Partial Computations in Constructive Type Theory

Scott F. Smith*
The Johns Hopkins University

February 21, 1991

Abstract

Constructive type theory as conceived by Per Martin-Löf has a very rich type system, but partial functions cannot be typed. This also makes it impossible to directly write recursive programs. In this paper a constructive type theory **Red** is defined which includes a *partial type* constructor $\overline{A}$; objects in the type $\overline{A}$ may diverge, but if they converge, they must be members of A. A *fixed point typing principle* is given to allow typing of recursive functions. The extraction paradigm of type theory, whereby programs are automatically extracted from constructive proofs, is extended to allow extraction of fixed points. There is a Scott fixed point induction principle for reasoning about these functions. Soundness of the theory is proven. Type theory becomes a more expressive programming logic as a result.

1 Introduction

Constructive Type Theory (CTT), originally developed by Per Martin-Löf [Mar82], is a foundational theory of constructive mathematics. It is also of interest to computer scientists because it contains a programming language with a very rich type system, and may be used to effectively reason about programs. Two features of type theory make it a particularly good programming logic. The collection of types is broad and flexible, so very precise and modular typings may be given to programs (MacQueen's modules for ML

Author's address: Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218 USA. Email: scott@cs.jhu.edu. Fax: 301.338.6134. Phone: 301.338.5299

[Mac86] were largely inspired by type theory). And, the connection between a (constructive) proof and a program is made explicit to the point that a program can be automatically *extracted* from a proof, equating the tasks of programming and proving, and offering a possibly better paradigm for the development of correct programs.

A great shortcoming of CTT as a programming logic is the inability to type and reason about partial functions and general recursive functions, because all function spaces are total. The main contribution of this paper is the extension of CTT to have types for partial functions, along with a useful collection of principles that apply to them. A new type constructor $\overline{A}$ is added which additionally allows nonterminating computations to inhabit the underlying type A; $A \to \overline{A}$ is then a space of partial functions on A. We present a new CTT **Red** to illustrate these ideas. **Red** is a theory in which we may directly type both partial and total computations, achieving the union of the notions of CTT type and programming language type. There are three main principles governing the use of partial types. If an object is in a partial type $\overline{A}$ and halts, 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 general recursive programs. This *fixed point typing principle* is the subject of much of the metamathematical investigation of this paper, for its justification is difficult and its uses surprisingly broad. Importantly, it allows arbitrary recursive programs to be written as well as extracted from proofs, and we will argue this increases the potential use of the extraction paradigm. Lastly, a Scott-style fixed point induction rule allows for direct inductive reasoning about partial computations. Fixed point induction and the fixed point typing principle are together call the *fixed point principles.* An alternative axiomitization for partial types, **Pure Red**, allows instances of the fixed point principles to be derived from more basic properties of computations.

1.1 Predacessors to this work

This work grew from a desire to extend the Nuprl type theory developed at Cornell University by Constable and others [CAB+86] to encompass non-total computations. Initial results were published in [CS87].

Nuprl is a descendant of the Intuitionistic Type Theory of Martin-Löf[Mar73, Mar82, Mar84], and in particular of [Mar82, Mar84], which we refer to as ETT ("Extensional type theory"), while his 1973 theory is ITT ("Intensional Type Theory"). The Nuprl theory is built upon ETT in that it

has all of the basic type constructors, and in addition has a subset type $\{x{:}A \mid P(x)\}$ which corresponds to the subset axiom of set theory, a general recursive type definition mechanism which subsumes Martin-Löf's W type of well-orderings, and a quotient type. Nuprl also takes a different view of computation: untyped computations are sensible, and types are collections of what were originally untyped computations; the type-free computation rules reflect this. Feferman class theories ([Fef75, Fef89]; some related theories are PX [HN89] and IOCC [Tal90]) take a similar approach. **Red** is even closer to Feferman class theory than existing CTTs: it includes a type E of all expressions, and a type-free equality $a \sim b$. In ETT and Nuprl, part of type formation is the definition of what it means for two terms to be equal in the type, $a = a' \in A$. In **Red** and in Feferman class theory, types/classes do not come with an equality, so either the type-free equality is used, or some other equality is explicitly defined.

1.2 Outline of the paper

The primary intention of this paper is to define principles for extending CTT by adding partial types, and to show a more general, usable and expressive theory is obtained.

First, elementary concepts of CTT are reviewed in section 2. In section 3 the type theory **Red** is defined. Section 4 fulfills the important aim of showing the usability of the theory through numerous examples. Then, a family of extensions to **Red** is given in section 5, including **Pure Red**, an alternative theory for typing and reasoning about fixed point computations, a classical extension **Classical Red**, and an extension **Full Red**, which includes a more extensive collection of types. The soundness of **Red** is proven in section 6. Little other metamathematical development is undertaken, for soundness is the primary objective.

2 Background

This section is a short survey of the important issues in the design and use of constructive type theories; for fuller descriptions and examples, see one or more of [CAB$^+$86, NPS90, BCMS89].

Some of the simpler systems that are subsystems of CTT include the typed λ-calculus [HS86, LS86], and Gödel's functionals of finite type (see [Ste72], ch. 7 or [Bar84], app. A).

There are also a family of impredicative type theories, the Calculus of Constructions and extensions thereof [CH88, Luo89], which because of their inherent self-referentiality differ significantly from Martin-Löf-style theories. Partial types may also be defined for impredicative theories, but that is beyond the scope of this paper.

Martin-Löf has written extensively on how type theory is a foundation for constructive mathematics [Mar83, Mar82, Mar84], and gives clear and careful justifications for the meanings of assertions in the theory.

A type theory has as its language a collection of *terms*. Some of these terms represent types, others computations (the two are not strictly separated); *types* are collections of terms. *Assertions* (called judgements by Martin-Löf) express the truths of type theory; the statement $a \in A$ asserts that a 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 fixed formal system, for more types, computations, and rules may be added at some future date; the theory is thus *open-ended*.

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.

2.1.1 Atomic data types

N is a type of natural numbers $0, 1, 2, \ldots$; E is a type of all terms.

2.1.2 Atomic propositional types

These types represent atomic propositions via the principle of propositions as types (discussed in section 2.3). In **Red**, *a in A* is a type which represents the assertion $a \in A$; types of this form are empty if the proposition is false, and have one placeholder member if true.

2.1.3 Functions

The type $A \to B$ is the space of all total functions from A to B. There is no provision for defining partial functions in traditional CTT. The function space may be generalized to a *dependent* function space $x{:}A \to B(x)$ where $B(x)$ depends on x: the range type $B(x)$ may depend on the value the

function is applied on. These types are also notated $\Pi x{:}A.B$. An informal example is

$$f \in n{:}\mathrm{N} \to \overbrace{\mathrm{N} \times \ldots \times \mathrm{N}}^{n};$$

$f(m)$ is thus an m-ary tuple of natural numbers. Dependent functions are useful for abstracting code with respect to type, and were an inspiration for MacQueen's implementation of modules in ML [Mac86].

2.1.4 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 $x{:}A \times B(x)$: the type $B(x)$ depends on the member of the type A. Such types are also notated $\Sigma x{:}A.B(x)$. This introduces a left-to-right dependency in the product: first an element a is placed in A, and b can be placed in $\in B(a)$, to give $\langle a, b \rangle \in x{:}A \times B(x)$.

2.1.5 Unions

The type $A + B$ is the disjoint union of types A and B. If $a \in A$, a is injected into the union by $inl(a) \in A + B$, and similarly $inr(b) \in A + B$.

2.1.6 Subtypes

$\{x{:}A \mid P(x)\}$ denotes those elements of A which have property $P(x)$. Subtypes are information hiders, because members of $P(x)$ cannot be accessed.

2.1.7 Recursive and infinite types

$rec(X.A(X))$ denotes the type of least membership which is a solution to the type equation $X = A(X)$, 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 $rec(t.\mathbf{1} + B \times X)$; constructive ordinals may be expressed by the type $rec(X.\mathbf{1} + \mathrm{N} \to X)$. The W-type of ETT may be expressed by the recursive type $rec(X.x{:}A \times B \to X)$.

$inf(X.A(X))$ denotes the greatest type which is a solution to the type equation $X = A(X)$. For instance, $inf(X.\mathrm{N} \times X)$ represents a stream of natural numbers. Recursive and infinite types for type theory have been developed by Mendler [Men87, CM85, MPC86, CAB+86].

2.1.8 Universes

Type universes ("large types") have types as members. In Nuprl and ETT there is an infinite hierarchy of universes $\mathrm{U}_1, \mathrm{U}_2, \ldots$; U_1 has as members all of those types closed under the type constructors excluding the U_i. U_2 is then constructed by closing over all of the U_1 types plus the large type U_1. In this fashion a hierarchy is constructed. Below it will be shown how the presence of universes directly leads to higher-order logical reasoning.

This hierarchy is *predicative* because new objects are always defined solely in terms of existing objects. In The Calculus of Constructions [CH88], there are *impredicative* large types *Prop* and *Type*, with $\mathit{Prop} \in \mathit{Type}$. *Prop* is impredicative: $X{:}\mathit{Prop} \to X \in \mathit{Prop}$, so types in *Prop* may quantify over *Prop* itself. Because of this self-referentiality, it is impossible to inductively define those types that are in *Prop*.

2.2 Equality

In Nuprl and ETT, for a type to exist means not only to know its members, but also to know when two members are equal. The assertion $a \in A$ is replaced with a partial equivalence relation (PER)* $a = a' \in A$, meaning a and a' are equal members of the type A. For example, function equality could be defined to be extensional equality: given two functions $f, f' \in A \to B$,

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

The disadvantage to a PER approach is it forces a particular equality on terms that is sometimes not the desired one; also, for ETT and Nuprl, equals cannot always be substituted for equals.

Another solution is to have a type-free equality on terms $a \sim b$. It is possible then to always allow substitution of equals for equals. **Red** takes this approach, as does Feferman class theory [Fef75].

2.3 The propositions-as-types principle

Formal logical reasoning may be embedded in CTT 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 member. This

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

principle is due to Curry, Howard, and DeBruijn amongst others, but importantly it was Scott [Sco70] who first formulated a theory where the connection was semantic and not proof-theoretic, because in his theory no extra axioms are needed to reason constructively.

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

$$P \stackrel{\text{def}}{=} \forall n{:}\mathrm{N}.\ \mathrm{prime}(n) \vee \neg\mathrm{prime}(n)$$

is to have a decision procedure for whether n is prime or not. In type theory, the type

$$P^* \stackrel{\text{def}}{=} n{:}\mathrm{N} \to \mathit{prime}(n) + \neg\mathit{prime}(n)$$

represents the above proposition, which has as members functions which given a natural number n, return whether or not n is prime. Any member of this type is a *validation* for P.

DEFINITION 2.1 *Propositions are translated to types as follows:*

$$\begin{array}{ll}
(A \Rightarrow B)^* & \stackrel{\text{def}}{=} A^* \to B^* \\
(A \,\&\, B)^* & \stackrel{\text{def}}{=} A^* \times B^* \\
(A \vee B)^* & \stackrel{\text{def}}{=} A^* + B^* \\
(\forall x{:}A.\ B)^* & \stackrel{\text{def}}{=} x{:}A^* \to B^* \\
(\exists x{:}A.\ B)^* & \stackrel{\text{def}}{=} x{:}A^* \times B^* \\
\mathit{false}^* & \stackrel{\text{def}}{=} \textbf{void}, \textit{ an empty type} \\
\neg A^* & \stackrel{\text{def}}{=} A^* \to \textbf{void}
\end{array}$$

We can sketch how any member of P^* validates the proposition 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 $\mathit{inl}(a) \in A^* + B^*$ or $\mathit{inr}(b) \in A^* + B^*$. 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 $f \in A^* \to B^*$; 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 $p \in P(x)^*(a)$, so p validates $P(a)$; $\exists x{:}A.\ P(x)$ is thus validated.

CASE $\forall x{:}A.\ P(x)$ true: This means $f \in x{:}A \to P(x)^*$; thus, for all $a \in A$, $f(a) \in P(a)^*$, meaning $f(a)$ validates $P(a)$; $\forall x{:}A.\ P(x)$ is thus validated.

An atomic proposition is translated into an atomic type which is inhabited just when the proposition is true. For example, in **Red** $a \in A$ is translated to the type a *in* A; this type has member 0 just when $a \in A$. Type universes can also be viewed as proposition universes: $A \to \mathrm{U}_1$ is a type of predicates on A; quantifying over all objects $P \in (A \to U1)$ amounts to quantifying over all predicates in the first universe.

2.4 Extraction

The rules of CTT may be 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 any member of the type P^*. Refinement-style rules may be implemented with *extract forms*, which automatically synthesize the validation up the proof tree. For example, for product $A \times B$, the rule

$$\begin{array}{l} \vdash \langle a, b\rangle \in A \times B \\ \quad \vdash a \in A \\ \quad \vdash b \in B \end{array}$$

has extract form

$$\begin{array}{l} \vdash A \times B \text{ extract } \langle a,b\rangle \\ \quad \vdash A \text{ extract } a \\ \quad \vdash B \text{ extract } b. \end{array}$$

which means when proofs of A and B are complete, a and b can be extracted; the pair $\langle a, b\rangle$ is then extracted from $A \times B$. Recall that $A \times B$ is the type which represents conjunction; this is confirmed by the fact that the above rule is identical to an and-introduction rule. Extract forms of all of the rules of type theory may be formulated, and are part of the Nuprl type theory. It is also possible to automatically extract objects from non-extract style rules using unification, as in the Isabelle implementation of CTT [PN90]. In this system, the initial goal is entered as $?a \in A$, where $?a$ is a metavariable and A is a type representing the proposition a proof is desired of; as the proof proceeds, $?a$'s structure is incrementally revealed.

2.5 CTT and mechanical proof systems

Several computer implementations of CTTs have been undertaken, the most ambitious being the Nuprl system [CAB+86]. The Nuprl system implements the Nuprl logic, and includes some primitive procedures for automated proof construction, patterned after the tactics of LCF [GMW79].

A fair chunk of mathematics has been developed in Nuprl. Saddleback search and quicksort have both been proven correct in Nuprl [How88]. 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, How88]. Basin has given a constructive development of Ramsey theory in Nuprl [Bas88].

3 The Theory

We define here a type theory of partial typed computations, **Red**. **Red** is not a full type theory, but is the core of a full type theory; special effort has been made to include features that give theoretical difficulties, and less problematic features, even though very useful, are left out for conciseness of presentation. In section 5 we sketch an extension **Full Red** which is a full type theory, including type universes, subtypes, and recursive types, more the home we imagine for these concepts. Two other type theories for partial computations are presented in [Smi88], one of which is a direct addition of partial types to Nuprl, and the other which is more intensional than the theory given here.

3.1 The terms

The theory has one sort, terms, which includes types and computations. This means there is no forced 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 (at the term level, types are untyped terms).

DEFINITION 3.1 (TERMS) *The terms of* **Red** *are*

(i) countably many variables (never directly written),

(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, $a \sim b$, $a{\downarrow}$

(iv) computation constructors $pred(a)$, $succ(a)$, $if\_zero(a; b; c)$, $a.1$, $a.2$, $a(b)$

(v) radioactive terms a^{r}

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

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

The radioactive labels a^{r} are for the purposes of keeping track of certain subterms for showing types admissible, and can usually be ignored. Some conventions are defined to keep their imposition minimal. Unless otherwise stated, terms as written do not show their radioactivity: $\lambda x.b$ refers to either $\lambda x.b$ or $\lambda x.b^{\mathrm{r}}$. Given some term b (radioactive or not), b^{r} is b with all subterms radioactively labeled.

3.2 Assertions

All assertions are sequents, and take two forms: we may assert A to be a type, and assert a to be a member of type 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 $?$ is of the form $x_1{:}A_1, x_2{:}A_2, \ldots x_n{:}A_n$, and signifies reasoning takes place under assumptions $x_1 \in A_1$, $\ldots$, $x_n \in A_n$. Two forms of assertion may be made; the first is

$$? \vdash A\ \mathrm{Type}_i$$

which asserts that under assumptions $?$, A is a type at level $i \in \{0, 1, 2, 3, 4, 5\}$. $A\ \mathrm{Type}_0$ means A is a type, and will thus be abbreviated A Type. $A\ \mathrm{Type}_1$ if and only if A is *admissible*, meaning the fixed point typing principle and fixed point induction are valid for A. The type levels $2, 3, 4, 5$ are used in proving types admissible. The second form of assertion is

$$? \vdash a \in A$$

which asserts under assumptions $?$, a inhabits type A.

3.3 Rules and proofs

The rules will be presented refinement-style, giving a goal sequent, folowed by subgoal sequents which if proven imply the truth of the goal.

We confine various technical conventions for the presentation of the rules to this paragraph. The hypothesis list is always increasing going from goal to subgoals, even though subgoals do not list goal hypotheses. In the hypothesis list, x_i may occur free in any A_{i+j} for positive j. The free variables in the conclusion are no more than the x_i. α-conversion is an unmentioned rule. To improve readability, the assertion $0 \in a \mathrel{\sim} b$ will be abbreviated $a \mathrel{\sim} b$, 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, $x{:}a \mathrel{\sim} b$ will be abbreviated $a \mathrel{\sim} b$, since x is known to be 0. $\mathrm{Y} \stackrel{\mathrm{def}}{=} \lambda y.(\lambda y.y(x(x)))(\lambda y.y(x(x)))$, and $- \stackrel{\mathrm{def}}{=} (\lambda x.x(x))(\lambda x.x(x))$.

3.4 Computation

Values are terms which can be computed no further; they are the results. This reflects the lazy method of computation: 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 $\lambda x.-$, which inhabits the type $\mathrm{N} \to \overline{\mathrm{N}}$: under a nondeterministic reduction strategy, 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 computations: $a \mathrel{\sim} b$ asserts b is as defined a computation 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.

Abbreviate $(a \mathrel{\sim} b) \times (b \mathrel{\sim} a)$ as $a \sim b$, the equivalence of computations. The inhabiting object justifying the truth of $a \sim b$, $\langle 0, 0 \rangle$, is elided, following the convention for $\mathrel{\sim}$. Abbreviate $a \sim b \to 0 \mathrel{\sim} 1$ as $a \not\sim b$.

(Computation) $\quad ? \vdash t \sim t'$

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

t	t'
$(\lambda x.b)(a)$	$b[a/x]$
$\langle a, b\rangle.1$	a
$\langle a, b\rangle.2$	b
$succ(n)$	n', where n' is one larger than n
$pred(n)$	n', where n' is one smaller than n or 0 if $n = 0$
$if\_zero(0; a; b)$	a.

(Computation) $? \vdash if\_zero(a; b; c) \sim c$
$\vdash a \not\sim 0$

(Contradict) $?, a \sim b \vdash c \in C$

where a and b are different values by inspection.

(Sim refl) $? \vdash a \sim a$

(Sim trans) $?, a \sim b, b \sim c \vdash a \sim c$

(Sim subst) $?, a \sim b \vdash c[a] \sim c[b]$

(Type termination) $? \vdash a\downarrow$
$\vdash a \in A$

where A is by inspection not a bar type $\overline{B}$ or the type E.

(Bottom) $? \vdash - \sim a$

(Termination inherit) $?, a \sim b, a\downarrow \vdash b\downarrow$

The following three rules refine this rule for value constructs.

(Pair inherit) $?, a \sim b, a \sim \langle a.1, a.2\rangle \vdash b \sim \langle b.1, b.2\rangle$

(Func inherit) $?, a \sim b, a \sim \lambda x.a(x) \vdash b \sim \lambda x.b(x)$

where x is not free in a or b.

(Number inherit) $?, a \sim b, a \in \mathrm{N} \vdash a \sim b$

If an expression terminates, necessarily evaluated arguments must terminate. For some presentations it may be desirable to remove these axioms, because they may assume more about the evaluation process than wished.

(Pair left strict) $\quad ?, a.1\downarrow \vdash b \in B$
$\quad\quad a \sim \langle a.1, a.2\rangle \vdash b \in B$

There is a similar rule (Pair right strict).

(Func strict) $\quad ?, a(b)\downarrow \vdash c \in C$
$\quad\quad a \sim \lambda x.a(x) \vdash c \in C$

where x is a new variable not free in a.

(If arg strict) $\quad ?, \mathit{if\_zero}(a; b; c)\downarrow \vdash d \in D$
$\quad\quad a \in \mathrm{N} \vdash d \in D$

(If true strict) $\quad ?, \mathit{if\_zero}(0; b; c)\downarrow \vdash d \in D$
$\quad\quad b\downarrow \vdash d \in D$

(If false strict) $\quad ?, \mathit{if\_zero}(a; b; c)\downarrow \vdash d \in D$
$\quad\quad a \sim 0 \vdash 0 \underset{\sim}{} 1$
$\quad\quad b\downarrow \vdash d \in D$

(Succ strict) $\quad ?, \mathit{succ}(a)\downarrow \vdash b \in B$
$\quad\quad a \in \mathrm{N} \vdash b \in B$

There is a similar rule (Pred strict).

3.5 Membership

The expression $a\ \mathit{in}\ A$ reifies the assertion $a \in A$ as a type; $0 \in a\ \mathit{in}\ A$ just when $a \in A$. The inhabitant 0 is implicit in the rules below.

(Member intro) $\quad ? \vdash a\ \mathit{in}\ A$
$\quad\quad \vdash a \in A$

(Member elim) $\quad ? \vdash a \in A$
$\quad\quad \vdash a\ \mathit{in}\ A$

3.6 Term

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

(E intro) $\quad ? \vdash a \in \mathrm{E}$

3.7 Natural number

N is a type of natural numbers.

(N intro) $\quad ? \vdash a \in \mathrm{N}$

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

(N succ) $\quad ? \vdash succ(a) \in \mathrm{N}$
$\vdash a \in \mathrm{N}$

There is a symmetric rule (N pred).

(N induction) $\quad ?, a \in \mathrm{N} \vdash b(a) \in B(a)$
$a \sim 0 \vdash b(a) \in B(a)$
$a \not\sim 0, x{:}\mathrm{N}, x \not\sim 0, b(pred(x)) \in B(pred(x)) \vdash b(x) \in B(x)$

3.8 Dependent function

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

(Func intro) $\quad ? \vdash \lambda x.b \in x{:}A \to B(x)$
$x{:}A \vdash b \in B(x)$
$\vdash A$ Type

(Func elim) $\quad ? \vdash c \in C$
$b \sim \lambda x.b(x), b(a) \in B(a) \vdash c \in C$
$\vdash a \in A$
$\vdash b \in x{:}A \to B(x)$

where x is not free in b.

3.9 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.

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

The third subgoal assures that $x{:}A \times B(x)$ is a sensible type.

(Prod elim) $\quad ? \vdash c \in C$
$a \sim \langle a.1, a.2\rangle, a.1 \in A, a.2 \in B(a.1) \vdash c \in C$
$\vdash a \in x{:}A \times B(x)$

3.10 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 in A, fixed points of functions $f \in \overline{A} \to \overline{A}$ may be taken using the fixed point typing principle, and inductive properties may be proven via fixed point induction. In section 5.1, an alternate axiomatization for partial types, the theory **Pure Red**, is given.

(Bar intro) $\quad ? \vdash a \in \overline{A}$
$a{\downarrow} \vdash a \in A$
$A{\downarrow} \vdash A\ \mathrm{Type}$

(Bar elim) $\quad ? \vdash a \in A$
$\vdash a{\downarrow}$
$\vdash a \in \overline{A}$

(Fixed point) $\quad ? \vdash \mathrm{Y}(a) \in \overline{A}$
$\vdash a \in \overline{A} \to \overline{A}$
$\vdash A\ \mathrm{Type}_1$

Letting A be $B \to \overline{C}$, arbitrary partial functions may be typed; examples are given in the next section.

A Scott-style fixed point induction principle ([dS69]; see also [Man74] or [Pau87]) 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 6, it will be shown that their soundness proofs both follow from a single general lemma.†

† In fact, the fixed point typing principle can be derived from the induction rule given below (let $P(x)$ be x *in* $\overline{B}$, a be 0); because of their centrality, however, both rules are given.

(Induction)
$$\begin{array}{l} ? \vdash a \in P(\mathrm{Y}(f)) \\ \quad \vdash a \in P(-) \\ \quad x{:}B, y{:}P(x) \vdash a \in P(f(x)) \\ \quad \vdash f \in \overline{B} \to \overline{B} \\ \quad \vdash x{:}B \times P(x)\ \mathrm{Type}_1 \end{array}$$

The validation a is fixed over the induction, reflecting the fact that the validation for all cases must be constant. If the validation were not constant, the base case validation would be infinitely distant when it was combined to give a validation for $P(\mathrm{Y}(f))$, and would thus be inaccessible.

3.11 Type

$A\ \mathrm{Type}_1$ means A is admissible, i.e. the above fixed point principles apply. Informally, A is admissible if for any dependent product subterms $x{:}B \times C$ occurring in A, if there is in turn a dependent function $y{:}D \to E$ inside C, certain restrictions are met by the occurrences of x in D. A counterexample showing some fixed points cannot be typed is found in section 4.3, so the admissible types can never be all types in any rich type theory. What is defined here is just an approximation to all admissible types, an approach also taken in first-order axiomatizations of fixed point induction [Iga72, Pau87]. It is not possible to directly compare admissible collections of formulae because CTT is significantly different from first order logic, but they are roughly equivalent.

Five assertions $\mathrm{Type}_1 \ldots \mathrm{Type}_5$ are used in the definition of the admissible types to implement admissibility; these assertions represent what types are allowed at different locations in A as outlined above, and can be viewed as a finite automaton: progressing further down the syntax tree might cause the state to change, and for atomic types (leaves in the syntax tree), each state has true/false conditions. The type A above must be shown to be a Type_1, C must be shown a Type_2, and since x occurring in D must be restricted, it is necessary to label those x. This is the use of the radioactive term constructor r: x^{r} marks x as radioactive, i.e. inhabiting the left side of a dependent product. D must be a Type_3, with certain restrictions on all radioactive subterms. The collection of admissible types presented here is exactly what can be shown sound in the next section.

In the rules below, i ranges over $0, \ldots, 5$.

($\sim$ form)
$$? \vdash a \sim b\ \mathrm{Type}_i$$

where, for $i = 3, 4$, b contains no radioactive subterms, and for $i = 4, 5$, a contains no radioactive subterms.

($\downarrow$ form) $\quad ? \vdash a\downarrow \text{ Type}_i$

where, for $i = 4$, a contains no radioactive subterms.

(Member form) $\quad ? \vdash (a \ in \ A) \text{ Type}_i$
$\vdash A \text{ Type}_i$

where, for $i = 3, 4$, a contains no radioactive subterms.

(E form) $\quad ? \vdash \text{E Type}_i$

(N form) $\quad ? \vdash \text{N Type}_i$

(Func form) $\quad ? \vdash x{:}A \to B(x) \text{ Type}_i$
$\vdash A \text{ Type}_j$
$x{:}A \vdash B \text{ Type}_k$

where i, j, k are one of $0,0,0; 1,0,1; 2,3,2; 3,5,4; 4,4,4; 5,4,5$.

(Prod form) $\quad ? \vdash x{:}A \times B(x) \text{ Type}_i$
$\vdash A \text{ Type}_j$
$x'{:}A \vdash B(x') \text{ Type}_k$

where i, j, k are one of $0,0,0; 1,1,2; 2,2,2; 3,3,3; 4,4,4; 5,5,5$, and if i is 1 or 2, $x' = x^{\mathrm{r}}$, otherwise $x' = x$.

(Bar form) $\quad ? \vdash \overline{A} \text{ Type}_i$
$A\downarrow \vdash A \text{ Type}_i$

3.12 Miscellaneous

(Cut) $\quad ? \vdash a \in A$
$\vdash b \in B$
$x{:}B \vdash a \in A$

(Cut type) $\quad ? \vdash A \text{ Type}_i$
$\vdash b \in B$
$x{:}B \vdash A \text{ Type}_i$

(Hypothesis) $\quad ? \vdash x \in A$

where $x{:}A$ occurs in $?$.

(Subst) $$\begin{array}{l} ? \vdash c[a] \in C[a] \\ \quad \vdash a \sim b \\ \quad \vdash c[b] \in C[b] \end{array}$$

When performing substitutions in typehood assertions, the radioactive labelings must not be erased.

(Subst type) $$\begin{array}{l} ? \vdash C[a]\,\mathrm{Type}_i \\ \quad \vdash a \sim b \\ \quad \vdash C[b']\,\mathrm{Type}_i \end{array}$$

where for $i = 2, 3, 4, 5$, b' is b if a contains no radioactive subterms, and is b^{r} otherwise.

(Prop member) $?, x{:}A \vdash x \sim 0$

where A is b *in* B, $a \underset{\sim}{} b$, or $a{\downarrow}$.

4 Examples of the theory in use

Here we show how **Red** may be used for reasoning, concentrating on those types and rules not found in other type theories such as partial types and the fixed point principles. We demonstrate two uses for the fixed point typing principle: for typing fixed points, and for proving by extracting fixed point objects.

4.1 Propositions and extraction

Propositions are expressed as types, as described in section 2.3. For matters of readability, we let the traditional logical notation stand for types.

DEFINITION 4.1 *The logical expressions are hereafter defined as follows:*

$$\begin{array}{ll} A \Rightarrow B & \stackrel{\mathrm{def}}{=} A \to B \\ A \,\&\, B & \stackrel{\mathrm{def}}{=} A \times B \\ A \vee B & \stackrel{\mathrm{def}}{=} A + B \ \textit{(defined as } x{:}\mathrm{N} \times \textit{if\_zero}(x; A; B) \textit{ in } \mathbf{Red}\textit{)} \\ \neg A & \stackrel{\mathrm{def}}{=} A \to (0 \underset{\sim}{} 1) \\ \forall x{:}A.\, B & \stackrel{\mathrm{def}}{=} x{:}A \to B \\ \exists x{:}A.\, B & \stackrel{\mathrm{def}}{=} x{:}A \times B \end{array}$$

Red has no extract form rules to automatically synthesize members of types representing propositions, but is possible to implement extraction by unifying the rule and current sequent and having the member partially undefined; the Isabelle generic prover has such an implementation of type theory [PN90]. For purposes of this paper, some rules will be written in extract form, but it will be notational, indicating we expect to incrementally define the object in the type.

DEFINITION 4.2 $? \vdash A \text{ extract } a$ *is notation for* $? \vdash a \in A$.

4.2 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 $\mathrm{N} \to \mathrm{N}$; 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. $\overline{\overline{\mathrm{N}} \to \overline{\mathrm{N}}}$ is the type of partial functions for a lazy functional programming language. In general, the traditional types of programming languages inductively have bars around all constructors, and the traditional types of type theory have no bar constructors.

To avoid writing so many bars, we make the following notational convention for partial functions.

DEFINITION 4.3 $x{:}A \xrightarrow{\mathrm{p}} B \stackrel{\mathrm{def}}{=} \overline{x{:}A \to \overline{B(x)}}$

To express recursive functions, fixed points $\mathrm{Y}(f) \in A \xrightarrow{\mathrm{p}} B$ of functionals $f \in (A \xrightarrow{\mathrm{p}} B) \to (A \xrightarrow{\mathrm{p}} B)$ are typed using the fixed point typing principle, provided the type is admissible. 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 4.4 $\vdash$ *plus* $\in \mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}$, *where*

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

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

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

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

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

which follows directly by (Bar intro) and (Hypothesis). Consider next the case $y \not\sim 0$; computing gives

$$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N}, y \not\sim 0 \vdash \mathit{succ}(x(\mathit{pred}(y))) \in \overline{\mathrm{N}},$$

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

$$x{:}\mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}, y{:}\mathrm{N}, z{:}\mathrm{N}, y \not\sim 0, x(\mathit{pred}(y))(z)\!\downarrow \vdash x(\mathit{pred}(y))(z) \in \mathrm{N}.$$

At this point, since $\mathit{pred}(y), z \in \mathrm{N}$ (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 $x(\mathit{pred}(y))(z)\!\downarrow$, (Func strict) gives $x\!\downarrow$. Then, by (Bar elim), $x \in \mathrm{N} \to \mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}$, so $x(\mathit{pred}(y)) \in \mathrm{N} \xrightarrow{\mathrm{p}} \mathrm{N}$, and one more iteration of this process results in the conclusion above.
QED.

4.3 Not all fixed points are typable

To illustrate the limits of the fixed point typing principles, we show a type which cannot be admissible, taking as given the consistency of the theory, proved in section 6.

THEOREM 4.5 *Some* **Red** *types can not be admissible, i.e. there is a type D and term d such that $\vdash d \in \overline{D} \to \overline{D}$, but $\vdash \mathrm{Y}(d) \in \overline{D}$ would lead to a contradiction.*

PROOF. Define

$$D \stackrel{\mathrm{def}}{=} x{:}(\mathrm{N} \to \overline{\mathrm{N}}) \times \neg(\forall y{:}\mathrm{N}.\, x(y)\!\downarrow) \text{ and}$$
$$d \stackrel{\mathrm{def}}{=} \lambda z.\langle \lambda y.\mathit{if\_zero}(y; 0; (z.1)(\mathit{pred}(y))), \lambda w.0\rangle.$$

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

First, $\vdash d \in \overline{D} \to \overline{D}$ is shown. This proof will be at a high level and not refer to individual rules. Assume $z \in \overline{D}$, and show the pair to be in

D. For the left half of the pair, $f \stackrel{\text{def}}{=} \lambda y.\mathit{if\_zero}(y; 0; (z.1)(\mathit{pred}(y)))$ must be shown to be in $\mathrm{N} \to \overline{\mathrm{N}}$, which is straightforward, similar to Lemma 4.4 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 $z.1\downarrow$ by strictness. Thus, $z \in D$, so $z.2 \in \neg(\forall y{:}\mathrm{N}.\ z.1(y)\downarrow)$. But, if $z.1$ is not total, f will also not be total, contradicting our assumption. Thus, $d \in \overline{D} \to \overline{D}$.

If it were true that $\mathrm{Y}(d) \in \overline{D}$, $\mathrm{Y}(d) \in D$ would also follow because $\mathrm{Y}(d)\downarrow$ by directly computing. This would mean the function $\mathrm{Y}(d).1 \in \mathrm{N} \to \overline{\mathrm{N}}$ is not total, but this is a contradiction because it can be proved total by induction on natural numbers. Therefore, $\mathrm{Y}(d) \notin \overline{D}$, so since **Red** is proved consistent by Theorem 6.33, D is not admissible.
QED.

An informal sketch of why D admissibility fails is as follows. Recalling our admissibility rules,

$\vdash D\ \mathrm{Type}_1$ if
$x^{\mathrm{r}}{:}(\mathrm{N} \to \overline{\mathrm{N}}) \vdash \neg(\forall y{:}\mathrm{N}.\ x^{\mathrm{r}}(y)\downarrow)\ \mathrm{Type}_2$ (note radioactive labeling of x) if
$x^{\mathrm{r}}{:}(\mathrm{N} \to \overline{\mathrm{N}}) \vdash \forall y{:}\mathrm{N}.\ x^{\mathrm{r}}(y)\downarrow\ \mathrm{Type}_3$ if
$x^{\mathrm{r}}{:}(\mathrm{N} \to \overline{\mathrm{N}}) \vdash x^{\mathrm{r}}(y)\downarrow\ \mathrm{Type}_4$,

which cannot be provable because $\vdash a\downarrow \mathrm{Type}_4$ is not provable when a contains radioactive subterms.

4.4 Partial propositions and partial proofs

The bar operator may also be applied to types which represent propositions, giving types such as $\forall n{:}\mathrm{N}.\ \overline{\exists m{:}\mathrm{N}.\ P(m,n)}$, which are called *partial propositions*. $\overline{A}$ for any type A is trivially true under the propositions-as-types interpretation, because $- \in \overline{A}$. 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 a, proving a partial proposition is useful.

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 4.6 $\overline{\forall}x{:}A.\ B(x) \stackrel{\text{def}}{=} x{:}A \stackrel{\mathrm{p}}{\to} B(x)$

Consider for instance the type

$P \stackrel{\text{def}}{=} \overline{\forall}x{:}\mathrm{N}.\ \exists y{:}\mathrm{N}.\ y \sim \mathit{mult}(x)(x)$, where
$\mathit{mult} \stackrel{\text{def}}{=} \mathrm{Y}(\lambda z.\lambda x.\lambda y.\mathit{if\_zero}(x; y; \mathit{plus}(y, z(\mathit{pred}(y)))))$.

If $p \in P$ and for some particular $n \in \mathrm{N}$ $p(n)\downarrow$, then $p(n)$ validates $\exists y{:}\mathrm{N}.\, y \sim mult(n)(n)$. Validations for propositions are always total, whereas validations for partial propositions are partial. Propositions are thus to total correctness as partial propositions are to partial correctness.

4.4.1 Extracting recursive programs

One important extension to the extraction paradigm is the extraction of arbitrary recursive computations. This is significant, because a limitation of the extraction paradigm had been the inability to automatically extract recursive programs from constructive proofs. For instance, take the proposition

$$\forall l{:}List.\, \exists l'{:}List.\, sorted(l') \;\&\; permutation(l, l').$$

It is not possible to extract an order $n \cdot \log n$ sorting routine, because the only way to prove this theorem is by double induction on natural numbers, which always produces an order n^2 algorithm. With partial propositions, the order $n \cdot \log n$ program can be directly extracted. The following derived rule is one principle useful to this end.

LEMMA 4.7 *The following non-well-founded induction principle is a derived rule for* **Red**.

$$\begin{array}{l} \vdash \overline{\forall} x{:}A.\, B(x) \text{ extract } \mathrm{Y}(\lambda h.\lambda x.b) \\ \quad h{:}(\overline{\forall} x{:}A.\, B(x)), x{:}A \vdash \overline{B} \text{ extract } b \\ \quad \vdash \overline{\forall} x{:}A.\, B(x)\ \mathrm{Type}_1 \end{array}$$

PROOF. Direct from the fixed point typing principle.
QED.

We may use the fact $\overline{\forall} x{:}A.\, B(x)$ in the proof of itself, resulting in an extract object which is a recursive computation. 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.

4.4.2 An example of recursive function extraction

A primality tester is extracted from a partial proposition.

DEFINITION 4.8 *Define four predicates,*

$Leq(x,y) \stackrel{\text{def}}{=} \exists z{:}\mathrm{N}.\ plus(x,z) \sim y,$
$Divides(x,y) \stackrel{\text{def}}{=} \exists z{:}\mathrm{N}.\ Leq(2,z)\ \&\ mult(y)(z) \sim x,$
$Primeto(x,y) \stackrel{\text{def}}{=} \forall z{:}\mathrm{N}.\ Leq(2,z) \Rightarrow Leq(z,y) \Rightarrow \neg Divides(x,z),$ *and*
$Compositeto(x,y) \stackrel{\text{def}}{=} \exists z{:}\mathrm{N}.\ Leq(2,z)\ \&\ Leq(z,y)\ \&\ Divides(x,z).$

LEMMA 4.9 *Divides is decidable, i.e. there is a term div such that*

$$\vdash \forall x,y{:}\mathrm{N}.\ Divides(x,y) \vee \neg Divides(x,y)\ \text{extract}\ div.$$

PROOF. Exercise.

LEMMA 4.10 *We may show*

$$\vdash \overline{\forall} x{:}\mathrm{N}.\ \overline{\forall} y{:}\mathrm{N}.\ Primeto(x,y) \vee Compositeto(x,y)\ \text{extract}\ e,$$

where extract object e is

$$\mathrm{Y}(\lambda z,x,y.if\_zero(y;\langle 0,e_1\rangle;if\_zero((div(x)(y)).1;\langle 1,\langle y,e_2\rangle\rangle;z(x)(pred(y))))).$$

PROOF. Using the fixed point typing principle and (Bar intro), (Func intro), we obtain

$$z{:}(\overline{\forall} x{:}\mathrm{N}.\ \overline{\forall} y{:}\mathrm{N}.\ Primeto(x,y) \vee Compositeto(x,y)), x{:}\mathrm{N}, y{:}\mathrm{N} \vdash \overline{Primeto(x,y) \vee Compositeto(x,y)}\ \text{extract}\ e(z)(x)(y).$$

First, proceed by cases on whether $y \sim 0$ or not: If $y \sim 0$, $Primeto(x,y)$ follows; suppose then $y \not\sim 0$. Proceed by cases on $Divides(x)(y)$ (Lemma 4.9 may be (Cut) into the hypothesis list).
CASE $Divides(x,y)$: By computing and (Bar intro) we arrive at

$$\ldots, Divides(x,y) \vdash Primeto(x,y) \vee Compositeto(x,y)\ \text{extract}\ \langle 1,\langle y,e_2\rangle\rangle,$$

which is direct from the definition of *Divides* and *Compositeto*.
CASE $\neg Divides(x,y)$: Recursively use the hypothesis z for y one smaller, i.e. apply $z(x)(pred(y))$, so show

$$\ldots, \neg Divides(x,y) \vdash \overline{Primeto(x,y) \vee Compositeto(x,y)}\ \text{extract}\ z(x)(pred(y)).$$

By (Bar intro), assume $z(x)(pred(y))\downarrow$, prove the above unbarred type. With this assumption, by (Func strict) and (Func elim) we obtain $z(x)(pred(y)) \in Primeto(x,pred(y)) \vee Compositeto(x,pred(y))$ as a hypothesis, which from assumption $\neg Divides(x,y)$ implies the conclusion.
QED.

Letting y be $pred(x)$, a primality tester is obtained. This tester also has the feature that for composite numbers, one of the factors is returned by e.

LEMMA 4.11 *In* **Red** *we may prove*

$$\vdash \overline{\forall} x{:}\mathrm{N}.\ \mathit{Primeto}(x, \mathit{pred}(x)) \vee \mathit{Compositeto}(x, \mathit{pred}(x)) \text{ extract } \lambda x.e(x)(\mathit{pred}(x)).$$

PROOF. By direct instantiation of the above lemma.
QED.

Partial propositions extend the collection of statements that can be phrased 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 purely classical theory, and provides a practical reason for working at least to some degree within a constructive framework.

The fixed point induction principle is well-known, so no examples will be given here; see for instance [Man74].

5 A family of theories

A family of theories is constructed upon **Red**. Directed set principles **Pure Red** which obviate the need for the fixed point principles, a more complete type theory **Full Red**, and a classical extension **Classical Red** are the extensions discussed.

5.1 Pure Red

There is an alternative approach for typing and inductively reasoning about partial functions which does not need a collection of admissible types, and is thus less complicated. By adding more general computational principles and removing the fixed point typing principles, we obtain a theory, **Pure Red**, in which fixed points are typed from more basic principles. The principles used here are part of a different approach to programming language semantics, presented in [Smi91]. The key insight to this approach is that $\sim$-directed collections of computations can be viewed as a single computation, ignoring less defined elements. Basic results of this approach are presented in section 6.2, and **Pure Red** can be viewed as an axiomatization of that development.

5.1.1 Pure Red rules

Pure Red is defined as a variation on **Red**. The (Fixed point) and (Induction) rules are removed, and the radioactive labelings and assertions

A $\text{Type}_{1/2/3/4/5}$ and associated rules are removed; only A Type typehood assertions are needed.

Rules are added for reasoning about $\sim$-directed collections of computations. A new atomic assertion, $A \sim_M B$, means directed collection of computations A is at least as defined as collection B.

A general notion of collection of computations is needed, and for this subtypes will be used.

(Subtype form)
$$\begin{array}{l} ? \vdash \{x{:}A \mid B(x)\}\ \text{Type} \\ \quad \vdash A\ \text{Type} \\ \quad x{:}A \vdash B(x)\ \text{Type} \end{array}$$

(Subtype intro)
$$\begin{array}{l} ? \vdash a \in \{x{:}A \mid B(x)\} \\ \quad \vdash a \in A \\ \quad \vdash b \in B(a) \\ \quad x{:}A \vdash B(x)\ \text{Type} \end{array}$$

(Subtype elim)
$$\begin{array}{l} ?, x{:}\{x{:}A \mid B(x)\} \vdash c \in C \\ \quad x{:}A, z{:}B(x) \vdash c \in C \end{array}$$

where z is a new variable.

DEFINITION 5.1 *Define*

$$f@k \stackrel{\text{def}}{=} \text{Y}(\lambda y.\lambda x.\mathit{if\_zero}(x; -; f(y(\mathit{pred}(x)))))(k),$$
$$\{f@\} \stackrel{\text{def}}{=} \{x{:}\text{E} \mid \exists y{:}\text{N}.\ x \sim f@y\},$$
$$\{\text{Y}(f)\} \stackrel{\text{def}}{=} \{x{:}\text{E} \mid x \sim \text{Y}(f)\},\ \textit{and}$$
$$A \sim_M B \stackrel{\text{def}}{=} A \sim_M B\ \textit{and}\ B \sim_M A.$$

(Dirset form)
$$\begin{array}{l} ? \vdash A \sim_M B\ \text{Type} \\ \quad \vdash A\ \text{Type} \\ \quad \vdash B\ \text{Type} \\ \quad x{:}A, y{:}A \vdash \exists z{:}A.\ x \sim z\ \&\ y \sim z \\ \quad x{:}B, y{:}B \vdash \exists z{:}B.\ x \sim z\ \&\ y \sim z \end{array}$$

(Dirset intro)
$$\begin{array}{l} ? \vdash A \sim_M B \\ \quad x{:}A \vdash \exists y{:}B.\ x \sim y \\ \quad \vdash A \sim_M B\ \text{Type} \end{array}$$

where x is a new variable.

(Dirset refl) $\quad ? \vdash A \sim_M A$
$\quad\quad\quad\quad\quad\quad \vdash A \sim_M A\ \mathrm{Type}$

(Dirset trans) $\quad ?, A \sim_M B, B \sim_M C \vdash A \sim_M C$

(Dirset subst) $\quad ?, A \sim_M B \vdash c[A] \sim_M c[B]$

where $c[C] \stackrel{\mathrm{def}}{=} \{x{:}\mathrm{E} \mid \exists y{:}C.\ x \sim c[y]\}$.

(Dirset approx) $\quad ? \vdash \{\mathrm{Y}(f)\} \sim_M \{f@\}$

(Dirset inherit) $\quad ?, A \sim_M B, a \in A, a\downarrow \vdash c \in C$
$\quad\quad\quad\quad\quad\quad y{:}B, y\downarrow \vdash c \in C$

(Dirset pair inherit) $\quad ?, A \sim_M B, a \in A, a \sim \langle a.1, a.2\rangle \vdash c \in C$
$\quad\quad\quad\quad\quad\quad y{:}B, y \sim \langle y.1, y.2\rangle \vdash c \in C$

(Dirset func inherit) $\quad ?, A \sim_M B, a \in A, a \sim \lambda x.a(x) \vdash c \in C$
$\quad\quad\quad\quad\quad\quad y{:}B, y \sim \lambda x.a(x) \vdash c \in C$

where x is not free in a and $x \neq y$.

5.1.2 An Example

Since it is not immediately apparent how fixed points may be typed using the above rules, a small example is given. From the proof it can be seen how a wide range of fixed points can be typed.

LEMMA 5.2 *In* **Pure Red**, *if* $\vdash f \in (\mathrm{N} \stackrel{\mathrm{p}}{\to} \mathrm{N}) \to (\mathrm{N} \stackrel{\mathrm{p}}{\to} \mathrm{N})$, *then* $\vdash \mathrm{Y}(f) \in \mathrm{N} \stackrel{\mathrm{p}}{\to} \mathrm{N}$.

PROOF. First, $\vdash \forall x{:}\mathrm{N}.\ (f@k)(x) \in \mathrm{N} \stackrel{\mathrm{p}}{\to} \mathrm{N}$ can be proved by N-induction on x. Suppose $\mathrm{Y}(f)\downarrow$, and show $\mathrm{Y}(f) \in \mathrm{N} \to \overline{\mathrm{N}}$. By (Dirset approx), $\{\mathrm{Y}(f)\} \sim_M \{f@\}$, so by (Dirset inherit), $f@k\downarrow$ for some k; thus, $f@k \in \mathrm{N} \to \overline{\mathrm{N}}$, so $f@k \sim \lambda x.(f@k)(x)$ by (Func elim), and then $\mathrm{Y}(f) \sim \lambda x.\mathrm{Y}(f)(x)$ by (Func inherit). Applying (Func intro) and (Bar intro), we must show $\mathrm{Y}(f)(x) \in \mathrm{N}$ given hypotheses $x{:}\mathrm{N}$ and $\mathrm{Y}(f)(x)\downarrow$. Using (Dirset subst), we have $((-)(x))[\{f@\}] \sim_M ((-)(x))[\{\mathrm{Y}(f)\}]$; thus, $(f@k)(x)\downarrow$ for some k, so $(f@k)(x) \in \mathrm{N}$. $(f@k)(x) \sim \mathrm{Y}(f)(x)$, so $\mathrm{Y}(f)(x) \in \mathrm{N}$ by (N inherit).
QED.

From the proof the drawback of **Pure Red** is most evident: simple typings have long proofs. In any implementation of this theory, a tactic would automatically complete these proofs, or even better tactics could proven correct in a formal metatheory and proof construction avoided [ACHA90]. Instances of the fixed point induction principle can also be internally derived.

LEMMA 5.3 $\vdash \lambda k.\lambda x.0 \in \forall k{:}\mathrm{N}.\ a[\{f@k\}] \sim b[\{f@k\}] \Rightarrow a[\{\mathrm{Y}(f)\}] \sim b[\{\mathrm{Y}(f)\}]$

PROOF. Theorem 6.16 can be directly reflected into the logic, using (Direct approx), (Dirset subst), and (Dirset intro).
QED.

5.1.3 Type-theoretic semantics

Martin-Löf gives type-theoretic semantics (called "direct semantics" in [NPS90]) of CTT in [Mar82]. Since Martin-Löf intends CTT to be a foundation for constructive mathematical practice, a semantics in terms of some other theory (like what is to be presented in section 6) is of little use. In type-theoretic semantics, types are defined by giving what values are members, and when two values are equal. The collection of types is not fixed, for the theory is open-ended and types may be added at some future date. Partial types in **Red** do not fit well into this framework because the fixed point typing principle goes beyond what it means to inhabit a partial type. **Pure Red** has no such objectionable behavior, because meaning of $\overline{A}$ may be directly defined in terms of the meaning of A: $a \in \overline{A}$ if, provided a terminates, $a \in A$. This exactly specifies the meaning of the type. Practically, this means **Red** is less open to extension because the collection of admissible types needs to be altered, and **Pure Red** is a more flexible theory.

5.2 Full Red

Numerous other type and computation constructors should be part of a full implementation of **Red**. We sketch such a theory, **Full Red**, and refer the reader to [CAB+86, Smi88] for rules and semantics for these types and terms.

DEFINITION 5.4 **Full Red** *is the type theory* **Red** *extended with rules for union types* $A+B$, *type universes* U_i, *recursive and inductive types* $rec(T.B(T))$, $ind(T.B(T))$, *subtypes* $\{x{:}A \mid B(x)\}$, *and sequencing of computations* $let x = a\ in\ b$.

These types are described briefly in section 2. The sequencing constructor $let\ x = a\ in\ b$ is computed by first evaluating a, and then evaluating $b[a/x]$. This constructor allows call-by-value function calls $\lambda_v x.b$ to be expressed by $(\lambda y.\ let\ x = y\ in\ b)$, and is also recursion-theoretically [CS88] and category-theoretically [Mog89] important for full expressiveness.

5.3 Classical Red

The standard classical interpretation of constructive type theory is to interpret $A \to B$ as the space of all set-theoretic functions from A to B. However, in presence of partial types and the fixed point typing principle, this interpretation is unsound.

LEMMA 5.5 *Function spaces* $\overline{\mathrm{N}} \to \mathrm{N}$ *in* **Red** *cannot be interpreted as all set-theoretic functions from the set* $\mathrm{N} \cup \{-\}$ *to the set* N.

PROOF. Supposing this were so; then, a function $h \in \overline{\mathrm{N}} \to \mathrm{N}$ solves the halting problem for the programming language of **Red**.

$$h(n) \stackrel{\mathrm{def}}{=} \begin{cases} 0 & \text{if } \neg n\downarrow \\ 1 & \text{if } n\downarrow \end{cases}$$

However, consider $d \stackrel{\mathrm{def}}{=} \mathrm{Y}(\lambda x.\mathit{if\_zero}(h(x); 1; -))$. $\vdash d \in \overline{\mathrm{N}}$ is easily proved using the fixed point typing principle. If $d\downarrow$, then $h(d) \sim 1$, but this means $\neg d\downarrow$; a similar contradiction results if $\neg d\downarrow$ is hypothesized. Since **Red** is consistent (Theorem 6.33), this cannot be possible.
QED.

It is in fact possible to abstractly develop much of basic recursive function theory in **Red**; see [CS88]. This also rules out an extension to include excluded middle $A \vee \neg A$.

LEMMA 5.6 *Adding the rule* $? \vdash A \vee \neg A$ *to* **Red** *results in an inconsistent theory.*

PROOF. With such a rule, $\forall x{:}\overline{\mathrm{N}}.\ x\downarrow \vee \neg x\downarrow$ could be proven, but then the extract object solves the halting problem defined above.
QED.

The classical extension proposed here is a more modest sort, and one that makes **Classical Red** a limited theory of *recursive mathematics*: all propositions must be justified by constructions, but it is possible to use nonconstructive means to prove properties of the constructions. Recursive mathematics subsumes Markov's principle, which states non-diverging computations terminate.

Two reasoning principles are added: a case analysis on whether a given computation terminates, and a case analysis on whether $a \sim b$.

Classical Red is **Red** extended with the following axioms:

(Termination case) $? \vdash a \in A$
$b{\downarrow} \vdash a \in A$
$b \sim - \vdash a \in A$

(Equivalence case) $? \vdash a \in A$
$b \sim c \vdash a \in A$
$b \not\sim c \vdash a \in A$

6 Semantics

Here a semantics for **Red** is given, which shows it to be a soundly constructed type theory: some types are uninhabited, meaning some propositions are false.

The semantic method used here is the non-type-theoretic semantics of Allen [All87b, All87a]. This method can be viewed as an elaboration of the notion of computable term found in Tait's strong normalization proof of the typed λ-calculus ([Tai67]; see also [Ste72, Bar84]), and also can be viewed as a formal version of the so-called type-theoretic semantics given to ETT by Martin-Löf in [Mar82].

Types are properties of type-free computations in **Red** so untyped computations have independent meaning, with untyped equivalence $\sim$, and types are then defined as collections of untyped computations. In this paper, untyped computations are given an operational interpretation, but a denotational interpretation could also be given.

Once untyped computations and equivalences thereupon are interpreted, the types and their inhabitants are inductively defined. Justification of the individual rules is then straightforward except for the fixed-point typing principles, which require a long proof. The full proof is large enough that a simplified version will be given, together with the key ideas that allow the general proof to be constructed. All mathematics in this section is constructive, with the exception of the justification of **Classical Red** at the end of the section.

6.1 Interpretation of computations

Since an operational meaning is given to terms, terms are interpreted in the semantics as themselves. We will thus use the same notation for a term and its denotation. To get the appropriate equality, types need to be destructable computationally, since they are destructable by the introduction

and elimination rules of **Red**. Otherwise, all types with the same outer constructor would be equivalent. For this purpose, a term constructor is added to the terms defined in section 3, *destr_type*$(A; b)$, which will access parts of the types. The atomic assertions $a \sim b$ and $a\!\downarrow$ are interpreted by relations $\sim$ and $\downarrow$ defined below; the same names will again be used by abuse of notation.

First, an operational interpreter for untyped computations is defined. Some notation: $a = b$ means a and b are identical modulo α-conversion. We present a rewriting interpreter like the $\xrightarrow{v}$ relation of [Plo75], using the more convenient notion of a reduction context from [FFK87].

DEFINITION 6.1 Computations *are terms that are outermost a computation constructor, i.e.* $pred(a)$, $succ(a)$, $if\_zero(a; b; c)$, $a.1$, $a.2$, $a(b)$, *or* $destr\_type(A; b)$. *Radioactively labeled computation constructors are also computation constructors.*

Computations are terms that may be further evaluated; closed terms are either computations or values. A lazy evaluation strategy is deterministic, so at most one reduction applies. We use reduction contexts to isolate the next redex to be reduced in a term.

DEFINITION 6.2 *A reduction context* $r[\![=]\!]$ *is inductively of the form*

$=$ *or* $s[\![=]\!](a)$, *or* $if\_zero(s[\![=]\!]; a; b)$, *or* $pred(s[\![=]\!])$, *or* $succ(s[\![=]\!])$, *or* $s[\![=]\!].1$, *or* $s[\![=]\!].2$, *or* $destr\_type([\![=]\!]; a)$,

where $s[\![=]\!]$ *is a reduction context, and* a *and* b *are closed terms.*

"=" is used to represent a hole in a reduction context and not "$-$", because it will be necessary below to use both kinds of holes at once.

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 6.3 $\mapsto_1$, *single-step computation, is the least relation such that*

$r[\![succ(a)]\!] \mapsto_1 r[\![a']\!]$, *where* $a + 1 = a'$
$r[\![pred(a)]\!] \mapsto_1 r[\![a']\!]$, *where* $a \dot{-} 1 = a'$
$r[\![if\_zero(0; b; c)]\!] \mapsto_1 r[\![b]\!]$
$r[\![if\_zero(a; b; c)]\!] \mapsto_1 r[\![c]\!]$, *where* $a \in \{1, 2, \ldots\}$
$r[\![(\lambda x.b)(a)]\!] \mapsto_1 r[\![b[a/x]]\!]$
$r[\![\langle a, b\rangle.1]\!] \mapsto_1 r[\![a]\!]$
$r[\![\langle a, b\rangle.2]\!] \mapsto_1 r[\![b]\!]$
$r[\![destr\_type(T; b)]\!] \mapsto_1 r[\![b(T_1)(T_2)]\!]$,

where T is a type, and $(x{:}A\ op\ B)_1 = A$, $(x{:}A\ op\ B)_2 = \lambda x.B$, $(\overline{A})_1 = A$, $(\overline{A})_2 = 0$, $(a{\downarrow})_1 = a$, $(a{\downarrow})_2 = 0$, $(a\ in\ A)_1 = a$, $(a\ in\ A)_2 = A$, $(a \sim b)_1 = a$, $(a \sim b)_2 = b$.

For each line $r[\![a]\!] \mapsto_1 r[\![a']\!]$ in the above definition, a is a *redex* and a' its *contractum*. It is necessary to show this relation defines at most one redex to reduce at once.

LEMMA 6.4 *All closed computations a can be expressed uniquely as $a = r[\![b]\!]$ for some reduction context $r[\![=]\!]$ and redex b.*

PROOF. By induction on the term structure. If a is not a computation, the result is trivial; suppose inductively that terms smaller than a can be factored, and proceed by cases on the outermost constructor of a. If $a = c(d)$, then there are two subcases: if c is a value, let r be $(=)$, and b be a; if c is a computation, inductively $c = r'[\![f]\!]$ uniquely, so let r be $(r'[\![=]\!])(d)$, and b be f. The other possibilities for a proceed similarly.
QED.

DEFINITION 6.5 *Over closed terms, $\mapsto^*$ is the reflexive, transitive closure of $\mapsto_1$; $a \mapsto b$ iff $a \mapsto^* b$, where b is a value.*

Computation is defined for closed terms only; open terms are interpreted as an infinite family of closed terms. Terminating computations are those computations with a value: $a{\downarrow}$ iff $a \mapsto b$ for some b.

6.2 Equivalence and ordering of computations

Red has an untyped equality $\sim$, more generally, an ordering on terms $\underset{\sim}{}$. These relations may be modeled either denotationally or operationally. Denotationally, for lazy computation a model where $\lambda x.- \not\sim -$ is necessary (i.e. we need non-sensible models[‡] [Bar84]), but most denotational models for the λ-calculus do not meet this criterion. Plotkin [Plo76] gives a non-sensible model for LCF by explicitly adding a $-$ element and *lifting* the rest, but LCF is a typed language; Ong [Ong88] shows this solution also works in the pure untyped lazy λ-calculus.

The solution taken is to give a purely operational interpretation of these relations [Smi91]. This has the advantage of simpler (even constructive) mathematics, and also leads to improved axiomatizations for computations, the **Pure Red** theory of section 5.1. Also, in [Smi91], it is proven that this

[‡]One person's nonsense is anothers sense.

operational model may be directly extended to a fully abstract cpo model, meaning nothing is lost by taking this operational approach. Here a terse exposition of the results relevant to **Red** soundness is given; for a more detailed exposition, consult [Smi91].

There are two general classes of operational equivalence that are commonly used: *observational congruence*, $a \sim_{\text{obs}} b$, and reduction observation equivalence, $a \sim_{\text{robs}} b$. $a \sim_{\text{obs}} b$ means a and b behave identically when placed in any program context $c[-]$, and $a \sim_{\text{robs}} b$ means a and b behave identically when placed in any reduction context $r[\![=]\!]$. The two orderings are in fact provably the same for many language, but $\sim_{\text{robs}}$ is more amenable to mathematical analysis, and is thus the ordering used herein, abbreviated $\sim$.

The relations on terms are initially defined for closed terms only, and then extended to arbitrary terms. Notation is then needed for the closing of a term. Define *closing substitutions* σ to range over substitutions $[a_1/x_1, a_2/x_2, \ldots]$, where all a_i are closed. $\sigma(a)$ denotes $a[a_1/x_1, a_2/x_2, \ldots]$, and is by definition closed.

DEFINITION 6.6 *(i) For closed a and b, $a \lesssim_0 b$ iff for all reduction contexts $r[\![=]\!]$, if $r[\![a]\!]\downarrow$, then $r[\![b]\!]\downarrow$.*

(ii) For arbitrary a and b, $a \lesssim b$ iff for all closing σ, $\sigma(a) \lesssim_0 \sigma(b)$.

Also very useful for **Red** is the equivalence $\sim$ induced by this ordering, for all $\sim$-equal terms are always treated identically, and may thus be freely substituted for one another.

DEFINITION 6.7 *$a \sim b$ iff $a \lesssim b$ and $b \lesssim a$.*

LEMMA 6.8 *(i) $\lesssim$ is transitive and reflexive;*

(ii) $- \lesssim a$;

(iii) $a \sim a'$ where a is a redex and a' its contractum;

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

Up to this point a partial ordering on computations has been defined which respects computation. Four important results are next proven about this ordering: the substitutivity of $\lesssim$, two properties which are the operational analogues the least fixed-point and continuity properties of domains, and a fixed point induction principle.

To prove the congruence of $\sim$, $\underset{\sim}{}$ is proved substitutive, which follows directly from the substitutivity of $\sim_0$. The proof here is adapted from [MT91].

LEMMA 6.9 *$\sim_0$ is substitutive, i.e. if $a \sim_0 b$ then $c[a] \sim_0 c[b]$.*

PROOF.

To show $c[a] \sim_0 c[b]$, which by definition means $r[\![c[a]]\!]\downarrow$ implies $r[\![c[b]]\!]\downarrow$, we generalize this statement to allow a/b to occur elsewhere in r; this will be notated by $r[-][\![=]\!]$, using hole $=$ for the reduction point, and holes $-$ for other occurrences of a/b in r. The proof then proceeds by induction on the length of the computation of terminating $r[a][\![c[a]]\!]$ for arbitrary c and r. For the base case $r[a][\![c[a]]\!]$ is a value, so the conclusion is direct. Assume there is a proof for shorter computations, and consider what happens in one step of computation

$$r[a][\![c[a]]\!] \mapsto_1 r[a][\![d]\!].$$

We can assume without loss of generality that $r[\![=]\!]$ is a true reduction context, i.e. $c[a]$ is a redex: if $c[a]$ is not a redex, the redex must be a subterm of $c[a]$, so $r[a][\![c[a]]\!]$ can be written $r'[a][\![c'[a]]\!]$, shifting part of c out of the reduction context. We need not worry about the reduction context being inside a, because then $r'[\![a]\!]\downarrow$, so $r'[\![b]\!]\downarrow$ by assumption.

Observe that a occurring outside of the reduction context are untouched when one step of computation is performed; this may be validated by inspection of the reduction rules. Now, consider $c[a]$ in a reduction context: if c is not $(-)$, inspection of the reduction rules shows a inside c is also just carried over to the right side of the rule, except in a few special cases. Putting off the special cases, assume a is not touched; we may write the single step as

$$r[a][\![c[a]]\!] \mapsto_1 r[a][\![c'[a]]\!]$$

for some c'. The right side of the reduction is one step shorter, so applying the induction hypothesis gives $r[b][\![c'[b]]\!]\downarrow$, and since $r[b][\![c[b]]\!] \mapsto_1 r[b][\![c'[b]]\!]$, $r[b][\![c[b]]\!]\downarrow$. Returning to the special cases, if $c[-] = (-)(d[-])$,

$$r[a][\![a(d[a])]\!] \mapsto_1 r[a][\![a'[d[a]/x]]\!], \text{ where } a = \lambda x.a'.$$

By induction hypothesis, $r[b][\![a'[d[b]/x]]\!]\downarrow$; thus, $r[b][\![a(d[b])]\!]\downarrow$ by reversing the computation step. This term may also be written $r'[b][\![a]\!]$ for $r' = r[-][\![(=)(d[b])]\!]$; observe r' is a reduction context. Then, from assumption $a \sim_0 b$, $r'[b][\![b]\!]\downarrow$, so reverting to previous notation, $r[b][\![b(d[b])]\!]\downarrow$, proving

this case. The other special cases, $c[-] = (-).1/2, pred(-), succ(-)$, are similar.

The only case remaining is when $c = (-)$,

$$r[a][\![a]\!] \mapsto_1 r[a][\![a']\!].$$

Applying the induction hypothesis to the right side treating a' as fixed, $r[b][\![a']\!]\downarrow$, so $r[b][\![a]\!]\downarrow$ as well. So, since $a \sim_0 b$, $r[b][\![b]\!]\downarrow$ directly.
QED.

THEOREM 6.10 *$\sim$ is substitutive, i.e. if $a \sim b$ then $c[a] \sim c[b]$.*

PROOF. Given $a[\sigma] \sim_0 b[\sigma]$, show $c[a][\sigma] \sim_0 c[b][\sigma]$, which may equivalently be written $c[a[\sigma]][\sigma] \sim_0 c[b[\sigma]][\sigma]$; this is a direct consequence of Lemma 6.9.
QED.

The congruence of $\sim$ is an immediate corollary. This allows justification of the substitution of equal terms in any context in type theory, the (Subst) rule.

COROLLARY 6.11 *$\sim$ is a congruence, i.e. if $a \sim b$ then $c[a] \sim c[b]$.*

PROOF. Direct from Theorem 6.10.

Next an ordering $\sim_M$ on $\sim$-directed sets of terms is defined which takes the place of least upper bounds and continuity principles of denotational semantics. $\sim_M$ is a critical part of the proof of soundness of the fixed point typing principle and fixed point induction for **Red**, and is directly axiomatized in **Pure Red** to allow fixed points to be typed without an explicit fixed point principle.

DEFINITION 6.12 *a set[§] of terms C is* directed *iff for every $a, b \in C$, $a \sim c$ and $b \sim c$ for some $c \in C$.*

It is possible to consider directed sets as a collective representation of a single computation, because the only incompatibility between terms in a directed set is that one may halt while another diverges. In such a case we ignore the diverging computations, taking the most defined elements. An ordering like $\sim$, but between directed sets of terms, $A \sim_M B$, may then be defined. This ordering is a direct generalization of $\sim$, and is useful for characterizing the relation between sets of terms.

DEFINITION 6.13 *(i) $A \sim_{M,0} B$ iff A, B closed and directed and for all reduction contexts $r[\![=]\!]$ and $a \in A$, if $r[\![a]\!]\downarrow$, then $r[\![b]\!]\downarrow$ for some $b \in B$.*

[§]Sets here are simple collections. $a \in A$ means a is a member of set A.

(ii) $A \sim_M B$ *iff* $\sigma(A) \sim_M \sigma(B)$ *for all closing substitutions* σ $(\sigma(A) \stackrel{\text{def}}{=} \{\sigma(a) \mid a \in A\})$.

(iii) $A \sim_M B$ *iff* $A \sim_M B$ *and* $B \sim_M A$.

THEOREM 6.14 *Three substitutivity properties can be proven.*

(i) $\sim_{M,0}$ *substitutive, i.e. if* $A \sim_{M,0} B$ *then* $c[A] \sim_{M,0} c[B]$.

(ii) $\sim_M$ *is substitutive.*

(iii) $\sim_M$ *is a congruence.*

PROOF. To show $\sim_{M,0}$ substitutive, show for arbitrary $a, r[\![=]\!]$ that if $r[\![c[a]]\!]\downarrow$, $r[\![c[b]]\!]\downarrow$ for some b. The remainder of the proof proceeds very similarly to $\sim_0$ substitutivity, Lemma 6.9: generalize and induct on computation length. The only difference is the assumption we are reasoning under, in the other case $a \sim_0 b$, and in this case $A \sim_{M,0} B$. To give the difference in proofs, it suffices to consider the case $c = (-)$, where

$$r[a][\![a]\!] \mapsto_1 r[a][\![a']\!].$$

Applying the induction hypothesis to the right side treating a' as fixed, $r[b][\![a']\!]\downarrow$, so $r[b][\![a]\!]\downarrow$ as well. So, since $A \sim_{M,0} B$, $r[b][\![b']\!]\downarrow$ for some b'. Then, by directedness of B, there exists b'' s.t. $b, b' \sim_0 b''$, and $r[b''][\![b'']\!]\downarrow$, proving this case.

$\sim_M$ substitutivity and $\sim$ congruence are then direct, as in Theorem 6.10 and Corollary 6.11.

QED.

For **Red** an important directed set to study is

$$\{-, f(-), f(f(-)), \ldots, f^k(-), \ldots\},$$

which is shown $\sim_M$-equal to $\{\mathrm{Y}(f)\}$. Intuitively, this means in any particular program context, some finite-depth recursion stack will suffice to compute a recursive function properly.

LEMMA 6.15 $\{\mathrm{Y}(f)\} \sim_M \{f^k(-) \mid k \in N\}$

PROOF. Take f to be closed. The right-to-left direction is direct by computing; for the other direction, note $\mathrm{Y}(f) \sim f'(f')$, $f' \stackrel{\text{def}}{=} (\lambda x.f(x(x)))$, so it suffices to show $\{f'(f')\} \sim_M \{f^k(-) \mid k \in \mathrm{N}\}$. Expanding definitions, the goal becomes

$r[\![f'(f')]\!]\downarrow$ implies $r[\![f^k(-)]\!]\downarrow$ for some k.

The rest of the proof parallels the proof of $\sim_0$ substitutivity (Lemma 6.9); we generalize and induct on computation length. 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 $r[f^k(-)][\![f^k(-)]\!]\downarrow$. By induction hypothesis, for some k', $r[f^{k'}(-)][\![f(f^{k'}(-))]\!]\downarrow$, so since $f^{k'}(-) \sim f^{k'+1}(-)$, $r[f^{k'+1}(-)][\![f^{k'+1}(-))]\!]\downarrow$, and letting k be $k'+1$, the goal is proven.
QED.

All that remains is to show the atomic case of fixed point induction, $a[\mathrm{Y}(f)] \sim b[\mathrm{Y}(f)]$, holds. This theorem will directly justify admissibility of these types.

THEOREM 6.16 (ATOMIC FIXED POINT INDUCTION) *If for all k $a[f^k(-)] \sim b[f^k(-)]$,*
$a[\mathrm{Y}(f)] \sim b[\mathrm{Y}(f)]$.
PROOF. By $\sim_M$ substitutivity (Theorem 6.14) and Lemma 6.15, $\{a[f^k(-)] \mid k \in \mathrm{N}\} \sim_M \{a[\mathrm{Y}(f)]\}$ and $\{b[f^k(-)] \mid k \in \mathrm{N}\} \sim_M \{b[\mathrm{Y}(f)]\}$. Then, using the fact $\{a[f^k(-)] \mid k \in \mathrm{N}\} \sim_M \{b[f^k(-)] \mid k \in \mathrm{N}\}$ (by definition of $\sim_M$) and the above equivalences, the result is immediate.
QED.

6.3 Defining the types and their inhabitants

Given the collection of terms and basic relations thereupon, the types and their inhabitants are simultaneously defined. Types are defined by induction on their structure. 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 $x{:}A \to B(x)$ 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* $x{:}A \to B(x)$ is considered well-formed. The solution taken here is to simultaneously define types and their inhabitants. [All87b, All87a] gives a more complete description of this technique. It is suggested on first reading for the reader to skip ahead to lemmas 6.22, 6.23, and 6.24, which give properties of the inductive definition; the details of how the definition is constructed are secondary.

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

DEFINITION 6.17 *A* type interpretation *is a three-place relation* $\tau(A, \epsilon, \vec{\gamma})$ *where* A *is a term,* ϵ *is a one-place relation on terms, and* $\vec{\gamma}$ *is a 6-tuple of truth conditions (0-place relations)* $\gamma_0 \ldots \gamma_5$.

$\tau(A, \epsilon, \vec{\gamma})$ means A is a type with its members specified by ϵ, and γ_i is true just when A Type_i. The following definitions make this more clear:

DEFINITION 6.18

A $\text{Type}_{\tau,i}$ *iff* $\exists \epsilon, \vec{\gamma}.\ \tau(A, \epsilon, \vec{\gamma})$ *and* $\vec{\gamma}_i$
$a \in_\tau A$ *iff* $\exists \epsilon, \vec{\gamma}.\ \tau(A, \epsilon, \vec{\gamma})$ *and* γ_0 *and* $\epsilon(a)$.

A $\text{Type}_{\tau,i}$ means A is a type at level i in interpretation τ. A Type_τ will abbreviate A $\text{Type}_{\tau,0}$. $a \in_\tau A$ means a is a member of the type A. $\forall a.$ refers to arbitrary terms a, and $\forall a^{\text{r}}.$ refers to arbitrary radioactive terms a^{r}. To give a proper type definition, a monotonic operator is defined, which we then take a fixed point of. For this, type interpretations need to be ordered.

DEFINITION 6.19 *Interpretation* τ' *contains interpretation* τ, *written* $\tau \sqsubseteq_{\text{I}} \tau'$, *iff*

for all $A, \epsilon, \vec{\gamma}$, *if* $\tau(A, \epsilon, \vec{\gamma})$ *and* γ_0, *then* $\tau'(A, \epsilon, \vec{\gamma})$.

The desired type interpretation ν is now inductively defined.

DEFINITION 6.20 ν *is the least fixed-point of the following monotonic operator* Ψ *on type interpretations:*

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

EITHER $T \mapsto \text{E}$, *in which case*
  γ_i *iff true, and*
  $\forall t.\ \epsilon(t)$ *is true*
OR $T \mapsto \text{N}$, *in which case*
  γ_i *iff true, and*
  $\forall t.\ \epsilon(t)$ *iff* $t \sim n$, *where* n *is* $0, 1, 2, \ldots$
OR $T \mapsto a \sim b$, *in which case*
  $\gamma_{0/1/2/3/4/5}$ *iff for* $i = 3$, b *contains no radioactive subterms, and for* $k = 4, 5$, a *contains no radioactive subterms, and*
  $\forall t.\ \epsilon(t)$ *iff* $t \sim 0$ *and* $a \sim b$
OR $T \mapsto a{\downarrow}$, *in which case*
  $\gamma_{0/1/2/3/4/5}$ *iff for* $k = 4$, a *contains no radioactive subterms, and*
  $\forall t.\ \epsilon(t)$ *iff* $t \sim 0$ *and* $a{\downarrow}$

OR $T \mapsto a$ *in* A, *in which case*
$\gamma_{0/1/2/3/4/5}$ *iff* $A\ \mathrm{Type}_{\tau,0/1/2/3/4/5}$
where for $k = 3, 4$, a *contains no radioactive subterms, and*
$\forall t.\ \epsilon(t)$ *iff* $t \sim 0$ *and* $a \in_\tau A$
OR $T \mapsto x{:}A \to B(x)$, *in which case*
$\gamma_{0/1/2/3/4/5}$ *iff* $A\ \mathrm{Type}_{\tau,0/0/3/5/4/4}$ & $\forall a \in_\tau A.\ B(a)\ \mathrm{Type}_{\tau,0/1/2/4/4/5}$, *and*
$\forall t.\ \epsilon(t)$ *iff* $t \sim \lambda x.b$ & $\forall a \in_\tau A.\ t(a) \in_\tau B(a)$
OR $T \mapsto x{:}A \times B(x)$, *in which case*
$\gamma_{0/3/4/5}$ *iff* $A\ \mathrm{Type}_{\tau,0/3/4/5}$ & $\forall a \in_\tau A.\ B(a)\ \mathrm{Type}_{\tau,0/3/4/5}$, *and*
$\gamma_{1/2}$ *iff* $A\ \mathrm{Type}_{\tau,1/2}$ & $\forall a^{\mathrm{r}}.\ a^{\mathrm{r}} \in_\tau A \Rightarrow B(a^{\mathrm{r}})\ \mathrm{Type}_{\tau,2/2}$, *and*
$\forall t.\ \epsilon(t)$ *iff* $t \sim \langle a, b\rangle$ & $a \in_\tau A$ & $b \in_\tau B(a)$
OR $T \mapsto \overline{A}$, *in which case*
$\gamma_{0/1/2/3/4/5}$ *iff* $A{\downarrow} \Rightarrow A\ \mathrm{Type}_{\tau,0/1/2/3/4/5}$, *and*
$\forall t.\ \epsilon(t)$ *iff* $t{\downarrow} \Rightarrow t \in_\tau A$

LEMMA 6.21 Ψ *is monotonic over* $\sqsubseteq_{\mathrm{I}}$.

PROOF. Suppose $\sigma \sqsubseteq_{\mathrm{I}} \tau$; show $\Psi(\sigma) \sqsubseteq_{\mathrm{I}} \Psi(\tau)$. Abbreviate $\Psi(\sigma)$ as σ' and $\Psi(\tau)$ as τ'. Arbitrary T, ϵ, γ; suppose $\sigma'(T, \epsilon, \vec{\gamma})$; show $\tau'(T, \epsilon, \vec{\gamma})$. We will just look at part of one of the more interesting cases of the proof:

CASE $T \mapsto x{:}A \to B$: Since $\sigma'(T, \epsilon, \vec{\gamma})$, $A\ \mathrm{Type}_\sigma$ and for all $a \in_\sigma A$, $B(a)\mathrm{Type}_\sigma$. To show $\tau'(T, \epsilon, \vec{\gamma})$, first show $A\mathrm{Type}_\tau$, and $\forall a \in_\tau A.B(a)\mathrm{Type}_\tau$. $A\ \mathrm{Type}_\tau$ follows directly from the assumption because $A\ \mathrm{Type}_\sigma$ implies $A\ \mathrm{Type}_\tau$ by the definition of $\sqsubseteq_{\mathrm{I}}$. To prove the second statement, take arbitrary $a \in_\tau A$, and show $B(a)\ \mathrm{Type}_\tau$. $a \in_\tau A$ means $\tau(A, \epsilon', \vec{\gamma}')$ for some ϵ' and $\vec{\gamma}'$; by the definition of $\sqsubseteq_{\mathrm{I}}$, $\sigma(A, \epsilon', \vec{\gamma}')$, so $a \in_\sigma A$. Thus $B(a)\ \mathrm{Type}_\sigma$, which means $B(a)\ \mathrm{Type}_\tau$ by the definition of $\sqsubseteq_{\mathrm{I}}$. The other two subcases needed to show $\tau(T, \epsilon, \vec{\gamma})$ are similar.
QED.

The least fixed-point ν of Ψ is thus sensible because Ψ is a monotonic operator. The least fixed point construction also gives rise to an induction principle on the structure of this definition, used frequently in the lemmas to follow. Since hereafter we will be working the the type interpretation ν, subscripts may be dropped: $a \in A \stackrel{\mathrm{def}}{=} a \in_\nu A$, and $A\ \mathrm{Type} \stackrel{\mathrm{def}}{=} A\ \mathrm{Type}_\nu$.

With this definition in place, numerous lemmas about these relations can be proven either directly or by straightforward induction on the definition.

LEMMA 6.22 *(i)* $a \in A$ *implies* $A\ \mathrm{Type}$.

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

(iii) *If* $A\ \mathrm{Type}_{1/2/3/4/5}$ *then* $A\ \mathrm{Type}_0$.

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

(v) $a[b] \in A[B]$ *iff* $a[b^{\mathrm{r}}] \in A[B^{\mathrm{r}}]$, *and* $A[B]\ \mathrm{Type}_{0/1}$ *iff* $A[B^{\mathrm{r}}]\ \mathrm{Type}_{0/1}$.

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 term to be types, and for what it means to be an inhabitant of the different types. Variables a, b, c, A, B are arbitrary closed terms.

LEMMA 6.23 *Type formation is characterized by the following properties.*

(i) $\mathrm{E}\ \mathrm{Type}_{0/1/2/3/4/5}$, $\mathrm{N}\ \mathrm{Type}_{0/1/2/3/4/5}$.

(ii) $a{\downarrow}\ \mathrm{Type}_{0/1/2/3/4/5}$ *iff for* Type_4, *a contains no radioactive subterms.*

(iii) $a \underset{\sim}{} b\ \mathrm{Type}_{0/1/2/3/4/5}$ *iff for* $\mathrm{Type}_{3,4}$, *b contains no radioactive subterms, and for* $\mathrm{Type}_{4,5}$, *a contains no radioactive subterms.*

(iv) *a in* $A\ \mathrm{Type}_{0/1/2/3/4/5}$ *iff* $A\mathrm{Type}_{0/1/2/3/4/5}$ *and for* $\mathrm{Type}_{3,4}$, *a contains no radioactive subterms.*

(v) $x{:}A \to B(x)\ \mathrm{Type}_{0/1/2/3/4/5}$ *iff* $A\ \mathrm{Type}_{0/0/3/5/4/4}$ *and for all* $a \in A$, $B(a)\ \mathrm{Type}_{0/1/2/4/4/5}$.

(vi) $x{:}A{\times}B(x)\mathrm{Type}_{0/3/4/5}$ *iff* $A\mathrm{Type}_{0/3/4/5}$ *and for all* $a \in A$, $B(a)\mathrm{Type}_{0/3/4/5}$.

(vii) $x{:}A \times B(x)\ \mathrm{Type}_{1/2}$ *iff* $A\ \mathrm{Type}_{1/2}$ *and for all* $a^{\mathrm{r}} \in A$, $B(a^{\mathrm{r}})\ \mathrm{Type}_{2/2}$.

(viii) $\overline{A}\ \mathrm{Type}_{0/1/2/3/4/5}$ *iff* $A{\downarrow}$ *implies* $A\ \mathrm{Type}_{0/1/2/3/4/5}$.

LEMMA 6.24 *Type membership is characterized by the following properties.*

(i) $c \in a{\downarrow}$ *iff* $c \sim 0$ *and* $a{\downarrow}$.

(ii) $c \in a \underset{\sim}{} b$ *iff* $c \sim 0$ *and* $a \underset{\sim}{} b$.

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

(iv) $c \in (a\ in\ A)$ *iff* $(a\ in\ A)\ \mathrm{Type}$ *and* $c \sim 0$ *and* $a \in A$.

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

(vi) $c \in x{:}A \times B(x)$ *iff* $x{:}A \times B(x)\,\mathrm{Type}$ *and* $c \sim \langle c.1, c.2\rangle$ *and* $c.1 \in A$ *and* $c.2 \in B(c.1)$.

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

6.4 Soundness of the fixed point principles

The fixed point typing principle is not true for all types, as shown in section 4.3. Here we show the principle is in fact true for the Type_1 types: if $f \in \overline{A} \to \overline{A}$, and $A\,\mathrm{Type}_1$, then $\mathrm{Y}(f) \in \overline{A}$. The fixed point induction principle is also proven sound. The general proof technique employed is induction on the definition of $A\,\mathrm{Type}_1$. The complete proof of validity for all Type_1 types is too cumbersome to present because a number of complicating factors, so instead a complete proof for a restricted collection of Type_1 types is presented, and the key ideas involved in generalizing this proof to all Type_1 types are given.

Dependent product types $x{:}B \times C$ cause all of the complications for **Red**, so they are restricted to be independent, i.e. x does not occur free in C.

DEFINITION 6.25 $A\,\mathrm{Type}_1^-$ *iff* $A\,\mathrm{Type}_1$ *and for all subterms of* A *of the form* $x{:}B \times C$, x *is not free in* C.

The type levels 2, 3, 4, 5 are only useful in the full proof; without truly dependent products, levels 2–5 are equivalent to level 1. We will use the shorthand $f^{j\uparrow}$ for $\{f^k(-) \mid k \geq j\}$; when it is not ambiguous, notation will be stretched to make assertions such as $f^{j\uparrow} \in A$.

THEOREM 6.26 (RESTRICTED FIXED POINT TYPING) *If* $A\,\mathrm{Type}_1^-$ *and* $f^k(-) \in \overline{A}$ *for all* k, *then* $\mathrm{Y}(f) \in \overline{A}$.

PROOF. This theorem is proved by generalizing to an arbitrary context $a[-]$ containing $f^k(-)$ or $\mathrm{Y}(f)$, from which the theorem directly follows. Show

$$\text{if } A\,\mathrm{Type}_1^- \text{ and } \exists j.\, \forall k \geq j.\, a[f^k(-)] \in A,\ a[\mathrm{Y}(f)] \in A.$$

Proceed by induction on the definition of the type A, $\nu(A, \epsilon, \vec{\gamma})$—inductively the result is assumed true for all the types A is defined in terms of.

From the assumption $a[f^{j\uparrow}(-)] \in A$, show $a[\mathrm{Y}(f)] \in A$ by case analysis on the form of A.

CASE $A \mapsto$ E or N: For E, the result is trivial, for N it is computationally clear: $\mathrm{Y}(f) \sim f(Y(f))$ by computing, and $- \underset{\sim}{} \mathrm{Y}(f)$, so $f^k(-) \underset{\sim}{} \mathrm{Y}(f)$ by Theorem 6.10.
CASE $A \mapsto b{\downarrow}$ or $b \underset{\sim}{} c$: Computationally obvious just as above, because $a[f^{j\uparrow}]$ and $a[\mathrm{Y}(f)]$ can at most compute to 0.
CASE $A \mapsto x{:}B \to C(x)$: To show $a[\mathrm{Y}(f)] \in x{:}B \to C(x)$ it suffices to show

$$a[\mathrm{Y}(f)] \sim \lambda x.d \text{ and for all } b \in B,\ a[\mathrm{Y}(f)](b) \in C(b),$$

using Lemma 6.24 (uses of which we refrain from citing hereafter). The first condition is computationally direct from the assumption
$a[f^{j\uparrow}(-)] \sim \lambda x.d'$. For the second condition, assume $b \in B$; by assumption,

$$\exists j.\, \forall k \geq j.\, a[f^k(-)](b) \in C[b].$$

So, since A is defined in terms of $C[b]$, the induction hypothesis may be applied to complete the proof of this case (let $a[-]$ be $a(-)(b)$).
CASE $A \mapsto x{:}B \times C$: To show $a[\mathrm{Y}(f)] \in x{:}B \times C$ it suffices to show

$$a[\mathrm{Y}(f)] \sim \langle b, c\rangle \text{ and } a[\mathrm{Y}(f)].1 \in B \text{ and } a[\mathrm{Y}(f)].2 \in C$$

The first condition is computationally direct from the assumption $a[f^{j\uparrow}(-)] \sim \langle b', c'\rangle$. For the second condition $a[\mathrm{Y}(f)].1 \in A$, by assumption,

$$\exists j.\, \forall k \geq j.\, a[f^k(-)] \in B.$$

So, since A is defined in terms of B, the induction hypothesis may be applied to prove this condition. For the third condition, $a[\mathrm{Y}(f)].1$ is not free in C by the definition of Type_1^-, so it is symmetric to the previous condition.
CASE $A \mapsto \overline{B}$: To show $a[\mathrm{Y}(f)] \in \overline{B}$ it suffices to show

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

So, assuming $a[\mathrm{Y}(f)]{\downarrow}$, show $a[\mathrm{Y}(f)] \in B$. Since $\{f^k(-)\} \sim_M \{\mathrm{Y}(f)\}$ (by Lemma 6.15) and by the definition of $\sim_M$, there is some j' for which $a(f^{j'}(-)){\downarrow}$; all larger approximations also clearly terminate. By assumption,

$$\exists j.\, \forall k \geq j.\ \text{ if } a[f^k(-)]{\downarrow} \text{ then } a[f^k(-)] \in B.$$

Picking j'' to be $\max(j, j')$,

$$\exists j''.\, \forall k \geq j''.\ a[f^k(-)] \in B,$$

which inductively yields the result.
QED.

The proof of the fixed point principles for all Type_1 types is now outlined. The Type_1^- types above have no dependent products because dependent products can introduce subterms $\text{Y}(f)$ and $f^k(-)$ into types ($\langle \text{Y}(f), 0\rangle \in x{:}A \times ()x \sim a$ implies $0 \in (\text{Y}(f) \sim a)$), which in some cases can lead to contradictions—see section 4.3. All occurrences of $\text{Y}(f)$ and $f^k(-)$ in types are then radioactively labeled, so their use can be restricted.

The type levels 1/2/3/4/5 are organized so that a different fact is true for each level, reflected in the following lemma, from which the fixed point principles directly follow.

LEMMA 6.27 (FIXED POINT PREDICATE) *For arbitrary (non-radioactive) terms a and A,*

For A Type_1, if $a[f^k(-)] \in A$ then $a[\text{Y}(f)] \in A$,
for $A[f^k(-)^{\text{r}}]$, $A[\text{Y}(f)^{\text{r}}]$ Type_2, if $a[f^{k\uparrow}(-)] \in A[f^{k\uparrow}(-)]$ then $a[\text{Y}(f)] \in A[\text{Y}(f)]$,
for $A[f^{k\uparrow}(-))^{\text{r}}]$, $A[\text{Y}(f))^{\text{r}}]$ Type_3, if $a[\text{Y}(f)] \in A[\text{Y}(f)]$ then $a[f^{k\uparrow}(-)] \in A[f^{k\uparrow}(-)]$,
for $A[f^{k\uparrow}(-))^{\text{r}}]$, $A[\text{Y}(f))^{\text{r}}]$ Type_4, nothing is known,
for $A[f^{k\uparrow}(-))^{\text{r}}]$, $A[\text{Y}(f))^{\text{r}}]$ Type_5, if $a[f^k(-)] \in A[f^k(-)]$ then
$a[\text{Y}(f)] \in A[\text{Y}(f)]$ and $a[f^{k\uparrow}(-)] \in A[f^{k\uparrow}(-)]$.

Proofs of the above statements are interdependent: the level 2 proof is used in the level 1 proof for the case of the dependent product, the level 3 proof is used in the level 2 proof for the case of a dependent function space, the level 5 proof is used in the level 3 proof also for the case of a dependent function space. The proof of all of the above facts must then proceed by simultaneous induction.

The base cases, where A is an atomic type, are direct with one exception: for the Type_2 case, $b[f^{j\uparrow}(-)] \sim b'[f^{j\uparrow}(-)]$ implies $b[\text{Y}(f)] \sim b'[\text{Y}(f)]$ is exactly Theorem 6.16.

A further complicating factor in an inductive proof is there is no single type to perform induction on; there is only a family of types $\{A[f^{j\uparrow}(-)], A[\text{Y}(f)]\}$. To obtain an induction principle over this family of types, it is necessary to rephrase Definition 6.20 in terms of families of types of the above form. This yields an inductive definition $\hat{\Psi}(\hat{\tau}) = \hat{\tau}'$, with least fixed point $\hat{\nu}(\hat{A}, \hat{\epsilon}, \vec{\hat{\gamma}})$, where $\hat{A}$ is a family of types of the above form, $\hat{\epsilon}$ is an $\hat{A}$-indexed family of membership relations, and $\vec{\hat{\gamma}}$ is an $\hat{A}$-indexed family of typehood relations.

This new definition is then shown to be unchanged when it overlaps the old definition.

LEMMA 6.28 *Letting* $\hat{A}$ *be* $\{A\}$, $\hat{\nu}(\hat{A}, \hat{\epsilon}, \vec{\hat{\gamma}})$ *for some* $\hat{\epsilon}, \vec{\hat{\gamma}}$ *iff* $\nu(A, \epsilon, \vec{\gamma})$ *for some* $\epsilon, \vec{\gamma}$. *Furthermore,* $\hat{\epsilon}(A) = \epsilon$, *and* $\hat{\gamma}_i(A) = \gamma_i$ *for* $i = 0 \ldots 5$.

The fixed point predicate Lemma (stated in somewhat different form) may then be proven by direct induction on the type definition $\hat{\nu}$, and the fixed point principles are immediate corollaries.

THEOREM 6.29 (GENERAL FIXED POINT TYPING) *If* $f \in \overline{A} \to \overline{A}$ *and* $\overline{A}\,\mathrm{Type}_1$, *then* $\mathrm{Y}(f) \in \overline{A}$.

PROOF. From the fixed point predicate Lemma (6.27), let A be $\overline{A}$ and $a[-]$ be $(-)$.
QED.

THEOREM 6.30 (FIXED POINT INDUCTION) *If* $a \in P(-)$, $(\forall b \in B.\ \forall c.\ c \in P(x) \Rightarrow a \in P(f(x)))$, $x{:}B \times P(x)\,\mathrm{Type}_1$, *and* $f \in \overline{B} \to \overline{B}$, *then* $a \in P(\mathrm{Y}(f))$.

PROOF. From the assumptions, prove $a \in P(\mathrm{Y}(f))$. First, we show $a \in P(f^k(-))$ for all k. For $k = 0$, this is an assumption; if $a \in P(f^k(-))$, then from the second and fourth assumptions, $a \in P(f^{k+1}(-))$.

For all $b^{\mathrm{r}} \in \overline{B}$, $P(b^{\mathrm{r}})\,\mathrm{Type}_2$, so from the second case of the fixed point predicate lemma (Lemma 6.27) $a \in P(\mathrm{Y}(f))$.
QED.

6.5 Soundness

All of the pieces of the interpretation are in place, and it only remains to show each of the rules are sound. Let $\mathcal{C}$ stand for either $b \in B$ or $B\,\mathrm{Type}_{0/1/2/3/4/5}$.

DEFINITION 6.31 *Validity of the sequents*

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

are defined as

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

where if x_i *is a radioactive variable* x_i^{r}, a_i *is a radioactively labeled term* a_i^{r}.

THEOREM 6.32 (SOUNDNESS OF INTERPRETATION) *For* **Red**, *if* $? \vdash \mathcal{C}$ *then* $? \models \mathcal{C}$.

PROOF. For most of the rules, soundness follows directly from Lemmas 6.22, 6.23, and 6.24. The computation rules follow directly from the definitions of $\sim$, and Lemma 6.8; substitutivity of $\sim$, (Sim subst), is exactly Theorem 6.10. The fixed point typing principle (Fixed point) is Theorem 6.29, and fixed point induction (Induction) is Theorem 6.30.
QED.

THEOREM 6.33 (CONSISTENCY) **Red** *is logically consistent, i.e. there exist types which cannot be proven inhabited.*

PROOF. The type $0 \sim 1$ by the semantics has no members, so for no a is it the case that $\models a \in 0 \sim 1$. Therefore, for no a is $\vdash a \in 0 \sim 1$ provable, by Theorem 6.32.
QED.

For the extensions presented in section 5, soundness proofs are sketched. The **Pure Red** subset types are justified in [CAB+86]; for the directed set principles, the type $A \sim_M B$ is interpreted by the relation $\sim_M$, Definition 6.13. (Dirset subst) soundness is Theorem 6.14, (Dirset approx) is Lemma 6.15, and the other principles are straightforward from the definition of $\sim_M$. For soundness of the types of **Full Red**, see [CAB+86]. The principles of **Classical Red** are direct if one takes a classical view of this section.

7 Conclusion

A powerful and useful extension to CTT to make it a more complete programming logic has been presented. The biggest shortcoming is the inelegance of the fixed point typing principle and the subsequent need to divide types into admissible and non-admissible types. An alternative solution which doesn't share these shortcomings is **Pure Red**, where notions of admissible type are unnecessary, because fixed points may be typed and fixed point induction justified from more basic principles of computation.

Another alternative not discussed in this paper is the use of an intensional induction principle; see [Smi88]. This requires the type theory to be intensional, making equality reasoning difficult.

There has recently been similar work to give a category-theoretic framework for partial computations via monads [Mog89]. Moggi's approach is more abstract than that carried out here, because the monad operator $T(A)$ represents general computations over type A, not just potential divergence $\overline{A}$. But because of its generality, no principles for typing or inductively reasoning about fixed points exist. Crole and Pitts [CP90] fill this gap by

adding a fixed point object to the monad, from which fixed points may be typed. A full programming logic has yet to be built using this methodology.

Acknowledgements

The original work this paper is built on [CS87] was carried out jointly with my thesis advisor Robert Constable, who also suggested the problem. Stuart Allen and Doug Howe also gave many useful comments and criticisms.

References

[ACHA90] S. F. Allen, R. L. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 95–105, 1990.

[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.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[Bas88] D. A. Basin. Building theories in Nuprl. Technical Report 88-932, Department of Computer Science, Cornell University, 1988.

[BCMS89] R. Backhouse, P. Chisholm, G. Malcom, and E. Saaman. Do-it-yourself type theory (part 1). *Formal Aspects of Computing*, 1:19–84, 1989.

[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.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[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.

[CP90] R. Crole and A. Pitts. New foundations for fixpoint computations. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE, 1990.

[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.

[dS69] J. W. deBakker and D. Scott. A theory of programs. unpublished notes, 1969.

[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.

[Fef89] S. Feferman. Logics for termination and correctness of functional programs. In *Logic from computer science*. MSRI, 1989.

[FFK87] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[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.

[HN89] S. Hayashi and H. Nakano. *PX: a Computational Logic*. MIT press, 1989.

[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.

[How88] 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.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[Iga72] 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.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[Luo89] Z. Luo. ECC, an extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 386–395. IEEE, 1989.

[Mac86] D. B. MacQueen. Using dependent types to express modular structure. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, 1986.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[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.

[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.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[Men87] P. F. Mendler. Inductive definition in type theory. Technical Report 87-870, Department of Computer Science, Cornell University, September 1987. Ph.D. Thesis.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, 1989.

[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.

[MT91] I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. to appear in *Functional Programming*, 1991.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's type theory.* Oxford Science Publications, 1990.

[Ong88] C.-H. L. Ong. Fully abstract models of the lazy lambda calculus. In *Symposium on the Foundations of Computer Science*, pages 368–376, 1988.

[Pau87] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge, 1987.

[Plo75] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[Plo76] G. Plotkin. A powerdomain construction. *Siam J. Computing*, 5:452–487, 1976.

[PN90] L. C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. preprint, 1990.

[Sco70] D. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture notes in Mathematics*, pages 237–275, Berlin, 1970. Spinger-Verlag.

[Smi88] S. F. Smith. Partial objects in type theory. Technical Report 88-938, Department of Computer Science, Cornell University, August 1988. Ph.D. Thesis.

[Smi91] S. F. Smith. From operational to denotational semantics. In *MFPS 1991*, Lecture notes in Computer Science, 1991. (To appear).

[Ste72] S. Stenlund. *Combinators, Lambda-Terms and Proof Theory.* D. Reidel, Dordrecht, Holland, 1972.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type. *J. Symbolic Logic*, 32:198–212, 1967.

[Tal90] C. L. Talcott. A theory for program and data type specification. In *DISCO*, 1990.