Admissibility of Fixpoint Induction over Partial Types

Karl Crary

Cornell University

Abstract. Partial types allow the reasoning about partial functions in type theory. The partial functions of main interest are recursively computed functions, which are commonly assigned types using fixpoint induction. However, fixpoint induction is valid only on *admissible* types. Previous work has shown many types to be admissible, but has not shown any dependent products to be admissible. Disallowing recursion on dependent product types substantially reduces the expressiveness of the logic; for example, it prevents much reasoning about modules, objects and algebras.

In this paper I present two new tools, *predicate-admissibility* and *monotonicity,* for showing types to be admissible. These tools show a wide class of types to be admissible; in particular, they show many dependent products to be admissible. This alleviates difficulties in applying partial types to theorem proving in practice. I also present a general least upper bound theorem for fixed points with regard to a computational approximation relation, and show an elegant application of the theorem to compactness.

1 Introduction

One of the earliest logical theorem provers was the LCF system [12, 20], based on the logic of partial computable functions [22, 23]. Although LCF enjoyed many groundbreaking successes, one problem it faced was that, although it supported a natural notion of *partial* function, it had difficulty expressing the notion of a *total* function. Later theorem provers based on constructive type theory, such as Nuprl [5], based on Martin-Löf type theory [19], and Coq [3], based on the Calculus of Constructions [9], faced the opposite problem; they had a natural notion of total functions, but had difficulty dealing with partial functions. The lack of partial functions seriously limited the scope of those theorem provers, because it made them unable to reason about programs in real programming languages where recursion does not always necessarily terminate.

This problem was addressed by Constable and Smith [7], who introduced into their type theory the *partial type* $\overline{T}$, which is like a "lifted" version of T. The type $\overline{T}$ contains all members of T as well as all divergent terms. Using the partial type, partial functions from A to B may be given the type $A \to \overline{B}$. That is, when applied to an argument in A, such a function either diverges or converges to a result in B.

In a partial type theory, recursively defined objects may be typed using the *fixpoint principle:* if f has type $\overline{T} \to \overline{T}$ then $\mathit{fix}(f)$ has type $\overline{T}$. However, the fixpoint principle is not valid for every type T; it is only valid for types that are *admissible.* This phenomenon was not unknown to LCF; LCF used the related device of fixpoint induction, which was valid only for admissible predicates. When the user attempted to invoke fixpoint induction, the system would automatically check that the goal was admissible using a set of syntactic rules [16].

Despite their obvious uses in program analysis, partial types have seen little use in theorem proving systems [8, 4, 2]. This is due in large part to the fact that too few types have been known to be admissible. Smith, in his doctoral dissertation [24], gave a significant class of admissible types for a Nuprl-like theory, but his class required product types to be non-dependent. The type $\Sigma x{:}A.B$ (where x appears free in B) was explicitly excluded. Partial type extensions to Coq [2] were just as restrictive, assuming function spaces to be the only type constructor. Excluding dependent products is quite a strong restriction; they are used in encodings of modules [18], objects [21], algebras [17], and even such simple devices as variant records. Furthermore, ruling out dependent products disallows reasoning using fixpoint induction as in LCF. (This is explained further in Section 2.3.) Finally, the restriction is particularly unsatisfying since most types used in practice do turn out to be admissible, and may be shown so by metatheoretical reasoning.

In this paper I present a very wide class of admissible types using two devices, a condition called *predicate-admissibility* and a monotonicity condition. In particular, many dependent products may be shown to be admissible. Predicate-admissibility relates to when the limit of a chain of type approximations contains certain terms, whereas admissibility relates to the membership of a single type. The term "predicate-admissibility" stems from its similarity to the notion of admissibility of predicates in domain theory (and LCF), where there has been considerable research (this work was influenced by Igarashi [16], for example), but I will not discuss the connection in this paper. Monotonicity is a simpler condition that will be useful for showing types admissible that do not involve partiality.

The paper is organized as follows: In Section 2 I lay out the theory for which these results are formalized. In Section 3 I prove some computational lemmas needed for the admissibility results. The primary result is a least upper bound theorem for fixed points with regard to a computational approximation relation. This result is quite general, and may be applied more widely than just to the purposes for which I use it. I present my main results in Section 4, beginning with a summary of Smith's admissibility class and then widening the class using predicate-admissibility and monotonicity. Concluding remarks appear in Section 5. Most proofs have been omitted in this paper due to space limitations; those proofs appear in the companion technical report [10].

	Type Formation	Introduction	Elimination
universe i	$\mathbb{U}_i$ (for $i \geq 1$)	type formation operators	
disjoint union	$T_1 + T_2$	$inj_1(e)$ $inj_2(e)$	$case(e, x_1.e_1, x_2.e_2)$
function space	$\Pi x{:}T_1.T_2$	$\lambda x.e$	$e_1 e_2$
product space	$\Sigma x{:}T_1.T_2$	$\langle e_1, e_2 \rangle$	$\pi_1(e)$ $\pi_2(e)$
natural numbers	$\mathbb{N}$	$0, 1, 2, \ldots$	assorted operations
equality	$t_1 = t_2$ *in* T	$\star$	
partial type	$\overline{T}$		
convergence	t *in*! T	$\star$	

Fig. 1. Type Theory Syntax

2 The Type Theory

The type theory in which I formalize the results of this paper is a variant of the Nuprl type theory [5] extended with partial types (that is, types containing possibly divergent objects). This theory is a subset of the type theory of Crary [11] and is similar to Smith's theory [24]. The major difference between the theory used here and Smith's is that the latter does not provide a notion of equality; the ramifications of handling equality are discussed in Constable and Crary [6] and at greater length in Crary [11].

2.1 Preliminaries

As data types, the theory contains natural numbers (denoted by $\mathbb{N}$), disjoint unions (denoted by $T_1 + T_2$), dependent products[1] (denoted by $\Sigma x{:}T_1.T_2$), and dependent function spaces (denoted by $\Pi x{:}T_1.T_2$). When x does not appear free in T_2, I write $T_1 \times T_2$ for $\Sigma x{:}T_1.T_2$ and $T_1 \to T_2$ for $\Pi x{:}T_1.T_2$. As usual, alpha-equivalent terms are considered identical. When t_1 and t_2 are alpha-equivalent, I write $t_1 \equiv t_2$.

Types themselves are terms in the theory and belong to a predicative hierarchy of universes, $\mathbb{U}_1, \mathbb{U}_2, \mathbb{U}_3$, etc. The universe $\mathbb{U}_1$ contains all types built from the base types only (*i.e.,* built without universes), and the universe $\mathbb{U}_{i+1}$ contains all types build from the base types and the universes $\mathbb{U}_1, \ldots, \mathbb{U}_i$. In particular, no universe is a member of itself. Propositions are interpreted as types using the propositions-as-types principle [14], but that is not central to the purposes of this paper.

Each type T comes with an intrinsic equality relation denoted by $t_1 = t_2 \in T$. Membership is also derived from this relation; $t \in T$ when $t = t \in T$. The equality

[1] These are sometimes referred to in the literature as dependent sums, but I prefer the terminology to suggest the connection to the non-dependent type $T_1 \times T_2$.

relation is introduced into the type theory as the type $t_1 = t_2$ *in* T, which is inhabited by the term $\star$ when $t_1 = t_2 \in T$ and is empty otherwise, provided that $t_1, t_2 \in T$. If either of t_1 or t_2 does not belong to T, then $t_1 = t_2$ *in* T is not well-formed. (Note that $t_1 = t_2 \in T$ is a metatheoretical assertion whereas $t_1 = t_2$ *in* T is a type in the theory.) The empty type *Void* is defined as $0 = 1$ *in* $\mathbb{N}$.

The *partial type* $\overline{T}$ is like a "lifted" version of T; it contains all the members of T as well as all divergent terms. Partial functions from A to B may then be given the type $A \rightarrow \overline{B}$. Two terms are equal in $\overline{T}$ if they both diverge, or if they both converge and are equal in T.

Convergence is expressed within the type theory by the type t *in*! T, which is inhabited by the term $\star$ when $t \in \overline{T}$ and t converges, and is empty if $t \in \overline{T}$ but t does not converge. If $t \notin \overline{T}$ then t *in*! T is not well-formed. (Again, note that t *in*! T is a type in the theory, but convergence, which is defined formally below, is a metatheoretical assertion.)

2.2 Computation

Underlying the type theory is the computation system shown in Figure 2. The computation system is defined by a small-step evaluation relation (denoted by $t_1 \mapsto t_2$), and a set of canonical terms. Whether a term is canonical is governed by its outermost operator; the canonical terms are those appearing in the first and second columns of Figure 1. The computation system is call-by-name and contains operators for constructing and destructing functions, pairs and disjoint unions. The computation system also contains various standard operations for computing and analyzing natural numbers, but these are not particularly interesting and are omitted from Figure 2. Of particular interest is the operator *fix*, which allows the recursive definition of objects is evaluated by the rule $\mathit{fix}(f) \mapsto f(\mathit{fix}(f))$.[2] Two important properties of evaluation are that evaluation is deterministic and canonical forms are terminal:

Proposition 1 If $t \mapsto t_1$ and $t \mapsto t_2$ then $t_1 \equiv t_2$. Moreover, if t is canonical then $t \not\mapsto t'$ for any t'.

If $t \mapsto^* t'$ and t' is canonical then I say that t converges (abbreviated $t{\downarrow}$) and t converges to t' (abbreviated $t \Downarrow t'$). Note that if $t \Downarrow t_1$ and $t \Downarrow t_2$ then $t_1 \equiv t_2$ and that if t is canonical then $t \Downarrow t$. If there exists an infinite sequence $t_1, t_2, \ldots$ where $t_i \mapsto t_{i+1}$ then I say that t_1 diverges (abbreviated $t_1{\uparrow}$).

The computation system is used to define the relation $t_1 = t_2 \in T$, which specifies the memberships of types and when terms are equal in those types.[3]

[2] The use of a *fix* operator greatly simplifies the presentation of these results (particularly the proof of Theorem 8), but it could be eliminated and replaced with the Y combinator. Similarly, the choice of a call-by-name computation system simplifies the formalism, but is also not critical to the results.

[3] Since the definition contains negative occurrences of $t_1 = t_2 \in T$, it is not immediately clear that it is a valid definition. Allen [1] and Harper [13] have shown how such a definition may be converted to a conventional inductive definition.

$$\frac{f \mapsto f'}{f\ e \mapsto f'\ e} \qquad \frac{t \mapsto t'}{\pi_i(t) \mapsto \pi_i(t')} \qquad \frac{t \mapsto t'}{case(t, x.e_1, x.e_2) \mapsto case(t', x.e_1, x.e_2)}$$

$$\frac{}{(\lambda x.e)t \mapsto e[t/x]} \qquad \frac{}{\pi_i(\langle t_1, t_2 \rangle) \mapsto t_i} \qquad \frac{}{case(inj_i(t), x.e_1, x.e_2) \mapsto e_i[t/x]}$$

$$\frac{}{fix(f) \mapsto f\ fix(f)}$$

Fig. 2. The Computation System

$t \in T$	iff $t = t \in T$
$T\ \ type$	iff $\exists i.\, T = T \in \mathbb{U}_i$
$t_1 = t_2 \in T$	iff $\exists t_1', t_2', T'.\, (t_1 \Downarrow t_1') \wedge (t_2 \Downarrow t_2') \wedge (T \Downarrow T') \wedge$ $(t_1' = t_2' \in T')$
$inj_1(a) = inj_1(a') \in A + B$	iff $A + B\ \ type \wedge a = a' \in A$
$inj_2(b) = inj_2(b') \in A + B$	iff $A + B\ \ type \wedge b = b' \in B$
$\lambda x.b = \lambda x.b' \in \Pi x{:}A.B$	iff $\Pi x{:}A.B\ \ type \wedge$ $\forall a, a'.\, a = a' \in A \Rightarrow b[a/x] = b'[a'/x] \in B[a/x]$
$\langle a, b \rangle = \langle a', b' \rangle \in \Sigma x{:}A.B$	iff $\Sigma x{:}A.B\ \ type \wedge (a = a' \in A) \wedge (b = b' \in B[a/x])$
$n = n' \in \mathbb{N}$	iff $n \equiv n'$ (n, n' natural numbers)
$\star = \star \in (a = a'\ in\ A)$	iff $(a = a'\ in\ A)\ \ type \wedge (a = a' \in A)$
$t = t' \in \overline{T}$	iff $\overline{T}\ \ type \wedge (t\downarrow \Leftrightarrow t'\downarrow) \wedge (t\downarrow \Rightarrow t = t' \in T)$
$\star = \star \in (a\ in!\ A)$	iff $(a\ in!\ A)\ \ type \wedge a\downarrow$

Fig. 3. Type Definitions

Part of the definition (for types other than universes) appears in Figure 3; the full definition appears in the companion technical report [10]. This equality relation is constructed to respect evaluation: if $t \in T$ and $t \mapsto t'$ then $t = t' \in T$.

2.3 The Fixpoint Principle

The central issue of this paper is the *fixpoint principle:*

$$f \in \overline{T} \to \overline{T} \Rightarrow fix(f) \in \overline{T}$$

The fixpoint principle allows us to type recursively defined objects, such as recursive functions. Unfortunately, unlike in programming languages, where the principle can usually be invoked on arbitrary types, expressive type theories such as the one in this paper contain types for which the fixpoint principle is not valid. I shall informally say that a type is *admissible* if the fixpoint principle is valid for that type and give a formal definition in Section 4. To make maximum use of a partial type theory, one wants as large a class of admissible types as possible.

Smith [24] showed that any type is admissible provided that it is constructed without using *dependent* product spaces (*i.e.,* Σ types) or universes. However, prohibiting dependent products is a strong restriction. Dependent products are used in, for example, encodings of modules [18], objects [21], algebras [17], and even such simple devices as variant records. Furthermore, the restriction also disallows reasoning about programs using the closely related principle of fixpoint induction

$$\frac{P[\bot] \quad P[e] \Rightarrow P[f(e)]}{P[\mathit{fix}\,(f)]}$$

which may be encoded with the fixpoint principle [24], but only by using dependent products.

In Section 4 I will explore two wide classes of admissible types, one derived from a *predicate-admissibility* condition and another derived from a monotonicity condition. But first, it is worthwhile to note that there are indeed inadmissible types:

Theorem 2 There exist inadmissible types.

Proof. This example is due to Smith [24]. Let T be the type of functions that do not halt for all inputs, and let f be the function that halts on zero, and on any other n immediately recurses with $n-1$. This is formalized as follows:

$$T \stackrel{\text{def}}{=} \Sigma h{:}(\mathbb{N} \to \overline{\mathbb{N}}).\ ((\Pi x{:}\mathbb{N}.\, h\ x\ \mathit{in}!\ \mathbb{N}) \to \mathit{Void})$$
$$f \stackrel{\text{def}}{=} \lambda p.\langle \lambda x.\ \mathit{if}\ x \leq 0\ \mathit{then}\ 0\ \mathit{else}\ \pi_1(p)(x-1), \lambda y.\star\rangle$$

Intuitively, any finite approximation of $\mathit{fix}\,(f)$ will recurse some limited number of times and then give up, placing it in T, but $\mathit{fix}\,(f)$ will halt for every input, excluding it from T. Formally, the function f has type $\overline{T} \to \overline{T}$, but $\mathit{fix}\,(f) \notin \overline{T}$. (The proof of this appears in the companion technical report [10].) Therefore T is not admissible.

3 Computational Lemmas

Before presenting my main results in Section 4, I first require some lemmas about the computational behavior of the fixpoint operator. The central result is that $\mathit{fix}\,(f)$ is the least upper bound of the finite approximations $\bot, f(\bot), f(f(\bot)), \ldots$ with regard to a computational approximation relation defined below. The compactness of *fix* (if $\mathit{fix}\,(f)$ halts then one of its finite approximations halts) will be a simple corollary of this result. However, the proof of the least upper bound theorem is considerably more elegant than most proofs of compactness.

3.1 Computational Approximation

For convenience, throughout this section we will frequently consider terms using a unified representation scheme for terms: A term is either a variable or a compound term $\theta(x_{11} \cdots x_{1k_1}.t_1, \ldots, x_{n1} \cdots x_{nk_n}.t_n)$ where the variables $x_{i1}, \ldots, x_{ik_i}$

are bound in the subterm t_i. For example, the term $\Pi x{:}T_1.T_2$ is represented $\Pi(T_1, x.T_2)$ and the term $\langle t_1, t_2\rangle$ is represented $\langle\rangle(t_1, t_2)$.

Informally speaking, a term t_1 approximates the term t_2 when: if t_1 converges to a canonical form then t_2 converges to a canonical form with the same outermost operator, and the subterms of t_1's canonical form approximate the corresponding subterms of t_2's canonical form. The formal definition appears below and is due to Howe [15]. Following Howe, when R is a binary relation on closed terms, I adopt the following convention extending R to possibly open terms: if t and t' are possibly open then $t\ R\ t'$ if and only if $\sigma(t)\ R\ \sigma(t')$ for every substitution σ such that $\sigma(t)$ and $\sigma(t')$ are closed.

Definition 3 (Computational Approximation)

- Let R be a binary relation on closed terms and suppose e and e' are closed. Then $e\ C(R)\ e'$ exactly when if $e \Downarrow \theta(\vec{x}_1.t_1, \ldots, \vec{x}_n.t_n)$ then there exists some closed $e'' = \theta(\vec{x}_1.t'_1, \ldots, \vec{x}_n.t'_n)$ such that $e' \Downarrow e''$ and $t_i\ R\ t'_i$.
- $e \leq_0 e'$ whenever e and e' are closed.
- $e \leq_{i+1} e$ if and only if $e\ C(\leq_i)\ e'$
- $e \leq e'$ if and only if $e \leq_i e'$ for every i

The following are facts about computational approximation that will be used without explicit reference. The first two follow immediately from the definition, the third uses determinism and the last is proven using Howe's method [15].

Proposition 4

- $\leq$ and $\leq_i$ are reflexive and transitive.
- If $t \mapsto t'$ then $t' \leq t$ and $t' \leq_i t$.
- If $t \mapsto t'$ then $t \leq t'$ and $t \leq_i t'$.
- (**Congruence**) If $e \leq e'$ and $t \leq t'$ then $e[t/x] \leq e'[t'/x]$.

3.2 Finite Approximations

With this notion of computational approximation in hand, we may now show that the terms $\bot, f\,\bot, f(f\,\bot), \ldots$ form a chain of approximations to the term $\mathit{fix}(f)$. Let $\bot$ be the divergent term $\mathit{fix}(\lambda x.x)$. Since $\bot$ never converges, $\bot \leq t$ for any term t. Let f^i be defined as follows:

$$f^0 \stackrel{\text{def}}{=} \bot$$
$$f^{i+1} \stackrel{\text{def}}{=} f(f^i)$$

Certainly $f^0 \leq f^1$, since $f^0 \equiv \bot$. By congruence, $f(f^0) \leq f(f^1)$, and thus $f_1 \leq f_2$. Similarly, $f^i \leq f^{i+1}$ for all i. Thus $f^0, f^1, f^2, \ldots$ forms a chain; I now wish to show that $\mathit{fix}(f)$ is an upper bound of the chain. Certainly $f^0 \leq \mathit{fix}(f)$. Suppose $f^i \leq \mathit{fix}(f)$. By congruence $f(f^i) \leq f(\mathit{fix}(f))$. Thus, since $\mathit{fix}(f) \mapsto f(\mathit{fix}(f))$, it follows that $f^{i+1} \equiv f(f^i) \leq f(\mathit{fix}(f)) \leq \mathit{fix}(f)$. By induction it follows that $\mathit{fix}(f)$ is an upper bound of the chain. The following corollary follows from congruence and the definition of approximation:

Corollary 5 If there exists j such that $e[f^j/x]\downarrow$ then $e[\mathit{fix}(f)/x]\downarrow$. Moreover, the canonical forms of $e[f^j/x]$ and $e[\mathit{fix}(f)/x]$ must have the same outermost operator.

3.3 Least Upper Bound Theorem

In this section I summarize the proof of the least upper bound theorem. To begin, we need a lemma stating a general property of evaluation. Lemma 6 captures the intuition that closed, noncanonical terms that lie within a term being evaluated are not destructed; they either are moved around unchanged (the lemma's first case) or are evaluated in place with the surrounding term left unchanged (the lemma's second case). The variable x indicates positions where the term of interest is found and (in the second case) the variable y indicates which of those positions, if any, is about to be evaluated.

Lemma 6 If $e_1[t/x] \mapsto e_2$, and $e_1[t/x]$ is closed, and t is closed and noncanonical, then either

1. there exists e_2' such that for any closed t', $e_1[t'/x] \mapsto e_2'[t'/x]$, or
2. there exist e_1' and t' such that $e_1 \equiv e_1'[x/y]$, $t \mapsto t'$ and for any closed t'', $e_1'[t'', t/x, y] \mapsto e_1'[t'', t'/x, y]$.

It is worthwhile to note that Propositions 1 and 4 and Lemma 6 are the only general properties of evaluation used in the proof of the least upper bound theorem, and that these properties are true in computational systems with considerable generality. Consequently, the theorem may be used in a variety of applications beyond the computational system of this paper.

Lemma 7 shows that *fix* terms may be simulated by sufficiently large finite approximations. The lemma is simplified by using computational approximation instead of evaluation for the simulation, which makes it unnecessary to track which of the approximations are unfolded and which are not, an issue that often complicates compactness proofs.

Lemma 7 For all f, e_1 and e_2 (where f is closed and x is the only free variable of e_1), there exist j and e_2' such that if $e_1[\mathit{fix}(f)/x] \mapsto^* e_2$ then $e_2 \equiv e_2'[\mathit{fix}(f)/x]$ and for all $k \geq j$, $e_2'[f^{k-j}/x] \leq e_1[f^k/x]$.

Theorem 8 (Least Upper Bound) For all f, t and e (where f is closed), if $\forall j.\, e[f^j/x] \leq t$, then $e[\mathit{fix}(f)/x] \leq t$.

There are two easy corollaries to the least upper bound theorem. One is that $\mathit{fix}(f)$ is the least fixed point of f, and the other is compactness.

Corollary 9 (Least Fixed Point) For all closed f and t, if $f(t) \leq t$ then $\mathit{fix}(f) \leq t$.

Proof. Certainly $f^0 \equiv \bot \leq t$. Then $f^1 \equiv f(f^0) \leq f(t) \leq t$. Similarly, by induction, $f^j \leq t$ for any j. Therefore $\mathit{fix}(f) \leq t$ by Theorem 8. □

Corollary 10 (Compactness) If f is closed and $e[\mathit{fix}(f)/x]\downarrow$ then there exists some j such that $e[f^j/x]\downarrow$. Moreover, the canonical forms of $e[\mathit{fix}(f)/x]$ and $e[f^j/x]$ must have the same outermost operator.

Proof. Suppose there does not exist j such that $e[f^j/x]\downarrow$. Then $e[f^j/x] \leq \bot$ for all j. By Theorem 8, $e[\mathit{fix}(f)/x] \leq \bot$. Therefore $e[\mathit{fix}(f)/x]$ does not converge, but this contradicts the assumption,[4] so there must exist j such that $e[f^j/x]\downarrow$. Since $e[f^j/x] \leq e[\mathit{fix}(f)/x]$, the canonical forms of $e[f^j/x]$ and $e[\mathit{fix}(f)/x]$ must have the same outermost operator. □

4 Admissibility

I am now ready to begin specifying some wide classes of types for which the fixpoint principle is valid. First we define admissibility. The simple property of validating the fixpoint principle is too specific to allow any good closure conditions to be shown easily, so we generalize a bit to define admissibility. A type is *admissible* if the upper bound $t[\mathit{fix}(f)]$ of an approximation chain $t[f^0], t[f^1], t[f^2], \ldots$ belongs to a type whenever a cofinite subset of the chain belongs to the type. This is formalized as Definition 12, but first I define some convenient notation.

Notation 11 For any natural number j, the notation $t^{[j]_f}$ means $t[f^j/w]$, and the notation $t^{[\omega]_f}$ means $t[\mathit{fix}(f)/w]$. Also, the f subscript is dropped when the intended term f is unambiguously clear.

Definition 12 A type T is *admissible* (abbreviated $\mathrm{Adm}(T)$) if:

$$\forall f, t, t'.\ (\exists j.\ \forall k \geq j.\ t^{[k]} = t'^{[k]} \in T) \Rightarrow t^{[\omega]} = t'^{[\omega]} \in T$$

As expected, admissibility is sufficient to guarantee applicability of the fixpoint principle:

Theorem 13 For any T and f, if T is admissible and $f = f' \in \overline{T} \to \overline{T}$ then $\mathit{fix}(f) = \mathit{fix}(f') \in \overline{T}$.

Proof. $\overline{T}$ *type* since $\overline{T} \to \overline{T}$ *type*. Note that $f^j = f'^j \in \overline{T}$ for every j. Suppose $\mathit{fix}(f)\downarrow$. By compactness, $f^j\downarrow$ for some j. Since $f^j = f'^j \in \overline{T}$, it follows that $f'^j\downarrow$ and thus $\mathit{fix}(f')\downarrow$ by Corollary 5. Similarly $\mathit{fix}(f')\downarrow$ implies $\mathit{fix}(f)\downarrow$. It remains to show that $\mathit{fix}(f) = \mathit{fix}(f') \in T$ when $\mathit{fix}(f)\downarrow$. Suppose again that $\mathit{fix}(f)\downarrow$. As before, there exists j such that $f^j\downarrow$ by compactness. Hence $f^j = f'^j \in T$. Since T is admissible, $\mathit{fix}(f) = \mathit{fix}(f') \in T$. □

A number of closure conditions exist on admissible types and are given in Lemma 14. Informally, compound types other than dependent products are admissible so long as their component types in positive positions are admissible.

[4] Although this proof is non-constructive, a slightly less elegant constructive proof may be derived directly from Lemma 7.

Base types—natural numbers, convergence types, and (for this lemma only) equality types—are always admissible. These are essentially Smith's admissible types, except that for a function type to be admissible Smith requires that its domain type also be admissible.

Lemma 14

- $\mathrm{Adm}(A + B)$ if $A + B$ *type* and $\mathrm{Adm}(A)$ and $\mathrm{Adm}(B)$
- $\mathrm{Adm}(\Pi x{:}A.B)$ if $\Pi x{:}A.B$ *type* and $\forall a \in A.\, \mathrm{Adm}(B[a/x])$
- $\mathrm{Adm}(A \times B)$ if $A \times B$ *type* and $\mathrm{Adm}(A)$ and $\mathrm{Adm}(B)$
- $\mathrm{Adm}(\mathbb{N})$
- $\mathrm{Adm}(a = a'\ in\ A)$ if $(a = a'\ in\ A)$ *type*
- $\mathrm{Adm}(\overline{A})$ if $\overline{A}$ *type* and $\mathrm{Adm}(A)$
- $\mathrm{Adm}(a\ in!\ A)$ if $(a\ in!\ A)$ *type*

Proof. The proof follows the same lines as Smith's proof, except that handling equality adds a small amount of complication to the proof. I show the function case by way of example.

Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \Pi x{:}A.B$. I need to show that $t^{[\omega]} = t'^{[\omega]} \in \Pi x{:}A.B$. By assumption $\Pi x{:}A.B$ *type*. Both $t^{[j]}$ and $t'^{[j]}$ converge to lambda abstractions, so, by Corollary 5, $t^{[\omega]} \Downarrow \lambda x.b$ and $t'^{[\omega]} \Downarrow \lambda x.b'$ for some terms b and b'. Suppose $a = a' \in A$. To get that $b[a/x] = b'[a'/x] \in B[a/x]$ it suffices to show that $t^{[\omega]}a = t'^{[\omega]}a' \in B[a/x]$. Since $\mathrm{Adm}(B[a/x])$, it suffices to show that $\forall k \geq j.\, t^{[k]}a = t'^{[k]}a' \in B[a/x]$, which follows from the supposition. □

Unfortunately, Lemma 14 can show the admissibility of a product space only if it is *non-dependent*. Dependent products do not have an admissibility condition similar to that of dependent functions. This reason for this is as follows: Admissibility states that a *single fixed type* contains the limit of an approximation chain if it contains a cofinite subset of that chain. For functions, disjoint union, partial types, and non-dependent products it is possible to decompose prospective members in such a way that admissibility may be applied to a single type (such as the type $B[a/x]$ used in the proof of Lemma 14). In contrast, for a dependent product, the right-hand term's desired type depends upon the left-hand term, which is changing at the same time as the right-hand term. Consequently, there is no single type into which to place the right-hand term.

However, understanding the problem with dependent products suggests a solution, to generalize the definition of admissibility to allow the type to vary. This leads to the notion of *predicate-admissibility* that I discuss in the next section.

4.1 Predicate-Admissibility

Definition 15 A type T is *predicate-admissible* for x in S (abbreviated $\mathrm{Adm}(T \mid x : S)$) if:

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j.\, \forall k \geq j.\, e^{[k]} \in S \wedge t^{[k]} = t'^{[k]} \in T[e^{[k]}/x]) \Rightarrow t^{[\omega]} = t'^{[\omega]} \in T[e^{[\omega]}/x]$$

Predicate-admissibility of the right-hand side (along with admissibility of the left) is sufficient to show the admissibility of a dependent product type:

Lemma 16 The type $\Sigma x{:}A.B$ is admissible if $\Sigma x{:}A.B$ *type* and $\mathrm{Adm}(A)$ and $\mathrm{Adm}(B \mid x : A)$.

Proof. Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \Sigma x{:}A.B$. It is necessary to show that $t^{[\omega]} = t'^{[\omega]} \in \Sigma x{:}A.B$. Both $t^{[j]}$ and $t'^{[j]}$ converge to pairs, so, by Corollary 5, $t^{[\omega]} \Downarrow \langle a, b\rangle$ and $t'^{[\omega]} \Downarrow \langle a', b'\rangle$ for some terms a, b, a' and b'. To get that $a = a' \in A$ it suffices to show that $\pi_1(t^{[\omega]}) = \pi_1(t'^{[\omega]}) \in A$. Since $\mathrm{Adm}(A)$, it suffices to show that $\forall k \geq j.\, \pi_1(t^{[k]}) = \pi_1(t'^{[k]}) \in A$, which follows from the supposition.

To get that $b = b' \in B[a/x]$ (the interesting part), it suffices to show that $\pi_2(t^{[\omega]}) = \pi_2(t'^{[\omega]}) \in B[\pi_1(t^{[\omega]})/x]$. Since $\mathrm{Adm}(B \mid x : A)$, it suffices to show that $\pi_1(t^{[\omega]}) \in A$, which has already been shown, and $\forall k \geq j.\, \pi_1(t^{[k]}) \in A \wedge \pi_2(t^{[k]}) = \pi_2(t'^{[k]}) \in B[\pi_1(t^{[k]})/x]$, which follows from the supposition. □

The conditions for predicate-admissibility are more elaborate, but also more general. I may immediately state conditions for types other than functions. Informally, compound types other than functions are predicate-admissible so long as their component types are predicate-admissible, and base types are always predicate-admissible.

Lemma 17

- $\mathrm{Adm}(A + B \mid y : S)$ if $\forall s \in S.\, (A + B)[s/y]$ *type* and $\mathrm{Adm}(A \mid y : S)$ and $\mathrm{Adm}(B \mid y : S)$.
- $\mathrm{Adm}(\Sigma x{:}A.B \mid y : S)$ if $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ *type* and $\mathrm{Adm}(A \mid y : S)$ and $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathrm{Adm}(\mathbb{N} \mid y : S)$
- $\mathrm{Adm}(a_1 = a_2 \ in \ A \mid y : S)$ if $\forall s \in S.\, (a_1 = a_2 \ in \ A)[s/y]$ *type* and $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(\overline{A} \mid y : S)$ if $\forall s \in S.\, \overline{A}[s/y]$ *type* and $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a \ in! \ A \mid y : S)$ if $\forall s \in S.\, (a \ in! \ A)[s/y]$ *type*

Predicate-admissibility of a function type is more complicated because a function argument with the type $A[e^{[\omega]}/x]$ does not necessarily belong to any of the finite approximations $A[e^{[j]}/x]$. To settle this, it is necessary to require a *coadmissibility* condition on the domain type. Then a function type will be predicate-admissible if the domain is weakly coadmissible and the codomain is predicate-admissible.

Definition 18 A type T is *weakly coadmissible* for x in S (abbreviated $\mathrm{WCoAdm}(T \mid x : S)$) if:

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j.\, \forall k \geq j.\, e^{[k]} \in S) \wedge t = t' \in T[e^{[\omega]}/x] \Rightarrow (\exists j.\, \forall k \geq j.\, t = t' \in T[e^{[k]}/x])$$

A type T is *coadmissible* for x in S (abbreviated $\mathrm{CoAdm}(T \mid x : S)$) if:

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j.\, \forall k \geq j.\, e^{[k]} \in S) \wedge t^{[\omega]} = t'^{[\omega]} \in T[e^{[\omega]}/x] \Rightarrow \\ (\exists j.\, \forall k \geq j.\, t^{[k]} = t'^{[k]} \in T[e^{[k]}/x])$$

Lemma 19 $\mathrm{Adm}(\Pi x{:}A.B \mid y : S)$ whenever $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ *type* and $\mathrm{WCoAdm}(A \mid y : S)$ and $\forall s \in S, a \in A[s/y].\, \mathrm{Adm}(B[a/x] \mid y : S)$

Clearly coadmissibility implies weak coadmissibility. A general set of conditions listed in Lemma 20 establish weak and full coadmissibility for various types. Weak and full coadmissibility are closed under disjoint union and dependent sum formation, and full coadmissibility is additionally closed under equality-type formation. I use both notions of coadmissibility, rather than just adopting one or the other, because full coadmissibility is needed for equality types but under certain circumstances weak coadmissibility is easier to show (Proposition 21 below).

Lemma 20

- $A + B$ is (weakly) coadmissible for y in S if $\forall s \in S.\, (A + B)[s/y]$ *type* and A and B are (weakly) coadmissible for y in S
- $\mathrm{WCoAdm}(\Sigma x{:}A.B \mid y : S)$ if $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ *type* and $\mathrm{WCoAdm}(A \mid y : S)$ and $\forall s \in S, a \in A[s/y].\, \mathrm{WCoAdm}(B[a/x] \mid y : S)$
- $\mathrm{CoAdm}(\Sigma x{:}A.B \mid y : S)$ if $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ *type* and $\mathrm{CoAdm}(A \mid y : S)$ and $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathbb{N}$ is strongly or weakly coadmissible for y in any S
- $\mathrm{CoAdm}(a_1 = a_2\ in\ A \mid y : S)$ if $\forall s \in S.\, (a_1 = a_2\ in\ A)[s/y]$ *type* and $\mathrm{CoAdm}(A \mid y : S)$
- $\overline{A}$ is (weakly) coadmissible for y in S if $\forall s \in S.\, \overline{A}[s/y]$ *type* and A is (weakly) coadmissible for y is S
- $a\ in!\ A$ is strongly or weakly coadmissible for y in S if $\forall s \in S.\, (a\ in!\ A)[s/y]$ *type*

When T does not depend upon S, predicate-admissibility and weak coadmissibility become easier to show:

Proposition 21 Suppose x does not appear free in T. Then:

- $\mathrm{Adm}(T)$ if $\mathrm{Adm}(T \mid x : S)$ and S is inhabited
- $\mathrm{Adm}(T \mid x : S)$ if $\mathrm{Adm}(T)$
- $\mathrm{WCoAdm}(T \mid x : S)$

There remains one more result related to predicate-admissibility. Suppose one wishes to show $\mathrm{Adm}(T \mid x : S)$ where T depends upon x. There are two ways that x may be used in T. First, T might contain an equality type where x appears in one or both of the equands. In that case, predicate-admissibility can be shown with the tools discussed above. Second, T may be an expression that computes a type from x. In this case, T can be simplified using untyped reasoning [15], but another tool will be needed if T performs any case analysis.

Lemma 22 Consider a type $case(d, x.A, x.B)$ that depends upon y from S. Suppose there exist T_1 and T_2 such that:

- $\forall s \in S.\, d[s/y] \in (T_1 + T_2)[s/y]$
- $\forall s \in S, t \in T_1[s/y].\, A[s, t/y, x]$ *type*
- $\forall s \in S, t \in T_2[s/y].\, B[s, t/y, x]$ *type*

Then the following are the case:

- $\mathrm{Adm}(case(d, x.A, x.B) \mid y : S)$ if $\mathrm{Adm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ and $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$
- $\mathrm{WCoAdm}(case(d, x.A, x.B) \mid y : S)$ if $\mathrm{WCoAdm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ and $\mathrm{WCoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$
- $\mathrm{CoAdm}(case(d, x.A, x.B) | y : S)$ if $\mathrm{CoAdm}(A[\pi_1(z), \pi_2(z)/y, x] | z : (\Sigma y{:}S.T_1))$ and $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$

4.2 Monotonicity

In some cases a very simple device may be used to show admissibility. We say that a type is monotone if it respects computational approximation, and it is easy to show that all monotone types are admissible.

Definition 23 A type T is *monotone* (abbreviated $\mathrm{Mono}(t)$) if $t = t' \in T$ whenever $t \in T$ and $t \leq t'$.

Lemma 24 All monotone types are admissible.

Proof. Let f, t and t' be arbitrary and suppose there exists j such that $t^{[j]} = t'^{[j]} \in T$. Since $t^{[j]} \leq t^{[\omega]}$ and $t'^{[j]} \leq t'^{[\omega]}$, it follows that $t^{[j]} = t^{[\omega]} \in T$ and $t'^{[j]} = t'^{[\omega]} \in T$. The result follows directly. □

All type constructors are monotone except universes and partial types, which are never monotone. The proof of this fact is easy [15].

Proposition 25

- $\mathrm{Mono}(A + B)$ if $\mathrm{Mono}(A)$ and $\mathrm{Mono}(B)$
- $\mathrm{Mono}(\Pi x{:}A.B)$ if $\mathrm{Mono}(A)$ and $\forall a \in A.\, \mathrm{Mono}(B[a/x])$
- $\mathrm{Mono}(\Sigma x{:}A.B)$ if $\mathrm{Mono}(A)$ and $\forall a \in A.\, \mathrm{Mono}(B[a/x])$
- $\mathrm{Mono}(\mathbb{N})$, $\mathrm{Mono}(a_1 = a_2 \in A)$, and $\mathrm{Mono}(a\ in!\ A)$

It is worthwhile to note that all these admissibility results are proved constructively, with the exception of the full coadmissibility of partial types, which is necessarily classical (an algorithm that computed the index j could be used to solve the halting problem).

Recall the inadmissible type T from Theorem 2. That type fails the predicate-admissibility condition because of the negative appearance of a function type, which could not be shown weakly coadmissible, and it fails the monotonicity condition because it contains the partial type $\overline{\mathbb{N}}$.

5 Conclusions

An interesting avenue for future investigation would be to find some negative results characterizing inadmissible types. Such negative results would be particularly interesting if they could be given a syntactic character, like the results of this paper. Along these lines, it would be interesting to find whether the inability to show coadmissibility of function types represents a weakness of this proof technique or an inherent limitation.

The results presented above provide *metatheoretical* justification for the fixpoint principle over many types. In order for these results to be useful in theorem proving, they must be introduced into the logic. One way to do this, and the way it is presently done in my implementation of partial types in the Nuprl theorem proving system, is to introduce types to represent the assertions $\mathrm{Adm}(T)$, $\mathrm{Adm}(T \mid x : S)$, etc., that are inhabited exactly when the underlying assertion is true (in much that same way as the equality type is inhabited exactly when the equands are equal), and to add rules relating to these types that correspond to the lemmas of Section 4. This brings the tools into the system in a semantically justifiable way, but it is unpleasant in that it leads to a proliferation of new types and inference rules stemming from discoveries outside the logic. It would be preferable to deal with admissibility within the logic. A theory with intensional reasoning principles, such as the one proposed in Constable and Crary [6], would allow reasoning about computation internally. Then these results could be proved within the theory and the only extra rule that would be required would be a single rule relating admissibility to the the fixpoint principle.

However they are placed into the logic, these results allow for recursive computation on a wide variety of types. This make partial types and fixpoint induction a useful tool in type-theoretic theorem provers. It also makes it possible to reason about many recursive programs that used to be barred from the logic because they could not be typed.

References

1. S. Allen. A non-type-theoretic definition of Martin-Löf's types. In *Second IEEE Symposium on Logic in Computer Science*, pages 215–221, Ithaca, New York, June 1987.
2. P. Audebaud. Partial objects in the calculus of constructions. In *Sixth IEEE Symposium on Logic in Computer Science*, pages 86–95, Amsterdam, July 1991.
3. B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.
4. D. A. Basin. An environment for automated reasoning about partial functions. In *Ninth International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.
5. R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

6. R. L. Constable and K. Crary. Computational complexity and induction for partial computable functions in type theory. Technical report, Department of Computer Science, Cornell University, 1997.
7. R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *Second IEEE Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.
8. R. L. Constable and S. F. Smith. Computational foundations of basic recursive function theory. In *Third IEEE Symposium on Logic in Computer Science*, pages 360–371, Edinburgh, Scotland, July 1988.
9. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
10. K. Crary. Admissibility of fixpoint induction over partial types. Technical Report TR98-1674, Department of Computer Science, Cornell University, Apr. 1998.
11. K. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1998. Forthcoming.
12. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
13. R. Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14:71–84, 1992.
14. W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
15. D. J. Howe. Equality in lazy computation systems. In *Fourth IEEE Symposium on Logic in Computer Science*, 1989.
16. S. Igarashi. Admissibility of fixed-point induction in first-order logic of typed theories. Technical Report AIM-168, Computer Science Department, Stanford University, May 1972.
17. P. B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Jan. 1995.
18. D. MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, Jan. 1986.
19. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 1982.
20. L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.
21. B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994.
22. D. Scott. Outline of a mathematical theory of computation. In *Fourth Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.
23. D. Scott. Lattice theoretic models for various type-free calculi. In *Fourth International Congress of Logic, Methodology and Philosophy of Science*. North-Holland, 1972.
24. S. F. Smith. *Partial Objects in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Jan. 1989.