Extracting Recursive Programs in Type Theory

Scott F. Smith
The Johns Hopkins University, Baltimore, MD 21218 USA

Abstract

Martin-Löf's constructive type theory is a foundational theory of mathematics and programming. The key to using type theory as a logic is the *formulas as types* principle, whereby propositional assertions are directly expressed by types. Furthermore, using the *extraction method* programs can automatically be extracted from proofs. One weakness of the use of the extraction method to date, however, is that it is impossible to extract arbitrary recursively defined programs from proofs, because all functions in type theory must be total. We show that under some extensions to type theory extraction of recursive programs is direct and useful.

1 Introduction

We believe the motivations and directions of this work are best seen in the light of historical development, and will thus take a short historical digression into intuitionism, realizability, proofs as programs, and formulas as types.

The most basic tenet of the philosophy of intuitionism, as set forth by Brouwer, is that a mathematical proof is a mental construction (see [8]). Starting from Heyting, logicians have been making Brouwer's non-formal philosophy more formal and precise. Heyting developed a formal logic for intuitionism, Heyting arithmetic, and argued that proofs in this logic are mental constructions. Kleene made Heyting's argument precise in his realizability interpretation of logic [9], the basic idea of which is worth recounting. Heyting stated that a proof of $\forall x.A(x)$ consisted of a construction which from any x can produce a proof of $A(x)$. For Kleene, this was made precise as follows. The realizer e of a formula $\forall x.A(x)$ is a computable function such that for every x, $e(x)$ realizes $A(x)$. This pattern also holds for the other connectives: some explicit computation realizes the implicit computation inherent in intuitionistic logic. Computer scientists, of which Constable [4] was an early exponent, saw great practical potential in Kleene's method, because realizers of theorems are equivalently programs that provably meet a specification. For instance, the theorem $\forall x.\exists y.x < y \ \& \ prime(y)$ expresses the fact that there are infinitely many prime numbers: this is the specification. The realizer is a program which given x returns a prime number larger than x. This realizer is thus a program that meets the specification. The art of programming may thus be equated with the art of proving. This insight is often called the "proofs as programs" paradigm [3].

A related thread was the development of a "theory of realizers," making e *realizes* A the judgement of the theory; formulas A may be viewed as collections of terms that realize them, i.e. as *types*, and e *realizes* A may be rephrased $e \in A$. Early versions of such theories were developed by Scott and others [14,

and references therein], but it was the Constructive Type Theory (CTT) of Martin-Löf [10, 11] that brought this idea to full fruition. From this perspective, then, formulas are viewed as types. Putting this together with the "proofs as programs" paradigm, we have proofs of formulas equated with programs that are members of types.

Theorem proving systems have been constructed that allow formal proof to be equated with program construction: given a formal proof in intuitionistic logic, the system automatically extracts the realizer from the proof. One such system is λ-PRL [2]. It is worth pondering this for a moment: in λ-PRL, the user apparently is building a proof, but is also simultaneously (and perhaps even unknowingly) building a program. This gives a very different view of how one programs, as will be seen in examples later in this paper. Although it would at first glance seem an artificial way to program, one must keep in mind that the resulting program provably meets the specification embodied in the theorem. Other systems have been constructed that extend this idea to constructive type theory; instead of proving a theorem, the user is applying rules of the type theory to show a type has some member (any member), a program. As rules are applied, the program is automatically incrementally constructed. This procedure is known as *extraction*. The Nuprl system is a primary example of such a system [5].

Now we come to the reason for this digression: one weakness of the aforementioned systems is the style of programming is more restricted than most programmers would accept. There is no possibility of directly writing fixed-point programs (*i.e.* the language does not allow for arbitrary recursion) by this method. Work we have done provides a solution to this problem, however. Constructive Type Theory has been extended by adding new types and principles, and using these principles we here show that direct extraction of fixed-point programs is possible. By enriching the collection of programs that may be directly extracted, the extraction paradigm becomes a more viable method for deriving correct programs. The price we must pay is that we become directly concerned with questions of termination of programs, because some of the constructions might not be total.

Partial type theory is CTT with types for computations which might not terminate added. This is accomplished by adding a type constructor: given a type A, the type $\overline{A}$ ("A bar") is the type of computations over A. A naive interpretation of $\overline{A}$ would be $A \cup \{\bot\}$, where $\bot$ is a never terminating computation. In a rich enough partial type theory [6, 17] it is possible to directly extract fixed-point computations from proofs.

The object of this paper is to show how to carry out extraction of fixed-point programs in a type-theoretic setting. We first review basic concepts of type theory, then outline the extensions that allow fixed-point programs to be extracted. With this come new notions of proposition and proof, *partial proposition* and *partial proof*, both of which are discussed in detail. There follows an example to illustrate the actual extraction process; some readers might find it useful to skip ahead to the examples to get a flavor of the method. This paper is written to more give an overview of and intuitions behind the method; a complete collection of rules, semantics, a full proof of soundness, and other metatheoretic results are found in [17].

2 Type Theory

This section is a short survey of the constructive type theory (CTT) that is necessary for basic comprehension of this paper; for fuller descriptions and examples, see [5, 18]. Before beginning, however, it is worth mentioning that we are here solely concerned with the *predicative* theories based on Martin-Löf's Extensional Type Theory [11, 18]; impredicative theories such as the Calculus of Constructions [7] are out of our purview.

A type theory has as its language a collection of untyped *terms*. Some of these terms represent types, others computations; the two, perhaps surprisingly, are not separated. The terms include numbers and basic operations such as $0, 1, 2, \ldots$, $pred(a)$, $succ(a)$, $if\_zero(a; b; c)$; pairing and projection $\langle a, b\rangle$, $a.1$, $a.2$; injection and decision terms $inl(a)$, $inr(b)$, $decide(a; b; c)$; and functions and application $\lambda x.a$, $a(b)$. This language is thus a small functional programming language. It is possible to write fixed-point functions using the Y-combinator $\mathrm{Y} \stackrel{\mathrm{def}}{=} \lambda y.(\lambda x.y(x(x)))(\lambda x.y(x(x)))$. Note however that even though such programs may be written, they might not be typable. On untyped terms we write $a\downarrow$ to mean the computation a terminates, and write $a \sim b$ meaning a is computationally equal to b. This is an untyped equality, so our presentation differs from Martin-Löf and Nuprl theories, where equalities are intimately associated with types. See [16] for foundations of this untyped equality.

Types are collections of terms. Judgments express the truths of type theory; the judgment $a \in A$ asserts that a is a member of the type A. The rules (not presented here) give meaning to the judgments.

2.1 Types

The expressiveness of type theory is largely due to the diversity of types that are definable. Here we survey some common types, making sure to include all that are used in the remainder of the paper.

First, we have atomic data types. N is a type of natural numbers $0, 1, 2, \ldots$.

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 often 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.

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 $\langle a, b\rangle \in x{:}A \times B(x)$.

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 $inr(b) \in A + B$.

A type *void* that is empty is needed to represent the unprovable proposition. The assertion $a \in A$ is expressed within the type theory as the type a *in* A,

which is inhabited (by the placeholder 0) just when $a \in A$ holds. A type $a \sim b$ is inhabited by 0 just when a and b are computationally equal. This approach is actually a departure from Martin-Löf type theory, because there each type comes with its own definition of equality on the type, whereas here equality is untyped and universal.

2.2 Formulas as types

As discussed in the introduction, one of the defining features of constructive type theory is that it is a theory of realizability. Formulas are viewed as types for which the members are the realizers of the formula. To prove the formula, show that when it is viewed as a type, the type is inhabited, i.e. has some member. Here is an example. To constructively prove

$$\forall n{:}\mathrm{N}.\ prime(n) \vee \neg prime(n)$$

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

$$n{:}\mathrm{N} \rightarrow prime(n) + \neg prime(n)$$

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 thus a realizer for the proposition.

Formulas are defined in terms of types as follows:

$$\begin{array}{ll}
A \Rightarrow B & \stackrel{\mathrm{def}}{=} A \rightarrow B \\
A \ \& \ B & \stackrel{\mathrm{def}}{=} A \times B \\
A \vee B & \stackrel{\mathrm{def}}{=} A + B \\
\neg A & \stackrel{\mathrm{def}}{=} A \rightarrow void \\
\forall x{:}A.\ B(x) & \stackrel{\mathrm{def}}{=} x{:}A \rightarrow B(x) \\
\exists x{:}A.\ B(x) & \stackrel{\mathrm{def}}{=} x{:}A \times B(x)
\end{array}$$

2.3 Refinement proof and 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 realize 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. A sample rule, the introduction rule for (non-dependent) products, is

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

where the first line is the goal and the indented lines are the subgoals.

To prove P we need to find any member. Take the specific case of $P = A \ \& \ B$, i.e. $P = A \times B$. We need members/realizers for both A and B, and to be able to form them into a pair. But, at the time of use of this rule, it is

not expected that the complete realizers will be known. What is desired, then, is to allow postponement of construction of a and b. Following the Isabelle implementation of CTT [13], it is possible to implement rules as unification steps. The initial goal is entered as $\vdash ?p \in P$, where $?p$ is a metavariable and P is a type to be proved; as the proof proceeds, $?p$'s structure is incrementally constructed via unification steps. For instance, if $P = A \times B$, an application of the above rule unifies p with $\langle ?a, ?b \rangle$, giving subgoals of $\vdash ?a \in A$ and $\vdash ?b \in B$. Further application of rules to these two subgoals will instantiate $?a$ and $?b$ as well. Nuprl directly implements extraction via special rules, but for the purposes of presenting proofs in this paper, we use Isabelle notation.

3 Partial types

The extensions to type theory that allow fixed-point functions to be typed and thus extracted from proofs are now given.

The bar type $\overline{A}$ is the type of (possibly diverging) computations over A; this gives a very general notion of partiality. For example, consider the (total) function space on natural numbers $\mathrm{N} \to \mathrm{N}$; there are seven distinct 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 $\overline{\overline{\mathrm{N}} \to \overline{\mathrm{N}}}$. $\mathrm{N} \to \overline{\mathrm{N}}$ is a type of partial functions, allowing also the function itself to diverge. This will in fact be the type generally used herein to express partial functions, notated $x{:}A \xrightarrow{\mathrm{p}} B(x) \stackrel{\mathrm{def}}{=} \overline{x{:}A \to B(x)}$, and $A \xrightarrow{\mathrm{p}} B \stackrel{\mathrm{def}}{=} \overline{A \to B}$.

The three main principles for reasoning about bar types are as follows, ignoring issues of type well-formedness.

introduction $a \in \overline{A}$ if $a\!\downarrow$ implies $a \in A$

elimination If $a \in \overline{A}$ and $a\!\downarrow$ then $a \in A$

fixed point typing If the type $A \xrightarrow{\mathrm{p}} B$ is *admissible* and
$f \in (A \xrightarrow{\mathrm{p}} B) \to (A \xrightarrow{\mathrm{p}} B)$, then $\mathrm{Y}(f) \in A \xrightarrow{\mathrm{p}} B$.

The fixed point typing principle gives types to functions defined using the Y-combinator, and it may be shown that all partial recursive functions can be typed in this language. This principle also extends the extraction paradigm to allow fixed-point programs to be extracted from proofs. One complication of this principle is the necessity of showing types are admissible, because the principle is not valid for all types. The original presentations of partial type theory [6, 15] had too restrictive an admissibility condition: dependent products could not appear at all, meaning existential quantifiers could not be used. For instance, the example given at the end of this paper would not be expressible in the earlier versions of partial type theory. Recently the collection of admissible types has been greatly extended, making the principle much more useful [17]; a rough explanation of admissibility is as follows:

> A is *admissible* if for any dependent product subterms $x{:}B \times C$ occurring in A, and 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.

The entire theory is presented and proved sound in [17], where we refer the reader to for details. We mention that Audebaud [1] has recently developed a partial type theory for the Calculus of Constructions. Martin-Löf has also developed a partial type theory (see [12]), but his theory has partial types *only*, so all types viewed as propositions are true. This is equivalent in our theory to having bars over all type constructors.

4 Partial propositions

We sketch how these new concepts may be used for reasoning, with particular emphasis on proving by extracting fixed point objects.

The bar operator may be applied to types which represent propositions, giving types such as $\overline{\forall n{:}\mathrm{N}.\ \overline{\exists m{:}\mathrm{N}.\ P(m,n)}}$, called *partial propositions*. However, are such forms of proposition at all useful? $\overline{A}$ for any type A is trivially true under the formulas-as-types interpretation, because $\bot \in \overline{A}$. However, if $a \in \overline{A}$ and $a{\downarrow}$, then $a \in A$, so A is true. Thus, a partial proposition is very useful when its realizer terminates. Since a may diverge it does not fit the Brouwerian notion of a construction, so $a \in \overline{A}$ cannot truly be considered a realizer; such terms a are *partial proofs*. This can thus be seen to be an extension of the Brouwerian notion of construction to encompass partial constructions, much as partial functions extend total ones.

In general, all of the propositional connectives have partial analogues, defined as follows: $A\overline{\&}B \stackrel{\mathrm{def}}{=} \overline{A \,\&\, B}$, $A\overline{\vee}B \stackrel{\mathrm{def}}{=} \overline{A \vee B}$, $A\overline{\Rightarrow}B \stackrel{\mathrm{def}}{=} \overline{A \Rightarrow \overline{B}}$, $\overline{\forall}x{:}A.B(x) \stackrel{\mathrm{def}}{=} \overline{\forall x{:}A.\ \overline{B(x)}}$, and $\overline{\exists}x{:}A.\ B(x) \stackrel{\mathrm{def}}{=} \overline{\exists x{:}A.\ B(x)}$. The most interesting partial propositions are those with universal quantifiers, for $\overline{\forall}x{:}A.\ B(x)$ is the type $x{:}A \stackrel{\mathrm{p}}{\rightarrow} B(x)$, meaning its realizers are partial functions.

One important extension to the extraction paradigm is the direct extraction of fixed-point computations. This is significant, because direct extraction of fixed-points is not possible without bar types, limiting the usefulness of extraction. For instance, take the proposition that partially specifies a sorting function,

$$\forall l{:}\mathit{List}.\ \exists l'{:}\mathit{List}.\ \mathit{Ordered}(l').$$

Suppose we wished to give a "mergesort" proof of this theorem to guarantee an order $n \cdot \log n$ algorithm. Using partial propositions, the proof proceeds on the same lines as the standard mergesort procedure, extracting a mergesort algorithm. Without bar types an ordering must be defined and proved well-founded, and the proof can proceed by induction on this ordering. In the case of mergesort this ordering is nontrivial to define and reason about, and results in a more convoluted algorithm. So, if extraction is to be useful in developing efficient algorithms, partial propositions are necessary. Additionally, partial functions that cannot be proved total may nonetheless be extracted from partial propositions; these functions and their correctness properties cannot even be expressed without bar types.

Before giving an example of fixed-point function extraction, it is shown how the fixed point principle can in general be used to extract fixed-point functions. Suppose the goal to prove was $\vdash ?e \in \overline{\forall}x{:}A.\ B(x)$; by using the fixed

point principle, it suffices to show from the assumption $y \in \overline{\forall}x{:}A.\ B(x)$ that $\vdash ?e' \in \overline{\forall}x{:}A.\ B(x)$ (unify $?e = \mathrm{Y}(\lambda y.?e')$). In the proof, uses of the hypothesis y amount to fixed-point calls, but if the user is not careful a divergent proof may be constructed. For instance, $?e'$ could trivially be y, since we are assuming what we want to prove. However, in this case the extract object is $\mathrm{Y}(\lambda y.y)$, which always diverges.

5 An example

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

DEFINITION 1 *Define three predicates,*

$$x\ \mathit{Divides}\ y \stackrel{\mathrm{def}}{=} 1 < x\ \&\ \exists z{:}\mathrm{N}.\ x * z \sim y$$
$$x\ \mathit{Is\_Prime\_Thru}\ y \stackrel{\mathrm{def}}{=} \forall z{:}\mathrm{N}.\ 1 < z \leq y \Rightarrow \neg(z\ \mathit{Divides}\ x)$$
$$x\ \mathit{Is\_Composite\_Thru}\ y \stackrel{\mathrm{def}}{=} \exists z{:}\mathrm{N}.\ 1 < z \leq y\ \&\ z\ \mathit{Divides}\ x$$

LEMMA 2 *We may show*

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

The extract object $?e$ will be a fixed-point function such that $?e(x)(\mathit{pred}(x))$ decides whether or not x is prime; furthermore, if x is composite, one of its factors will be returned. Since any programmer should be able to tell by inspection that $?e(x)(\mathit{pred}(x))$ will converge for any number x, this proof can thus be used to give a (non-partial) proof that any particular number is either prime or composite.

PROOF. Using the fixed point typing principle, it suffices to assume

$$\vdash z \in (\overline{\forall}x{:}\mathrm{N}.\ \overline{\forall}y{:}\mathrm{N}.\ x\ \mathit{Is\_Prime\_Thru}\ y \vee x\ \mathit{Is\_Composite\_Thru}\ y), x \in \mathrm{N}, y \in \mathrm{N},$$

and show

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

where $?e \stackrel{\mathrm{def}}{=} \mathrm{Y}(\lambda z.\lambda x.\lambda y.?e')$. This means we are extracting a fixed-point function from this proof; inside $?e'$, applications of z are recursive calls, which proof-theoretically amount to uses of the hypothesis z. First, proceed by cases on $y \sim 0$; if this is the case the proof is trivial, so assume $y \not\sim 0$. Proceed by cases on $y\ \mathit{Divides}\ x$ (we take as given a proof $\mathit{div}(x, z) \in (z\ \mathit{Divides}\ x) \vee \neg(z\ \mathit{Divides}\ x)$).
CASE $y\ \mathit{Divides}\ x$: Clearly then x is composite, so the right disjunct can be proven.
CASE $\neg(y\ \mathit{Divides}\ x)$: Recursively use the hypothesis z for y one smaller, i.e. apply $z(x)(\mathit{pred}(y))$, and the result also follows.
QED.
The full extract object $?e$ is

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

For instance, $?e(6)(5)$, a test if 6 is prime, computes to $inr(\langle 3, \langle 0, 0\rangle\rangle)$, meaning 3 *Divides* 6 (studying the proof will convince the reader that the largest factor is returned if the number is composite).

This shows that it is possible to extract fixed-point programs from partial propositions in straightforward fashion. This in turn makes the extraction method of program derivation more viable.

References

[1] P. Audebaud. Partial objects in the calculus of constructions. In *Sixth Symposium on Logic in Computer Science*, 1991.

[2] J. L. Bates. *A Logic For Correct Program Development.* PhD thesis, Cornell University, 1979.

[3] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):113–136, 1985.

[4] R. L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233, Ljubljana, 1971.

[5] 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.

[6] 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.

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

[8] M. Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.

[9] S. C. Kleene. On the interpretation of intuitionistic number theory. *JSL*, 10:109–124, 1945.

[10] 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.

[11] 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.

[12] E. Palmgren. Domain interpretations of Martin-Löf's partial type theory. *Annals of Pure and Applied Logic*, 48:135–196, 1990.

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

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

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

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

[17] S. F. Smith. Partial computations in constructive type theory. Submitted to *Journal of Logic and Computation*, 1991.

[18] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.