International Journal of Foundations of Computer Science

HYBRID PARTIAL-TOTAL TYPE THEORY

SCOTT F. SMITH
Department of Computer Science, The Johns Hopkins University
Baltimore, Maryland 21218 USA
scott@cs.jhu.edu

ABSTRACT

In this paper a hybrid type theory **HTT** is defined which combines the programming language notion of partial type with the logical notion of total type into a single theory. A new *partial type* constructor $\overline{A}$ is added to the type theory: objects in $\overline{A}$ may diverge, but if they converge, they must be members of A. A fixed point typing rule is given to allow for typing of fixed points. The underlying theory is based on ideas from Feferman's Class Theory and Martin Löf's Intuitionistic Type Theory. The extraction paradigm of constructive type theory is extended to allow direct extraction of arbitrary fixed points. Important features of general programming logics such as LCF are preserved, including the typing of all partial functions, a partial ordering $\lesssim$ on computations, and a fixed point induction principle. The resulting theory is thus intended as a general-purpose programming logic. Rules are presented and soundness of the theory established.

Keywords: Constructive Type Theory, Logics of Programs, Least Fixed Points

1. Introduction

In the seventies, Scott proposed a Logic for Computible Functions.[31] This theory axiomatized an ordering $\sqsubseteq$ on programs based on the domain-theoretic ordering, and included rules for typing fixed points and a fixed-point induction principle. Milner and others extended and implement Scott's ideas in the Edinburgh LCF system.[16]

Another line of research developing programming logics grew out of work by logicians. Martin-Löf's Intuitionistic Type Theory[22,7] has at its core a functional programming language, and contains a rich collection of types for typing programs. Through the duality of types and propositions, proofs and programs are linked. Another related approach is Feferman's class theory,[12,13,18,35]— classes are are arbitrary collections of untyped computations, and a rich array of classes can be defined. In both of these approaches, however, the standard notion of function space is a *total* one, and neither theory has general principles for typing fixed points as members of partial function spaces, or for ordering terms via $\sqsubseteq$, so many of the concepts expressible in LCF are missing.

The goal of this paper is to extend type/class theory to make it possible to type and reason about partial functions as is possible in LCF. This is thus a work

of synthesis, and the resulting theory is a hybrid partial-total type theory. It is also a hybrid of class theory and type theory; in fact the foundations bear more resemblance to class theory. We refer to it as a "type theory" only in a generic sense. The purpose of **HTT** should not be confused with the purpose of Intuitionistic Type Theory. We aim solely for a practical foundation that is the core of a usable programming logic, not for a philosophical foundation.

We start with constructive type theory and add a new collection of types, the *partial types* $\overline{A}$. There are three main principles governing the use of partial types. If an object is in a partial type $\overline{A}$ and its computation always terminates, it is in the type A of total objects. It is possible to type fixed points of functions on partial types, extending the programming power of type theory to arbitrary recursive programs. This fixed point typing rule is the subject of much of the metamathematical investigation of this paper, for its justification is somewhat difficult. With this rule, arbitrary recursive programs can be typed and in addition can be extracted from proofs following the proof-as-programs interpretation of constructive type theory. There is also a Scott-style fixed point induction principle.

This work grew from Constable's desire to extend the Nuprl type theory[7] to encompass ideas of LCF. Early results may be found in (Ref. 9,32,34). The Nuprl type theory is derived from Martin-Löf's ITT_0.[22] One important difference is that Nuprl takes an untyped view of computation: untyped computations are sensible, in particular it is possible to compute expressions before they are typed. Feferman-style class theories[12,13,18,35] take a similar approach: classes are collections of untyped computations. **HTT** borrows additional ideas from class theory: it includes a type E of all expressions and a type-free equality judgement $a \cong b$ and ordering $a \sqsubseteq b$, making pure untyped reasoning possible. The author's original interest in this mix of type theory and class theory is due to Howe.[19] In ITT_0 and Nuprl, on the other hand, types come with an equivalence on their members, $a = a' \in A$. Thus, types are PER's. The notion of types as PER's does not fit as well with partiality, for fundamental to partiality is the idea of an ordering $\sqsubseteq$. Fixed-point induction for instance is an uninteresting principle without an atomic ordering $\sqsubseteq$, nontrivial equivalences cannot be directly established by the principle. It thus might seem that the idea of a partial ordering relation (POR), $a \sqsubseteq a' \in A$, would be the natural way to generalize a PER and develop hybrid type theory. However this idea proves problematic. We thus believe the approach taken here is the most natural setting for a hybrid type theory.

1.1. Outline of the paper

In section 2 **HTT** is defined. Section 3 illustrates uses of the theory through examples. Lastly, semantics is given and soundness proved in section 4. We assume the reader has some familiarity with constructive type theory.[7,22,27]

2. The Theory

We now define **HTT**, a hybrid theory of partial and total typed computations.

HTT is not a full-featured type theory, but is the core of one; some important features not included are higher-order principles such as type universes, subtypes and recursive types, and classical reasoning principles. These are removed so we may focus on the core ideas in this presentation.

2.1. The terms

The theory has one sort, terms, which includes both types and computations. This means there is no rigid separation of types and terms; it is only in how the terms are used that the separation lies. We have an untyped language with numbers, pairing and projection, functions and application, and types.

Definition 1 (Terms) *The terms of* **HTT** *are*

(i) Variables x, y, z,...,

(ii) Data constructors $0, 1, 2, \ldots$, $\langle a, b\rangle$, $\lambda x.a$,

(iii) Type constructors E, N, $\overline{A}$, $x{:}A{\times}B$, $x{:}A{\to}B$, a `in` A, $a \sqsubseteq b$, $a{\downarrow}$

(iv) Computation constructors `pred`(a), `succ`(a), `if_zero`$(a; b; c)$, $\pi_1(a)$, $\pi_2(a)$, $a(b)$

where inductively a, b, c, A, B range over terms, and x ranges over variables.

We let a–t, A–T range over terms. Although terms and types are of the same sort, we informally use capital letters to denote what is intended to be a type and small letters, terms. Notions of *bound* and *free* variables, *open* and *closed* terms and substitution of b for x in a, $a[b/x]$, are standard (we rename bound variables to avoid capture); α-variants will be considered equal. We define a notion of contextual substitution: contexts are terms with holes $\bullet$; $a[\bullet], A[\bullet]\ldots$ range over contexts, and $a[b]$ denotes the replacement of all holes occurring in $a[\bullet]$ with b, possibly capturing free variables in b. The *values* (also called *canonical* terms) are outermost a data or type constructor, and are terms which cannot be computed further.

2.2. Judgements

All judgements are sequents, and take two forms: we may assert A to be a type, "A Type", and assert a to be a member of type A, "$a \in A$". The rules are organized such that in the process of showing a to inhabit A, A will be shown to be a type.

An assumption list Γ is of the form $x_1{:}A_1, x_2{:}A_2, \ldots x_n{:}A_n$, and signifies reasoning takes place under assumptions $x_1 \in A_1$, ..., $x_n \in A_n$. Two forms of judgement may be made; the first is

$\Gamma \vdash A$ Type

which asserts that under assumptions Γ, A is a type. The second is

$\Gamma \vdash a \in A$

which asserts under assumptions Γ, a inhabits type A. Note it is an invariant that $a \in A$ always implies A Type: for a type to be inhabited it first must be well-formed.

2.3. Rules and proofs

Before presenting the rules, some conventions are given. In the hypothesis list $x_1{:}A_1, x_2{:}A_2, \ldots x_n{:}A_n$, x_i may occur free in any A_{i+j} for positive j, and free variables in the conclusion are no more than the x_i. α-conversion is an unmentioned rule. The judgement $0 \in a \sqsubseteq b$ will be abbreviated $a \sqsubseteq b$, likewise for $a{\downarrow}$ and a `in` A—since there is at most one inhabiting object, 0, it need not be mentioned. Also, in hypothesis lists, $x{:}(a \sqsubseteq b)$ will be abbreviated $a \sqsubseteq b$, since x is known to be 0. $\mathtt{Y} \stackrel{\text{def}}{=} \lambda y.(\lambda x.y(x(x)))(\lambda x.y(x(x)))$, and $\mathtt{bot} \stackrel{\text{def}}{=} (\lambda x.x(x))(\lambda x.x(x))$. We also make a convention that when writing a dependent type as $x{:}A{\rightarrow}B(x)$ or $x{:}A{\times}B(x)$, x will not occur free in B. This means the only x bound by $x{:}A$ is the x appearing in the application.

2.4. Computation

A call-by-name method of computation is used: the outermost constructor is computed; if it is a value, no computation is performed. A *terminating* computation is thus one which computes to a value. The deterministic nature of the computation system is an important part of the partial type definition, because computations may terminate but still contain undefined components. Take for example $\lambda x.\mathtt{bot}$, which inhabits the type $\mathrm{N}{\rightarrow}\overline{\mathrm{N}}$: for some reduction strategies, this term would not be in normal form and thus could not inhabit any type but a partial type $\overline{A}$ for some A.

Two types directly assert properties of untyped terms: $a \sqsubseteq b$ asserts b is as defined as a, and $a{\downarrow}$ asserts a terminates. In accordance with the principle of propositions-as-types, these types are inhabited (by the placeholder 0) just when they are true. Equivalence $a \cong b$ abbreviates $(a \sqsubseteq b){\times}(b \sqsubseteq a)$. The inhabiting object justifying the truth of $a \cong b$, $\langle 0, 0\rangle$, is often elided, following the above convention for $\sqsubseteq$. $a \ncong b$ abbreviates $a \cong b{\rightarrow}0 \sqsubseteq 1$, and $a \not\sqsubseteq b$ abbreviates $a \sqsubseteq b{\rightarrow}0 \sqsubseteq 1$.

$$\text{(Computation)} \quad \frac{}{\Gamma \vdash t \cong t'}$$

where t and t' are one of the following pairs:

t	t'
$(\lambda x.b)(a)$	$b[a/x]$
$\pi_1(\langle a, b\rangle)$	a
$\pi_2(\langle a, b\rangle)$	b
$\mathtt{succ}(n)$	$n+1$
$\mathtt{pred}(n+1)$	n
$\mathtt{if\_zero}(0; a; b)$	a

$$\text{(Comp if)} \quad \frac{\Gamma \vdash a \ncong 0 \qquad \Gamma \vdash a \in \mathrm{N}}{\Gamma \vdash \mathtt{if\_zero}(a; b; c) \cong c}$$

$$(\sqsubseteq \text{ refl}) \quad \frac{}{\Gamma \vdash a \sqsubseteq a}$$

$$(\lesssim \text{ trans}) \quad \frac{\Gamma \vdash a \lesssim b \qquad \Gamma \vdash b \lesssim c}{\Gamma \vdash a \lesssim c}$$

$$(\lesssim \text{ subst}) \quad \frac{\Gamma \vdash a \lesssim b}{\Gamma \vdash c[a] \lesssim c[b]}$$

$$(\lesssim \text{ extensionality}) \quad \frac{\Gamma, x{:}\mathrm{E} \vdash (\lambda x.a)(x) \lesssim (\lambda x.b)(x)}{\Gamma \vdash \lambda x.a \lesssim \lambda x.b}$$
where x is not free in a or b.

$$(\text{Bottom}) \quad \frac{}{\Gamma \vdash \mathtt{bot} \lesssim a}$$

$$(\text{Termination inherit}) \quad \frac{\Gamma \vdash a \lesssim b \qquad \Gamma \vdash a\downarrow}{\Gamma \vdash b\downarrow}$$

$$(\text{Pair inherit}) \quad \frac{\Gamma \vdash a \lesssim b \qquad \Gamma \vdash a \cong \langle \pi_1(a), \pi_2(a)\rangle}{\Gamma \vdash b \cong \langle \pi_1(b), \pi_2(b)\rangle}$$

$$(\text{Func inherit}) \quad \frac{\Gamma \vdash a \lesssim b \qquad \Gamma \vdash a \cong \lambda x.a(x)}{\Gamma \vdash b \cong \lambda x.b(x)}$$
where x is not free in a or b

$$(\text{Number inherit}) \quad \frac{\Gamma \vdash a \lesssim b \qquad \Gamma \vdash a \in \mathrm{N}}{\Gamma \vdash a \cong b}$$

$$(\text{Pair left strict}) \quad \frac{\Gamma \vdash \pi_1(a)\downarrow}{\Gamma \vdash a \cong \langle \pi_1(a), \pi_2(a)\rangle}$$

$$(\text{Pair right strict}) \quad \frac{\Gamma \vdash \pi_2(a)\downarrow}{\Gamma \vdash a \cong \langle \pi_1(a), \pi_2(a)\rangle}$$

$$(\text{Func strict}) \quad \frac{\Gamma \vdash a(b)\downarrow}{\Gamma \vdash a \cong \lambda x.a(x)}$$
where x is a new variable not free in a

$$(\text{If arg strict}) \quad \frac{\Gamma \vdash \mathtt{if\_zero}(a; b; c)\downarrow}{\Gamma \vdash a \in \mathrm{N}}$$

$$(\text{Succ strict}) \quad \frac{\Gamma \vdash \mathtt{succ}(a)\downarrow}{\Gamma \vdash a \in \mathrm{N}}$$

$$(\text{Pred strict}) \quad \frac{\Gamma \vdash \mathtt{pred}(a)\downarrow}{\Gamma \vdash a \ \mathtt{in}\ \mathrm{N} \times a \not\cong 0}$$

The strictness rules are surprisingly important for proofs in that they characterize the evaluation order.

2.5. Membership

The expression a `in` A reifies the judgement $a \in A$ as a type; $0 \in (a$ `in` $A)$ just when $a \in A$. The placeholder member 0 is implicit in the rules below.

$$\text{(Member intro)} \quad \frac{\Gamma \vdash a \in A}{\Gamma \vdash a \texttt{ in } A}$$

$$\text{(Member elim)} \quad \frac{\Gamma \vdash a \texttt{ in } A}{\Gamma \vdash a \in A}$$

2.6. Term

E is a type of all terms. A more useful theory is obtained by admitting E as a type, quantifying over E allows abstract reasoning about untyped computations.

$$\text{(E intro)} \quad \frac{}{\Gamma \vdash a \in \mathrm{E}}$$

2.7. Natural numbers

N is a type of natural numbers, with standard rules.

$$\text{(N intro)} \quad \frac{}{\Gamma \vdash a \in \mathrm{N}}$$

Where a is one of 0, 1, 2,....

$$\text{(N succ)} \quad \frac{\Gamma \vdash a \in \mathrm{N}}{\Gamma \vdash \texttt{succ}(a) \in \mathrm{N}}$$

There is a symmetric rule (N pred).

$$\text{(N succpred)} \quad \frac{\Gamma \vdash a \in \mathrm{N} \qquad \Gamma \vdash a \ncong 0}{\Gamma \vdash \texttt{succ}(\texttt{pred}(a)) \cong a}$$

$$\text{(N predsucc)} \quad \frac{\Gamma \vdash a \in \mathrm{N}}{\Gamma \vdash \texttt{pred}(\texttt{succ}(a)) \cong a}$$

$$\text{(N induction)} \quad \frac{\Gamma \vdash b(0) \in B(0) \qquad \Gamma \vdash a \in \mathrm{N} \qquad \Gamma, x{:}\mathrm{N}, b(x) \texttt{ in } B(x) \vdash b(\texttt{succ}(x)) \in B(\texttt{succ}(x))}{\Gamma \vdash b(a) \in B(a)}$$

2.8. Dependent function

Dependent functions $x{:}A{\rightarrow}B(x)$ are also commonly notated $\Pi x{:}A.B(x)$. We let $A{\rightarrow}B$ abbreviate $x{:}A{\rightarrow}B$ where x is not free in B.

$$\text{(Func intro)} \quad \frac{\Gamma, x{:}A \vdash b \in B(x) \qquad \Gamma \vdash A \text{ Type}}{\Gamma \vdash \lambda x.b \in x{:}A{\rightarrow}B(x)}$$

$$\text{(Func elim)}\quad \frac{\Gamma \vdash a \in A \qquad \Gamma \vdash b \in x{:}A{\to}B(x)}{\Gamma \vdash b(a) \in B(a)}$$
where x is not free in b.

$$\text{(Func lam)}\quad \frac{\Gamma \vdash a \in A \qquad \Gamma \vdash b \in x{:}A{\to}B(x)}{\Gamma \vdash b \cong \lambda x.b(x)}$$
where x is not free in b.

2.9. Dependent product

Dependent products $x{:}A{\times}B(x)$ are also commonly notated $\Sigma x{:}A.B(x)$, but "dependent product" is a more apt computational description. These are types of pairs for which the type of the second component of the pair depends on the value of the first component. Let $A{\times}B$ abbreviate $x{:}A{\times}B$ where x is not free in B.

$$\text{(Prod intro)}\quad \frac{\Gamma \vdash a \in A \qquad \Gamma \vdash b \in B(a) \qquad \Gamma, x{:}A \vdash B(x)\ \text{Type}}{\Gamma \vdash \langle a, b\rangle \in x{:}A{\times}B(x)}$$

$$\text{(Prod elim left)}\quad \frac{\Gamma \vdash a \in x{:}A{\times}B(x)}{\Gamma \vdash \pi_1(a) \in A}$$

$$\text{(Prod elim right)}\quad \frac{\Gamma \vdash a \in x{:}A{\times}B(x)}{\Gamma \vdash \pi_2(a) \in B(\pi_1(a))}$$

$$\text{(Prod elim pair)}\quad \frac{\Gamma \vdash a \in x{:}A{\times}B(x)}{\Gamma \vdash a \cong \langle \pi_1(a), \pi_2(a)\rangle}$$

The third assumption of (Prod intro) assures that $x{:}A{\times}B(x)$ is a sensible type. This antecedent is not found in (Func intro) because there it follows from the antecedent $x{:}A \vdash b \in B(x)$.

2.10. Partial type

The bar type $\overline{A}$ is the type of (possibly diverging) computations over A. Three principles axiomatize partial types: terminating objects in types $\overline{A}$ are also in A, fixed points of functions $f \in \overline{A}{\to}\overline{A}$ may be taken using the fixed point typing rule, and inductive properties may be proven via fixed point induction. The latter two properties only hold for certain *admissible* types $\mathcal{A}$, defined below.

$$\text{(Bar intro)}\quad \frac{\Gamma, a{\downarrow} \vdash a \in A \qquad \Gamma, A{\downarrow} \vdash A\ \text{Type}}{\Gamma \vdash a \in \overline{A}}$$

$$\text{(Bar elim)}\quad \frac{\Gamma \vdash a{\downarrow} \qquad \Gamma \vdash a \in \overline{A}}{\Gamma \vdash a \in A}$$

$$\text{(Fixed point)}\quad \frac{\Gamma \vdash a \in \overline{A}{\to}\overline{A}}{\Gamma \vdash \mathtt{Y}(a) \in \overline{A}}$$
where $A \in \mathcal{A}$

Letting A be $B \rightarrow \overline{C}$, partial functions may be typed with (Fixed point); examples of its use are found in the next section. (Fixed point) is not generally true for all types; a counterexample appears in section 3.3.

A Scott-style fixed point induction principle[11,21,29] allows inductive properties of partial functions to be proven. It is closely related to the fixed point typing principle: both share the same admissible formulae, and in section 4, it will be shown that their soundness proofs both follow from a single general lemma.

$$\text{(FP induction)} \quad \frac{\Gamma \vdash a(f(\mathtt{bot})) \in P(f(\mathtt{bot})) \qquad \Gamma \vdash f \in \overline{B} \rightarrow \overline{B} \quad \Gamma, x{:}B, y{:}P(x) \vdash a(f(x)) \in P(f(x))}{\Gamma \vdash a(\mathtt{Y}(f)) \in P(\mathtt{Y}(f))} \quad \text{where } x{:}\overline{B} \times P(x) \in \mathcal{A}$$

The set of admissible types $\mathcal{A}$ are those types for which the fixed point principle and fixed point induction are valid. Informally, A is defined to be admissible if for any dependent product subterms $x{:}B \times C$ occurring in A, if there is in turn a dependent function $y{:}D \rightarrow E$ inside C, x may not occur in D. What is defined here is one approximation to a set of admissible types, an approach also taken in first-order axiomatizations of fixed point induction.[16,20,29] The general question of admissibility is undecidable.

We give a very simple inductive definition of admissible types. A more liberal *semantic* as opposed to *syntactic* constraint is possible,[34] but the resultant complexity of such an approach places it beyond the scope of the present paper.

Definition 2 *Given a set of variables X, the admissible types $\mathcal{A}$ are inductively defined as follows:*

$$\begin{array}{rcl} \mathcal{A} & = & x{:}\mathcal{A} \rightarrow \mathcal{A} + x{:}\mathcal{A} \times \mathcal{B}^{\{x\}} + a \ \mathtt{in} \ \mathcal{A} + \overline{\mathcal{A}} + a \lesssim b + a{\downarrow} + \mathrm{E} + \mathrm{N} \\ \mathcal{B}^X & = & x{:}C \rightarrow \mathcal{B}^X + x{:}\mathcal{B}^X \times \mathcal{B}^{X \cup \{x\}} + a \ \mathtt{in} \ \mathcal{B}^X + \overline{\mathcal{B}^X} + a \lesssim b + a{\downarrow} + \mathrm{E} + \mathrm{N} \end{array}$$

where C contains no free occurrences of variables in X and is by inspection a well-formed type.

Note how equivalence assertions $a \cong b$ are not admissible. This is one reason why the theory herein is based around $\lesssim$ instead of $\cong$, and why the PER approach to type equivalence falls short in a partial setting.

2.11. Type

$$(\lesssim \text{ form}) \quad \frac{}{\Gamma \vdash a \lesssim b \ \mathrm{Type}}$$

$$(\downarrow \text{ form}) \quad \frac{}{\Gamma \vdash a{\downarrow} \ \mathrm{Type}}$$

$$\text{(Member form)} \quad \frac{\Gamma \vdash A \ \mathrm{Type}}{\Gamma \vdash (a \ \mathtt{in} \ A) \ \mathrm{Type}}$$

(E form) $\dfrac{}{\Gamma \vdash \mathrm{E}\ \mathrm{Type}}$

(N form) $\dfrac{}{\Gamma \vdash \mathrm{N}\ \mathrm{Type}}$

(Func form) $\dfrac{\Gamma \vdash A\ \mathrm{Type} \qquad \Gamma, x{:}A \vdash B(x)\ \mathrm{Type}}{\Gamma \vdash x{:}A \rightarrow B(x)\ \mathrm{Type}}$

(Prod form) $\dfrac{\Gamma \vdash A\ \mathrm{Type} \qquad \Gamma, x{:}A \vdash B(x)\ \mathrm{Type}}{\Gamma \vdash x{:}A \times B(x)\ \mathrm{Type}}$

(Bar form) $\dfrac{\Gamma, A\!\downarrow \vdash A\ \mathrm{Type}}{\Gamma \vdash \overline{A}\ \mathrm{Type}}$

2.12. Miscellaneous

(Contradict) $\dfrac{}{\Gamma, a \sqsubseteq b \vdash c \in C}$ where $a \not\lesssim b$ by inspection

(Cut) $\dfrac{\Gamma \vdash b \in B \qquad \Gamma, x{:}B \vdash a \in A}{\Gamma \vdash a \in A}$

(Cut type) $\dfrac{\Gamma \vdash b \in B \qquad \Gamma, x{:}B \vdash A\ \mathrm{Type}}{\Gamma \vdash A\ \mathrm{Type}}$

(Hypothesis) $\dfrac{}{\Gamma \vdash x \in A}$ where $x{:}A$ occurs in Γ.

(Subst) $\dfrac{\Gamma \vdash a \cong b \qquad \Gamma \vdash c[a] \in C[a]}{\Gamma \vdash c[b] \in C[b]}$

(Subst type) $\dfrac{\Gamma \vdash a \cong b \qquad \Gamma \vdash C[a]\ \mathrm{Type}}{\Gamma \vdash C[b]\ \mathrm{Type}}$

(Type termination) $\dfrac{\Gamma \vdash a \in A}{\Gamma \vdash a\!\downarrow}$ where A is by inspection not of the form $\overline{B}$ or E.

(Prop member) $\dfrac{}{\Gamma, x{:}A \vdash x \cong 0}$ where A is $(b\ \mathtt{in}\ B)$, $a \sqsubseteq b$, or $a\!\downarrow$.

3. Examples of the theory in use

We now give examples illustrating how **HTT** may be used to reason about programs, concentrating on the partial types and associated fixed point rules. We demonstrate two uses for the fixed point typing rule: for typing fixed points, and for proving by extracting fixed point objects.

3.1. Propositions and extraction

Propositions are expressed as types, using to the now-standard embedding. First, disjoint sums may be defined using existing types.

Definition 3

$$A + B \stackrel{\text{def}}{=} x{:}\mathrm{N}\times \mathtt{if\_zero}(x; A; B),$$

$$\mathtt{inl}(a) \stackrel{\text{def}}{=} \langle 0, a\rangle,$$

$$\mathtt{inr}(a) \stackrel{\text{def}}{=} \langle 1, a\rangle,$$

$$\mathtt{decide}(a, b, c) \stackrel{\text{def}}{=} \mathtt{if\_zero}(\pi_1(a); b(\pi_2(a)); c(\pi_2(a))).$$

Definition 4 *Logical expressions are defined in terms of types as follows:*

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

3.2. Using partial types

The partial type operator $\overline{A}$ gives a general notion of partiality. For example, consider the (total) function space on natural numbers N→N; there are seven partial type versions, $\overline{\mathrm{N}{\to}\mathrm{N}}$, $\mathrm{N}{\to}\overline{\mathrm{N}}$, $\overline{\mathrm{N}}{\to}\mathrm{N}$, $\overline{\mathrm{N}{\to}\overline{\mathrm{N}}}$, $\overline{\overline{\mathrm{N}}{\to}\mathrm{N}}$, $\overline{\mathrm{N}}{\to}\overline{\mathrm{N}}$, and $\overline{\overline{\mathrm{N}}{\to}\overline{\mathrm{N}}}$. $\overline{\mathrm{N}{\to}\mathrm{N}}$ is the type of terms which, if they terminate, are total functions on natural numbers. $\mathrm{N}{\to}\overline{\mathrm{N}}$ is closer to what we would consider a partial function: the argument of the function always terminates, but the result might not terminate. $\overline{\mathrm{N}{\to}\overline{\mathrm{N}}}$ is also a type of partial functions, allowing the function itself to diverge. This will in fact be the type generally used herein to express partial functions, it has the advantage over the previous form that partial functions compose. $\overline{\overline{\mathrm{N}}{\to}\overline{\mathrm{N}}}$ is the type of partial functions for a call-by-name functional programming language. Fully partial types, as used in a programming language, may be defined by inductively barring all constructors in the type. Note $\overline{\overline{A}} = \overline{A}$.

For convenience we use the following abbreviation for partial functions:

$$x{:}A \stackrel{\mathrm{p}}{\to} B(x) \stackrel{\text{def}}{=} \overline{x{:}A{\to}\overline{B(x)}}.$$

To express recursive functions, fixed points $\mathtt{Y}(f) \in A \stackrel{\mathrm{p}}{\to} B$ of functionals $f \in (A \stackrel{\mathrm{p}}{\to} B){\to}(A \stackrel{\mathrm{p}}{\to} B)$ are typed using the fixed point typing principle. In this general

manner all recursive functions may be typed. We give an example of typing a simple recursive addition function of two curried arguments.

Lemma 1 $\vdash \mathtt{plus} \in \mathrm{N}{\to}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}$, *where*

$\mathtt{plus} \stackrel{\mathrm{def}}{=} \lambda z.\mathtt{Y}(\lambda x.\lambda y.\mathtt{if\_zero}(y; z; \mathtt{succ}(x(\mathtt{pred}(y)))))$.

Proof. By the fixed point typing principle and (Bar intro), (Func intro), it suffices to show

$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N} \vdash \mathtt{if\_zero}(y; z; \mathtt{succ}(x(\mathtt{pred}(y)))) \in \overline{\mathrm{N}}$.

Since $y \in \mathrm{N}$, we can proceed by cases on $y \cong 0$ or not, using (N induction). First, consider the case where $y \cong 0$; computing with (Computation) gives

$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N}, y \cong 0 \vdash z \in \overline{\mathrm{N}}$,

which follows directly by (Bar intro). Consider next the case $y \ncong 0$. By (N Pred), (Hyp), (Succ strict) and (N Succ),

$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N}, y \ncong 0, \mathtt{succ}(x(\mathtt{pred}(y)))\!\downarrow \vdash \mathtt{succ}(x(\mathtt{pred}(y))) \in \mathrm{N}$.

By (Bar intro),

$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N}, y \ncong 0 \vdash \mathtt{succ}(x(\mathtt{pred}(y))) \in \overline{\mathrm{N}}$,

which by (N succ) and (Bar intro) gives

$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N}, y \ncong 0, x(\mathtt{pred}(y))\!\downarrow \vdash x(\mathtt{pred}(y)) \in \mathrm{N}$.

At this point, since $\mathtt{pred}(y) \in \mathrm{N}$ (by (N pred)), we only need to apply the function x. However, as written, x may not be a function because it may diverge, but since $x(\mathtt{pred}(y))\!\downarrow$, (Func strict) gives $x\!\downarrow$. Then, by (Bar elim), $x \in \mathrm{N}{\to}\overline{\mathrm{N}}$, so $x(\mathtt{pred}(y)) \in \overline{\mathrm{N}}$, so by (Bar elim), $x(\mathtt{pred}(y)) \in \mathrm{N}$. □

3.3. Not all fixed points are typable

To illustrate the limits of the fixed point typing principle, we show a type which cannot be admissible if the theory is to be consistent.

Theorem 1 *There is a type D and term d such that $\vdash d \in \overline{D}{\to}\overline{D}$, but $\vdash \mathtt{Y}(d) \in \overline{D}$ implies $\vdash 0 \cong 1$.*

Proof. Define

$D \stackrel{\mathrm{def}}{=} x{:}(\mathrm{N}{\to}\overline{\mathrm{N}}){\times}\neg(\forall y{:}\mathrm{N}.\, x(y)\!\downarrow)$ and

$d \stackrel{\mathrm{def}}{=} \lambda z.\langle\lambda y.\mathtt{if\_zero}(y; 0; (\pi_1(z))(\mathtt{pred}(y))), \lambda w.0\rangle$.

D states "there exists a function x that is not total"; $\mathtt{Y}(d)$ is a pair consisting of a total function and the proof object $\lambda w.0$. $\mathtt{Y}(d) \in D$ is then a contradiction, because a total function is asserted non-total.

First, show $\vdash d \in \overline{D}{\to}\overline{D}$. Using (Fixed point), assume $z \in \overline{D}$, and show the pair to be in D. For the left half of the pair, $f \stackrel{\mathrm{def}}{=} \lambda y.\mathtt{if\_zero}(y; 0; (\pi_1(z))(\mathtt{pred}(y)))$ must be shown to be in $\mathrm{N}{\to}\overline{\mathrm{N}}$. This proof is similar to the proof of Lemma 1 above. For the right half of the pair, we show $\neg(\forall y{:}\mathrm{N}.\, f(y)\!\downarrow)$: suppose the antecedent were true; then, $z\!\downarrow$ because $\pi_1(z)\!\downarrow$ by strictness. Thus, $z \in D$, so $\pi_2(z) \in \neg(\forall y{:}\mathrm{N}.\, \pi_1(z)(y)\!\downarrow)$. But, if $\pi_1(z)$ is not total, f will also not be total, contradicting our assumption. Thus, $d \in \overline{D}{\to}\overline{D}$.

Since we assume $\vdash \mathtt{Y}(d) \in \overline{D}$, $\vdash \mathtt{Y}(d) \in D$ follows by computing. From this it follows that the function $\pi_1(\mathtt{Y}(d)) \in \mathrm{N}{\to}\overline{\mathrm{N}}$ is not total. However this is a contradiction as $\vdash \pi_1(\mathtt{Y}(d)) \in \mathrm{N}{\to}\mathrm{N}$ is provable using natural number induction. Therefore,

$\vdash \neg(\mathtt{Y}(d)\ \mathtt{in}\ \overline{D})$, and $\vdash 0 \cong 1$. $\square$

3.4. Partial propositions and partial proofs

The bar operator may also be applied to types which represent propositions, giving types such as $\forall n{:}\mathrm{N}.\ \overline{\exists m{:}\mathrm{N}.\ P(m,n)}$. We call such types *partial propositions*. $\overline{A}$ for any type A is trivially true under the propositions-as-types interpretation, because $\mathtt{bot} \in \overline{A}$. However, if $a \in \overline{A}$ and $a\!\downarrow$, then $a \in A$, so A is true. If we can potentially show termination of a, proving a partial proposition is useful. Partial propositions can be viewed as a logical notion of partial correctness.

Partial propositions are most relevant for universal quantifiers, because their extract objects are functions; a partial function type has as analogue a partial universal quantifier.

Definition 5 $\overline{\forall}x{:}A.\ B(x) \stackrel{\mathrm{def}}{=} x{:}A \stackrel{\mathrm{p}}{\rightarrow} B(x)$

Consider for instance the type

$P \stackrel{\mathrm{def}}{=} \overline{\forall}x{:}\mathrm{N}.\ \exists y{:}\mathrm{N}.\ y \cong \mathtt{mult}(x)(x)$, where

$\mathtt{mult} \stackrel{\mathrm{def}}{=} \mathtt{Y}(\lambda z.\lambda x.\lambda y.\mathtt{if\_zero}(x; 0; \mathtt{plus}(y, z(\mathtt{pred}(x))(y))))$.

If $p \in P$ and for some particular $n \in \mathrm{N}$ $p(n)\!\downarrow$, then $p(n)$ inhabits the type $\exists y{:}\mathrm{N}.y \cong \mathit{mult}(n)(n)$. Inhabiting objects validating propositions are always total, whereas objects inhabiting partial propositions are partial.

3.4.1. Extracting recursive programs

An extension to the extraction paradigm is possible via the notion of partial proposition: arbitrary recursive computations may be directly extracted from proofs. Consider the following derived rule.

Lemma 2 *The following non-well-founded induction principle is a derived rule in* **HTT**.

$$\textit{(Partial Induction)} \qquad \frac{\Gamma, h{:}(\overline{\forall}x{:}A.\ B(x)), x{:}A \vdash b \in \overline{B}(x)}{\Gamma \vdash \mathtt{Y}(\lambda h.\lambda x.b) \in \overline{\forall}x{:}A.\ B(x)} \quad \textit{where } \overline{\forall}x{:}A.\ B(x) \in \mathcal{A}$$

Proof. Direct from the fixed point typing principle. $\square$

When using this rule we may use the fact $\overline{\forall}x{:}A.\ B(x)$ in the proof of itself, resulting in an extract object which is a fixed-point computation. There is a well-known analogy between programming constructs and proof constructs: implication corresponds to function abstraction, disjunction corresponds to if-then-else, etc. The above rule gives the proof analogue to the fixed-point programming construct. This rule is not sound in total type theory, so there the fixed-point construct has no corresponding rule. For example, consider the partial proposition

$\overline{\forall}l{:}\mathtt{List}.\ \exists l'{:}\mathtt{List}.\ \mathtt{sorted}(l') \wedge \mathtt{permutation}(l, l')$.

Using (Partial Induction), we can assume what we are trying to prove, and thus give a proof where the proof recursion is exactly the algorithm recursion. This induction is not necessarily well-founded, because if we apply the induction hypothesis in a fashion that the value is not decreasing, the resulting extract object will loop

forever. For instance, using this rule $\mathtt{Y}(\lambda h.\lambda x.h(x))$ proves all $\overline{\forall}$ propositions. For this reason partial proofs will often not be appropriate.

3.4.2. An Example of Recursive Function Extraction

A sketch of the extraction of a fixed-point primality tester from a partial proposition is now given.

Definition 6

$\mathtt{minus} \stackrel{\text{def}}{=} \mathtt{Y}(\lambda z.\lambda x.\lambda y.\mathtt{if\_zero}(x; 0; \mathtt{if\_zero}(y; x; z(\mathtt{pred}(x))(\mathtt{pred}(y)))))$

$x\ \mathtt{Leq}\ y \stackrel{\text{def}}{=} \mathtt{minus}(x, y) \cong 0$

$y\ \mathtt{Divides}\ x \stackrel{\text{def}}{=} 2\ \mathtt{Leq}\ y \wedge \exists z{:}\mathrm{N}.\ \mathtt{mult}(y)(z) \cong x$

$x\ \mathtt{Is\_Prime\_Thru}\ y \stackrel{\text{def}}{=} \forall z{:}\mathrm{N}.\ 2\ \mathtt{Leq}\ z\ \Rightarrow\ z\ \mathtt{Leq}\ y\ \Rightarrow\ \neg(z\ \mathtt{Divides}\ x)$

$x\ \mathtt{Is\_Composite\_Thru}\ y \stackrel{\text{def}}{=} \exists z{:}\mathrm{N}.\ 2\ \mathtt{Leq}\ z\ \wedge\ z\ \mathtt{Leq}\ y\ \wedge\ z\ \mathtt{Divides}\ x$

Lemma 3 $\vdash e \in \overline{\forall} x{:}\mathrm{N}.\ \overline{\forall} y{:}\mathrm{N}.\ x\ \mathtt{Is\_Prime\_Thru}\ y \vee x\ \mathtt{Is\_Composite\_Thru}\ y.$

The extract object e will be a fixed-point function such that $e(x)(\mathtt{pred}(x))$ decides whether or not x is prime; furthermore, if x is composite, one of its factors will be returned. An induction argument could then be given to prove this function is in fact total.

Proof. Using the fixed point typing principle, assume
$\vdash z \in (\overline{\forall} x{:}\mathrm{N}.\ \overline{\forall} y{:}\mathrm{N}.\ x\ \mathtt{Is\_Prime\_Thru}\ y \vee x\ \mathtt{Is\_Composite\_Thru}\ y), x \in \mathrm{N}, y \in \mathrm{N}$,
and show

$\vdash e' \in \overline{x\ \mathtt{Is\_Prime\_Thru}\ y \vee x\ \mathtt{Is\_Composite\_Thru}\ y}$,

where $e \stackrel{\text{def}}{=} \mathtt{Y}(\lambda z.\lambda x.\lambda y.e')$ and e' is to be determined. Inside e', uses of z are recursive calls, which logically are uses of the hypothesis z. If $y \cong 0$ or $y \cong 1$ the proof is trivial, so assume $2\ \mathtt{Leq}\ y$ is inhabited. Proceed by cases on $y\ \mathtt{Divides}\ x$, taking as given the obvious term $\mathtt{div}$ and proof of $\vdash \mathtt{div}(x)(z) \in (z\ \mathtt{Divides}\ x) \vee \neg(z\ \mathtt{Divides}\ x)$.

case $y\ \mathtt{Divides}\ x$: Clearly then x is composite, so the right disjunct can be proven letting z be y, noting $2\ \mathtt{Leq}\ y$ by assumption.

case $\neg(y\ \mathtt{Divides}\ x)$: Recursively use the hypothesis z for y one smaller, i.e. apply $z(x)(\mathtt{pred}(y))$, and the result also follows. □

The full extract object e is

$\mathtt{Y}(\lambda z, x, y.\mathtt{if\_zero}(y; \mathtt{inl}(\lambda z.\lambda w_1\lambda w_2.0); \mathtt{if\_zero}(\mathtt{pred}(y); \mathtt{inl}(\lambda z.\lambda w_1\lambda w_2.0);$
$\mathtt{decide}(\mathtt{div}(x)(y); \lambda w.\mathtt{inr}(\langle y, \langle 0, 0, w\rangle\rangle); \lambda w.z(x, \mathtt{pred}(y))))))$.

For instance, $e(6)(5)$, a test if 6 is prime, computes to $\mathtt{inr}(\langle 3, \langle 0, 0\rangle\rangle)$, meaning $3\ \mathtt{Divides}\ 6$ (studying e will convince the reader that the largest factor is returned if the number is composite). Note that since this proof term terminated, we now have a (total) proof that 6 is not prime. This notion of proof may be particularly well suited to reflected proof search, for the proof searcher will be a partial proof which, if it terminates, produces an actual proof.

Partial propositions extend the collection of statements that can be phrased when reasoning constructively, giving a more expressive logic. The notion of partial proposition makes no sense in a purely classical theory since proofs may not have computational content. See (Ref. 5) for some applications of partial proofs in theorem proving.

4. Semantics

A semantics for **HTT** is now given which shows it to be a soundly constructed theory. A simultaneous inductive definition of types and their members is given. Our approach is based on the work of Allen,[3,2] for which there is some precedence in the literature[1,6,17]. We refer the reader to these references for a more detailed description of the technique, a terse treatment will be given here.

Types are properties of type-free computations in **HTT** so untyped computations have independent meaning. An untyped equivalence $\cong$ is defined, and types are defined as sets of untyped computations. In this paper, untyped equivalence is defined operationally following Morris[25] and Plotkin[30]: two terms are considered equivalent iff no program context can distinguish between the two upon execution.

Once operational equivalence on terms is defined, the types and their inhabitants may be inductively defined as outlined above. Establishing soundness of the rules is then straightforward except for the fixed-point rules. These rules require a proof by induction on the structure of the admissible types.

4.1. Interpretation of computations

Since an operational meaning is given to terms, terms are interpreted in the semantics as themselves, i.e. a term model is constructed. Our presentation of operational semantics follows the approach of (Ref. 33,23), and we refer the reader to these papers for the details missing here. One contribution found in these references is a definition of operational equivalence over directed sets of computations, $\cong_{\mathrm{s}}$. The following property of this ordering, Theorem 4 below, is behind the proof of soundness of the fixed point rules:

$$\{\mathtt{Y}(f)\} \cong_{\mathrm{s}} \{f^k(\mathtt{bot}) \mid k \in N\}.$$

To obtain the appropriate notion of untyped equivalence, types need to be destructable. Otherwise, all types with the same outer constructor would be operationally equivalent, for no program context could distinguish between them. For this purpose, the collection of terms is extended.

Definition 7 *The* extended terms *are the terms as defined in Definition 1 with in addition the computation constructor* $\mathtt{destr}(A)$.

For the remainder of this paper, we will work over extended terms. First, an operational interpreter for untyped computations is defined. We present a single-step rewriting interpreter, using the more convenient notion of a reduction context to isolate the next redex.[14]

Definition 8 Computations *are terms that are outermost a computation constructor, i.e.* $\texttt{pred}(a)$, $\texttt{succ}(a)$, $\texttt{if\_zero}(a;b;c)$, $\pi_1(a)$, $\pi_2(a)$, $a(b)$, *or* $\texttt{destr}(A)$. Values *are terms that are outermost a data constructor or type constructor.*

Computations are terms that may be further evaluated; closed terms are either computations or values. A call-by-name evaluation strategy is deterministic, so at most one reduction applies. Recall earlier we had defined contexts $c[\bullet]$, terms with holes in which other terms may be placed. Reduction contexts are a special form of context that isolate the position of the next reduction.

Definition 9 *A reduction context* $R[\bullet]$ *is inductively of the form*

$\bullet$, $R[\bullet](a)$, $\texttt{if\_zero}(R[\bullet];a;b)$, $\texttt{pred}(R[\bullet])$,
$\texttt{succ}(R[\bullet])$, $\pi_1(R[\bullet])$, $\pi_2(R[\bullet])$, *or* $\texttt{destr}([\bullet])$,

where $R[\bullet]$ *is a reduction context, and* a *and* b *are terms.*

Single-step computation $\mapsto_1$ is next defined. Since the reduction context isolates the next reduction to perform, it is only a matter of performing the reductions at the point isolated by the reduction context.

Definition 10 $\mapsto_1$, *single-step computation, is the least relation such that*

$$\begin{array}{rcl}
R[\texttt{succ}(a)] & \mapsto_1 & R[a+1] \\
R[\texttt{pred}(a+1)] & \mapsto_1 & R[a] \\
R[\texttt{if_zero}(0;b;c)] & \mapsto_1 & R[b] \\
R[\texttt{if_zero}(a;b;c)] & \mapsto_1 & R[c], \textit{ where } a \in \{1,2,\ldots\} \\
R[(\lambda x.b)(a)] & \mapsto_1 & R[b[a/x]] \\
R[\pi_1(\langle a,b\rangle)] & \mapsto_1 & R[a] \\
R[\pi_2(\langle a,b\rangle)] & \mapsto_1 & R[b] \\
R[\texttt{destr}(T)] & \mapsto_1 & R[\langle \Pi_1(T), \Pi_2(T)\rangle],
\end{array}$$

where $\Pi_1(T)$ *and* $\Pi_2(T)$ *are defined as follows.*

T	$\Pi_1(T)$	$\Pi_2(T)$
$x{:}A{\rightarrow}B$	0	$\langle A, \lambda x.b\rangle$
$x{:}A{\times}B$	1	$\langle A, \lambda x.b\rangle$
$\overline{A}$	2	A
$a\downarrow$	3	a
a `in` A	4	$\langle a, A\rangle$
$a \sqsubseteq b$	5	$\langle a, b\rangle$
N	6	0
E	7	0

Definition 11 $\mapsto$ *is the transitive, reflexive closure of* $\mapsto_1$. $a\downarrow$ *iff* $a \mapsto b$ *for some value* b.

Note we overload the type syntax "$a\downarrow$" as a relation here; the appropriate meaning should be clear from context. A similar overloading of $\lesssim$ and $\cong$ will be defined below.

4.2. Equivalence and ordering of computations

HTT has an untyped ordering type $a \sqsubseteq b$. We give an operational interpretation of this ordering. Here a terse exposition of the results relevant to **HTT** soundness is

given; for a more detailed exposition and proofs, consult (Ref. 33,23). Operational ordering $a \lesssim b$ and equivalence $a \cong b$ are defined as follows.

Definition 12 *$a \lesssim b$ iff for all contexts $c[\bullet]$ such that $c[a]$ and $c[b]$ are closed, $c[a]\downarrow$ implies $c[b]\downarrow$. $a \cong b$ iff $a \lesssim b$ and $b \lesssim a$.*

In order to establish useful properties of $\lesssim$, it is useful to give an alternative characterization for which proving equivalences is easier. Let σ range over finite substitutions $[a_1/x_1, a_2/x_2, \ldots]$. $\sigma(a)$ denotes applying substitution σ to term a.

Definition 13 *$a \lesssim^{\mathrm{ciu}} b$ iff for all substitutions σ and all reduction contexts $R[\bullet]$ for which $R[\sigma(a)]$ and $R[\sigma(b)]$ are closed, $R[\sigma(a)]\downarrow$ implies $R[\sigma(b)]\downarrow$.*

This ordering is called ciu-ordering since the contexts are **c**losed **i**nstances of all **u**ses. The main characterization is then as follows.

Theorem 2 *$a \lesssim^{\mathrm{ciu}} b$ iff $a \lesssim b$*

Using this Theorem, a large number of properties of $\lesssim$ and $\cong$ may be established.

Lemma 4 *(i) $\lesssim$ is transitive and reflexive, and $\cong$ is an equivalence relation;*

(ii) $\lesssim$ is a pre-congruence: $a \lesssim b$ implies $c[a] \lesssim c[b]$ for all $c[\bullet]$;

(iii) $\cong$ is a congruence: $a \cong b$ implies $c[a] \cong c[b]$ for all $c[\bullet]$;

(iv) $\lesssim$ is extensional: $\lambda x.a \lesssim \lambda x.b$ iff for all c, $(\lambda x.a)(c) \lesssim (\lambda x.b)(c)$.

(v) $\mathtt{bot} \lesssim a$;

(vi) $a \cong a'$ if $a \mapsto a'$

(vii) If $a\downarrow$ and $a \lesssim b$, then $b\downarrow$.

(viii) If $a \lesssim b$ and $a \cong \langle \pi_1(a), \pi_2(a) \rangle$, then $b \cong \langle \pi_1(b), \pi_2(b) \rangle$;

(ix) If $a \lesssim b$ and $a \cong \lambda x.a(x)$, then $b \cong \lambda x.b(x)$, for fresh variable x;

(x) If $a \lesssim b$ and $a \in \{0, 1, 2, \ldots\}$, then $a \cong b$;

(xi) If $\pi_1(a)\downarrow$ or $\pi_2(a)\downarrow$, then $a \cong \langle \pi_1(a), \pi_2(a) \rangle$;

(xii) If $a(b)\downarrow$ then $a \cong \lambda x.a(x)$ for fresh variable x;

(xiii) If $\mathtt{if\_zero}(a; b; c)\downarrow$ then $a \in \{0, 1, 2, \ldots\}$;

(xiv) If $\mathtt{if\_zero}(0; b; c)\downarrow$ then $b\downarrow$;

(xv) If $\mathtt{if\_zero}(n; b; c)\downarrow$ and $n \in \{1, 2, 3, \ldots\}$, then $c\downarrow$;

(xvi) If $\mathtt{succ}(a)\downarrow$ or $\mathtt{pred}(a)\downarrow$ then $a \in \{0, 1, 2, \ldots\}$.

All of the computational rules may be justified by one of the cases of the above Lemma. Note, $\langle \mathtt{bot}, \mathtt{bot} \rangle \cong \lambda x.\mathtt{bot}$, but this artifact causes no problems.

Next the ordering $\lesssim_{\mathrm{s}}$ on $\lesssim$-directed sets of terms, and associated equivalence $\cong_{\mathrm{s}}$, are defined. As mentioned previously, properties of this ordering will aid in establishing the fixed point rules.

Definition 14 *A set of terms S is* directed *iff for every $a, b \in S$, $a \lesssim c$ and $b \lesssim c$ for some $c \in S$.*

We hereafter restrict ourselves to $\sqsubseteq$-directed sets of terms S containing only finitely many free variables, for otherwise it may be difficult to obtain fresh variables.

Definition 15 *$S \sqsubseteq_{\mathrm{s}} S'$ iff S and S' are $\sqsubseteq$- directed and for all reduction contexts $c[\bullet]$ such that $c[S]$ and $c[S']$ are closed, for all $s \in S$, if $c[s]\downarrow$, then $c[s']\downarrow$ for some $s' \in S'$. $S \cong_{\mathrm{s}} S'$ iff $S \sqsubseteq_{\mathrm{s}} S'$ and $S' \sqsubseteq_{\mathrm{s}} S$.*

The following properties are direct from the definitions.

Lemma 5

(i) $a \sqsubseteq b$ iff $\{a\} \sqsubseteq_{\mathrm{s}} \{b\}$.

(ii) $\sqsubseteq_{\mathrm{s}}$ is a pre-congruence, i.e. if $S \sqsubseteq_{\mathrm{s}} S'$ then $c[S'] \sqsubseteq_{\mathrm{s}} c[B]$.

(iii) $\cong_{\mathrm{s}}$ is a congruence, i.e. if $S \cong_{\mathrm{s}} S'$ then $c[S'] \cong_{\mathrm{s}} c[B]$.

(iv) $S \sqsubseteq_{\mathrm{s}} S'$ if for all $s \in S$, there is an $s' \in S'$ such that $s \sqsubseteq s'$.

As with $\cong$, the crucial tool in establishing $\cong_{\mathrm{s}}$ equivalences is an alternate characterization, which takes an analogous form.

Definition 16 *$S \sqsubseteq^{\mathrm{ciu}}_{\mathrm{s}} S'$ iff S and S' directed and for all reduction contexts $R[\bullet]$ such that $R[\sigma(S)]$ and $R[\sigma(S')]$ are closed, for all $s \in S$, if $R[\sigma(s)]\downarrow$, then $R[\sigma(S')]\downarrow$ for some $s' \in S'$.*

Theorem 3 *for all $\sqsubseteq$-directed sets S and S', $S \sqsubseteq_{\mathrm{s}} S'$ iff $S \sqsubseteq^{\mathrm{ciu}}_{\mathrm{s}} S'$.*

Using this alternate characterization, the key property may be proven. For some given function f, define

$$S_0 \stackrel{\mathrm{def}}{=} \{\mathtt{bot}, f(\mathtt{bot}), f(f(\mathtt{bot})), \ldots, f^k(\mathtt{bot}), \ldots\}.$$

Theorem 4 $\{\mathtt{Y}(f)\} \cong_{\mathrm{s}} S_0$.

Proof. $\{\mathtt{Y}(f)\} \supseteq_{\mathrm{s}} S_0$ is a direct induction; show $\sqsubseteq_{\mathrm{s}}$. Observe $\mathtt{Y}(f) \cong f'(f')$ where $f' = \lambda x.f(x(x))$, so it suffices to show $\{f'(f')\} \sqsubseteq_{\mathrm{s}} \{f^k(\mathtt{bot}) \mid k \in \mathrm{N}\}$. Expanding definitions, show

$R[f'(f')]\downarrow$ implies $R[f^k(\mathtt{bot})]\downarrow$ for some k.

To prove this we need to generalize the statement to allow the term in the reduction hole to appear elsewhere in R, and for the reduction point to be a more general form. We will notate this as $R[\bullet][\bullet]$, where the first hole can be anywhere in R and the second hole represents the reduction point. See (Ref. 34) for a more detailed description of this technique. The generalized goal is

$R[f'(f')][c[f'(f')]]\downarrow$ implies $R[f^k(\mathtt{bot})][c[f^k(\mathtt{bot})]]\downarrow$ for some k,

proved by induction on the length of computation. The only interesting case, where $f'(f')$ is touched, is

$R[f'(f')][f'(f')] \mapsto_1 R[f'(f')][f(f'(f'))]$,

with the goal to show $R[f^k(\mathtt{bot})][f^k(\mathtt{bot})]\downarrow$. By induction hypothesis, for some k', $R[f^{k'}(\mathtt{bot})][f(f^{k'}(\mathtt{bot}))]\downarrow$, so since $f^{k'}(\mathtt{bot}) \sqsubseteq f^{k'+1}(\mathtt{bot})$, and $f(f^{k'}(\mathtt{bot})) \cong f^{k'+1}(\mathtt{bot})$, $R[f^{k'+1}(\mathtt{bot})][f^{k'+1}(\mathtt{bot})]\downarrow$, and letting k be $k'+1$, the goal is proven. $\Box$

Intuitively, this proof formalizes the fact that in any particular program context, some finite-depth recursion stack will suffice to compute a recursive function to termination, provided we know the function terminates to begin with.

All that remains is to show the atomic case of fixed point induction. This theorem will directly justify admissibility for base types containing $\sqsubseteq$ for free occurrences of x.

Theorem 5 (Atomic fixed point induction) *If* $c[f^k(\mathtt{bot})] \sqsubseteq c'[f^k(\mathtt{bot})]$ *for all* k, $c[\mathtt{Y}(f)] \sqsubseteq c'[\mathtt{Y}(f)]$.

Proof. By Lemma 5 and Theorem 4, $\{c[f^k(\mathtt{bot})] \mid k \in N\} \cong_s \{c[\mathtt{Y}(f)]\}$ and $\{c'[f^k(\mathtt{bot})] \mid k \in N\} \cong_s \{c'[\mathtt{Y}(f)]\}$. Then, using the fact $\{c[f^k(\mathtt{bot})] \mid k \in N\} \sqsubseteq_s \{c'[f^k(\mathtt{bot})] \mid k \in N\}$ (by definition of $\sqsubseteq_s$) and the above equivalences, the result is immediate. □

4.3. Defining the types and their inhabitants

The types and their inhabitants may now be defined by simultaneous induction. In the presence of dependent types it is impossible to first define the types and then define membership relations for the types, because for a term $x{:}A{\rightarrow}B(x)$ to be a type, $B(a)$ must be a type for all members a of type A. This means the *members* of A must be defined before the *type* $x{:}A{\rightarrow}B(x)$ is considered well-formed.

Open terms are considered only when interpreting hypothetical judgements, so until that point all terms are implicitly taken to be closed.

Definition 17 *A* type interpretation *is a two-place relation* $\tau(A, \epsilon)$ *where* A *is a term, and* ϵ *is a one-place relation on terms.*

$\tau(A, \epsilon)$ means A is a type with its members specified by ϵ. The following definitions make this more clear:

Definition 18

A Type$_\tau$ *iff* $\exists\epsilon.\ \tau(A, \epsilon)$

$a \in_\tau A$ *iff* $\exists\epsilon.\ \tau(A, \epsilon)$ *and* $\epsilon(a)$.

A Type$_\tau$ means A is a type in interpretation τ, and $a \in_\tau A$ means a is a member of the type A.

The desired type interpretation ν is now inductively defined as the least fixed point of a monotone operator.

Definition 19 *Operator* Ψ *on type interpretations is defined as follows:*

$\Psi(\tau) \stackrel{\text{def}}{=} \tau'$, *where* $\tau'(T, \epsilon)$ *is true if and only if*

either $T \mapsto \mathrm{E}$, *in which case*
$\forall t.\ \epsilon(t)$

or $T \mapsto \mathrm{N}$, *in which case*
$\forall t.\ \epsilon(t)$ *iff* $t \cong n$, *where* n *is* $0, 1, 2, \ldots$

or $T \mapsto a \sqsubseteq b$, *in which case*
$\forall t.\ \epsilon(t)$ *iff* $t \cong 0$ *and* $a \lesssim b$

or $T \mapsto a{\downarrow}$, *in which case*
$\forall t.\ \epsilon(t)$ *iff* $t \cong 0$ *and* $a{\downarrow}$

or $T \mapsto a$ `in` A, *in which case*

A Type$_\tau$, *and*
$\forall t.\, \epsilon(t)$ *iff* $t \cong 0$ *and* $a \in_\tau A$

or $T \mapsto x{:}A{\rightarrow}B(x)$, *in which case*
A Type$_\tau \wedge \forall a \in_\tau A.\, B(a)$ Type$_\tau$, *and*
$\forall t.\, \epsilon(t)$ *iff* $t \cong \lambda x.b \wedge \forall a \in_\tau A.\, t(a) \in_\tau B(a)$

or $T \mapsto x{:}A{\times}B(x)$, *in which case*
A Type$_\tau \wedge \forall a \in_\tau A.\, B(a)$ Type$_\tau$, *and*
$\forall t.\, \epsilon(t)$ *iff* $t \cong \langle a, b\rangle \wedge a \in_\tau A \wedge b \in_\tau B(a)$

or $T \mapsto \overline{A}$, *in which case*
$A{\downarrow} \Rightarrow A$ Type$_\tau$, *and*
$\forall t.\, \epsilon(t)$ *iff* $t{\downarrow} \Rightarrow t \in_\tau A$

Lemma 6 Ψ *is a monotone operator on type interpretations.*

Definition 20 ν *is the least fixed-point of the monotone operator* Ψ.

Hereafter we will be working in type interpretation ν, and subscripts may be dropped: $a \in A \stackrel{\text{def}}{=} a \in_\nu A$, and A Type $\stackrel{\text{def}}{=} A$ Type$_\nu$. With this definition in place, numerous lemmas about these relations can be proven either directly or by straightforward induction on the definition of ν.

Lemma 7 *(i)* $a \in A$ *implies* A Type.

(ii) A Type *implies* $A{\downarrow}$.

(iii) $c[a] \in C[A]$ *and* $a \cong b$ *and* $A \cong B$ *implies* $c[b] \in C[B]$.

Now some lemmas about typehood and membership which follow directly from the definitions are proven. a, b, c, A, and B may be taken to be arbitrary closed terms.

Lemma 8 *Type formation is characterized by the following properties.*

(i) E Type, N Type.

(ii) $a{\downarrow}$ Type, $a \sqsubseteq\!\!\!\sim b$ Type.

(iii) a `in` A Type *iff* A Type.

(iv) $x{:}A{\rightarrow}B(x)$ Type *iff* A Type *and for all* $a \in A$, $B(a)$ Type.

(v) $x{:}A{\times}B(x)$ Type *iff* A Type *and for all* $a \in A$, $B(a)$ Type.

(vi) $\overline{A}$ Type *iff* $A{\downarrow}$ *implies* A Type.

(vii) $A \in \mathcal{A}$ *and* A *closed implies* A Type.

Lemma 9 *Type membership is characterized by the following properties.*

(i) $c \in E$ *iff* c *closed.*

(ii) $c \in a{\downarrow}$ *iff* $c \cong 0$ *and* $a{\downarrow}$.

(iii) $c \in a \sqsubseteq\!\!\!\sim b$ *iff* $c \cong 0$ *and* $a \sqsubseteq\!\!\!\sim b$.

(iv) $c \in \mathrm{N}$ *iff* $c \cong 0, 1, 2, \ldots$.

(v) $c \in (a\ \mathtt{in}\ A)$ *iff* $(a\ \mathtt{in}\ A)$ Type *and* $c \cong 0$ *and* $a \in A$.

(vi) $c \in x{:}A{\to}B(x)$ *iff* $x{:}A{\to}B(x)$ Type *and* $c \cong \lambda x.c(x)$ *and for all* $a \in A$, $c(a) \in B(c)$.

(vii) $c \in x{:}A{\times}B(x)$ *iff* $x{:}A{\times}B(x)$ Type *and* $c \cong \langle \pi_1(c), \pi_2(c) \rangle$ *and* $\pi_1(c) \in A$ *and* $\pi_2(c) \in B(\pi_1(c))$.

(viii) $c \in \overline{A}$ *iff* $\overline{A}$ Type *and if* $c{\downarrow}$, $c \in A$.

Most of the rules for individual types can be justified directly from the preceding Lemmas. The one exception is the fixed point rules, which require a more in-depth analysis.

4.4. Soundness of the fixed point rules

The fixed point typing principle is not true for all types, illustrated by the counterexample presented in section 3.3. Here we show the principle is in fact true for the admissible types: if $f \in \overline{A}{\to}\overline{A}$, and A is admissible, then $\mathtt{Y}(f) \in \overline{A}$. The fixed point induction principle is also proven sound. The theorems are proved by induction on the definition of the admissible types.

We will use the shorthand $f^{j\uparrow}$ for $\{f^k(\mathtt{bot}) \mid k \geq j\}$; when it is not ambiguous, notation will be stretched to make assertions such as "$f^{j\uparrow} \in A[f^{j\uparrow}/x]$", meaning $f^k(\mathtt{bot}) \in A[f^k(\mathtt{bot})/x]$ for all $k \geq j$.

Theorem 6 (Fixed point typing) *If* $A \in \mathcal{A}$ *and there exists* j *with* $f^k(\mathtt{bot}) \in \overline{A}$ *for all* $k \geq j$, *then* $\mathtt{Y}(f) \in \overline{A}$.

Proof. This theorem is proved by first generalizing to an arbitrary context $a[\bullet]$ containing $f^k(\mathtt{bot})$ or $\mathtt{Y}(f)$, from which the theorem directly follows. Types in $\mathcal{A}$ and $\mathcal{B}$ have different properties that are proved of them, so we establish two properties by simultaneous induction:

(i) For arbitrary $a[\bullet]$ and $A \in \mathcal{A}$, if there exists j such that $a[f^{j\uparrow}] \in A$ then $a[\mathtt{Y}(f)] \in A$.

(ii) For arbitrary $a[\bullet]$ and $A \in \mathcal{B}^{\{x\}}$ and x being the only free variable in A, if there exists j such that $a[f^{j\uparrow}(\mathtt{bot})] \in A[f^{j\uparrow}(\mathtt{bot})/x]$ then $a[\mathtt{Y}(f)] \in A[\mathtt{Y}(f)/x]$.

Proceed by induction on the definition of $\mathcal{A}$. Consider case $A \in \mathcal{A}$. From the assumption $a[f^{j\uparrow}(\mathtt{bot})] \in A$, show $a[\mathtt{Y}(f)] \in A$ by case analysis on the form of A.
case $A = \mathrm{E}$ or N: For E, the result is trivial, for N it is computationally clear: $\mathtt{Y}(f) \cong f(\mathtt{Y}(f))$ by computing, and $\mathtt{bot} \sqsubseteq \mathtt{Y}(f)$, so $f^k(\mathtt{bot}) \sqsubseteq \mathtt{Y}(f)$ by Theorem 4.
case $A = b{\downarrow}$ or $b \sqsubseteq c$: Computationally obvious just as above, because $a[f^{j\uparrow}]$ and $a[\mathtt{Y}(f)]$ can at most compute to 0.
case $A = b\ \mathtt{in}\ B$: To show $a[\mathtt{Y}(f)] \in b\ \mathtt{in}\ B$ it suffices to show
$a[\mathtt{Y}(f)] \cong 0$ and $b \in B$,

using Lemma 9 (uses of which we refrain from citing hereafter). Both of these conditions are direct by assumption.

case $A = x{:}B{\to}C$: To show $a[\mathtt{Y}(f)] \in x{:}B{\to}C$ it suffices to show

$a[\mathtt{Y}(f)] \cong \lambda x.d$ for some d and for all $b \in B$, $a[\mathtt{Y}(f)](b) \in C[b/x]$.

The first condition is computationally direct from the assumption $a[f^{j\uparrow}(\mathtt{bot})] \cong \lambda x.d'$. For the second condition, assume $b \in B$; by assumption, $a[f^{j\uparrow}](b) \in C[b/x]$. So, since A is defined in terms of $C[b]$, the induction hypothesis may be applied to complete the proof of this case (let $a[\bullet]$ there be $a(\bullet)(b)$).

case $A = x{:}B{\times}C$: To show $a[\mathtt{Y}(f)] \in x{:}B{\times}C$ it suffices to show

$a[\mathtt{Y}(f)] \cong \langle b, c\rangle$ and $\pi_1(a[\mathtt{Y}(f)]) \in B$ and $\pi_2(a[\mathtt{Y}(f)]) \in C[\pi_1(a[\mathtt{Y}(f)])/x]$

The first condition is computationally direct from the assumption $a[f^{j\uparrow}(\mathtt{bot})] \cong \langle b', c'\rangle$ for some j. For the second condition $\pi_1(a[\mathtt{Y}(f)]) \in B$, by assumption we have $\pi_1(a[f^{j\uparrow}]) \in B$ for some j. So, since A is defined in terms of B, the induction hypothesis may be applied to prove this condition. For the third condition, since $\pi_2(a[f^{j\uparrow}]) \in C[a[f^{j\uparrow}]/x]$ for some j, and $C \in \mathcal{B}^{\{x\}}$ and thus $C[a[x]/x] \in \mathcal{B}^{\{x\}}$, the conclusion follows by induction hypothesis.

case $A = \overline{B}$: To show $a[\mathtt{Y}(f)] \in \overline{B}$ it suffices to show

if $a[\mathtt{Y}(f)]{\downarrow}$ then $a[\mathtt{Y}(f)] \in B$.

So, assuming $a[\mathtt{Y}(f)]{\downarrow}$, show $a[\mathtt{Y}(f)] \in B$. Since $\{f^k(\mathtt{bot}) \mid k \in \mathrm{N}\} \cong_{\mathrm{s}} \{\mathtt{Y}(f)\}$ (by Theorem 4) and by the definition of $\cong_{\mathrm{s}}$, there is some j' for which $a(f^{j'}(\mathtt{bot})){\downarrow}$; all larger approximations also clearly terminate. By assumption,

$\forall k \geq j$. if $a[f^k(\mathtt{bot})]{\downarrow}$ then $a[f^k(\mathtt{bot})] \in B$

for some j. Picking j'' to be $\max(j, j')$, $a[f^{j''\uparrow}] \in B$, which inductively yields the result.

Now consider case (ii), $A \in \mathcal{B}^{\{x\}}$.

case $A = \mathrm{E}$ or N: same as in case (i).

case $A = b \sqsubseteq c$: By hypothesis, $a[f^{j\uparrow}] \in (b[f^{j\uparrow}] \sqsubseteq c[f^{j\uparrow}])$ for some j. It suffices to show $\{b[f^k(\mathtt{bot})] \mid k \geq j\} \sqsubseteq_{\mathrm{s}} \{c[f^k(\mathtt{bot})] \mid k \geq j\}$ by two uses of Theorem 4. And, this in turn follows by the assumption and Lemma 5 case (iv).

case $A = b{\downarrow}$: If $b[f^k(\mathtt{bot})]{\downarrow}$, then $b[\mathtt{Y}(f)]{\downarrow}$ by the definition of $\sqsubseteq$.

case $A = b$ **in** B: To show $a[\mathtt{Y}(f)] \in b[\mathtt{Y}(f)/x]$ **in** $B[\mathtt{Y}(f)/x]$ it suffices to show $a[\mathtt{Y}(f)] \cong 0$ and $b[\mathtt{Y}(f)/x] \in B[\mathtt{Y}(f)/x]$. Both of these conditions are direct by assumption.

case $A = y{:}B{\to}C$: First, B contains no free x by inspection of the definition of $\mathcal{B}^{\{x\}}$. To show $a[\mathtt{Y}(f)] \in y{:}B{\to}C[\mathtt{Y}(f)/x]$ it suffices to show

$a[\mathtt{Y}(f)] \cong \lambda y.d$ and for all $b \in B$, $a[\mathtt{Y}(f)](b) \in C[\mathtt{Y}(f)/x][b/y]$.

This in turn follows by similar reasoning as for $A \in \mathcal{A}$.

case $A = y{:}B{\times}C$: To show $a[\mathtt{Y}(f)] \in y{:}B[\mathtt{Y}(f)/x]{\times}C[\mathtt{Y}(f)/x]$ it suffices to show $a[\mathtt{Y}(f)] \cong \langle b, c\rangle$, $\pi_1(a[\mathtt{Y}(f)]) \in B[\mathtt{Y}(f)/x]$ and $\pi_2(a[\mathtt{Y}(f)]) \in C[\mathtt{Y}(f)/x][\pi_1(a[\mathtt{Y}(f)])/y]$.

The first condition is computationally direct from the assumption $a[f^{j\uparrow}(\mathtt{bot})] \cong \langle b', c'\rangle$ for some j. For the second condition $\pi_1(a[\mathtt{Y}(f)]) \in B[\mathtt{Y}(f)/x]$, by assumption,

$\exists j.\ \pi_1(a[f^{j\uparrow}]) \in B[f^{j\uparrow}/x]$.

So, the induction hypothesis may be applied to prove this condition. For the third condition, since $\pi_2(a[f^{j\uparrow}]) \in C[f^{j\uparrow}/x][\pi_1(a[f^{j\uparrow}])/x]$, and by the definition of $\mathcal{B}^{\{x\}}$, $C \in \mathcal{B}^{\{x\}}$, so $C[\pi_1(a[x])/y] \in \mathcal{B}^{\{x\}}$ and the conclusion follows by induction hypothesis.

case $A = \overline{B}$: To show $a[\mathtt{Y}(f)] \in \overline{B[\mathtt{Y}(f)/x]}$ it suffices to show

if $a[\mathtt{Y}(f)]\downarrow$ then $a[\mathtt{Y}(f)] \in B[\mathtt{Y}(f)/x]$.

So, assuming $a[\mathtt{Y}(f)]\downarrow$, show $a[\mathtt{Y}(f)] \in B[\mathtt{Y}(f)/x]$. Since $\{f^k(\mathtt{bot}) \mid k \in \mathrm{N}\} \cong_{\mathrm{s}} \{\mathtt{Y}(f)\}$ (by Theorem 4) and by the definition of $\cong_{\mathrm{s}}$, there is some j' for which $a(f^{j'}(\mathtt{bot}))\downarrow$; all larger approximations also clearly terminate. By assumption,

$\forall k \geq j$. if $a[f^k(\mathtt{bot})]\downarrow$ then $a[f^k(\mathtt{bot})] \in B[f^k(\mathtt{bot})/x]$

for some j. Picking j'' to be $\max(j, j')$, $a[f^{j''\uparrow}] \in B[f^{j''\uparrow}/x]$, which inductively yields the result. □

Theorem 7 (Fixed point induction) *If*

(i) $a(f(\mathtt{bot})) \in P(f(\mathtt{bot}))$,

(ii) $(\forall b \in B.\, a(b) \in P(b) \Rightarrow a(f(b)) \in P(f(b)))$,

(iii) $x{:}\overline{B}{\times}P(x) \in \mathcal{A}$, *and* $f \in \overline{B}{\rightarrow}\overline{B}$,

then $a(\mathtt{Y}(f)) \in P(\mathtt{Y}(f))$.

Proof. From the assumptions, prove $a(\mathtt{Y}(f)) \in P(\mathtt{Y}(f))$. First, proceed by cases on whether $f(\mathtt{bot}) \cong \mathtt{bot}$.[a]

case $f(\mathtt{bot}) \cong \mathtt{bot}$: Then, $f^n(\mathtt{bot})\uparrow$ for all n by a direct induction. Thus, $\{f^k(\mathtt{bot}) \mid k \in \mathrm{N}\} \cong_{\mathrm{s}} \{\mathtt{bot}\}$, and since $\{f^k(\mathtt{bot}) \mid k \in \mathrm{N}\} \cong_{\mathrm{s}} \{\mathtt{Y}(f)\}$ by Theorem 4, the conclusion follows by assumption (i).

case $f^k(\mathtt{bot})\downarrow$: We first show $a(f^k(\mathtt{bot})) \in P(f^k(\mathtt{bot}))$ for all $k \geq 1$. For $k = 1$, this is assumption (i); assuming $a(f^k(\mathtt{bot})) \in P(f^k(\mathtt{bot}))$ for $k \geq 1$, show $a(f^{k+1}(\mathtt{bot})) \in P(f^{k+1}(\mathtt{bot}))$. Since $f^k(\mathtt{bot}) \in \overline{B}$, $f^k(\mathtt{bot}) \in B$ by assumption $f^k(\mathtt{bot})\downarrow$. Thus from the assumption (ii), $a(f^{k+1}(\mathtt{bot})) \in P(f^{k+1}(\mathtt{bot}))$. $P \in \mathcal{B}^{\{x\}}$ by inspection of the definition of $\mathcal{B}$, so from the second case of the fixed point typing theorem (Theorem 6) letting j there be 1, $a(\mathtt{Y}(f)) \in P(\mathtt{Y}(f))$. □

4.5. Soundness of the rules

All of the pieces of the interpretation are in place, and it only remains to show each of the rules are sound. Let $\mathcal{G}$ stand for one of $b \in B$ and B Type.

Definition 21 *Validity of a sequent*

$x_1{:}A_1, x_2{:}A_2, \ldots x_n{:}A_n \models \mathcal{G}$

is defined as

$\forall a_1 \in A_1, a_2 \in A_2[a_1/x_1], \ldots, a_n \in A_n[a_1/x_1, \ldots, a_{n-1}/x_{n-1}].$
$\mathcal{G}[a_1/x_1, \ldots, a_n/x_n].$

Theorem 8 (Soundness) *If* $\Gamma \vdash \mathcal{G}$ *then* $\Gamma \models \mathcal{G}$.

[a] Observe this axiom thus is not constructively validated; a slightly weaker version with $\overline{B}$ replacing B is constructively valid.

Proof. The computation rules each follow directly from one of the cases of Lemma 4. The type formation rules follow from Lemma 8. Rules for individual types, with the exception of the fixed point rules, follow from Lemmas 7 and 9. (Fixed point) follows from Theorem 6, and (FP induction) follows from Theorem 7. The miscellaneous rules are either direct from the definition of $\models$ ((Cut), (Cut type), and (Hypothesis)), follow from Lemma 7 ((Subst) and (Subst type)), or are direct by inspection of the definition of ν ((Contradict), (Prop member), (Type termination)).

□

Theorem 9 (consistency) HTT *is logically consistent, i.e. there exist types which cannot be proven inhabited.*

Proof. The type $0 \sqsubseteq 1$ has no members by inspection of the definition of ν, so for no a is it the case that $\models a \in 0 \lesssim 1$. Therefore, for no a is $\vdash a \in 0 \sqsubseteq 1$ provable, by Theorem 8. □

5. Conclusion

A hybrid partial-total type theory has been presented. The theory has all of the features of a programming logic designed to type and reason about partial functions, as well as the standard total types of constructive type theory. The fixed point typing principle may be used over a wide range of types (in particular all of those types not containing the atomic proposition types a `in` A, $a\downarrow$, and $a \sqsubseteq b$). However, the collection of admissible types could be enriched still further. An alternative solution to achieve this is to axiomatize $\sqsubseteq_s$ within the theory; using this approach, fixed points may be typed and fixed point induction justified from more basic principles; see (Ref. 34). This hybridization is in principle possible in Martin-Löf style type theory, but as was discussed in the introduction, lack of an ordering $\sqsubseteq$ makes it very difficult to formulate a useful induction principle. See the Appendix of (Ref. 32) for a hybrid version of the Nuprl theory. The induction rule there is quite restrictive when compared to fixed point induction as axiomatized herein.

There has been some work to give a category-theoretic framework for partial computations in a total-type framework as we do here, using categorical monad constructions. Moggi's approach[24] is more abstract than that carried out here, because the monad operator $T(A)$ represents general computations over type A, not just potential divergence $\overline{A}$. Crole and Pitts[10] add a fixed point object to the monad, allowing fixed points to be typed. They work over base type N and simple functions $A \rightarrow B$ only, so the difficult issue of non-admissible types does not arise. Audebaud has developed a version of partial types for the Calculus of Constructions.[4] Again, the lack of (strong) dependent products in the Calculus of Constructions keeps all types admissible.

Martin-Löf has developed a Partial Type Theory. However his theory is not a hybrid—all types are partial, and thus the propositions-as-types principle fails and no logical reasoning is possible. Palmgren has elaborated upon Martin-Löf's ideas.[28]

By adding bar types we allow diverging computations to be typed. An alternate approach is to add types to allow arbitrary well-ordering types to be constructed and use these to prove functions are total.[26,8,15] Howewver there are weaknesses to this approach—it creates a significant overhead and does not allow general partial correctness results to be proven. There is also no technique for overhead-free extraction of fixed points, something the partial propositions allow in **HTT**.

Acknowledgements

Thanks to Robert Constable, Stuart Allen, Doug Howe, Nax Mendler, and Carolyn Talcott for comments on earlier versions of this work.

References

1. P. Aczel. Frege structures revisited. In B. Nordström and J. Smith, editors, *Proceedings of the 1983 Marstrand Workshop*, 1983.
2. 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.
3. 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.
4. P. Audebaud. Partial objects in the calculus of constructions. In *Sixth Symposium on Logic in Computer Science*, 1991.
5. 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.
6. M. J. Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.
7. 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.
8. 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.
9. 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.
10. R. L. Crole and A. H. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.
11. J. W. deBakker and D. Scott. A theory of programs. unpublished Notes, 1969.
12. 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.
13. S. Feferman. Logics for termination and correctness of functional programs. In *Proceedings of the conference "Logic from computer science"*. MSRI, 1989.
14. M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential

control. *Theoretical Computer Science*, 52:205–237, 1987.

15. D. Galmiche. Program development in constructive type theory. *Theoretical Computer Science*, 94:237–259, 1992.
16. 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.
17. R. Harper. Constructing type systems over an operational semantics. *J Symbolic Computation*, 14:71–84, 1991.
18. S. Hayashi and H. Nakano. *PX: a Computational Logic*. MIT press, 1989.
19. D. J. Howe, 1988. personal communication.
20. S. Igarashi. Admissibility of fixed-point induction in first-order logic of typed theories. Technical Report Stan-CS-72-287, Stanford University Computer Science Department, 1972.
21. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
22. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
23. I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. (Submitted; currently available as URL file://ftp.cs.jhu.edu/pub/scott/fosdt.ps.Z), 1994.
24. E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
25. J. H. Morris. *Lambda calculus models of programming languages*. PhD thesis, MIT, 1968.
26. B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.
27. B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's type theory*. Oxford Science Publications, 1990.
28. E. Palmgren. An information system interpretation of Martin-Löf's partial type theory with universes. *Information and Computation*, 106:26–60, 1993.
29. L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge, 1987.
30. G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.
31. D. Scott. Lattice theoretic models for various type-free calculi. In *Proceedings of the 4th International Congress in Logic, Methodology, and Philosophy of Science*, Amsterdam, 1972. North Holland.
32. S. F. Smith. Partial objects in type theory. Technical Report 88-938, Department of Computer Science, Cornell University, August 1988. Ph.D. Thesis.
33. S. F. Smith. From operational to denotational semantics. In *MFPS 1991*, volume 598 of *Lecture notes in Computer Science*, pages 54–76. Springer-Verlag, 1992.
34. S. F. Smith. Partial computations in constructive type theory. Technical Report 90-20, Department of Computer Science, The Johns Hopkins University 21218, 1992.
35. C. L. Talcott. A theory for program and data specification. *Theoretical Computer Science*, 104:129–159, 1993.