points we may type recursive programs. The fixed-point principle types fixed-points:

$$\begin{array}{l}
 [\text{fix}] \quad |> \text{fix}(f) \in \overline{A} \\
 \quad \quad \quad |> f \in \overline{A} \rightarrow \overline{A} \\
 \quad \quad \quad |> A \in V
 \end{array}$$

This rule defines new inhabitants of \overline{A} in terms of functions over \overline{A} , which is circular; this circularity can lead to paradoxes, so this rule is not valid for all U_α types, but is valid for the subcollection of *admissible types* which inhabit the universe V . V is in fact *defined* to have as inhabitants those types for which the fixed-point rule is valid. Higher V universes are also in principle possible, but they are not a part of this theory. The collection of admissible types is undecidable, so we here present a collection of rules which allows some of the admissible types to be proven to inhabit V . In these rules we can only define independent product types, but none of the other type constructors need be restricted. In section 4.4, questions of admissibility are explored in more depth.

The rules for V are:

$$[\text{V form}] \quad |> V \in U_2$$

V is a subcollection of U_1 :

$$\begin{array}{l}
 [\text{V sub}] \quad |> A \in U_1 \\
 \quad \quad \quad |> A \in V
 \end{array}$$

For forming admissible types there are the following rules, which parallel the U_α formation rules with the exception of dependent products.

$$[\text{is form adm}] \quad |> a \text{ is } b \in V$$

$$[\text{eval form adm}] \quad |> a \text{ eval } v \in V$$

$$[\text{ind form adm}] \quad |> a \text{ ind } b \in V$$

$$[\text{terminate form adm}] \quad |> a \text{ halts } \in V$$

$$\begin{array}{l}
 [\text{member form adm}] \quad |> (a \text{ in } A) \in V \\
 \quad \quad \quad |> A \in V
 \end{array}$$

[E form adm]	$ > E \in V$
[N form adm]	$ > N \in V$
[func form adm]	$ > x:A \rightarrow B \in V$ $ > A \in V$ $x:A \ > B \in V$
[prod form adm]	$ > x:A \times B \in V$ $ > A \in V$ $ > B \in V$

Where x does not occur free in B .

[bar form adm]	$ > \overline{A} \in V$ $A \text{ halts} \ > A \in V$
----------------	---

3.6.15 Computational induction

Computational induction is a principle for reasoning about arbitrary computations. Given the fact that a computation a terminates, the collection of computations induced by it form a well-founded ordering. The following rule defines an induction principle over this ordering:

[comp ind]	$a \text{ halts} \ > \text{fix}(\lambda h.\lambda a''.\lambda v.e[a/a'])(a)(0) \in P[a/x]$ $a':E, a' \text{ halts}, h:(a'':A \rightarrow a' \text{ ind } a'' \rightarrow P[a''/x])$ $ > e \in P[a'/x]$
------------	---

In section 3.12, we show how this principle may be used in proofs.

3.6.16 Miscellaneous

The cut rule allows arbitrary facts to be added to the hypothesis list:

[cut]	$ > a \in A$ $ > b \in B$ $x:B \ > a \in A$
-------	--

The hyp rule allows a goal to be proven if it is a hypothesis:

[hyp] $x_1:A_1, \dots, x_i:A_i, \dots, x_n:A_n \mid > x_i \in A_i$

For the atomic propositional types, there is a more general rule: we can show anything inhabits the type:

[hyp prop] $x_1:A_1, \dots, B, \dots, x_n:A_n \mid > B$
 where B is an atomic propositional type.

α -conversion is respected by the assertion $a \in A$, so to prove a sequent it is sufficient to prove an α -converted form of it: this is the rule [alpha].

3.7 Reasoning in the theory

Here we give some idea of how this theory may be used for reasoning about mathematical and computational objects. Constructive logical reasoning is carried out using the principle of propositions as types. We give examples of how the rules are used to prove propositions, concentrating on those types and rules not found in Nuprl such as partial types and the fixed-point rule, the type-free atomic propositions, and computational induction.

3.8 Logical reasoning

To carry out logical reasoning in the theory, propositions P are mapped to types P^* such that P is true if and only if P^* is inhabited. The mapping is given in section 2.2.3, and in section 3.5 the types which represent the atomic forms of assertion are given. For matters of readability, we let the traditional logical notation stand for types:

DEFINITION 5 *The logical expressions are defined as follows:*

$$\begin{aligned}
 A \Rightarrow B &\stackrel{\text{def}}{=} A \rightarrow B \\
 A \& B &\stackrel{\text{def}}{=} A \times B \\
 A \vee B &\stackrel{\text{def}}{=} A + B \\
 \neg A &\stackrel{\text{def}}{=} A \rightarrow (0 \text{ is } 1) \\
 \forall x:A. B &\stackrel{\text{def}}{=} x:A \rightarrow B \\
 \exists x:A. B &\stackrel{\text{def}}{=} x:A \times B
 \end{aligned}$$

$\forall x.P$ and $\exists x.P$ abbreviate $\forall x:E.P$ and $\exists x:E.P$, respectively. When treating propositions as types, the actual inhabitant of the type P^* is not very important; what matters most is that something inhabits the type. There is a method of proof which we use that takes advantage of this fact, called *extract-style proof*.

3.9 Extract-style proof

If we wish to prove a proposition P by finding an inhabitant of the type P^* , at the start of the proof we don't know what the inhabiting object of P^* will be. For example, suppose the top of our proof was

$$|> ? \in \forall x:N. \exists y:N. x < y \ \& \ \text{prime}(y).$$

The inhabiting object “?” is unknown and will be derived recursively. For this proof, we wish to first apply the [func intro] rule, and by looking at that rule it can be seen that ? must be of the form $\lambda x.??$:

$$\begin{aligned} |> \lambda x. ?? \in \forall x:N. \exists y:N. x < y \ \& \ \text{prime}(y) \\ \quad x:N \quad |> ?? \in \exists y:N. x < y \ \& \ \text{prime}(y) \\ \quad |> (\forall x:N. \exists y:N. x < y \ \& \ \text{prime}(y)) \in U_1 \end{aligned}$$

In Nuprl, rules are given in *extract form*: the inhabiting object of a type is automatically derived (it is *extracted* from the proof). For each rule there is a procedure associated with the rule which when given inhabitants of each consequent of the subgoals constructs an inhabitant of the goal's consequent type; extraction is discussed in section 2.3.1. In this presentation we don't wish to double the number of rules by adding extract forms and use instead an informal method which postpones what exactly inhabits a type until later in the proof. The symbols \exists_i will be used as metavariables ranging over expressions for unknown inhabitants of types. For the above rule application, we would write

$$\begin{aligned} |> \exists_1 \in \forall x:N. \exists y:N. x < y \ \& \ \text{prime}(y) \\ \quad x:N \quad |> \exists_2 \in \exists y:N. x < y \ \& \ \text{prime}(y) \\ \quad |> \forall x:N. \exists y:N. x < y \ \& \ \text{prime}(y) \in U_1 \end{aligned}$$

plus we must know that \exists_1 is $\lambda x.\exists_2$. This fact is evident from the rule, and it will usually not be given. If for each rule in a proof we always (implicitly) know how to find the inhabitant of the goal given inhabitants of the subgoals, and if the proof is complete, we can find an inhabitant of the top.

3.9.1 An extract-style proof

A simple fact is proven extract style here: we prove for arbitrary predicates $P \in \mathbf{N} \rightarrow \mathbf{U}_\alpha$ that

$$\neg(\exists n:\mathbf{N}. P(n)) \Rightarrow \forall n:\mathbf{N}. \neg P(n).$$

The proof will be complete except that formation subgoals (proving types are well formed) will be left off. We wish to prove something inhabits this type:

$$|> \exists_1 \in \neg(\exists n:\mathbf{N}. P(n)) \Rightarrow \forall n:\mathbf{N}. \neg P(n).$$

Applying [func intro] gives the subgoal

$$x:\neg(\exists n:\mathbf{N}. P(n)) \quad |> \exists_2 \in \forall n:\mathbf{N}. \neg P(n)$$

(we implicitly know that \exists_1 is $\lambda x.\exists_2$). Applying [func intro] again gives

$$n:\mathbf{N} \quad |> \exists_3 \in \neg P(n)$$

(the hypothesis $x:\neg(\exists n:\mathbf{N}. P(n))$ is not shown but it is implicitly still there). Recalling that $\neg P(n)$ is defined to be $P(n) \rightarrow (0 \text{ is } 1)$, we apply [func intro] one more time:

$$z:P(n) \quad |> 0 \text{ is } 1$$

Since the goal is an atomic propositional type, there is no need to keep track of the inhabitant, for it may be anything. Applying [func elim] to x gives subgoals

$$\begin{array}{l} |> x \in \neg(\exists n:\mathbf{N}. P(n)) \\ |> \exists_4 \in \exists n:\mathbf{N}. P(n) \\ v:0 \text{ is } 1, x(\exists_4) \text{ is } v \quad |> 0 \text{ is } 1 \end{array}$$

the first and third subgoals follow trivially by [hyp] and [hyp prop]. The second subgoal is true because we may instantiate the existential quantifier with n via [prod intro], giving subgoals

$$\begin{array}{l} |> n \in \mathbf{N} \\ |> \exists_5 \in P(n) \end{array}$$

Both of which follow by hypothesis.

QED.

Since the proof is now complete, we may extract what the \exists_i are, starting at the bottom: \exists_5 is z , \exists_4 is $\langle n, z \rangle$, \exists_3 is $\lambda z.0$, \exists_2 is $\lambda n.\lambda z.0$, and \exists_1 is $\lambda x.\lambda n.\lambda z.0$.

3.10 Defining Unions

The disjoint union type $A + B$ is used to represent logical disjunction. This type is built in to Nuprl, but here it is a defined notion.

A *derived rule* is an incomplete proof schema; the goal is the top of the proof, and the subgoals are the unproved leaves in the incomplete proof. It is a schema because there could be metavariables occurring in the proof. Derived rules can be used like regular rules because all they are doing is abbreviating part of a proof. Disjoint union $A + B$ is defined and derived rules for it are given.

DEFINITION 6 $A + B \stackrel{\text{def}}{=} n:N \times \text{if_zero}(n; A; B)$, where n is a variable which does not occur in A or B .

The reader may verify that the following are derived rules for $A + B$:

[union formation] $|> A + B \in U_\alpha$
 $|> A \in U_\alpha$
 $|> B \in U_\alpha$

[union intro left] $|> \langle 0, a \rangle \in A + B$
 $|> a \in A$

[union intro right] $|> \langle 1, b \rangle \in A + B$
 $|> b \in B$

[union elim] $x:A + B \quad |> c \in C$
 $y:A, \forall v. (x.2 \text{ eval } v \iff y \text{ eval } v) \quad |> c \in C$
 $z:B, \forall v. (x.2 \text{ eval } v \iff z \text{ eval } v) \quad |> c \in C$

With the above derived rules, there are enough tools now to reason using disjunction.

3.11 Using types and rules

The natural number type N , dependent types $x:A \rightarrow B$ and $x:A \times B$ and universes U_α are well known types, and their use will not be discussed here; see [CAB⁺86] for examples. The partial types \bar{A} and the propositional types $a \text{ is } b$, $a \text{ eval } b$, $a \text{ ind } b$, and $a \text{ halts}$ are new, so some further explanation of their use is in order.

3.11.1 Using partial types

The operator $\overline{}$ adds to any type A the possibility that its inhabitants might not evaluate to anything. This operator may be applied to all types, and this leads to types with both partial and total aspects. For example, consider the (total) function space $\mathbb{N} \rightarrow \mathbb{N}$; there are seven possible new types that may be formed by using the partial type operator, and each has a slightly different meaning (multiple bars over a single type are redundant). They are $\overline{\mathbb{N} \rightarrow \mathbb{N}}$, $\mathbb{N} \rightarrow \overline{\mathbb{N}}$, $\overline{\mathbb{N}} \rightarrow \mathbb{N}$, $\overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$, $\overline{\overline{\mathbb{N}}} \rightarrow \mathbb{N}$, $\overline{\mathbb{N}} \rightarrow \overline{\overline{\mathbb{N}}}$, and $\overline{\overline{\mathbb{N}}} \rightarrow \overline{\overline{\mathbb{N}}}$. $\overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$ is the type of expressions which if they converge are total functions on natural numbers. $\mathbb{N} \rightarrow \overline{\mathbb{N}}$ is closer to what we would consider a partial function: the argument of the function always converges, but the result might not converge. This corresponds to the notion of partiality of Turing machines. The type $\overline{\mathbb{N}} \rightarrow \mathbb{N}$ is not very useful because the argument may diverge but the function cannot. This means the function could not evaluate the argument. The type $\overline{\overline{\mathbb{N}}} \rightarrow \overline{\overline{\mathbb{N}}}$ is the type of partial functions used in lazy functional programming languages: functions themselves may diverge because they may be values of functions that might not converge, the argument to the function can diverge, and the result may diverge.

The bar operator may also be applied to types which represent propositions, giving types such as $\forall n:\mathbb{N}. \overline{\exists m:\mathbb{N}. P(m,n)}$, which are called *partial propositions*. Partial propositions are discussed in detail in section 5.3.

Potentially diverging computations are typed using the fixed-point rule [fix]. To show how simple functions may be typed we will prove

$$|> \text{fix}(\lambda f.\lambda x.\text{if_zero}(x; 6; f(\text{succ}(x)))) \in \mathbb{N} \rightarrow \overline{\mathbb{N}}.$$

Let $\text{fix}(\cdot)$ abbreviate the fix expression above. The [fix] rule allows us to show the function is in the type $\overline{\mathbb{N} \rightarrow \overline{\mathbb{N}}}$, so we [seq] in $\text{fix}(\cdot)$ in $\mathbb{N} \rightarrow \overline{\mathbb{N}}$, giving two subgoals

- (1) $|> \text{fix}(\cdot)$ in $\overline{\mathbb{N} \rightarrow \overline{\mathbb{N}}}$
- (2) $\text{fix}(\cdot)$ in $\mathbb{N} \rightarrow \overline{\mathbb{N}}$ $|> \text{fix}(\cdot) \in \mathbb{N} \rightarrow \overline{\mathbb{N}}$

(2) follows because $\text{fix}(\cdot)$ halts after one fix unrolling and one application, so we may concentrate on (1). Applying [member intro] and then [fix] gives subgoals

- (3) $|> \lambda f.\lambda x.\text{if_zero}(x; 6; f(\text{succ}(x))) \in \overline{\mathbb{N} \rightarrow \overline{\mathbb{N}}} \rightarrow \overline{\mathbb{N} \rightarrow \overline{\mathbb{N}}}$
- (4) $|> \overline{\mathbb{N} \rightarrow \overline{\mathbb{N}}} \in V,$

whose proofs are straightforward.

This method may be used to type complex functions over a wide range of types, but it must be kept in mind that not all types are admissible.

3.11.2 Untyped reasoning

The termination predicate $a \downarrow$ is useful in conjunction with bar types, for it provides the connection between partial types and total types: if $a \in \overline{A}$ and $a \downarrow$, then by [bar elim] a inhabits the type A .

The assertions $a =_{\alpha} b$, $a \mapsto b$, and $a \succ b$ are *intensional* assertions because they assert properties of computations besides their values. Most formal theories for reasoning about computation have no intensional assertions, and their presence here distinguishes this theory from others. Recall from section 3.5 that the assertion $a \in A$ respects computational equivalence \cong . The intensional assertions do *not* respect computation equivalence, however. For example, $(\lambda x.0)(0) \mapsto 0$ and $0 \cong \text{pred}(\text{succ}(0))$, but $(\lambda x.0)(0) \mapsto \text{pred}(\text{succ}(0))$ is false. Since assertions may be expressed in the logic via the atomic propositional types, there are then types (like $(\lambda x.0)(0) \text{ eval } 0$) which have subexpressions that do not respect computational equivalence. For this reason there is no general principle of substitutivity in the theory.

The type $a \text{ is } b$ is useful because often two different expression abstractions refer to the same thing; for example, if x and $\text{fix}(f)(a)$ refer to the same thing, we may equate the two by asserting $x \text{ is } \text{fix}(f)(a)$.

The evaluation mechanism is a fundamental part of a computational type theory because the notion of computation is basic. With the type $a \text{ eval } b$, we may reason about evaluation directly in the theory.

The type $a \text{ ind } b$ has two major uses in the theory: for one, with the [terminate ind] rule we can prove computations terminate if they are induced by a terminating computation. But most importantly it may be used as a structure to perform induction on, discussed in the next section.

The type of all expressions E gives us the ability to carry out type-free reasoning. For example, $\exists x, y : E. a \text{ eval } \langle x, y \rangle$ asserts a evaluates to a pair. As another example, $x : E \times (x \text{ eval } 0 + x \text{ eval } 1)$ is a two-element type.

3.12 Reasoning by induction

If an expression a terminates we may perform induction on the well-founded part of the induces relation via [comp ind]. This rule, found in section 3.6.15, may be rephrased in English as:

To prove some property $P(a)$ by induction given terminating a , let a' be an arbitrary terminating expression for which all a'' induced by it have

property $P(a'')$ true. If $P(a')$ can then be proven, $P(a)$ is true.

Since a' is an arbitrary terminating expression, we cannot prove any a'' will be induced by it. If this rule is to be usable, the property $P(a)$ must thus include as assumptions restrictions on the structure of a . One example is

$$P(a) = (\exists b:\mathbb{N}. a \text{ is } \text{fix}(f)(b)) \Rightarrow P'(a)$$

where f and P' are arbitrary expressions. This will be an induction over those expressions of the form $\text{fix}(f)(b)$ for some b .

We will now give an example of how to use this restriction in a proof about a function using [comp ind].

3.12.1 An example of computational induction

Consider the function

$$g = \lambda f.\lambda x.\text{if_zero}(x; 6; f(\text{succ}(x))).$$

To show

$$(1) \quad |> \exists_1 \in \forall e:\mathbb{E}. \text{fix}(g)(e) \text{ halts} \Rightarrow \text{fix}(g)(e) \text{ eval } 6,$$

apply [func intro] twice to give

$$(2) \quad e:\mathbb{E}, \text{fix}(g)(e) \text{ halts} \quad |> \text{fix}(g)(e) \text{ eval } 6.$$

Before applying [comp ind], the goal needs to be weakened by adding assumptions about the structure of the computation $\text{fix}(g)(e)$ as described above. Thus, [seq] the statement

$$\exists_2 \in (\exists b:\mathbb{N}. \text{fix}(g)(e) \text{ is } \text{fix}(g)(b)) \Rightarrow \text{fix}(g)(e) \text{ eval } 6$$

which gives subgoals

$$(3) \quad |> \exists_2 \in (\exists b:\mathbb{N}. \text{fix}(g)(e) \text{ is } \text{fix}(g)(b)) \Rightarrow \text{fix}(g)(e) \text{ eval } 6$$

$$(4) \quad z:((\exists b:\mathbb{N}. \text{fix}(g)(e) \text{ is } \text{fix}(g)(b)) \Rightarrow \text{fix}(g)(e) \text{ eval } 6) \quad |> \text{fix}(g)(e) \text{ eval } 6.$$

(4) follows by letting b in hypothesis z be a (apply [prod elim] and [func elim]).

(3) is proven by [comp ind] on $\text{fix}(g)(e)$, which we know terminates by assumption.

Applying [comp ind] to (3) gives the subgoal

$$(5) \quad \begin{array}{l} a':E, a' \text{ halts}, h:(\forall a'':E. a' \text{ ind } a'' \Rightarrow (\exists b':N. a'' \text{ is } \text{fix}(g)(b')) \Rightarrow a'' \text{ eval } 6) \\ |> \exists_3 \in (\exists b:N. a' \text{ is } \text{fix}(g)(b)) \Rightarrow a' \text{ eval } 6 \end{array}$$

By [func intro] we get

$$(6) \quad y:(\exists b:N. a' \text{ is } \text{fix}(g)(b)) \quad |> \quad a' \text{ eval } 6.$$

Eliminating y via [prod elim] gives

$$(7) \quad b:E, a' \text{ is } \text{fix}(g)(b) \quad |> \quad a' \text{ eval } 6.$$

Applying [is subst] to the consequent gives

$$(8) \quad |> \text{fix}(g)(b) \text{ eval } 6;$$

by [eval app] and [eval fix], this amounts to showing

$$(9) \quad |> \text{if_zero}(b; 6; \text{fix}(g)(\text{succ}(b))) \text{ eval } 6,$$

which by case analysis on b using [N if_zero] gives subgoals

$$(10) \quad b \text{ eval } 0 \quad |> \quad \text{if_zero}(b; 6; \text{fix}(g)(\text{succ}(b))) \text{ eval } 6$$

$$(11) \quad z:(b \text{ eval } 0 \rightarrow 0 \text{ is } 1) \quad |> \quad \text{if_zero}(b; 6; \text{fix}(g)(\text{succ}(b))) \text{ eval } 6.$$

We can prove (10) directly by computing the $\text{if_zero}()$ expression; (11) reduces via [eval if_zero false] to showing

$$(12) \quad |> \text{fix}(g)(\text{succ}(b)) \text{ eval } 6,$$

which may be proven with the induction hypothesis: apply [function elim] to hypothesis h at the value $\text{fix}(g)(\text{succ}(b))$ to give

$$(13) \quad |> h \in \forall a'':E. a' \text{ ind } a'' \Rightarrow \exists b':N. a'' \text{ is } \text{fix}(g)(b') \Rightarrow a'' \text{ eval } 6$$

$$(14) \quad |> \text{fix}(g)(\text{succ}(b)) \in E$$

$$(15) \quad \begin{array}{l} h':(a' \text{ ind } \text{fix}(g)(\text{succ}(b)) \Rightarrow \exists b':N. \text{fix}(g)(\text{succ}(b)) \text{ is } \text{fix}(g)(b') \Rightarrow \\ \text{fix}(g)(\text{succ}(b)) \text{ eval } 6) \quad |> \quad \text{fix}(g)(\text{succ}(b)) \text{ eval } 6. \end{array}$$

Subgoals (13) and (14) follow directly. To prove (15), doing two more [func elim]'s on the induction hypothesis h' gives subgoals

$$(16) \quad |> a' \text{ ind } \text{fix}(g)(\text{succ}(b))$$

$$(17) \quad |> \exists_4 \in \exists b':N. \text{fix}(g)(\text{succ}(b)) \text{ is } \text{fix}(g)(b')$$

$$(18) \quad \text{fix}(g)(\text{succ}(b)) \text{ eval } 6 \quad |> \quad \text{fix}(g)(\text{succ}(b)) \text{ eval } 6.$$

(17) follows by letting \exists_4 be $\langle succ(b), 0 \rangle$, and (18) is proved by [hyp prop]; all that remains to show is (16). Recalling from (7) that a' is $fix(g)(b)$, we may substitute this fact into (16) to give the subgoal

$$(19) \quad |> fix(g)(b) \text{ ind } fix(g)(succ(b))$$

which follows by a chain of [ind] rules:

$$fix(g)(b) \text{ ind } if_zero(b; 6; fix(g)(succ(b)) \text{ ind } fix(g)(succ(b))).$$

QED.

3.12.2 Induction on natural numbers

Using computational induction, a principle of induction on the natural numbers may be derived. To prove property $P(n)$, perform computational induction on the computation of $F(n)$, where

$$F \stackrel{\text{def}}{=} fix(\lambda f. \lambda x. if_zero(x; 0; f(pred(x)))).$$

F is a function whose computation is isomorphic to \mathbb{N} , so induction over this computation is equivalent to induction over \mathbb{N} . The [N ind func] rule guarantees this function is total, so $F(n) \downarrow$. The derived rule is

$$\begin{array}{l} n:\mathbb{N} \quad |> fix(\lambda h. \lambda n'. \lambda c. if_zero(n'; p_1; p_2))(n)(0) \in P(n) \\ n':\mathbb{N}, n' \text{ eval } 0 \quad |> p_1 \in P(n') \\ n':\mathbb{N}, x:\neg(n' \text{ eval } 0), h:T, y:P(pred(b)), y \text{ is } h(F(pred(b)))(0)(pred(b))(0) \\ \quad |> p_2 \in P(n') \end{array}$$

The type T of h in the second subgoal is irrelevant. The verification of this fact is left to the reader (perform computational induction on $\forall n':\mathbb{N}. (F(n) \text{ is } F(n') \Rightarrow P(n'))$).

3.13 Defining equalities

There is no notion of equivalence in the theory except the simple notion of computational equivalence; all other notions of equality are *defined*. For example, two inhabitants of the \mathbb{N} type are defined to be equal when they have the same value. To define equalities on arbitrary types, we use a convention.

DEFINITION 7 *Let a type with equality be a dependent pair*

$$A:U_\alpha \times E_A:(A \rightarrow A \rightarrow U_\alpha) \times \text{“}E_A \text{ is a p.e.r.}\text{”}$$

where “ E_A is a p.e.r.” is a type which expresses that fact that E_A is a partial equivalence relation, i.e. it is symmetric and transitive over A .

A partial equivalence is used because we may not wish to define the equality over all elements of A . Types with equality are thus pairs of the form $\langle A, \langle E_A, (\text{evidence that } E_A \text{ is a p.e.r.}) \rangle \rangle$. The evidence inhabits the type that states that E is a p.e.r., and its actual content is irrelevant.

Given such a pair, define

DEFINITION 8 $a = a'$ in $A \stackrel{\text{def}}{=} E_A(a)(a')$.

Since equality is a p.e.r., $a \in A$ does not imply $a = a'$ in A is inhabited. For natural numbers, the equality would be

$$\langle \mathbb{N}, \lambda n. \lambda n'. \langle (\exists m:\mathbb{N}. m \text{ eval } m \ \& \ n' \text{ eval } m), (\text{evidence}) \rangle \rangle.$$

Given two types with equalities $\langle A, E_A, \langle \text{evidence}_A \rangle \rangle$ $\langle B, E_B, \langle \text{evidence}_B \rangle \rangle$, the extensional (independent) function equality would be

$$\langle A \rightarrow B, \langle \lambda f, \lambda f'. \forall a, a': A. a = a' \text{ in } A \Rightarrow f(a) = f(a') \text{ in } B, (\text{evidence}) \rangle \rangle$$

The proof burden when reasoning about equalities is greater here than in theories with built-in equality, but we may define whatever equalities we like, including ones that are not definable in Nuprl.

Chapter 4

Semantics of partial object type theory

In this chapter we give semantics for the theory of chapter 3. We define relations which interpret the assertions in such a way that the rules are valid. The expressions are interpreted as themselves, so it could be called a term model construction. One result of this labor is the intuitionistic consistency of the theory: *0 is 1* cannot be proved inhabited.

The method used here is the non-type-theoretic semantics that has been developed by Allen [All87b, All87a]. This method can in turn be viewed as an elaboration of the notion of computable term found in Tait's strong normalization proof of the typed λ -calculus. Beeson [Bee82] has a semantics which also gives the consistency of CMCP, but it is of little use as a tool for understanding type theory. Allen's semantics, on the other hand, gives insight into type theory; this semantics has in fact proved useful in the development of partial object type theory. Mendler [Men87] has developed non-type-theoretic semantics for recursive types based on Allen's method, and presents the results in a set-theoretic framework. Here we follow Allen and carry out the construction in an extensional theory of relations.

This chapter parallels the previous one in the development of the notions of type theory except that here we take a closed view of the theory: the class of expressions and rules is fixed, and interpretations for the assertions are once and for all defined. Justification of the rules is straightforward except for the fixed-point rule, which involves quite a bit of extra work to show the types in V to be admissible. This proof first entails showing several facts about the computational behavior of fixed-points $fix(f)$; the types in V are then proved admissible by induction on their definition.

4.1 The expressions

The expressions we are interpreting are those defined in the previous chapter and no more:

DEFINITION 9 *The expressions are defined as the least solution of the following definition:*

The variables

a, b, c, \dots

the type constructors

$E, N,$
 $a \text{ is } b, a \text{ eval } b, a \text{ ind } b, a \text{ halts}, a \text{ in } A,$
 $\overline{A}, x:A \rightarrow B, x:A \times B,$
 U_1, U_2, \dots, V

the data constructors

$0, 1, 2, \dots, \langle a, b \rangle, \lambda x.a$

and the computation constructors

$pred(n), succ(n), if_zero(n; a; b),$
 $p.1, p.2, f(a), fix(f), seq(a; x.b)$

where $a, b, f, n, p, A,$ and B are expressions, and x is a variable.

Conventions on the use of variables, notions of bound and free variables, α -convertibility, and the notion of substitution are the same as in the previous chapter.

4.2 The type-free assertions

The four forms of type-free assertion, $a =_{\alpha} b$, $a \mapsto b$, $a \succ b$, and $a \downarrow$ are herein defined. They are defined for closed expressions only, because open expressions are interpreted in terms of closed ones.

α -equivalence

DEFINITION 10 $a =_\alpha b$ iff a and b are α -variants.

The notion of α -variant is the usual one, and the details will not be given here. This relation has a few obvious properties:

LEMMA 11 $a =_\alpha b$ is reflexive, symmetric, and transitive.

Evaluation

The evaluation relation is defined inductively as the following least fixed-point:

DEFINITION 12

$a \mapsto v$ iff

case a is a value:	$v =_\alpha a$
a is $f(c)$:	$f \mapsto \lambda x.b$ and $b[c/x] \mapsto v$
a is $p.1$:	$p \mapsto \langle a, b \rangle$ and $a \mapsto v$
a is $p.2$:	$p \mapsto \langle a, b \rangle$ and $b \mapsto v$
a is $\text{succ}(n)$:	$n \mapsto n'$, where v is one more than n'
a is $\text{pred}(n)$:	$n \mapsto n'$, where v is one less than n'
a is $\text{if_zero}(n; c; d)$:	$n \mapsto n'$, n' is $0, 1, 2, \dots$, and case n is 0 : $c \mapsto v$ otherwise: $d \mapsto v$
a is $\text{fix}(f)$:	$f(\text{fix}(f)) \mapsto v$
a is $\text{seq}(b; x.c)$:	$b \mapsto v'$ and $c[v'/x] \mapsto v$.

We can make the following simple observations:

LEMMA 13 if v is a value then $v \mapsto v$.

LEMMA 14 If $a \mapsto v$ then $v \mapsto v$.

PROOF. By induction on the definition of evaluation.

QED.

LEMMA 15 If $a \mapsto v$ and $a \mapsto v'$ then $v =_\alpha v'$.

PROOF. By induction on the definition of evaluation.

QED.

Computational equivalence, $a \cong b$, is defined as in the previous chapter. Given some a , there is a partial recursive function which returns v just when $a \mapsto v$, so \mapsto is a sensible notion of evaluation.

Termination

Terminating computations are those computations that have a value.

DEFINITION 16 $a \downarrow$ iff there exists v such that $a \mapsto v$.

Inducement

Inducement is defined in terms of evaluation, because the notion of inducement is a property of evaluation. First we define *direct inducement*, $a \succ_1 b$. The expressions that are directly induced by some computation a may be read off of the definition of evaluation:

DEFINITION 17 $a \succ_1 b$ iff

<i>case a is a value:</i>	<i>false</i>
a is $f(c)$:	$b =_\alpha f$ or $f \mapsto \lambda x.d$ and $b =_\alpha d[c/x]$
a is $p.1$:	$b =_\alpha p$ or $p \mapsto \langle c, d \rangle$ and $b =_\alpha c$
a is $p.2$:	$b =_\alpha p$ or $p \mapsto \langle c, d \rangle$ and $b =_\alpha d$
a is $\text{if_zero}(n; c; d)$:	$b =_\alpha n$ or $n \mapsto m$ and m is 0 and $b =_\alpha c$ or $m =_\alpha 1, 2, \dots$ and $b =_\alpha d$
a is $\text{succ}(n)$:	$b =_\alpha n$
a is $\text{pred}(n)$:	$b =_\alpha n$
a is $\text{fix}(f)$:	$b =_\alpha f(\text{fix}(f))$

The inducement relation is the transitive closure of direct inducement.

DEFINITION 18 $a \succ_n b$ iff $a \succ_1 a'$ and $a' \succ_{n-1} b$.

DEFINITION 19 $a \succ b$ iff $a \succ_n b$ for some n .

From these definitions, it is easy to see

LEMMA 20 \succ is transitive.

There is a useful connection between evaluation and inducement:

LEMMA 21 If $a \mapsto b$ and a is a computation, then $a \succ b$.

PROOF. This follows by straightforward induction on the definition of evaluation.
QED.

Computations induced by terminating computations themselves must terminate:

LEMMA 22 If $a \downarrow$ and $a \succ b$, then $b \downarrow$.

PROOF. Suppose $a \downarrow$ and $a \succ b$; $a \succ b$ means $a \succ_n b$ for some n . By induction on n , we show $a \downarrow$ and $a \succ_n b$ implies $b \downarrow$.

CASE $n = 1$: Given $a \downarrow$ and $a \succ_1 b$, show $b \downarrow$. An inspection of the definition of $a \succ_1 b$ shows that for each case, if $a \downarrow$, b also must terminate.

CASE $n > 1$: Given the inductive assumption that for arbitrary a and b , $a \downarrow$ and $a \succ_n b$ implies $b \downarrow$ and the assumptions $a \downarrow$ and $a \succ_{n+1} b$, show $b \downarrow$. By the definition of inducement, there is some a' such that $a \succ_1 a' \succ_n b$; by the base case argument above, a' must also terminate; by the induction hypothesis, this means $b \downarrow$.

QED.

An induction principle can be derived for \succ : if an expression terminates, the collection of expressions induced by it must be well-founded over \succ , giving an ordering over which to perform induction. First we define the well-founded part of \succ :

DEFINITION 23 $a \downarrow$ (read “ a is founded”) is a relation on expressions which is the least property of expression such that

$$a \downarrow \text{ iff for all } b, a \succ b \text{ implies } b \downarrow.$$

The well-founded expressions should at least include the terminating expressions, because this would mean to show an expression is founded it would then suffice to show it terminates.

LEMMA 24 For all a , $a \downarrow$ implies $a \downarrow$.

PROOF. Pick arbitrary a such that $a \mapsto v$; a is shown to be founded by induction on the definition of evaluation. Pick arbitrary a' and v' such that $a' \mapsto v'$; inductively assume that for all a'' and v'' , $a'' \mapsto v''$ implies $a'' \downarrow$, and show $a' \downarrow$.

CASE a' is a value: All values induce nothing, so they are founded.

CASE a' is $f(c)$: Then $f \mapsto \lambda x.d$ and $d[c/x] \mapsto v'$. To show $f(c)\Downarrow$ is to show that for all b induced by $f(c)$, $b\Downarrow$. If $f(c) \succ b$, either $f(c) \succ_1 f \succ b$, or $f(c) \succ_1 d[c/x] \succ b$. By the inductive hypothesis, f and $d[c/x]$ are both founded, so b must be founded by the definition of foundedness. The rest of the cases are very similar.

QED.

It is now possible to prove a principle of induction on induced computations is sound. This principle directly justifies [comp ind].

THEOREM 25 *Given some arbitrary predicate on expressions $P(a)$, if*

$$a\Downarrow \text{ and } \forall a'. a'\Downarrow \text{ and } (\forall a''. a' \succ a'' \text{ implies } P(a'')) \text{ implies } P(a'),$$

then $P(a)$ holds.

PROOF. Arbitrary a such that $a\Downarrow$; show $P(a)$ given

$$(1) \quad \forall a'. a'\Downarrow \text{ and } (\forall a''. a' \succ a'' \text{ implies } P(a'')) \text{ implies } P(a').$$

$a\Downarrow$ by lemma 24, so we may proceed by induction on the definition of $a\Downarrow$. We wish to show the more general result

$$(2) \quad a\Downarrow \text{ implies } P(a).$$

Arbitrary a' ; inductively assume

$$(3) \quad \forall a''. a' \succ a'' \text{ implies } a''\Downarrow \text{ implies } P(a''),$$

show $a'\Downarrow$ implies $P(a')$. Suppose $a'\Downarrow$; $P(a')$ follows directly from (1) and (3).

QED.

4.3 Defining the types and their inhabitants

Given the collection of expressions and type-free assertions that have now been defined, we may proceed to define the types and their inhabitants. This entails making one large inductive definition which constructs all of the types. It is not obvious that the types maybe built inductively, so a monotonic operator is defined which has as a fixed-point the desired type system; see [All87b, All87a] for a more thorough analysis of the technique used.

DEFINITION 26 *A type interpretation τ is a 3-place relation, written*

$(T \text{ Type with } \epsilon, \gamma)_\tau,$

where T is an expression, ϵ is a one-place relation, and γ is a truth condition.

The intended meaning of the above assertion is: T is a type with its members specified by ϵ , and γ is true just when T is a candidate for admissibility (later, these candidates are shown to in fact be admissible). The following definitions make this more clear:

DEFINITION 27

$A \text{ Type}_\tau$ iff $\exists \epsilon, \gamma. (T \text{ Type with } \epsilon, \gamma)_\tau$
 $A \text{ C-AType}_\tau$ iff $\exists \epsilon, \gamma. (T \text{ Type with } \epsilon, \gamma)_\tau$ and γ
 $a \in_\tau A$ iff $\exists \epsilon, \gamma. (T \text{ Type with } \epsilon, \gamma)_\tau$ and $\epsilon(a)$

$A \text{ Type}_\tau$ means A is a type (in interpretation τ), $A \text{ C-AType}_\tau$ means A is a candidate-admissible type, and $a \in_\tau A$ means a is a member of the type A .

Type interpretations may be ordered: a “larger” relation is defined to be true at more values.

DEFINITION 28 Interpretation τ' contains interpretation τ , written $\tau \sqsubseteq \tau'$, iff

$$\forall T, \epsilon, \gamma. (T \text{ Type with } \epsilon, \gamma)_\tau \Rightarrow (T \text{ Type with } \epsilon, \gamma)_{\tau'}.$$

The interpretation for all types of the theory will be carried out a universe level at a time:

σ_1 will be the base theory with no universes,
 σ_2 will have U_1 and V as types,
 \vdots
 σ_α will have U_1, \dots, U_α and V as types.

To construct the interpretation σ_α , each universe U_β for $\beta < \alpha$ will have as members those T such that $T \text{ Type}_{\sigma_\beta}$. Similarly, V will have as members those T such that $T \text{ C-AType}_{\sigma_1}$.

DEFINITION 29 Inductively assume that interpretations $\sigma_1, \dots, \sigma_{\alpha-1}$ have been constructed. Define σ_α as the least fixed-point of the following monotonic operator Ψ_α on interpretations:

$\Psi_\alpha(\tau) \stackrel{\text{def}}{=} \tau'$, where $(T \text{ Type with } \epsilon, \gamma)_{\tau'}$ is true if and only if

EITHER T evaluates to one of E , N , a is b , a eval b , a ind b , or a halts, in which case

γ is true, and

CASE $T \mapsto E$: $\forall t. \epsilon(t)$ is true

CASE $T \mapsto N$: $\forall t. \epsilon(t)$ iff $t \mapsto n$, where n is $0, 1, 2, \dots$

CASE $T \mapsto a$ is b : $\forall t. \epsilon(t)$ iff $a =_\alpha b$

CASE $T \mapsto a$ eval b : $\forall t. \epsilon(t)$ iff $a \mapsto b$

CASE $T \mapsto a$ ind b : $\forall t. \epsilon(t)$ iff $a \succ b$

CASE $T \mapsto a$ halts: $\forall t. \epsilon(t)$ iff $a \downarrow$

OR $T \mapsto a$ in A , in which case

$A \text{ Type}_\tau$,

γ iff $A \text{ C-AType}_\tau$, and

$\forall t. \epsilon(t)$ iff $a \in_\tau A$

OR $T \mapsto x:A \rightarrow B$, in which case

$A \text{ Type}_\tau$ & $\forall a \in_\tau A. B[a/x] \text{ Type}_\tau$,

γ iff $A \text{ C-AType}_\tau$ & $\forall a \in_\tau A. B[a/x] \text{ C-AType}_\tau$, and

$\forall t. \epsilon(t)$ iff $t \mapsto \lambda x.b$ & $\forall a \in_\tau A. b[a/x] \in_\tau B[a/x]$

OR $T \mapsto x:A \times B$, in which case

$A \text{ Type}_\tau$ & $\forall a \in_\tau A. B[a/x] \text{ Type}_\tau$,

γ iff $A \text{ C-AType}_\tau$ & $B \text{ C-AType}_\tau$, and

$\forall t. \epsilon(t)$ iff $t \mapsto \langle a, b \rangle$ & $a \in_\tau A$ & $b \in_\tau B[a/x]$

OR $T \mapsto \overline{A}$, in which case

$A \downarrow \Rightarrow A \text{ Type}_\tau$,

γ iff $A \downarrow \Rightarrow A \text{ C-AType}_\tau$, and

$\forall t. \epsilon(t)$ iff $t \downarrow \Rightarrow t \in_\tau A$

OR $T \mapsto U_\beta$ where $\beta < \alpha$, in which case

γ false, and

$\forall t. \epsilon(t)$ iff $t \text{ Type}_{\sigma_\beta}$

OR $T \mapsto V$ & $\alpha > 1$, in which case

γ false, and

$\forall t. \epsilon(t)$ iff $t \text{ C-AType}_{\sigma_1}$

LEMMA 30 Ψ_α is in fact monotonic over the ordering \sqsubseteq .

PROOF. Suppose $\sigma \sqsubseteq \tau$; show $\Psi_\alpha(\sigma) \sqsubseteq \Psi_\alpha(\tau)$. Abbreviate $\Psi_\alpha(\sigma)$ as σ' and $\Psi_\alpha(\tau)$ as τ' . Arbitrary T, ϵ, γ ; suppose $(T \text{ Type with } \epsilon, \gamma)_{\sigma'}$; show $(T \text{ Type with } \epsilon, \gamma)_{\tau'}$. We will just look at part of one of the more interesting cases of the proof:

CASE $T \mapsto x:A \rightarrow B$: Since $(T \text{ Type with } \epsilon, \gamma)_{\sigma'}$, we know

(1) $A \text{ Type}_\sigma$ and for all $a \in_\sigma A$, $B[a/x] \text{ Type}_\sigma$.

We wish to show $(T \text{ Type with } \epsilon, \gamma)_{\tau'}$, which means we first must show

(2) $A \text{ Type}_\tau$, and
 (3) $\forall a \in_\tau A. B[a/x] \text{ Type}_\tau$.

(2) follows directly from (1) because $A \text{ Type}_\sigma$ implies $A \text{ Type}_\tau$ by the definition of \sqsubseteq . To prove (3), take arbitrary $a \in_\tau A$, and show $B[a/x] \text{ Type}_\tau$. $a \in_\tau A$ means $(A \text{ Type with } \epsilon', \gamma')_\tau$ for some ϵ' and γ' ; by the definition of \sqsubseteq , $(A \text{ Type with } \epsilon', \gamma')_\sigma$, so $a \in_\sigma A$. Thus by (1), $B[a/x] \text{ Type}_\sigma$, which means $B[a/x] \text{ Type}_\tau$ by the definition of \sqsubseteq . The other two subcases needed to show $(T \text{ Type with } \epsilon, \gamma)_{\tau'}$ are similar.

QED.

The least fixed-point σ_α of Ψ_α is sensible because Ψ_α is a monotonic operator on the relation $(T \text{ Type with } \epsilon, \gamma)$: this may be justified by interpreting the relation $(T \text{ Type with } \epsilon, \gamma)$ as a set, and it may also be satisfactory to intuitionists if they accept inductive definitions over higher-order relations; the reader is referred to [All87b, All87a] for further discussion of this. It is possible to reason by induction directly on the structure of this definition, and this is how many properties of the interpretation are proven below.

DEFINITION 31 *The full interpretation σ_ω that includes all universe levels is defined as the union of all the σ_α :*

$$(T \text{ Type with } \epsilon, \gamma)_{\sigma_\omega} \text{ iff } (T \text{ Type with } \epsilon, \gamma)_{\sigma_\alpha} \text{ for some } \alpha.$$

If there is no subscript, it is implicitly the full interpretation σ_ω that is being referred to. This is also the case for the defined notation: $T \text{ Type}$ for instance means $T \text{ Type}_{\sigma_\omega}$.

With this definition in place, numerous lemmas about these relations can be proven either directly or by straightforward induction on the definitions. First we prove some lemmas which characterize typehood.

All types inhabit some universe level:

LEMMA 32 $A \text{ Type}$ iff $A \in U_\alpha$ for some α .

The candidate-admissible types are exactly those types in V :

LEMMA 33 $A \text{ C-AType}$ iff $A \in V$.

If an expression has an inhabitant, it is a type:

LEMMA 34 $a \in A$ implies $A \in U_\alpha$ for some α .

All types are convergent expressions:

LEMMA 35 $A \in U_\alpha$ implies $A \downarrow$.

U_1 contains all types in V :

LEMMA 36 $A \in V$ implies $A \in U_1$.

Types in lower universes are also found in higher universes:

LEMMA 37 $A \in U_\alpha$ implies $A \in U_\beta$ for all $\beta > \alpha$.

The membership relation respects computational equivalence:

LEMMA 38 $a \in A$ and $a \cong b$ and $A \cong B$ implies $b \in B$.

Now some lemmas about typehood and membership which follow directly from the definitions are proven. They can be taken as defining what it means for the different forms of expression to be types, and for what it means to be an inhabitant of the different types. Variables a, b, c, f, p, n, A are arbitrary closed expressions, d and B are arbitrary expressions with only x free in them, and α and β range over universe levels.

For the formation of types,

LEMMA 39 The expressions a is b , a eval b , a ind b , a halts, E , and N are all members of U_α .

LEMMA 40 a in $A \in U_\alpha$ iff $A \in U_\alpha$.

LEMMA 41 $x:A \rightarrow B \in U_\alpha$ iff $A \in U_\alpha$ and for all $a \in A$, $B[a/x] \in U_\alpha$.

LEMMA 42 $x:A \times B \in U_\alpha$ iff $A \in U_\alpha$ and for all $a \in A$, $B[a/x] \in U_\alpha$.

LEMMA 43 $\bar{A} \in U_\alpha$ iff $A \downarrow$ implies $A \in U_\alpha$.

LEMMA 44 $U_\beta \in U_\alpha$ iff $\beta < \alpha$.

There is a parallel collection of lemmas for forming candidate-admissible types.

LEMMA 45 *The expressions a is b , a eval b , a ind b , a halts, E , and N are all members of V .*

LEMMA 46 *a in $A \in V$ iff $A \in V$.*

LEMMA 47 *$x:A \rightarrow B \in V$ iff $A \in V$ and for all $a \in A$, $B[a/x] \in V$.*

LEMMA 48 *$x:A \times B \in V$ iff $A \in V$ and $B \in V$.*

LEMMA 49 *$\overline{A} \in V$ iff $A \downarrow$ implies $A \in V$.*

LEMMA 50 *$V \in U_\alpha$ if $\alpha > 1$.*

For inhabitants of types,

LEMMA 51

- (i) *$c \in a$ is b iff $a =_\alpha b$.*
- (ii) *$c \in a$ eval b iff $a \mapsto b$.*
- (iii) *$c \in a$ ind b iff $a \succ b$.*
- (iv) *$c \in a$ halts iff $a \downarrow$.*
- (v) *$c \in E$.*

LEMMA 52 *$n \in N$ iff $n \mapsto 0, 1, 2, \dots$*

LEMMA 53 *$c \in (a \text{ in } A)$ iff $(a \text{ in } A) \in U_\alpha$ for some α and $a \in A$.*

LEMMA 54 *$f \in x:A \rightarrow B$ iff $x:A \rightarrow B \in U_\alpha$ for some α and $f \mapsto \lambda x.d$ and for all $a \in A$, $d[a/x] \in B[a/x]$.*

LEMMA 55 *$p \in x:A \times B$ iff $x:A \times B \in U_\alpha$ for some α and $p \mapsto \langle a, b \rangle$ and $a \in A$ and $b \in B[a/x]$.*

LEMMA 56 *$a \in \overline{A}$ iff $\overline{A} \in U_\alpha$ for some α and $(a \downarrow)$ implies $a \in A$.*

4.4 Admissibility

Types in the universe V are candidate-admissible types by lemma 33; for a given type to be admissible, all partial functions over the type must have typable fixed-points:

DEFINITION 57 *A* A Type *iff* for all $f \in \overline{A} \rightarrow \overline{A}$, $fix(f) \in \overline{A}$.

We wish to show AC - A Type implies A Type, meaning all candidate-admissible types are admissible. The proof exploits equivalences between fixed-points and their finite approximations.

DEFINITION 58

$$\begin{aligned} \uparrow &\stackrel{\text{def}}{=} fix(\lambda x.x) \\ f^{\times k} &\stackrel{\text{def}}{=} \overbrace{f(f(\dots f(\uparrow)\dots))}^k \end{aligned}$$

$f^{\times k}$ is sometimes called the k -ary *unrolling* of f . All approximations $f^{\times k}$ to the fixed-point $fix(f)$ are properly typed, providing f is typed:

LEMMA 59 $f \in \overline{A} \rightarrow \overline{A}$ *implies* $f^{\times k} \in \overline{A}$.

PROOF. By induction on k , $f^{\times 0}$ is \uparrow and $\uparrow \in \overline{A}$; suppose $f^{\times k} \in \overline{A}$; then, $f(f^{\times k}) \in \overline{A}$, so $f^{\times k+1} \in \overline{A}$.

QED.

Given this fact, we show $fix(f) \in \overline{A}$ by showing the computational behavior of $fix(f)$ to be similar to the computational behavior of its approximations $f^{\times k}$. Intuitively, this similarly seems obvious: if $fix(f)$ halts, then there will be an approximation $f^{\times k}$ for some k which also halts. However, complications arise, as shown by the following example. Consider the case of $f \in \overline{N} \rightarrow \overline{N} \rightarrow \overline{N} \rightarrow \overline{N}$; we wish to show $fix(f) \in \overline{N} \rightarrow \overline{N}$. To prove this, suppose $fix(f) \downarrow$; show $fix(f) \in \overline{N} \rightarrow \overline{N}$. This means (by the intuitive argument above) $f^{\times k} \downarrow$ for some k . However, we also need to know $fix(f)$ has a value that is a λ -expression, because all inhabitants of function spaces must be λ -expressions. We know $f^{\times k} \in \overline{N} \rightarrow \overline{N}$ by lemma 59 above, so $f^{\times k} \mapsto \lambda x.b$. We would like to say $fix(f) \mapsto \lambda x.b$ which would prove our result, but this is not true, because the result may have $fix(f)$ occurring freely in it. For instance, letting f be $\lambda g.\lambda x.\langle 0, g(x) \rangle.1$, $fix(f) \mapsto \lambda x.\langle 0, fix(f)(x) \rangle.1$, whereas $f^{\times k} \mapsto \lambda x.\langle 0, f^{\times k-1}(x) \rangle.1$. This requires us to keep track of occurrences of $fix(f)$ and $f^{\times k}$ in the results of computations. This and other factors complicate the proof.

There are two parts to the proof of admissibility: some lemmas are proven which fully reflect the computational similarity between fixed-points and their approximations. Then, the candidate-admissible types are shown to be admissible by structural induction. Before we proceed with the proof, we show all of this work is necessary because there exist non-admissible types.

4.4.1 Non-admissible types

Since we cannot prove all U_1 types are admissible, it is reasonable to expect that some types cannot be admissible. This means that there would be a type D and a function $d \in \overline{D} \rightarrow \overline{D}$ such that $fix(d) \in \overline{D}$ is false. Such types do in fact exist.

THEOREM 60 *Some types are not admissible.*

PROOF. Define

$$\begin{aligned} D &\stackrel{\text{def}}{=} g:(\mathbb{N} \rightarrow \overline{\mathbb{N}}) \times \neg(\forall n:\mathbb{N}. g(n) \text{ halts}) \text{ and} \\ d &\stackrel{\text{def}}{=} \lambda g. \langle \lambda n. \text{if_zero}(n; 0; \text{seq}(\text{pred}(n); x.(g.1)(x))), \lambda x. 0 \rangle. \end{aligned}$$

It is possible to prove $d \in \overline{D} \rightarrow \overline{D}$: assume $g \in \overline{D}$, and show the pair to be in D . For the left half of the pair, $l \stackrel{\text{def}}{=} \lambda n. \text{if_zero}(n; 0; \text{seq}(\text{pred}(n); x.(g.1)(x)))$ must be shown to be in $\mathbb{N} \rightarrow \overline{\mathbb{N}}$, which is straightforward. For the right half of the pair, we show $\neg(\forall n:\mathbb{N}. l(n) \text{ halts})$: suppose the antecedent were true; then, $g \downarrow$ because $g.1 \downarrow$. Thus, $g \in D$, so $g.2 \in \neg(\forall n:\mathbb{N}. g.1(n) \text{ halts})$. But, if $g.1$ is not total, l will also not be total, contradicting our assumption. Thus, $d \in \overline{D} \rightarrow \overline{D}$.

If it were true that $fix(d) \in \overline{D}$, $fix(d) \in D$ would also follow because $fix(d) \downarrow$. This would mean the function $fix(d).1 \in \mathbb{N} \rightarrow \overline{\mathbb{N}}$ is not total, but this is a contradiction because it can be proved total by induction on natural numbers. Therefore, $fix(d) \notin \overline{D}$, so D is not admissible.

QED.

4.4.2 Computational lemmas

Lemmas characterizing the computational behavior of fixed-points are now proven. To simplify the proofs to be given, we define notation and some conventions for reasoning about some particular forms of substitution. All of the definitions presuppose some arbitrary closed expression f .

DEFINITION 61 *Let t be an expression with free variables amongst x_1, \dots, x_n . Given a vector of natural numbers $\vec{k} \stackrel{\text{def}}{=} k_1, \dots, k_n$, define*

$$t^{[\vec{k}]} \stackrel{\text{def}}{=} t[f^{\times k_1}/x_1, \dots, f^{\times k_n}/x_n].$$

$t^{[\vec{k}]}$ is thus a closed expression. The notation $t^{[\vec{k}]}$ is ambiguous because the variable names x_1, \dots, x_n are nowhere apparent. However, we can get by with ignoring them and will do so. α -variants are also identified. Often we wish to add a fixed amount m to each k_i of \vec{k} , so there is a special notation for this:

DEFINITION 62 *Let t and \vec{k} be as above, and let m be a natural number; define*

$$t^{[\vec{k}+m]} \stackrel{\text{def}}{=} t[f^{\times k_1+m}/x_1, \dots, f^{\times k_n+m}/x_n].$$

A similar notation also holds for $fix(f)$ substitutions:

DEFINITION 63 *Let t be as above; define*

$$t^{[\vec{\omega}]} \stackrel{\text{def}}{=} t[fix(f)/x_1, \dots, fix(f)/x_n].$$

Substitutions are always factored in; for example, $(\langle a, b \rangle)^{[\vec{k}]}$ is the same expression as $\langle a^{[\vec{k}]}, b^{[\vec{k}]} \rangle$. For a concrete example of this notation, let \vec{k} be 3, 2, 5, and x_1, x_2, x_3 be a, b, c . $\langle \langle a, b \rangle, \langle c, a \rangle \rangle^{[\vec{k}]}$ is

$$\langle \langle a, b \rangle, \langle c, a \rangle \rangle [f^{\times 3}/a, f^{\times 2}/b, f^{\times 5}/c].$$

These substitutions may be factored in, giving the equivalent expression

$$\langle \langle a, b \rangle^{[\vec{k}]}, \langle c, a \rangle^{[\vec{k}]} \rangle.$$

Superscripts are used to label substitutions, so \vec{k}^a is just another name for a substitution; the superscript a means that the substitution is intended for an expression a .

It is often desirable to merge substitutions when two expressions are put together to form a new expression.

DEFINITION 64 *Given expressions $a^{[\vec{k}^a]}$ and $b^{[\vec{k}^b]}$, define a single merged substitution \vec{k} such that $a^{[\vec{k}^a]} =_{\alpha} a^{[\vec{k}]}$ and $b^{[\vec{k}^b]} =_{\alpha} b^{[\vec{k}]}$.*

The two substitutions are appended, α -converting the free variables in one expression that both have in common.

The first computational fact we need is that if an expression halts, we may replace all occurrences of $fix(f)$ in the expression with large enough approximations, and the expression will still halt. This fact cannot be proven without a considerable strengthening of the statement, so even though the result is not deep, the proof is long.

LEMMA 65 $\forall f, t. t^{[\vec{\omega}]} \downarrow \Rightarrow \exists v. t^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. t^{[\vec{k}+n]} \mapsto v^{[\vec{m}+n]}$.

PROOF. Suppose $t^{[\vec{\omega}]} \downarrow$; we prove the consequent by induction on the computation of $t^{[\vec{\omega}]}$. Assume without loss of generality that the free variables in t are distinct from any bound variables in t (this means we need not worry about capture of superscripted substitutions).

CASE $t^{[\vec{\omega}]}$ is a value: $t^{[\vec{\omega}]} \mapsto t^{[\vec{\omega}]}$ and the result follows trivially.

CASE $t^{[\vec{\omega}]}$ is $f^{[\vec{\omega}^f]}(a^{[\vec{\omega}^a]})$: By the induction hypothesis, we have

- (1) $f^{[\vec{\omega}^f]} \mapsto \lambda x. b^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. f^{[\vec{k}+n]} \mapsto \lambda x. b^{[\vec{m}+n]}$
- (2) $b^{[\vec{\omega}^b]}[a^{[\vec{\omega}^a]}/x] \mapsto v^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. b^{[\vec{k}^b+n]}[a^{[\vec{k}^a+n]}/x] \mapsto v^{[\vec{m}+n]}$.

Therefore, $f^{[\vec{\omega}^f]}(a^{[\vec{\omega}^a]}) \mapsto v^{[\vec{\omega}]}$; we wish to show

- (3) $\exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. f^{[\vec{k}^f+n]}(a^{[\vec{k}^a+n]}) \mapsto v^{[\vec{m}+n]}$.

To arrive at this, we must show that we can unify b in (1) and (2) so that they will be the same approximation. It requires special care taken to how the quantifiers are instantiated to accomplish this. $b^{[\vec{m}+n]}$ must have been unrolled enough times to be a more defined approximation than $b^{[\vec{j}^b]}$, meaning $\vec{m} + n \geq \vec{j}^b$. n is thus picked to be large enough so that this will be the case. \vec{k}^b in (2) can then be picked to make the two b 's identical, meaning $\vec{k}^b = \vec{m} + n$. It is the need to *exactly* unify b in (1) and (2) that makes the proof difficult.

There is much tedious manipulation of quantifiers to be done since there are so many of them; some conventions will thus be used to simplify matters. Numerical subscripts will be used to indicate which formula a particular variable is from. For example, if \vec{k} is taken to be an arbitrary in formula (1), it will be given the name \vec{k}_1 . (1k) refers to formula (1) with quantifiers up to the \vec{k} quantifier removed.

Let \vec{j}_1 name \vec{j} in (1), and let \vec{j}_2 name \vec{j} in (2). Instantiate \vec{k} in (1) with \vec{j}_1 , let \vec{m}_1 name \vec{m} in (1), and pick n to be the least n_1 such that $n_1, \dots, n_1 \geq \vec{j}_2^b + \vec{m}_1$. Now,

we may pick \vec{j} in (3) to be $\vec{j}_1 + n_1$ for \vec{j}^f and \vec{j}_2^a for \vec{j}^a . Pick arbitrary \vec{k}_3 in (3j) such that $\vec{k}_3 \geq \vec{j}_3$. Now, re-instantiate \vec{k} in (1j) with $\vec{k}_3^f - n_1$; this is bigger than \vec{j}_1 because $\vec{k}_3^f \geq \vec{j}_1 + n_1$. Name \vec{m} in (1k) \vec{m}_1 ; we now have

$$(1m) \quad \forall n. f^{[\vec{k}_3^f - n_1 + n]} \mapsto \lambda x. b^{[\vec{m}_1 + n]},$$

which means

$$(1m') \quad \forall n. f^{[\vec{k}_3^f + n]} \mapsto \lambda x. b^{[\vec{m}_1 + n_1 + n]}.$$

Instantiate \vec{k} in (2j) with \vec{k}_3^a for \vec{k}^a ($\vec{k}_3^a \geq \vec{j}_3^a = \vec{j}_2^a$) and $\vec{m}_1 + n_1$ for \vec{k}^b ($n_1 \geq \vec{j}_2^b$, so $\vec{m}_1 + n_1 \geq \vec{j}_2^b$). Name \vec{m} in (2k) \vec{m}_2 , so we now have

$$(2m) \quad \forall n. b^{[\vec{m}_1 + n_1 + n]}[a^{[\vec{k}_3^a + n]}/x] \mapsto v^{[\vec{m}_2 + n]}.$$

Instantiate \vec{m} in (3k) with \vec{m}_2 , and take n to be arbitrary n_3 ; show

$$(3n) \quad f^{[\vec{k}_3^f + n_3]}(a^{[\vec{k}_3^a + n_3]}) \mapsto v^{[\vec{m}_2 + n_3]}.$$

This follows by computation from (1m') and (2m) instantiated with n_3 for n .

CASE $t^{[\vec{\omega}]}$ is $p^{[\vec{\omega}]} . 1$: By the induction hypothesis, we have

$$(1) \quad p^{[\vec{\omega}]} . 1 \mapsto \langle a^{[\vec{\omega}^a]}, b^{[\vec{\omega}^b]} \rangle \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. p^{[\vec{k} + n]} \mapsto \langle a^{[\vec{m}^a + n]}, b^{[\vec{m}^b + n]} \rangle$$

$$(2) \quad a^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. a^{[\vec{k} + n]} \mapsto v^{[\vec{m} + n]}.$$

Therefore, $p^{[\vec{\omega}]} . 1 \mapsto v^{[\vec{\omega}]}$; we wish to show

$$(3) \quad \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. p^{[\vec{k}^f + n]} . 1 \mapsto v^{[\vec{m} + n]}.$$

Let \vec{j}_1 name \vec{j} in (1), and let \vec{j}_2 name \vec{j} in (2). Instantiate \vec{k} in (1) with \vec{j}_1 , let \vec{m}_1 name \vec{m} in (1), and pick n to be the least n_1 such that $n_1, \dots, n_1 \geq \vec{j}_2 + \vec{m}_1^a$. Now, we may pick \vec{j} in (3) to be $\vec{j}_1 + n_1$. Pick arbitrary \vec{k}_3 in (3j) such that $\vec{k}_3 \geq \vec{j}_3$. Now, re-instantiate \vec{k} in (1j) with $\vec{k}_3 - n_1$; this is bigger than \vec{j}_1 because $\vec{k}_3 \geq \vec{j}_1 + n_1$. Name \vec{m} in (1k) \vec{m}_1 ; we now have

$$(1m) \quad \forall n. p^{[\vec{k}_3 - n_1 + n]} \mapsto \langle a^{[\vec{m}_1^a + n]}, b^{[\vec{m}_1^b + n]} \rangle,$$

which means

$$(1m') \quad \forall n. p^{[\vec{k}_3 + n]} \mapsto \langle a^{[\vec{m}_1^a + n_1 + n]}, b^{[\vec{m}_1^b + n_1 + n]} \rangle.$$

Instantiate \vec{k} in (2j) with $\vec{m}_1^a + n_1$ ($n_1 \geq \vec{j}_2$, so $\vec{m}_1^a + n_1 \geq \vec{j}_2^b$). Name \vec{m} in (2k) \vec{m}_2 , so we now have

$$(2m) \quad \forall n. a^{[\vec{m}_1^a + n_1 + n]} \mapsto v^{[\vec{m}_2 + n]}.$$

Instantiate \vec{m} in (3k) with \vec{m}_2 , and take n to be arbitrary n_3 ; show

$$(3n) \quad p^{[\vec{k}_3 + n_3].1} \mapsto v^{[\vec{m}_2 + n_3]}.$$

This follows by computation from (1m') and (2m) instantiated with n_3 for n .

CASE $t^{[\vec{\omega}]}$ is $p^{[\vec{\omega}].2}$: This is similar to the above case.

CASE $t^{[\vec{\omega}]}$ is $if_zero(a^{[\vec{\omega}]}; b^{[\vec{\omega}]}; c^{[\vec{\omega}]})$: By the induction hypothesis, we have

- (1) $a^{[\vec{\omega}]} \mapsto n_0$ & $\exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. a^{[\vec{k} + n]} \mapsto n_0$
- (2a) if n_0 is 0, $b^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]}$ & $\exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. b^{[\vec{k} + n]} \mapsto v^{[\vec{m} + n]}$.
- (2b) if n_0 is 1, 2, 3, ..., $c^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]}$ & $\exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. c^{[\vec{k} + n]} \mapsto v^{[\vec{m} + n]}$.

Therefore, $if_zero(a^{[\vec{\omega}]}; b^{[\vec{\omega}]}; c^{[\vec{\omega}]}) \mapsto v^{[\vec{\omega}]}$; we wish to show

$$(3) \quad \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. if_zero(a^{[\vec{\omega}]}; b^{[\vec{\omega}]}; c^{[\vec{\omega}]}) \mapsto v^{[\vec{m} + n]}.$$

Without loss of generality, assume n_0 is 0. Let \vec{j}_1 name \vec{j} in (1), and let \vec{j}_2 name \vec{j} in (2). Instantiate \vec{j} in (3) by letting \vec{j}^a be \vec{j}_1^a , \vec{j}^b be \vec{j}_2^b , and \vec{j}^c be $\vec{0}$. Pick arbitrary \vec{k}_3 in (3j) such that $\vec{k}_3 \geq \vec{j}_3$. Now, instantiate \vec{k} in (1j) with \vec{k}_3^a ; this is bigger than \vec{j}_1 . Name \vec{m} in (1k) \vec{m}_1 ; we now have

$$(1m) \quad \forall n. a^{[\vec{k}_3^a + n]} \mapsto n_0.$$

Instantiate \vec{k} in (2j) with \vec{k}_3^b ; this is bigger than \vec{j}_2 . Name \vec{m} in (2k) \vec{m}_2 , so we now have

$$(2m) \quad \forall n. b^{[\vec{k}_3^b + n]} \mapsto v^{[\vec{m}_2 + n]}.$$

Instantiate \vec{m} in (3k) with \vec{m}_2 , and take n to be arbitrary n_3 ; show

$$(3n) \quad if_zero(a^{[\vec{k}_3^a + n_3]}; b^{[\vec{k}_3^b + n_3]}; c^{[\vec{k}_3^c + n_3]}) \mapsto v^{[\vec{m}_2 + n_3]}.$$

This follows by computation from (1m) and (2m) instantiated with n_3 for n .

CASE $t^{[\vec{\omega}]}$ is $succ(a^{[\vec{\omega}]})$: By the induction hypothesis, we have

$$(1) \quad a^{[\vec{\omega}]} \mapsto n_0 \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. a^{[\vec{k}+n]} \mapsto n_0$$

Therefore, $\text{succ}(a^{[\vec{\omega}]}) \mapsto m_0$, where m_0 is one more than n_0 . We wish to show

$$(2) \quad \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. \text{succ}(a^{[\vec{k}+n]}) \mapsto m_0.$$

This fact follows directly from (1).

CASE $t^{[\vec{\omega}]}$ is $\text{pred}(a^{[\vec{\omega}]})$: This is similar to the previous case.

CASE $t^{[\vec{\omega}]}$ is $(\text{fix}(g^{[\vec{\omega}]}))$: By the induction hypothesis, we have

$$(1) \quad g^{[\vec{\omega}]}(\text{fix}(g^{[\vec{\omega}]})) \mapsto v^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. g^{[\vec{k}+n]}(\text{fix}(g^{[\vec{k}+n]})) \mapsto v^{[\vec{m}+n]};$$

therefore, $(\text{fix}(g^{[\vec{\omega}]})) \mapsto v^{[\vec{\omega}]}$. We wish to show

$$(2) \quad \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. \text{fix}(g^{[\vec{k}+n]}) \mapsto v^{[\vec{m}+n]}.$$

Let \vec{j}_1 name \vec{j} in (1); instantiate \vec{j} in (2) to be \vec{j}_1 . Pick arbitrary \vec{k}_2 in (2j) such that $\vec{k}_2 \geq \vec{j}_2$; instantiate \vec{k} in (1j) with \vec{k}_2 ; this is bigger than \vec{j}_1 . Name \vec{m} in (1k) \vec{m}_1 , and instantiate \vec{m} in (2) with \vec{m}_1 . Arbitrary n_2 in (2m); instantiate n in (1m) with n_2 . It then follows by computation that $g^{[\vec{k}_2+n_2]}(\text{fix}(g^{[\vec{k}_2+n_2]})) \mapsto v^{[\vec{m}_2+n_2]}$ implies $\text{fix}(g^{[\vec{k}_2+n_2]}) \mapsto v^{[\vec{m}_2+n_2]}$.

CASE $t^{[\vec{\omega}]}$ is $\text{fix}(f)$: This case arises when $t^{[\vec{\omega}]}$ is of the form $x[\text{fix}(f)/x]$ for some variable x . Since in this case \vec{j} and \vec{k} are unit-length vectors, they will be treated as natural numbers j and k . By the induction hypothesis,

$$(1) \quad f(\text{fix}(f)) \mapsto v^{[\vec{\omega}]} \ \& \ \exists j. \forall k \geq j. \exists \vec{m}. \forall n. f(f^{\times k+n}) \mapsto v^{[\vec{m}+n]}.$$

Therefore, $\text{fix}(f) \mapsto v^{[\vec{\omega}]}$; show

$$(2) \quad \exists j. \forall k \geq j. \exists \vec{m}. \forall n. f^{\times k+n} \mapsto v^{[\vec{m}+n]}.$$

Let j_1 name j in (1); instantiate j in (2) to be $j_1 + 1$. Arbitrary $k_2 \geq j_2$; instantiate k in (1) with $k_2 - 1$ ($k_2 - 1 \geq j_2 - 1$). Let \vec{m}_1 name \vec{m} in (1k), and instantiate \vec{m} in (2k) with \vec{m}_1 . Take n in (2m) to be arbitrary n_2 , and instantiate n in (1m) with n_2 ; we thus have

$$(1n) \quad f(f^{[k_1+1+n_2]}) \mapsto v^{[\vec{m}_1+n_2]}$$

and we wish to show

$$(2n) \quad f^{[k_1+n_2]} \mapsto v^{[\vec{m}_1+1+n_2]},$$

which follows directly.

CASE $t^{[\vec{\omega}]}$ is $seq(a^{[\vec{\omega}^a]}; x.b^{[\vec{\omega}^b]})$: By the induction hypothesis, we have

$$(1) \quad a^{[\vec{\omega}]} \mapsto u^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. a^{[\vec{k}+n]} \mapsto u^{[\vec{m}+n]}$$

$$(2) \quad b^{[\vec{\omega}^b]}[u^{[\vec{\omega}^a]}/x] \mapsto v^{[\vec{\omega}]} \ \& \ \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. b^{[\vec{k}^b+n]}[u^{[\vec{k}^a+n]}/x] \mapsto v^{[\vec{m}+n]}.$$

Therefore, $seq(a^{[\vec{\omega}^a]}; x.b^{[\vec{\omega}^b]}) \mapsto v^{[\vec{\omega}]}$; we wish to show

$$(3) \quad \exists \vec{j}. \forall \vec{k} \geq \vec{j}. \exists \vec{m}. \forall n. seq(a^{[\vec{k}^a+n]}; x.b^{[\vec{k}^b+n]}) \mapsto v^{[\vec{m}+n]}.$$

Let \vec{j}_1 name \vec{j} in (1), and let \vec{j}_2 name \vec{j} in (2). Instantiate \vec{k} in (1) with \vec{j}_1 , let \vec{m}_1 name \vec{m} in (1), and pick n to be the least n_1 such that $n_1, \dots, n_1 \geq \vec{j}_2^a + \vec{m}_1$. Now, we may pick \vec{j} in (3) to be $\vec{j}_1^a + n_1$ for \vec{j}^a and \vec{j}_2 for \vec{j}^b . Pick arbitrary \vec{k}_3 in (3j) such that $\vec{k}_3 \geq \vec{j}_3$. Now, re-instantiate \vec{k} in (1j) with $\vec{k}_3^a - n_1$ for \vec{k} ; this is bigger than \vec{j}_1 because $\vec{k}_3^a \geq \vec{j}_1 + n_1$. Name \vec{m} in (1k) \vec{m}_1 ; we now have

$$(1m) \quad \forall n. a^{[\vec{k}_3^a-n_1+n]} \mapsto u^{[\vec{m}_1+n]},$$

which means

$$(1m') \quad \forall n. a^{[\vec{k}_3^a+n]} \mapsto u^{[\vec{m}_1+n_1+n]}.$$

Instantiate \vec{k} in (2j) with \vec{k}_3^b for \vec{k}^b ($\vec{k}_3^b \geq \vec{j}_3^b = \vec{j}_2^b$) and $\vec{m}_1 + n_1$ for \vec{k}^u ($n_1 \geq \vec{j}_2^u$, so $\vec{m}_1 + n_1 \geq \vec{j}_2^u$). Name \vec{m} in (2k) \vec{m}_2 , so we now have

$$(2m) \quad \forall n. b^{[\vec{k}_3^b+n]}[u^{[\vec{m}_1+n_1+n]}/x] \mapsto v^{[\vec{m}_2+n]}.$$

Instantiate \vec{m} in (3k) with \vec{m}_2 , and take n to be arbitrary n_3 ; show

$$(3n) \quad seq(a^{[\vec{k}_3^a+n_3]}; x.b^{[\vec{k}_3^b+n_3]}) \mapsto v^{[\vec{m}_2+n_3]}.$$

This follows by computation from (1m') and (2m) instantiated with n_3 for n .

QED.

From this lemma, the fact we need follows directly.

LEMMA 66 $t^{[\vec{\omega}]} \downarrow$ implies $\exists \vec{j}. \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]} \downarrow$.

PROOF. This is an immediate consequence of the previous theorem.

QED.

The next lemma we need is to go in the opposite direction: if an approximation halts, the fixed-point also halts.

LEMMA 67 $\forall t, f, \vec{k}. t^{[\vec{k}]} \downarrow \Rightarrow \exists v, \vec{m}. t^{[\vec{k}]} \mapsto v^{[\vec{m}]} \ \& \ t^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]}$

PROOF. Suppose $t^{[\vec{k}]} \downarrow$; proceed by induction on the structure of computation.

CASE $t^{[\vec{k}]}$ is a value: Trivial.

CASE $t^{[\vec{k}]}$ is $g^{[\vec{k}]}(a^{[\vec{k}]})$: By the induction hypothesis,

$$g^{[\vec{k}]} \mapsto \lambda x. b^{[\vec{m}_1]} \ \& \ g^{[\vec{\omega}]} \mapsto \lambda x. b^{[\vec{\omega}]} \\ b^{[\vec{m}_1]}[a^{[\vec{m}]} / x] \mapsto v^{[\vec{m}_2]} \ \& \ b^{[\vec{\omega}]}[a^{[\vec{\omega}]} / x] \mapsto v^{[\vec{\omega}]}.$$

Thus,

$$g^{[\vec{k}]}(a^{[\vec{k}]}) \mapsto v^{[\vec{m}_2]} \ \& \ g^{[\vec{\omega}]}(a^{[\vec{\omega}]}) \mapsto v^{[\vec{\omega}]}.$$

CASE $t^{[\vec{k}]}$ is $p^{[\vec{k}]}.1$: By the induction hypothesis,

$$p^{[\vec{k}]} \mapsto \langle a^{[\vec{m}_1]}, b^{[\vec{m}_1]} \rangle \ \& \ p^{[\vec{\omega}]} \mapsto \langle a^{[\vec{\omega}]}, b^{[\vec{\omega}]} \rangle \\ a^{[\vec{m}_1]} \mapsto v^{[\vec{m}_2]} \ \& \ a^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]}.$$

Thus,

$$p^{[\vec{k}]}.1 \mapsto v^{[\vec{m}_2]} \ \& \ p^{[\vec{\omega}]}.1 \mapsto v^{[\vec{\omega}]}.$$

CASE $t^{[\vec{k}]}$ is $p^{[\vec{k}]}.2$: Similar to the above case.

CASE $t^{[\vec{k}]}$ is $if_zero(a^{[\vec{k}]}; b^{[\vec{k}]}; c^{[\vec{k}]})$: By the induction hypothesis,

$$a^{[\vec{k}]} \mapsto n_0 \ \& \ a^{[\vec{\omega}]} \mapsto n_0.$$

Assume without loss of generality that n_0 is zero. Then,

$$b^{[\vec{k}]} \mapsto v^{[\vec{m}_2]} \ \& \ b^{[\vec{\omega}]} \mapsto v^{[\vec{\omega}]}.$$

Thus,

$$if_zero(a^{[\vec{k}]}; b^{[\vec{k}]}; c^{[\vec{k}]}) \mapsto v^{[\vec{m}_2]} \ \& \ if_zero(a^{[\vec{\omega}]}; b^{[\vec{\omega}]}; c^{[\vec{\omega}]}) \mapsto v^{[\vec{\omega}]}.$$

CASE $t^{[\vec{k}]}$ is $succ(a^{[\vec{k}]})$: By the induction hypothesis,

$$a^{[\vec{k}]} \mapsto n_0 \ \& \ a^{[\vec{\omega}]} \mapsto n_0.$$

Thus,

$$succ(a^{[\vec{k}]}) \mapsto m_0 \ \& \ succ(a^{[\vec{\omega}]}) \mapsto m_0$$

where m_0 is one more than n_0 .

CASE $t^{[\vec{k}]}$ is $\text{pred}(a^{[\vec{k}]})$: Similar to previous case.

CASE $t^{[\vec{k}]}$ is $\text{fix}(g^{[\vec{k}]})$: By the induction hypothesis,

$$g^{[\vec{k}]}(\text{fix}(g^{[\vec{k}]})) \mapsto v^{[\vec{m}_1]} \ \& \ g^{[\vec{\omega}]}(\text{fix}(g^{[\vec{\omega}]})) \mapsto v^{[\vec{\omega}]}.$$

Therefore,

$$\text{fix}(g^{[\vec{k}]}) \mapsto v^{[\vec{m}_1]} \ \& \ \text{fix}(g^{[\vec{\omega}]}) \mapsto v^{[\vec{\omega}]}.$$

CASE $t^{[\vec{k}]}$ is $f^{\times k}$: This means $t^{[\vec{\omega}]}$ is $\text{fix}(f)$. $f^{\times k}$ is $f(f^{\times k-1})$, and to compute $\text{fix}(f)$ is to compute $f(\text{fix}(f))$; thus, using the result from the case for application above,

$$f(f^{\times k-1}) \mapsto v^{[\vec{m}]} \ \& \ f(\text{fix}(f)) \mapsto v^{[\vec{\omega}]}$$

for some \vec{m} .

CASE $t^{[\vec{k}]}$ is $\text{seq}(a^{[\vec{k}]}; x.b^{[\vec{k}]})$: By the induction hypothesis,

$$\begin{aligned} a^{[\vec{k}]} &\mapsto u^{[\vec{m}_1]} \ \& \ a^{[\vec{\omega}]} \mapsto u^{[\vec{\omega}]} \\ b^{[\vec{k}]}[u^{[\vec{m}_1]}/x] &\mapsto v^{[\vec{m}_2]} \ \& \ b^{[\vec{\omega}]}[u^{[\vec{\omega}]} / x] \mapsto v^{[\vec{\omega}]} . \end{aligned}$$

Thus,

$$\text{seq}(a^{[\vec{k}]}; x.b^{[\vec{k}]}) \mapsto v^{[\vec{m}_2]} \ \& \ \text{seq}(a^{[\vec{\omega}]}; x.b^{[\vec{\omega}]}) \mapsto v^{[\vec{\omega}]}.$$

QED.

4.4.3 Proof of admissibility

It is now possible to prove if the approximations $f^{\times k}$ are typed, the fixed-point may be typed; we prove a stronger result from which this statement follows as a corollary.

LEMMA 68 $\forall T, f, t. T \text{ C-AType} \Rightarrow (\exists \vec{j}. \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]} \in T) \Rightarrow t^{[\vec{\omega}]} \in T$.

PROOF. Arbitrary T and t , and suppose $T \text{ C-AType}$. We prove

$$(1) \quad (\exists \vec{j}. \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]} \in T) \Rightarrow t^{[\vec{\omega}]} \in T$$

by induction on the definition of the candidate-admissible type T . Assume (1) is true for all candidate-admissible types already defined. Assume the antecedent

$$(2) \quad \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]} \in T$$

for some \vec{j} ; we show $t^{[\vec{\omega}]} \in T$. Since T C-AType, we may analyze by cases the possibilities for T . But first, note that we may let \vec{k} in (2) be \vec{j} , so $t^{[\vec{j}]} \in T$.

CASE T evaluates to an atomic propositional assertion or to E: $t^{[\vec{j}]} \in T$ as just mentioned, so by lemma 51, $c \in T$ for any c ; thus, $t^{[\vec{\omega}]} \in T$.

CASE $T \mapsto N$: We wish to show $t^{[\vec{\omega}]} \in T$. $t^{[\vec{j}]} \in T$, so $t^{[\vec{j}]} \mapsto n'$; by lemma 67, $t^{[\vec{\omega}]} \mapsto n'$, so $t^{[\vec{\omega}]} \in T$.

CASE $T \mapsto x:A \rightarrow B$: We wish to show $t^{[\vec{\omega}]} \in T$, which by lemma 54 means we must show

$$(3) \quad t^{[\vec{\omega}]} \mapsto \lambda x.b \ \& \ \forall a \in A. b[a/x] \in B[a/x].$$

$t^{[\vec{j}]} \in T$, so $t^{[\vec{j}]} \mapsto \lambda x.b'$; therefore, by lemma 67, $t^{[\vec{\omega}]} \mapsto \lambda x.b$ for some b , showing the first part of (3). To show the second part, assume arbitrary $a \in A$, and show $b[a/x] \in B[a/x]$. $t^{[\vec{\omega}]}(a) \cong b[a/x]$, so it suffices to show $t^{[\vec{\omega}]}(a) \in B[a/x]$. $B[a/x]$ must be a candidate-admissible type, so we may inductively assume (1) is true for $B[a/x]$:

$$(4) \quad (\exists \vec{j}. \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]}(a) \in B[a/x]) \Rightarrow t^{[\vec{\omega}]}(a) \in B[a/x].$$

The consequent of (4) is the result we want, so we only need to show the antecedent. Instantiate \vec{j} with \vec{j} , let \vec{k} be arbitrary; we thus need $t^{[\vec{k}]}(a) \in B[a/x]$. From (2), we know that $t^{[\vec{k}]} \in T$, which means

$$(5) \quad t^{[\vec{k}]} \mapsto \lambda x.b'' \ \& \ \forall a \in A. b''[a/x] \in B[a/x].$$

Thus, $b''[a/x] \in B[a/x]$; $t^{[\vec{k}]}(a) \cong b''[a/x]$, so $t^{[\vec{k}]}(a) \in B[a/x]$, what we needed to show.

CASE $T \mapsto x:A \times B$: Since T C-AType, x does not occur freely in B . We wish to show $t^{[\vec{\omega}]} \in T$, which by lemma 55 means we must show

$$(6) \quad t^{[\vec{\omega}]} \mapsto \langle a, b \rangle \ \& \ a \in A \ \& \ b \in B.$$

$t^{[\vec{j}]} \in T$, so $t^{[\vec{j}]} \mapsto \langle a', b' \rangle$; therefore, by lemma 67, $t^{[\vec{\omega}]} \mapsto \langle a, b \rangle$ for some a and b , showing the first part of (6). In showing the second part, without loss of generality we just show $a \in A$. $t^{[\vec{\omega}]} \cdot 1 \cong a$, so it suffices to show $t^{[\vec{\omega}]} \cdot 1 \in A$. A must be a candidate-admissible type, so we may inductively assume (1) is true for A :

$$(7) \quad (\exists \vec{j}. \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]} \cdot 1 \in A) \Rightarrow t^{[\vec{\omega}]} \cdot 1 \in A.$$

The consequent of (7) is the result we want, so we just need to show the antecedent. Instantiate \vec{j} with \vec{j} , let \vec{k} be arbitrary; we thus need $t^{[\vec{k}]} \cdot 1 \in A$. From (2), we know that $t^{[\vec{k}]} \in T$, which means

$$(8) \quad t^{[\vec{k}]} \mapsto \langle a'', b'' \rangle \ \& \ a'' \in A \ \& \ b'' \in B.$$

Thus, $a'' \in A$; $t^{[\vec{k}]} \cdot 1 \cong a''$, so $t^{[\vec{k}]} \cdot 1 \in A$, what we needed to show.

CASE $T \mapsto \overline{A}$: We wish to show $t^{[\vec{\omega}]} \in T$, which by lemma 56 means we must show

$$(9) \quad t^{[\vec{\omega}]} \downarrow \Rightarrow t^{[\vec{\omega}]} \in A.$$

Suppose $t^{[\vec{\omega}]} \downarrow$; show $t^{[\vec{\omega}]} \in A$. By lemma 66,

$$(10) \quad \forall \vec{k} \geq \vec{j}'. t^{[\vec{k}]} \downarrow$$

for some arbitrary \vec{j}' . A must be a candidate-admissible type, so we may inductively assume (1) is true for A :

$$(11) \quad (\exists \vec{j}. \forall \vec{k} \geq \vec{j}. t^{[\vec{k}]} \in A) \Rightarrow t^{[\vec{\omega}]} \in A.$$

The consequent of (11) is the result we want, so we just need to show the antecedent. Instantiate \vec{j} in (11) with \vec{j}'' such that $\vec{j}'' \geq \vec{j} + \vec{j}'$; let \vec{k} be arbitrary, and show $t^{[\vec{k}]} \in A$. From (2), we know that $t^{[\vec{k}]} \in T$, which means

$$(12) \quad t^{[\vec{k}]} \downarrow \Rightarrow t^{[\vec{k}]} \in A.$$

The consequent of (12) is what we wish to show. Instantiate \vec{k} in (10) with \vec{k} , giving $t^{[\vec{k}]} \downarrow$. Thus, $t^{[\vec{k}]} \in A$.

QED.

The consequence of this lemma is that all candidate-admissible types are admissible.

THEOREM 69 *For all types A , A C-AType implies A AType.*

PROOF. Suppose A C-AType. To show A AType, let f be an arbitrary function such that $f \in \overline{A} \rightarrow \overline{A}$; show $\text{fix}(f) \in \overline{A}$. From lemma 68, this means

$$(\exists \vec{j}. \forall \vec{k} \geq \vec{j}. f^{\times k} \in \overline{A}) \Rightarrow \text{fix}(f) \in \overline{A};$$

the antecedent follows from lemma 59, so $\text{fix}(f) \in \overline{A}$.

QED.

We now have all of the results needed to prove the rules valid.

4.5 Consistency of the rules

All of the interpretation is now in place, and it only remains to show that the rules are true under this interpretation. Sequents are defined and interpreted exactly as in section 3.5.

4.5.1 Conventions

Recall that each sequent has an implicit list of hypotheses $x_1:A_1, \dots, x_n:A_n$; only the relevant hypotheses in the list are shown. We define some abbreviations for hypothesis lists and substitutions over lists.

DEFINITION 70

$$\vec{a} \stackrel{\text{def}}{=} a_1, a_2, \dots, a_n.$$

$$\vec{A} \stackrel{\text{def}}{=} A_1, A_2, \dots, A_n.$$

$$\vec{a} \in \vec{A} \text{ abbreviates } a_1 \in A_1 \text{ and } a_2 \in A_2[a_1/x_1] \text{ and } \dots \\ \text{and } a_n \in A_n[a_1, \dots, a_{n-1}].$$

$$\tilde{a} \stackrel{\text{def}}{=} a[a_1/x_1, \dots, a_n/x_n].$$

$$\tilde{a}^{-x} \stackrel{\text{def}}{=} \text{if } x \text{ is } x_i \text{ for some } i, \text{ then} \\ a[a_1/x_1, \dots, a_{i-1}/x_{i-1}, a_{i+1}/a_{i+1}, \dots, a_n/x_n] \\ \text{else } a[a_1/x_1, \dots, a_n/x_n].$$

where a_1, a_2, \dots, a_n are closed expressions, A_1 is closed, only x_1 occurs free in A_2, \dots , only x_1, \dots, x_{n-1} occurs free in A_n .

To prove

$$x_1:A_1, \dots, x_n:A_n \mid > b \in B$$

thus amounts to showing for arbitrary $\vec{a} \in \vec{A}$ that $\tilde{b} \in \tilde{B}$. Substitutions are automatically factored into expressions; we use the notation \tilde{a}^{-x} to indicate x is free in a and should not be substituted for. Thus, $(\lambda x.b)[a_1/x_1, \dots, a_n/x_n]$ is $\lambda x.\tilde{b}^{-x}$.

4.5.2 Proof of consistency

We may now prove each of the rules is correct.

THEOREM 71 *The rules of the theory of chapter 3 are valid under the interpretation σ_ω .*

PROOF. The rules will be proved in the order they are given in chapter 3.

Universes

[U form] Arbitrary $\vec{a} \in \vec{A}$; show $U_\alpha \in U_\beta$ for $\alpha < \beta$. This follows directly from lemma 44.

[U cumulativity] Arbitrary $\vec{a} \in \vec{A}$; show $\tilde{A} \in U_\alpha$. By instantiating the subgoal with \vec{a} , we get $\tilde{A} \in U_\beta$ with $\alpha > \beta$. $\tilde{A} \in U_\alpha$ follows directly from lemma 37.

Is

[is form] Use lemma 39.

[is reflex] Use lemmas 11 and 51.

[is sym] Arbitrary $\vec{a} \in \vec{A}$; A_i is \tilde{b} is \tilde{a} ; show $c \in \tilde{a}$ is \tilde{b} for arbitrary c . Since \tilde{a} is closed, \tilde{a} is \tilde{a} ; a similar argument applies to \tilde{b} . $a_i \in \tilde{b}$ is \tilde{a} , so $\tilde{b} =_\alpha \tilde{a}$ by lemma 51. By the symmetry of $=_\alpha$, $\tilde{a} =_\alpha \tilde{b}$, so $c \in \tilde{a}$ is \tilde{b} .

[is trans] Similar to [is sym] proof.

[is contradiction] Arbitrary $\vec{a} \in \vec{A}$; a_i is \tilde{a} is \tilde{b} ; show $\tilde{c} \in \tilde{C}$. By the condition imposed on the rule, \tilde{a} and \tilde{b} are not α -convertible, so the type \tilde{a} is \tilde{b} cannot be inhabited. The assumption list is thus contradictory, so anything follows.

[is subst] Arbitrary $\vec{a} \in \vec{A}$; A_i is \tilde{a} is \tilde{b} ; show $\tilde{c}^{-x}[\tilde{a}/x] \in \tilde{C}^{-x}[\tilde{a}/x]$. By the subgoal, $\tilde{c}^{-x}[\tilde{b}/x] \in \tilde{C}^{-x}[\tilde{b}/x]$; since a and b are α -variants, $\tilde{c}^{-x}[\tilde{a}/x] =_\alpha \tilde{c}^{-x}[\tilde{b}/x]$ and $\tilde{C}^{-x}[\tilde{a}/x] =_\alpha \tilde{C}^{-x}[\tilde{b}/x]$. Membership respects α -conversion, so $\tilde{c}^{-x}[\tilde{a}/x] \in \tilde{C}^{-x}[\tilde{a}/x]$.

[is decision] Arbitrary $\vec{a} \in \vec{A}$; show $\tilde{c} \in \tilde{C}$. $\tilde{a} =_\alpha \tilde{b}$ is either true or false; if it is true, $\tilde{c} \in \tilde{C}$ follows by the first subgoal, and if $\tilde{a} =_\alpha \tilde{b}$ is false, by the second subgoal.

Evaluation

[eval form] Use lemma 39.

[eval value] Use lemma 13.

[eval idemp] Use lemma 14.

[eval unique] Use lemma 15.

[eval app] [eval proj left], [eval proj right], [eval succ], [eval pred], [eval if_zero true], [eval if_zero false], [eval fix], and [eval seq] follow directly from the definition of evaluation, definition 12.

[eval app unique] Arbitrary $\vec{a} \in \vec{A}$; A_i is $\tilde{f}(\tilde{a})$ eval \tilde{v} , and we need to show $\tilde{b} \in \tilde{B}$. $\tilde{b} \in \tilde{B}$ follows from the third subgoal if we can show $\tilde{f}'(\tilde{a})$ eval \tilde{v} . Since $\tilde{f}(\tilde{a}) \mapsto \tilde{v}$, $\tilde{f} \mapsto \lambda x.b$ for some b and $b[\tilde{a}/x] \mapsto \tilde{v}$. From the first two subgoals, \tilde{f} and \tilde{f}' both evaluate to the same \tilde{v}' ; \tilde{v}' is thus $\lambda x.b$, so $\tilde{f}' \mapsto \lambda x.b$ and $\tilde{f}'(\tilde{a}) \mapsto \tilde{v}$.

[eval proj left unique] Arbitrary $\vec{a} \in \vec{A}$; A_i is $\tilde{p}.1$ eval \tilde{v} , show $\tilde{b} \in \tilde{B}$. Since $\tilde{p}.1 \mapsto \tilde{v}$, $\tilde{p} \mapsto \langle a, b \rangle$ and $a \mapsto \tilde{v}$. Instantiating x with a and y with b in subgoal one, \tilde{p} eval $\langle a, b \rangle$ and $a \mapsto \tilde{v}$ follow by assumption, so $\tilde{b} \in \tilde{B}$. The other uniqueness rules, [eval proj right unique], [eval succ unique], [eval pred unique], [eval if_zero true unique], [eval if_zero false unique], [eval fix unique], and [eval seq unique] have similar proofs.

Equivalence

[equiv exp] Arbitrary $\vec{a} \in \vec{A}$; show $\tilde{a} \in \tilde{A}$. From subgoals 3 and 4, we have $\tilde{a} \mapsto \tilde{v}$ iff $\tilde{a}' \mapsto \tilde{v}$, so $\tilde{a} \cong \tilde{a}'$. By subgoal one, $\tilde{a}' \in \tilde{A}$; therefore, by lemma 38, $\tilde{a} \in \tilde{A}$. The proofs for [equiv type] and [equiv hyp] are similar.

Inducement

[ind form] Use lemma 39.

[ind trans] Use lemma 20.

[ind eval] Arbitrary $\vec{a} \in \vec{A}$; A_i is \tilde{a} eval \tilde{b} , need to show $\tilde{a} \succ \tilde{b}$. By the subgoal, $\tilde{a} \neq_\alpha \tilde{b}$. Since $\tilde{a} \mapsto \tilde{b}$, \tilde{a} must be a computation, so we may apply lemma 21 to give $\tilde{a} \succ \tilde{b}$.

[ind app], [ind app arg], [ind proj left], [ind proj right], [ind proj left arg], [ind proj right arg], [ind succ], [ind pred], [ind if_zero true], [ind if_zero false], [ind if_zero arg], [ind fix], [ind seq], and [ind seq arg] can be read directly off of the definition of \succ_1 , definition 17.

Termination

[terminate form] Use lemma 39.

[terminate intro], [terminate elim] These rules follow directly from the definition of termination, definition 16.

[terminate ind] Use lemma 22.

[terminate total] Use lemmas 52, 54, 55, and 35.

Membership

[member form] Use lemma 40.

[member intro] and [member elim] follow directly from lemma 53.

Expression

[E form] Use lemma 39.

[E intro] Use lemma 51.

Natural numbers

[N form] Use lemma 39.

[N intro] Use lemma 52.

[N succ] Arbitrary $\vec{a} \in \vec{A}$; show $\text{succ}(\tilde{n}) \in \mathbb{N}$. By the subgoal, $\tilde{n} \in \mathbb{N}$, so by lemma 52, $\tilde{n} \mapsto m$ for m one of $0, 1, 2, \dots$. Therefore by the definition of evaluation, $\text{succ}(\tilde{n}) \mapsto m'$ where m' is one more than m . Thus $\text{succ}(\tilde{n}) \in \mathbb{N}$. [N pred] has a similar proof.

[N if_zero] Arbitrary $\vec{a} \in \vec{A}$; show $\text{if_zero}(\tilde{n}; \tilde{a}; \tilde{b}) \in \vec{A}$. By subgoal one, $\tilde{n} \in \mathbb{N}$, so $\tilde{n} \mapsto m$ for m one of $0, 1, 2, \dots$.

CASE m is 0: $\tilde{n} \mapsto 0$, so we may instantiate subgoal two to get $\tilde{a} \in \vec{A}$; since membership respects computational equivalence, $\text{if_zero}(\tilde{n}; \tilde{a}; \tilde{b}) \in \vec{A}$.

CASE m is not 0: This case is similar.

[N ind func] We need to show $\text{fix}(f)(n) \in \mathbb{N}$ for all $n \in \mathbb{N}$; this follows by straightforward induction on n .

Dependent function space

[func form] Arbitrary $\vec{a} \in \vec{A}$; show $x:\vec{A} \rightarrow \vec{B}^{-x} \in U_\alpha$. By lemma 41, this means showing

$$(1) \quad \vec{A} \in U_\alpha \text{ and for all } a \in \vec{A}, \vec{B}^{-x}[a/x] \in U_\alpha.$$

Subgoal one proves $\vec{A} \in U_\alpha$, and subgoal two proves for all $a \in \vec{A}$, $\vec{B}^{-x}[a/x] \in U_\alpha$.

[func intro] Arbitrary $\vec{a} \in \vec{A}$; show $\lambda x.\vec{b}^{-x} \in \vec{A} \rightarrow \vec{B}^{-x}$. By lemma 54, this is equivalent to showing

$$(2) \quad x:\vec{A} \rightarrow \vec{B}^{-x} \text{ Type and}$$

$$(3) \quad \text{for all } a \in \vec{A}, \vec{b}^{-x}[a/x] \in \vec{B}^{-x}[a/x].$$

To prove (2), we need to show the equivalent of (1) above: by subgoal two, $\vec{A} \in U_\alpha$, and by subgoal one,

$$(4) \quad \text{for all } a \in \vec{A}, \vec{b}^{-x}[a/x] \in \vec{B}^{-x}[a/x].$$

Thus, for all $a \in \tilde{A}$, $\tilde{B}^{-x}[a/x] \in U_\beta$ for some β by lemma 34. Letting γ be $\max\{\alpha, \beta\}$, $x:\tilde{A} \rightarrow \tilde{B}^{-x} \in U_\gamma$ by lemma 37. Therefore, $x:\tilde{A} \rightarrow \tilde{B}^{-x}$ Type. (3) is exactly (4), which we already mentioned follows directly from subgoal one.

[func lam] Arbitrary $\tilde{a} \in \tilde{A}$; show $\tilde{f} \in \tilde{A} \rightarrow \tilde{B}^{-x}$. By subgoal one, $\tilde{f} \in y:\tilde{C} \rightarrow \tilde{D}^{-y}$, so by lemma 54, $\tilde{f} \mapsto \lambda z.d$ for some z and d . Thus, by lemma 38, we need only to show $\lambda z.d \in \tilde{A} \rightarrow \tilde{B}^{-x}$. The rest of the proof is similar to the proof for [func intro].

[func elim] Arbitrary $\tilde{a} \in \tilde{A}$; show $\tilde{c} \in \tilde{C}$. By subgoal three, we have

$$\text{for all } b \in \tilde{B}^{-x}[\tilde{a}/x], \tilde{F}(\tilde{A}) =_\alpha b \text{ implies } \tilde{c} \in \tilde{C}.$$

Instantiating b with $\tilde{F}(\tilde{A})$, $b \in \tilde{B}^{-x}[\tilde{A}/x]$ because $\tilde{f} \in x:\tilde{A} \rightarrow \tilde{B}^{-x}$ by subgoal one and $\tilde{a} \in \tilde{A}$ by subgoal two, and with these two facts, lemma 54 implies $\tilde{f}(\tilde{a}) \in \tilde{B}^{-x}[\tilde{a}/x]$. $\tilde{f}(\tilde{a}) =_\alpha \tilde{f}(\tilde{a})$ follows trivially, so $\tilde{c} \in \tilde{C}$.

Dependent product space

[prod form] This parallels the [func form] rule.

[prod intro] Arbitrary $\tilde{a} \in \tilde{A}$; show $\langle \tilde{a}, \tilde{b} \rangle \in x:\tilde{A} \times \tilde{B}^{-x}$. By lemma 55, this is equivalent to showing

$$x:\tilde{A} \times \tilde{B}^{-x} \in U_\alpha \text{ and } \tilde{a} \in \tilde{A} \text{ and } \tilde{b} \in \tilde{B}^{-x}[\tilde{a}/x].$$

By subgoal one, $\tilde{a} \in \tilde{A}$, so $\tilde{A} \in U_\beta$ for some β by lemma 34. By subgoal three, for all $a' \in \tilde{A}$, $\tilde{B}^{-x}[a'/x] \in U_\alpha$. Therefore, by lemma 42, $x:\tilde{A} \times \tilde{B}^{-x} \in U_\gamma$, where γ is $\max\{\alpha, \beta\}$.

All that remains to show is $\tilde{a} \in \tilde{A}$ and $\tilde{b} \in \tilde{B}^{-x}[\tilde{a}/x]$: these two facts follow directly from subgoals one and two, respectively.

[prod elim] Arbitrary $\tilde{a} \in \tilde{A}$; show $\tilde{c} \in \tilde{C}$. From subgoal two we have

$$(5) \quad \text{for all } a \in \tilde{A} \text{ and } b \in \tilde{B}^{-x}[a/x], \tilde{p} \mapsto \langle a, b \rangle \text{ implies } \tilde{c} \in \tilde{C}.$$

Since by subgoal one $\tilde{p} \in x:\tilde{A} \times \tilde{B}^{-x}$, $\tilde{P} \mapsto \langle a, b \rangle$ for some a and b . Instantiating subgoal two with a for x and b for y , $a \in \tilde{A}$, $b \in \tilde{B}^{-x}[a/x]$ and $\tilde{p} \mapsto \langle a, b \rangle$ are all true, so $\tilde{c} \in \tilde{C}$.

Partial types

[bar form] Use lemma 43.

[bar intro] Arbitrary $\vec{a} \in \vec{A}$; show $\tilde{a} \in \overline{\vec{A}}$. By lemma 56, this is equivalent to showing $\vec{A} \in \overline{U_\alpha}$ and $\tilde{a} \downarrow$ implies $\tilde{a} \in \vec{A}$. These two facts follow directly from subgoals two and one, respectively.

[bar elim] Arbitrary $\vec{a} \in \vec{A}$; show $\tilde{a} \in \vec{A}$. By subgoals one and two, $\tilde{a} \downarrow$ and $\tilde{a} \in \overline{\vec{A}}$, so by lemma 56, $\tilde{a} \in \vec{A}$.

Fixed-point

[fix] In section 4.4, we showed that all of the candidate-admissible types (which are exactly the V types) are admissible; thus, if $A \in V$, A AType by theorem 69. This means for all $f \in \vec{A} \rightarrow \vec{A}$, $fix(f) \in \vec{A}$. The correctness of the rule is an immediate consequence of this.

[V form] Use lemma 50.

[V sub] this follows by lemma 36.

The proofs of correctness of the rules for forming V types, [`<type> form adm`], are very similar to the proofs of the rules for forming U_α types, except they use lemmas about candidate-admissible types, lemmas 45 through 49.

Computational induction

[comp ind] Arbitrary $\vec{a} \in \vec{A}$; A_i is \tilde{a} halts. Letting

$$F(t) \stackrel{\text{def}}{=} fix(\lambda h. \lambda a''. \lambda v. \tilde{e}[t/a'])(t)(0),$$

show

$$(1) \quad F(\tilde{a}) \in \tilde{P}[\tilde{a}/x].$$

By the computational induction theorem (theorem 25), it suffices to show

$$(2) \quad \begin{aligned} &\tilde{a} \downarrow \text{ and for all } a', a' \downarrow \text{ and} \\ &\quad (\text{for all } a'', a' \succ a'' \text{ implies } F(a'') \in \tilde{P}[a''/x]) \\ &\quad \text{implies } F(a') \in \tilde{P}[a'/x]. \end{aligned}$$

$\tilde{a} \downarrow$ by assumption; let a' be an arbitrary expression such that $a' \downarrow$ and

$$(3) \quad \text{for all } a'', a' \succ a'' \text{ implies } F(a'') \in \tilde{P}[a''/x];$$

show $F(a') \in \tilde{P}[a'/x]$. Subgoal one states

(4) $a' \downarrow$ and $h \in (a'' : E \rightarrow a' \text{ ind } a'' \rightarrow \tilde{P}[a'/x])$ implies $\tilde{e} \in \tilde{P}[\tilde{a}/x]$

for arbitrary a' and h . Letting a' in (4) be a' , $a' \downarrow$ follows by assumption; let h be $F(a')$; by computation and the definition of \rightarrow , to show h is in the proper type is to show

(5) for all a'' , $a' \succ a''$ implies $\tilde{e} \in \tilde{P}[a''/x]$.

$\tilde{e} \cong F(a'')$, so (5) follows by (3). Thus, from (4) we obtain $\tilde{e} \in \tilde{P}[\tilde{a}/x]$; by computational equivalence, this means $F(a') \in \tilde{P}[a'/x]$, what we needed to show.

Miscellaneous

[cut] Arbitrary $\vec{a} \in \vec{A}$; show $\tilde{a} \in \tilde{A}$. By the first subgoal, $\tilde{b} \in \tilde{B}$; instantiating the second subgoal with \tilde{b} for x gives $\tilde{a} \in \tilde{A}$.

[hyp] Trivial.

[hyp prop] Arbitrary $\vec{a} \in \vec{A}$; this means $a_i \in \tilde{B}$, where \tilde{B} is an atomic propositional type. $c \in \tilde{B}$ for arbitrary c follows directly from lemma 51.

[alpha] This rule follows from the respect of \in and sequent truth for α -conversion.
QED.

All of the rules are true under this interpretation, so since 0 is 1 is uninhabited semantically, it is not provable using the rules.

Chapter 5

Topics in partial object type theory

This chapter is a loose collection of consequences of partial object type theory. Type theory can now be viewed as a full-fledged programming logic since we may reason in a facile manner about partial objects as well as total ones. LCF has proven itself to be a viable programming logic, and we show that partial object type theory is also viable by showing that it compares favorably with LCF.

The fixed-point principle and computational induction are the two principles which give reasoning power to partial object type theory. The two are more closely related than it would first appear: the fixed-point principle is also an induction principle, and computational induction is also a principle for typing fixed-points.

Some other consequences discussed are a new notion of proposition which gives a novel form of logical reasoning, and some results that show there are unsolvable problems in the theory.

We conclude with remarks about alternate ways of developing partial types. Since the theory presented in chapter 3 is not the final word, these remarks should give future designers something to work from.

5.1 Type theory as a programming logic

Programming logics are, as might be guessed, logics for reasoning about programs. Hoare pioneered this field in his development of an axiomatic assertion-based logic for reasoning about programs [Hoa69]. The assertion $\{P\} A \{Q\}$ means if P is true

and program A is executed to completion, facts Q then hold. The program may alter variables used in P and Q , so these assertions are for reasoning about state-based systems of programming. Much has been written about how this style of reasoning can be used to develop correct programs; for instance, see [Gri81]. In the PL/CV system [CO78], state-based programs are labeled with assertions and proofs that the programs meet the assertions; a verifier then can check that the programs are correct. This is an example of a system which uses the computer to aid in reasoning about programs.

Other efforts have concentrated on reasoning about functional programs: functional programs are more mathematically concise and thus easier to reason about. DeBakker and Scott and Park amongst others interpreted computable functions as continuous mathematical functions; this approach inspired the development of new and more powerful induction principles for reasoning about functional programs (Manna gives a review of these early results [Man74]). This work was soon followed by Scott and Strachey's revolutionary treatment of programs as continuous functions over lattices [SS71]. This approach has since become a standard mathematical interpretation of programming languages [Sco76, Sto77]. Milner conceived of Edinburgh LCF [GMW79] as a formal system for reasoning about continuous functions over a complete partial order.

Type theory may be used to reason about programs [Nor81, How88a, Moh86], but it cannot be compared directly with programming logics like LCF because it deals poorly with partial objects. Partial object type theory, on the other hand, is on a more equal footing, and it is informative to compare it with LCF. Before we do this, however, it is worthwhile to consider what in general makes good programming logic.

5.1.1 What makes good programming logic?

A simple but powerful notion of programming is easiest to understand, and extra features must be critically important to the language if the increased complexity in reasoning is to be justified; the functional style of programming is thus a good choice. Typing greatly aids writing and reasoning about programs. In fact, typing a program makes an assertion about its meaning, and so is a form of reasoning about programs. The more expressive types are, the more possible it is to encode information about the program behavior in the type.

The logic should be expressive, because it is also easier to reason (in the sense that proofs will be shorter) in a more powerful logic. The logic must allow for assertions about the behavior of programs to be made. Most importantly, there must be tech-

niques for proving functions meet some specification, i.e. proving partial and total correctness. To this end it must be possible to perform induction on the structure of the domain of the function (for total correctness), and to perform induction on the structure of the function execution (for partial correctness). There should also be facilities for reasoning about equivalences between programs.

With these points in mind, we may proceed to the comparison of LCF and partial object type theory.

5.1.2 LCF and type theory compared

The computation language of the two is similar: there are λ -expressions for functions, a fixed-point constructor to represent recursive programs, and some simple atomic elements (booleans `tt` and `ff` in LCF, and numbers in type theory).

LCF has a simple polymorphic type system, of the sort that its metalanguage ML has. One advantage of this simple type system is that the typing of computations is decidable. In type theory, the typing of computations is not decidable, but to compensate there is a much richer type system, illustrated by the list of types found in section 2.2.1. This means it is possible to place more information about program behavior in the type information. Using the subtype constructor, it is possible for the complete program specification to be given in the type, so a program becomes typable if and only if it meets its specification; for instance,

$$x:\mathbb{N} \rightarrow \{y:\mathbb{N} \mid y = x * x \in \mathbb{N}\}$$

is the type of all functions that compute squares. The property of being a partial or total function is easily expressed by the type alone: $\mathbb{N} \rightarrow \mathbb{N}$ is the type of total functions, and $\overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$ is one type of partial functions. In LCF, all functions are typed as partial functions, and totality is a property which may be proved about them. The ability to place this information in the type is a big advantage of partial object type theory.

Partial functions are proved total by the same technique in both theories: induction is performed on some well-founded ordering; if recursive calls are smaller by this ordering than the value the function was called on, the function terminates. In other approaches to proving totality [BM79, Gri81], the function is labeled with a well-ordering that shows the function to be total; here the well-founded ordering is part of the proof of totality.

Type theory has a substantially more powerful logic than LCF, one reason being that it is possible to quantify over predicates. For some domains of reasoning, this

extra expressive power is important. However, LCF is classical and type theory is constructive, so it is difficult to compare reasoning procedures in the two theories directly. A case in point is that in constructive logic it is more difficult to prove termination, because $\neg t\uparrow$ does not necessarily imply $t\downarrow$. However, the inducement property in type theory may be used to compensate for this: if a induces b and $a\downarrow$, then $b\downarrow$ also. To prove this fact in LCF, we would assume b diverged, and show that a then diverged, which would be a contradiction, meaning (by excluded middle), that b converged.

It is difficult to draw any quantitative conclusions from the above informal discussion, but it can be said that partial object type theory compares favorably with LCF. A comparison of how one may reason by induction in the two theories is part of a larger story which is told in the next section.

5.2 Fixed-points and induction

A different light is shed on fixed-point induction from the perspective of partial object type theory. We show that fixed-point induction is a special case of the fixed-point principle. Surprisingly, we also find that computational induction may be used to type fixed-points. What emerges from these two facts is that the fixed-point principle and computational induction are both general principles for typing and reasoning about computations. Computational induction is a real induction principle, because it is an induction on a well-founded ordering. Fixed-point induction is not really an induction principle; it can be viewed as a metaprinciple about what facts are provable via computational induction. What emerges from these comparisons is a clearer picture of what it means to reason about computations.

5.2.1 A unified fixed-point principle

Fixed-point induction can be viewed as a special case of the fixed-point principle. Fixed-point induction is used to prove properties P about functions f by induction on the finite unrollings of the function:

$$\frac{P(\perp), \forall f. P(f) \Rightarrow P(F(f))}{P(\text{fix}(F))}$$

If the predicate P is true for all finite approximations to the fixed-point, this rule implies P is true of the fixed-point. This is not a real induction principle, because it

is not valid for all predicates P , but only for *admissible* P 's. It is possible for P to be true at all approximations, but to fail to hold at the limit; for example,

$$\begin{aligned} F &\stackrel{\text{def}}{=} \lambda f. \lambda x. \text{if } x = 0 \text{ then } 0 \text{ else } f(x - 1) \\ P(\text{fix}(F)) &\stackrel{\text{def}}{=} \exists x. f(x) = \perp. \end{aligned}$$

$P(\overbrace{F(\dots F(\perp)\dots)}^k)$ is true for all k , but $P(\text{fix}(F))$ is false. Syntactic restrictions are then placed on formulas P with which fixed-point induction may be used. In practice, many formulas of interest are admissible. Sometimes, however, the proposition needs to be re-phrased to make it admissible. Other times, no admissible version can be found.

Fixed-point induction can be derived in type theory using the fixed-point principle. Versions that are similar to fixed-point induction are derivable in many formulations of type theory, but to get the exact principle, we need to assume that the theory has subtypes, and to accept the classical fact that computations either converge or diverge.

Suppose $F \in (\overline{A \rightarrow \overline{B}}) \rightarrow (\overline{A \rightarrow \overline{B}})$, and $\text{fix}(F) \in A \rightarrow \overline{B}$. Define

$$S \stackrel{\text{def}}{=} \{f: A \rightarrow \overline{B} \mid P(f)\}.$$

If $\text{fix}(F) \in \overline{S}$ can be shown, we have as a consequence that $P(\text{fix}(F))$ is true.

THEOREM 72 *We may derive the following rule:*

$$\begin{array}{l} |> \text{fix}(F) \in \overline{S} \\ |> F \in (\overline{A \rightarrow \overline{B}}) \rightarrow (\overline{A \rightarrow \overline{B}}) \\ |> P(F(\uparrow)) \\ f: A \rightarrow \overline{B}, P(f) \quad |> P(F(f)) \end{array}$$

provided \overline{S} is admissible.

PROOF. Given the subgoals hold, show $\text{fix}(F) \in \overline{S}$. Applying the fixed-point principle and a few more proof steps gives the subgoal

$$f: \overline{S} \quad |> P(F(f)).$$

Proceed by cases on $f \downarrow \vee f \uparrow$. If $f \uparrow$, we need to show $P(F(\uparrow))$; this is a hypothesis. If $f \downarrow$, we may remove the bar from the hypothesis, giving the sequent

$$f: A \rightarrow \overline{B}, P(f) \quad |> P(F(f)).$$

This is also a hypothesis.

QED.

The only difference between the derived rule and the fixed-point induction rule is that the base case is $P(F(\uparrow))$ instead of $P(\uparrow)$. The admissibility restriction for fixed-point induction on P is transferred to an admissibility restriction on \bar{S} in the fixed-point principle. Subtypes cause problems similar to the problems caused by dependent products, so a bigger collection of admissible types is needed than the ones found in chapter 3 if this derived rule is to be useful.

The result of this construction is that the fixed-point principle emerges as a single method for proving properties about fixed-points, be it just typing them, or performing more complex inductive reasoning.

5.2.2 Computational induction reconsidered

We now show how the computational induction principle can be used to type fixed-points. Consider the following example:

$$F \stackrel{\text{def}}{=} \lambda f. \lambda x. \text{if_zero}(x; 1; f(x - 1)) \in (\mathbb{N} \rightarrow \bar{\mathbb{N}}) \rightarrow (\mathbb{N} \rightarrow \bar{\mathbb{N}}).$$

LEMMA 73 *Without use of the fixed-point principle, we may prove $\text{fix}(F) \in \mathbb{N} \rightarrow \bar{\mathbb{N}}$.*

PROOF. Pick arbitrary $n \in \mathbb{N}$, suppose $\text{fix}(F)(n) \downarrow$; we need to show $\text{fix}(F)(n) \in \mathbb{N}$. Applying computational induction gives the subgoal

$$\begin{array}{l} n:\mathbb{N}, \text{fix}(F)(n) \downarrow, \forall n':\mathbb{N}. \text{fix}(F)(n) \text{ ind } \text{fix}(F)(n') \Rightarrow \text{fix}(F)(n') \in \mathbb{N} \\ |> \text{fix}(F)(n) \in \mathbb{N} \end{array}$$

which by computing amounts to showing

$$|> \text{if_zero}(n; 1; \text{fix}(F)(n - 1)) \in \mathbb{N}.$$

CASE $n = 0$: Trivial.

CASE $n \neq 0$: We wish to show

$$|> \text{fix}(F)(n - 1) \in \mathbb{N},$$

which follows from the induction hypothesis upon instantiating n' by $n - 1$.

QED.

This technique allows a wide range of functions to be typed, but there is no simple way to describe them. Fixed-points are typed by making sure all computation paths

are typed; if it is impossible to describe all computation paths, the function will not be typable. For example, if in the above function we changed the base case from 1 to $g(f)(x)$ for some function g , we would have to examine how g computed to make sure $g(f)(x) \in \mathbb{N}$. If the range type of the function has nested bars, nested uses of computational induction may be required to type the function.

5.2.3 Two principles compared

We have seen that the fixed-point principle and the computational induction principle are both general principles for typing and reasoning about computations. It would thus be possible to build a usable partial object type theory that incorporated only one of the two.

This then raises the question of how these two principles compare to one another. The two techniques are used quite differently; notice for example that the previous proof is much longer than the trivial proof of the same fact using the fixed-point principle.

Computational induction is a real induction principle, for there is an underlying well-founded structure of induced computations that induction is being performed on. The fixed-point principle, on the other hand, is more properly viewed as a metaprinciple: for certain types, a metatheorem shows that fixed-points always exist. Without such a metatheorem, it would be difficult to explain why some types had fixed-points and other didn't. One advantage is that because the fixed-point principle is a metaprinciple, it reflects some metafacts into the theory that would otherwise not be there. This means it is sometimes more powerful than computational induction. For example, the statement $\forall f: \overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}. \text{fix}(f) \in \overline{\mathbb{N}}$ is provable with fixed-point induction, but not with computational induction, because it is impossible to analyze the structure of an arbitrary function in the theory; to type a fixed-point using computational induction requires analyzing how the function computes.

Fixed-point induction also has weaknesses: if any partial function $f \in \overline{A} \rightarrow \overline{A}$ has no typable fixed-point, then the type A is not admissible for fixed-point induction, even though some other function $g \in \overline{A} \rightarrow \overline{A}$ may have a typable fixed-point. $\text{fix}(g)$ could perhaps still be typed using computational induction. Another problem with admissibility restrictions is that the collection of admissible types is never complete, so there is always the chance that a sensible result cannot be proven because of arbitrary admissibility restrictions.

The fixed-point principle is difficult to justify foundationally. Metaprinciples about open-ended computation systems are in general difficult, because it is hard

to formulate metaprinciples about something we do not completely know. Computational induction is foundationally sound, however, because inducement is a basic property of computation.

5.3 Partial propositions

The types $x:A \rightarrow B$, $x:A \times B$, $A \rightarrow B$, $A \times B$, and $A + B$ have been shown to be useful in representing propositions. What is the meaning of \overline{A} as a proposition? Such propositions have enough interesting properties that we will give them a name, *partial propositions*. \overline{A} is trivially true, because $\uparrow \in \overline{A}$. However, if $a \in \overline{A}$ and $a \downarrow$, then $a \in A$, so A is true. If we can potentially show termination of the validation, proving a partial proposition would then be a useful exercise.

Consider the type $P \stackrel{\text{def}}{=} \forall x:\mathbb{N}. \overline{\exists y:\mathbb{N}. y = x * x}$. If $p \in P$ and we know for some particular $n \in \mathbb{N}$ that $p(n) \downarrow$, then $p(n)$ validates the unbarred proposition $\exists y:\mathbb{N}. y = n * n$. This makes p potentially useful. Validations for propositions are always total objects, whereas validations for partial propositions are partial objects. Propositions are thus to total correctness as partial propositions are to partial correctness; since we know partial correctness to be a useful notion, partial propositions should also be useful. The reasons why it would be desirable to prove a partial proposition are the same as the reasons for proving partial correctness: the total proposition is either intractable or unprovable. It is unknown whether there are an infinite number of pairwise primes, that is, pairs of primes of the form $p, p + 2$. We thus could not prove

$$\forall n. \exists m. m, m + 2 \text{ are the } n\text{-th set of pairwise primes.}$$

However, we may prove

$$p \in \forall n. \overline{\exists m. m, m + 2 \text{ are the } n\text{-th set of pairwise primes}}$$

by defining $p(n)$ to search through primes for the n -th pair. If the computation of $p(100)$ terminated, it would give a validation for the existence of 100 pairwise primes.

Proofs of partial propositions could also involve induction over a structure not known to be well-founded. Using the fixed-point principle, we may derive the following non-well-founded induction principle, written in extract form:

$$\begin{aligned} | > \forall x:A. \overline{B(x)} (\text{extract } fx(\lambda h. \lambda x. b)) \\ h:(\forall x:A. \overline{B(x)}), x:A \quad | > \overline{B} (\text{extract } b) \end{aligned}$$

for admissible B . We may use the fact $\forall x:A. \overline{B(x)}$ in the proof of itself, a proof-theoretic analogue of recursion.

An important feature of Nuprl is that it is possible to program by proving a statement and extracting its constructive content; this is discussed in section 2.3.1. In Nuprl, it is only possible to develop total programs in this manner, because the constructions must always terminate. Proving partial propositions will give an extraction which is a partial program, so in partial object type theory we may also develop partial functions using the extraction paradigm. The non-well-founded induction principle above is one principle for proving a partial proposition which will extract an unbounded computation.

Partial propositions extend the collection of statements that can be made when reasoning constructively, giving a more expressive logic. Since there is no construction to validate statements in classical logic, the notion of partial proposition makes no sense in a classical theory.

5.4 Abstract computability theory

One of the surprising results that emerges from partial object type theory is that it is possible to prove the existence of unsolvable problems [CS88]. Since the computation system is taken to be open-ended, the computations cannot be indexed, so these unsolvable problems must be shown to exist by means other than the standard ones found in [HR67] or [Soa87]. Other researchers have studied abstract treatments of recursion theory [Wag69, Str68, Fri69]; the approach here is a different abstract view.

It is not possible to solve the “halting problem”, i.e. it is impossible to tell, for arbitrary $t \in \overline{\mathbb{N}}$, whether t halts. That is,

THEOREM 74 *There can be no function $h \in \overline{\mathbb{N}} \rightarrow \mathbb{N}$ such that $h(t) = 1$ iff t halts.*

PROOF. Assume that h exists; then we can define the function

$$d = \text{fix}(\lambda x. \text{if_zero}(h(x); 1; \uparrow)) \in \overline{\mathbb{N}}.$$

$d \in \overline{\mathbb{N}}$ follows by the fixed-point rule because the body is in the type $\overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$. By computing the fix term, we have $d = \text{if_zero}(h(d); 1; \uparrow) \in \overline{\mathbb{N}}$. If $h(d) \neq 1$, then d should diverge, but in fact $d = 1$; so it converges. We reach a similar contradiction if $h(d) = 1$. So the assumption that h exists is in error. The argument depends essentially on the self-referential nature of $\text{fix}(f)$. This argument cannot be valid in a classical type theory which takes $A \rightarrow B$ to denote the type of all classical functions

from A into B , because in that case there surely is a function $\overline{N} \rightarrow \mathbb{N}$ solving the halting problem. This argument thus shows that the type-theoretic notion of partial function is different from the classical notion.

Other results, including Rice's theorem, notions of reducibility and the existence of complete sets can be developed in this context. By adding new notions of computation such as the ability to dovetail computations or the ability to count computation steps, it is possible to prove more theorems; each new notion of computation gives rise to a new cluster of theorems. In the standard development of recursion theory there is no such clustering because once a universal machine is defined, all forms of computation may be simulated. A close study of this clustering phenomenon could give more insights into the nature of computation.

5.5 Building a partial object type theory

There are features of partial object type theory that impose special problems in the design of a theory that includes them. It is not necessary to have both a fixed-point principle and computational induction but both have advantages, so it is not clear whether it would be acceptable to remove one of the principles. We consider what difficulties each of them poses in the construction of a theory.

5.5.1 Expressing computational induction

First, let us consider computational induction. The key to expressing computational induction is expressing inducement. If we want a mathematical interpretation of the theory, we cannot interpret inducement, so computational induction must be left out.

In a mathematical interpretation (defined in section 2.2.5), computations are interpreted as mathematical objects and the actual computation process is lost. We don't wish to go into the details of how partial objects can be given a mathematical interpretation; what we wish to show instead is how certain problems arise in *any* mathematical interpretation.

The theory of chapter 3 can have no mathematical interpretation because it is an explicitly computational theory. The assertions $a \text{ is } b$, $a \text{ eval } b$, and $a \text{ ind } b$ express intensional properties of computations which are not present in mathematical interpretations. The question is, what sort of theory can be formulated if we wish to have a mathematical interpretation?

The direct way which it is possible to reason about computation in chapter 3 would not be possible, but the greatest loss is that computational induction cannot be expressed because inducement cannot be expressed. However, the fixed-point principle could be used to prove facts by induction. We can thus tentatively conclude that a theory could be formulated that has a mathematical interpretation, but that sacrifices would be made.

If we abandon the possibility of having a mathematical interpretation, the problem of how intensional assertions are to be expressed remains. If they are to be expressed using types as in chapter 3, the theory can have no built-in notion of equality that is nontrivial and a congruence, because such an equality would have to hold across inducement and evaluation:

$$\begin{aligned} a = b \text{ and } c \mapsto b \text{ implies } c \mapsto a \\ a = b \text{ and } b \succ c \text{ implies } a \succ c; \end{aligned}$$

any equality satisfying these requirements must be trivial. There are at least three solutions: the first is presented in chapter 3, accepting the fact that there is no built-in equality. Another solution is found in appendix A: Nuprl already has equality reasoning, so intensional reasoning is carried out by new forms of assertion. However, this solution leaves something to be desired, because we cannot form arbitrary propositions using *ind* and *eval* like “*a ind b* \Rightarrow *a eval b*.” A third solution is to define intensional properties of computation at the metalevel by somehow Gödelizing computations; it is not problematic to perform intensional reasoning over the Gödelizations.

5.5.2 Expressing the fixed-point principle

The fixed-point principle is easier to use than computational induction, but it has at least three disadvantages. First, admissibility conditions are hard to implement. In chapter 3, a separate universe V was used for admissible types. However, if we wished to have a more refined collection of admissible types, the collection of rules for V would become large. Second, admissibility conditions are arbitrary, so some proofs will fail because the type is not admissible; this is an undesirable feature for any logic to have. Third, the fixed-point principle is difficult to justify foundationally.

It is possible to significantly extend to collection of admissible types, making the fixed-point principle a powerful technique for reasoning about functions by induction. Only experience with actual proofs will tell how useful computational induction and the fixed-point principle will ultimately be.

Appendix A

A partial object Nuprl

In this appendix we define new rules for Nuprl which make it a theory for reasoning about partial objects. We add bar types \overline{T} , a convergence predicate $t \text{ in! } T$, a universe of admissible types \mathbb{V} , and new computations $\text{fix}(f)$ and $\text{seq}(a;b)$. Equality on bar types is defined as

$$t = t' \in \overline{T} \text{ iff } \overline{T} \text{ Type and } (t \downarrow \iff t' \downarrow) \text{ and } t \downarrow \Rightarrow t = t' \in T.$$

Several features of the Nuprl theory make it difficult to add rules for reasoning about partial objects; most problematic is the requirement that all types come with a notion of equality which is always respected when the type is used. This makes it more difficult to perform type-free reasoning, because there is no notion of equality that can be placed on type-free assertions like $a \text{ ind } b$ and $a \text{ eval } b$. $t \text{ halts}$ is also cannot be made a sensible type in Nuprl because we cannot say when two such types are equal; however, we can have a typed convergence predicate: $t \text{ in! } T$ means $t \in \overline{T}$ and $t \downarrow$.

It is thus necessary to extend the sequent-style assertions of Nuprl to give the ability to reason about computations. In Nuprl, one form of sequent is used, $H \gg A \text{ ext } a$; for partial objects we need three extensions to this form. Since inducement and evaluation cannot be expressed as types, there are two new sequent forms for reasoning about them:

$$H \gg a \text{ ind } b$$

and

$$H \gg a \text{ eval } b.$$

$a \text{ ind } b$ means that in computing a , b is in turn computed. $a \text{ eval } b$ means a evaluates to b .

We also allow for a new kind of hypothesis to be used in sequents:

$$H, h: [\text{fix}(f)(a) \text{ at } A \text{ over } q, r.P] \gg T$$

This hypothesis expresses the induction hypothesis of computational induction. In the theory of chapter 3 the induction hypothesis can be expressed directly in the logic, so there is no need for a new form of hypothesis to be added.

The rules below extend the current Nuprl theory to be a partial object type theory; only rules for atom, list, and quotient types are missing, but they are straightforward. In comparison to the theory of chapter 3, more rules are needed here to reason effectively about bar types. This is again due to the limits placed on type-free reasoning in Nuprl: facts that are provable in the theory of chapter 3 from other principles may need to be added here as axioms.

A.1 The rules

formation

1. $H \gg \overline{T}$ in U_i by **intro**
 $\gg T$ in \overline{U}_i

canonical

1. $H \gg \text{fix}(f)$ in \overline{A} by **fix_intro**
 $\gg f$ in $\overline{A} \rightarrow \overline{A}$
 $\gg A$ in v

2. $H \gg t$ in T by **canonical**
 $\gg t$ in \overline{T}

Where t is a canonical expression.

3. $H \gg t$ in \overline{T} by **totality**
 $\gg t$ in T

partial int

1. $H \gg -t$ in $\overline{\text{int}}$
 $\gg t$ in $\overline{\text{int}}$
2. $H \gg m \text{ op } n$ in $\overline{\text{int}}$ by intro
 $\gg m$ in $\overline{\text{int}}$
 $\gg n$ in $\overline{\text{int}}$
3. $H \gg \text{ind}(e; x, y.t_d; t_b; x, y.t_u)$ in $\overline{T}[e/z]$
by intro [over $z.\overline{T}$] [new u, v]
 $\gg e$ in $\overline{\text{int}}$
 $u:\text{int}, u < 0, v:T[u+1/z] \gg t_d[u, v/x, y]$ in $\overline{T}[u/z]$
 $\gg t_b$ in $\overline{T}[0/z]$
 $u:\text{int}, 0 < u, v:T[u-1/z] \gg t_u[u, v/x, y]$ in $\overline{T}[u/z]$
4. $H \gg \text{int_eq}(a; b; t; t')$ in \overline{T} by intro
 $\gg a$ in $\overline{\text{int}}$
 $\gg b$ in $\overline{\text{int}}$
 $a=b$ in $\text{int} \gg t$ in \overline{T}
 $(a=b \text{ in } \text{int}) \rightarrow \text{void} \gg t'$ in \overline{T}
5. $H \gg \text{less}(a; b; t; t')$ in \overline{T} by intro
 $\gg a$ in $\overline{\text{int}}$
 $\gg b$ in $\overline{\text{int}}$
 $a < b \gg t$ in \overline{T}
 $(a < b) \rightarrow \text{void} \gg t'$ in \overline{T}

partial union

1. $H \gg \text{decide}(e; x.t_l; y.t_r)$ in $\overline{T}[e/z]$
by intro [over $z.\overline{T}$] using $\overline{A|B}$ [new u, v]
 $\gg e$ in $\overline{A|B}$
 $u:A, e=\text{inl}(u)$ in $A|B \gg t_l[u/x]$ in $\overline{T}[\text{inl}(u)/z]$
 $v:B, e=\text{inr}(v)$ in $A|B \gg t_r[v/y]$ in $\overline{T}[\text{inr}(v)/z]$

partial function

1. $H \gg f(a)$ in $\overline{B}[a/x]$ by intro using $\overline{x:A \rightarrow B}$
 $\gg f$ in $\overline{x:A \rightarrow B}$

>> a in A

partial product

1. H >> $\text{spread}(e; x, y.t)$ in $\overline{T}[e/z]$
 by intro [over $z.\overline{T}$] using $\overline{w:A\#B}$ [new u, v]
 >> e in $\overline{w:A\#B}$
 $u:A, v:B[u/w], e=\langle u, v \rangle$ in $w:A\#B$ >> $t[u, v/x, y]$ in $\overline{T}[\langle u, v \rangle/z]$

partial set

1. H >> a in $\overline{\{x:A|B\}}$ by intro at U_i [new y]
 >> a in \overline{A}
 a in! A >> $B[a/x]$
 $y:A$ >> $B[y/x]$ in U_i
2. $H, u:\overline{\{x:A|B\}}, H'$ >> T ext ($\backslash y.t$)(u) by elim u at U_i [new y]
 $y:A$ >> $B[y/x]$ in U_i
 $y:\overline{A}, [y$ in! $A \rightarrow B[y/x]], u=y$ in \overline{A} >> $T[y/u]$ ext t

Note that the second new hypotheses of the second subgoal is hidden.

partial universe

V is the universe of types that have fixed-points. All of the formation rules except for dependent product formation have V counterparts, just as in the theory of chapter 3; these rules will thus not be listed individually.

1. H >> V in U_2
2. H >> A in U_1
 >> A in V

partial equality

1. H >> $a=a'$ in \overline{A} by equality
 >> $(a$ in! $A) \rightarrow (a'$ in! $A)$
 >> $(a'$ in! $A) \rightarrow (a$ in! $A)$
 $(a$ in! $A), (a'$ in! $A)$ >> $a=a'$ in A

2. $H \gg a=a'$ in A by convergence 1
 $\gg a=a'$ in \overline{A}
 $\gg a$ in A

termination

1. $H \gg (t \text{ in! } T)$ in U_i by intro
 $\gg t$ in \overline{T}
2. $H \gg \text{axiom}$ in $(t \text{ in! } T)$ by intro
 $\gg t \text{ in! } T$
3. $H \gg t \text{ in! } T$ by intro
 $\gg t$ in T

where T is a type of the form $x:A\#B$, $x:A \rightarrow B$, $A|B$, $a < b$, $a=b$ in A , a in! A , U_i , or V .

4. $H \gg t \text{ in! } T$ by induce
 $\gg t' \text{ in! } T'$
 $\gg t' \text{ ind } t$
 $\gg t$ in \overline{T}
 T in $\overline{U_i} \gg T$ in U_i
5. $H, h:(t \text{ in! } T) \gg t$ in T by elim h

sequencing

1. $H \gg \text{seq}(a;b)$ in \overline{T} by intro
 $\gg a$ in $\overline{T'}$
 $\gg b$ in \overline{T}
2. $\text{seq}(a;b)$ in $\overline{T} \gg b$ in \overline{T} by elim

computational induction

1. $H \gg P[\text{fix}(f), a/q, r]$
 - ext $\text{fix}(\backslash h, a'.t)(a)$ by `fix_induction`
 - $\gg \text{fix}(f)$ in $A \rightarrow \overline{B}$
 - $\gg \text{fix}(f)(a)$ in! B
 - $\gg a$ in A
 - $a': A, \text{fix}(f)(a')$ in! $B, [\text{fix}(f)(a')$ at A over $q, r.P]$
 - $\gg P[\text{fix}(f), a'/q, r]$ ext t

The hypothesis $[\text{fix}(f)(a')$ at A over $q, r.P]$ asserts that

$$P[\text{fix}(f), a'/q, r]$$

is valid for all $\text{fix}(f)(a'')$ induced by $\text{fix}(f)(a')$. The following rule reflects this.

2. $H, h: [\text{fix}(f)(a)$ at A over $q, r.P]$
 - $\gg P[\text{fix}(f), a'/q, r]$ ext $h(a')$ by `comp_hyp h`
 - $\gg \text{fix}(f)(a)$ ind $\text{fix}(f)(a')$
 - $\gg a'$ in A

evaluation

1. $H \gg a \text{ op } b \text{ eval } c$ by `comp`
 - $\gg a \text{ op } b = c$ in `int`

where c is a canonical integer.
2. $H \gg \text{int_eq}(a; b; c; d)$ eval e by `comp true`
 - $\gg a=b$ in `int`
 - $\gg c$ eval e
3. $H \gg \text{less}(a; b; c; d)$ eval e by `comp true`
 - $\gg a < b$
 - $\gg c$ eval e
4. $H \gg \text{ind}(n; x, y. a; b; x, y. c)$ eval d plus
 - $\gg 0 < n$
 - $\gg c[n, (\text{ind}(n-1; x, y. a; b; x, y. c))/x, y]$ eval d

5. $H \gg \text{spread}(a;x,y.b) \text{ eval } c \text{ by comp}$
 $\gg a \text{ eval } \langle e, f \rangle$
 $\gg b[e, f/x, y] \text{ eval } c$
6. $H \gg a(b) \text{ eval } c \text{ by comp}$
 $\gg a \text{ eval } \backslash x.d$
 $\gg d[b/x] \text{ eval } c$
7. $H \gg \text{decide}(a;x.b;y.c) \text{ eval } d \text{ by comp}$
 $\gg a \text{ eval inl}(l)$
 $\gg b[l/x] \text{ eval } d$
8. $H \gg \text{seq}(a;b) \text{ eval } c \text{ by comp}$
 $\gg a \text{ in! } A$
 $\gg b \text{ eval } c$
9. $H \gg \text{fix}(a) \text{ eval } b \text{ by comp}$
 $\gg a(\text{fix}(a)) \text{ eval } b$

inducement

1. $H \gg a \text{ ind } b \text{ by eval}$
 $\gg a \text{ eval } b$
 where a is not canonical.
2. $H \gg a \text{ ind } c \text{ by trans}$
 $\gg a \text{ ind } b$
 $\gg b \text{ ind } c$
3. $H \gg a \text{ op } b \text{ ind } a \text{ by arg left}$
4. $H \gg \text{int_eq}(a;b;c;d) \text{ ind } c \text{ by comp true}$
 $\gg a = b \text{ in int}$
5. $H \gg \text{int_eq}(a;b;c;d) \text{ ind } a \text{ by arg left}$
6. $H \gg \text{less}(a;b;c;d) \text{ ind } c \text{ by comp true}$
 $\gg a < b$
7. $H \gg \text{less}(a;b;c;d) \text{ ind } a \text{ by arg left}$
8. $H \gg \text{ind}(n;x,y.a;b;x,y.c) \text{ ind } c[n, (\text{ind}(n-1;x,y.a;b;x,y.c))/x, y]$
 by comp plus
 $\gg 0 < n$

9. $H \gg \text{ind}(n; x, y. a; b; x, y. c)$ ind n by arg
10. $H \gg \text{spread}(a; x, y. b)$ ind $b[l, r/x, y]$ by comp
 $\gg a$ eval $\langle l, r \rangle$
11. $H \gg \text{spread}(a; x, y. b)$ ind a by arg
12. $H \gg a(b)$ ind $d[b/x]$ by comp
 $\gg a$ eval $\backslash x. d$
13. $H \gg a(b)$ ind a by arg
14. $H \gg \text{decide}(a; x. b; y. c)$ ind $b[l/x]$ by comp left
 $\gg a$ eval $\text{inl}(l)$
15. $H \gg \text{decide}(a; x. b; y. c)$ ind a by arg
16. $H \gg \text{seq}(a; b)$ ind b by comp
 $\gg a$ in! T
17. $H \gg \text{seq}(a; b)$ ind a by arg
18. $H \gg \text{fix}(a)$ ind $a(\text{fix}(a))$ by comp

Bibliography

- [All87a] S. F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.
- [All87b] S. F. Allen. A non-type-theoretic semantics for type-theoretic language. Technical Report 87-866, Department of Computer Science, Cornell University, September 1987. Ph.D. Thesis.
- [Bas88a] D. A. Basin. Building theories in Nuprl. Technical Report 88-932, Department of Computer Science, Cornell University, 1988.
- [Bas88b] D. A. Basin. An environment for automated reasoning about partial functions. In *9th International Conference On Automated Deduction*, volume 310 of *Lecture notes in Computer Science*, pages 101–110, 1988.
- [Bat79] J. L. Bates. *A Logic For Correct Program Development*. PhD thesis, Cornell University, 1979.
- [BC85] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):113–136, 1985.
- [Bee82] M. J. Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.
- [Blu67] M. Blum. On the size of machines. *Information and Control*, 11:257–265, 1967.
- [BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. ACM Monograph series. Academic Press, New York, 1979.

- [CAB⁺86] R. L. Constable, S. F. Allen, H. Bromley, W. R. Cleveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. P. Mendler, P. Panangaden, J. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [CH85] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184. Springer-Verlag, 1985.
- [CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
- [Cle87] R. C. Cleaveland. Type-theoretic models of concurrency. Technical Report 87-837, Department of Computer Science, Cornell University, May 1987. Ph.D. Thesis.
- [CM85] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture notes in Computer Science*, pages 61–78, Berlin, 1985. Springer-Verlag.
- [CO78] R. L. Constable and M. J. O'Donnell. *A Programming Logic*. Winthrop, 1978.
- [CS87] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.
- [CS88] R. L. Constable and S. F. Smith. Computational foundations of basic recursive function theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*. IEEE, 1988.
- [CZ84] R. L. Constable and D. R. Zlatin. The type theory PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, 1984.
- [dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration, Lecture notes in Mathematics vol. 125*, pages 29–61, New York, 1970. Springer-Verlag.

- [dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.
- [Dum77] M. Dummett. *Elements of Intuitionism*. Oxford University Press, 1977.
- [Ehr88] T. Ehrhard. A categorical semantics of constructions. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 264–273. IEEE, 1988.
- [Fef75] S. Feferman. A language and axioms for explicit mathematics. In J. N. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture notes in Mathematics*, pages 87–139. Springer-Verlag, 1975.
- [Fri69] H. Friedman. Axiomatic recursive function theory. In *Logic Colloquium '69*, pages 385–404. North-Holland, 1969.
- [Gir71] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'Analyse, et son application à l'Élimination des coupures dans l'Analyse et la Théorie des types. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971. North-Holland.
- [Gir86] J.-Y. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.
- [GMW79] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture notes in Computer Science*. Springer-Verlag, 1979.
- [Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
- [HN87] S. Hayashi and Hiroshi Nakano. PX: a computational logic. Technical Report RIMS-573, RIMS, Kyoto University, 1987.
- [Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, October 1969.
- [How86] D. J. Howe. Implementing number theory, an experiment with Nuprl. In *Proceedings of the Eighth International Conference on Automated Deduction*, volume 230 of *Lecture notes in Computer Science*, pages 404–415. Springer-Verlag, 1986.

- [How87] D. J. Howe. The computational behaviour of Girard's paradox. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 205–214. IEEE, 1987.
- [How88a] D. J. Howe. Automating reasoning in an implementation of constructive type theory. Technical Report 88-925, Department of Computer Science, Cornell University, June 1988. Ph.D. Thesis.
- [How88b] D. J. Howe. Equality in lazy computation systems. Manuscript, 1988.
- [HR67] Jr. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
- [Kre86] C. Kreitz. Implementing automata theory. Technical Report 86-779, Department of Computer Science, Cornell University, 1986.
- [Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
- [Mar71] P. Martin-Löf. A theory of types. Report 71-3, Department of Mathematics, University of Stockholm, February 1971.
- [Mar73] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. F. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, Amsterdam, 1973. North-Holland.
- [Mar80] P. Martin-Löf. Intuitionistic type theory. Notes by Giovanni Sambin of a series of lectures given in Padova, June, 1980.
- [Mar82] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
- [Mar83] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes from lectures given at Siena, April, 1983.
- [Men87] P. F. Mendler. Inductive definition in type theory. Technical Report 87-870, Department of Computer Science, Cornell University, September 1987. Ph.D. Thesis.

- [Moh86] C. Mohring. Algorithm development in the calculus of constructions. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 84–91. IEEE, 1986.
- [MPC86] N. P. Mendler, P. Panangaden, and R. L. Constable. Infinite objects in type theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 249–255, 1986.
- [Nor81] B. Nordstrom. Programming in constructive set theory: Some examples. In *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 290–341, Portsmouth, England, 1981.
- [Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
- [Rey74] J. C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.
- [Sco70] D. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture notes in Mathematics*, pages 237–275, Berlin, 1970. Spinger-Verlag.
- [Sco76] D. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.
- [Soa87] R. I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, 1987.
- [SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRC-6, Oxford University, 1971.
- [Ste72] S. Stenlund. *Combinators, Lambda-Terms and Proof Theory*. D. Reidel, Dordrecht, Holland, 1972.
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, 1977.
- [Str68] H. R. Strong. Algebraically generalized recursive function theory. *IBM J. Res. Devel.*, 12:465–475, 1968.

- [Wag69] E. G. Wagner. Uniformly reflexive structures: On the nature of Gödelizations and relative computability. In *Studies In Logic and The Foundations Of Mathematics — Logic Colloquim '69*, volume 61. North-Holland, 1969.