A Time to Build: Constructive Programming in Lucid*

Michael R. Levy
Department of Computer Science
University of Victoria
P.O. Box 1700
Victoria, B.C.
Canada V8W 2Y2

Abstract
This paper presents the beginnings of an extension to Per Martin-Löf’s Constructive Type Theory that can be used to derive or verify Lucid programs. It does this by showing how intensional logic can be formulated constructively, using Lucid for the constructions. Simple examples of Lucid derivations are given.

1 Introduction

Formal constructive logic systems yield pleasing relationships between mathematics and Programming. From the perspective of a mathematician, a program can be regarded as a construction for a given proof. As a matter of philosophy, a mathematician may choose to regard as unproven any claimed theorem for which no proofs with constructions exist. A programmer, on the other hand, can view a theorem as a program specification, and its proof as the systematic derivation of a particular program. Programmers can also use the systems as very enhanced type-checkers: they can be used as program verification systems.

The most explored theories (as far as programming is concerned) are Per Martin-Löf’s Constructive Type Theory (CTT)[2] and Constable’s Nuprl[1].

*This research was supported by a grant from The Natural Sciences and Engineering Research Council of Canada.
Both systems use Intuitionistic Logic, formulated using an extension of Natural Deduction. The programming language is essentially $\lambda$-calculus. It certainly is applicative, although it is possible to model imperative languages, essentially in the obvious way, that is, by using a variable of type state.

In this paper, we present the beginnings of an extension of Constructive Type Theory using intensional logic rather than intuitionistic predicate logic. The programming language is, therefore, Lucid[3]. This theory can be used to derive or verify Lucid programs.

2 Constructive Formal Systems

A constructive expression of the form

$$a \in A$$

can be read in four equivalent ways:

1. $a$ is an element of the set $A$
2. $a$ is an object of type $A$
3. $a$ is evidence of the proposition $A$
4. $a$ is a program for the problem $A$

The third reading is used to obtain the constructive notion of truth: a proposition $A$ is true if it has evidence. Since a proposition can be regarded as a set, we can say that a proposition is true if it is non-empty.

Intensional propositions can only be assigned truth values in a given context. We therefore regard every intensional proposition as being dependent on context. Once the context has been established, then the conventional constructive view of truth can be used. Thus the extensional proposition ‘Socrates is Dead now’ is non-empty. The intensional proposition ‘Socrates is Dead’ contains a higher-order object, namely a program that maps a context either to the empty set or to a set of evidence for the proposition. Certain intensions are same as their extensions. All sets introduced in CTT before its extension will be assumed to have this property. In particular, we will use $N$ to denote either the natural numbers or the set of infinite sequences of natural numbers. The extended rules that are presented here can be thought of as rules for creating extensions of such sets. We will assume all the rules of CTT, without repeating them. This paper is self-contained, as
long as the reader is willing to accept intuitive arguments about elementary arithmetic and induction.

CTT uses an extended form of Natural Deduction to formulate its rules. Each rule is of the form

\[
\frac{a \in A \ b \in B \ c \in C \ldots [d \in D \ e \in E \ldots]}{f \in F}
\]

which can be read (ignoring the constructions) 'if A, B, C etc. can be proven under the assumptions D, E etc., then F is proven and the assumptions are discharged'.

One of the appealing aspects of CTT is the connection it makes between induction and recursion. This is done using rules of induction that yield primitive recursive programs as members of the corresponding proposition. For example, the induction rule for natural numbers is stated as follows:

\[
\frac{n \in N \ a \in P(0) \ f(k, y) \in P(k + 1) [k \in N, y \in P(k)]}{\text{rec}(n, a, f) \in P(n)}
\]

If some of the evidence terms are omitted, Peano's usual induction axiom is obtained:

\[
\frac{n \in N \ P(0) \ P(k + 1) [P(k)]}{P(n)}
\]

\text{rec} is a defined operator, essentially primitive recursion. Because Lucid makes an explicit connection between induction and sequences, the corresponding Lucid axiom is much more direct.

3 Towards Intensional Constructive Type Theory

We will illustrate the approach taken with Lucid using two of the important rules, namely \text{fby} introduction and \text{asa} introduction. These rules allow proofs that introduce Lucid variables using \text{fby} and \text{asa} expressions respectively.

Firstly, some notation. We will use Roman capitals for intensional propositions. All such propositions will be assumed to be time dependent. Thus we may write \(L(t)\) for the extension of \(L\) at time \(t\). We define

1. \textit{first} \(L = L(0)\)
2. \textit{next} \(L(t) = L(t + 1)\)
3. eventually \( L = \exists t \cdot L(t) \)

4. firsttime \( L = \min\{t \mid L(t)\} \)

In reading a rule say \( \frac{\phi}{\psi} \), we assume that time is fixed for \( A \). That is, each intension above the line is assumed true at some fixed point in time. \( B \) is an intension, in the usual sense.

The first rule introduces the Lucid operator \( asa \).

\[
\frac{e \in P(firsttime Q)}{a \in eventually Q}
\]

\[
\quad \quad \quad \quad [a, e asa Q] \in P
\]

This can be read as follows: if \( e \) is the program satisfying \( P \) the first time that \( Q \) is true, and if it can be shown that \( Q \) is eventually true, then \( e asa Q \) is the program that always satisfies \( P \). The construction for \( Q \) will usually be discarded in Lucid expression derivations.

The second rule introduces \( fby \).

\[
\frac{a \in first P}{\quad e(X) \in next P}
\]

\[
\quad \quad \quad \quad X \text{ where } X = a fby e(X) \in P
\]

CTT uses a predicate \( Eq \) to denote equality of objects, as in \( Eq(N, A, B) \). The following rules relate Lucid expressions to equality.

\[
\frac{X = A fby B(X) \in N}{fst \in Eq(N, first X, A)}
\]

\[
\frac{X = A fby B(X) \in N}{nxt \in Eq(N, next X, B(X))}
\]

\[
\frac{fst \in Eq(N, first X, A) \quad nxt \in Eq(N, next X, B(X))}{X = A fby B(X) \in N}
\]

One other rule that is used in the following examples is for the Existential quantifier.

\[
\frac{P(t)[x \in A]}{t \in \exists x \in A \cdot P(x)}
\]

Here are two small examples that illustrate the use of the rules with Lucid. The first example is the derivation of a Lucid variable \( X \) with the property that \( next X > X \).

Firstly, we define
\[ A > B \equiv \exists z \in N \cdot Eq(N, A, B + 1 + z) \]

Then we want to prove that

\[ \exists X \in N \cdot next X > X \]

In general, constructive proofs can be used for program derivation. In this case, the proof is essentially a simple verification.

1. \( X = 0 \ fby \ X + 1 \) \hspace{1cm} \text{assume}
2. \( \text{nxt} \in Eq(N, \text{next} \ X, X + 1) \) \hspace{1cm} \text{rule 5}
3. \( 0 \in \exists Z \in N \cdot Eq(N, \text{next} \ X, X + 1 + Z) \) \hspace{1cm} \text{rule 6}
4. \( \text{next} \ X > X \) \hspace{1cm} \text{definition of} \ >
5. \( [X = 0 \ fby \ X + 1, X] \in \exists Z \cdot \text{next} \ X > X \) \hspace{1cm} \text{rule 6}

The second example illustrates the use of the induction rule. The requirement is to find a lucid expression \( E \) such that the value of \( E \) at time \( t \) is

\[ \sum_{i=1}^{t} i \]

Let \( P(t) = \{ x \mid x = \sum_{i=1}^{t} i \} \). As a simple lemma, we can show that if \( I = 1 \ fby \ I + 1 \) then \( I_k = k + 1 \). This lemma uses the conventional CTT induction axiom. Here is the derivation of \( E \).

1. \( 0 \in first P \) \hspace{1cm} \text{arithmetic}
2. Let time stand still at \( k \)
3. \( E \in P \) \hspace{1cm} \text{assume}
4. \( I = 1 \ fby \ I + 1 \in N \) \hspace{1cm} \text{assume}
5. \( I = k + 1 \) \hspace{1cm} \text{lemma}
6. \( E = \sum_{i=1}^{k} i \in P \) \hspace{1cm} \text{defn of} \ P \ \text{equality on 3}
7. \( \sum_{i=1}^{k+1} i \in next P \) \hspace{1cm} \text{defn of} \ P \n
8. \( k + 1 + \sum_{i=1}^{k} i \in next P \) \hspace{1cm} \text{defn of} \ \Sigma \n
9. \( I + E \in next P \) \hspace{1cm} \text{equality}
10. \( E \ where \ E = 0 \ fby \ I + E \in P \) 

and time may proceed \hspace{1cm} \text{by rule 2}

Hence the result is established, under the assumption that \( I = 1 \ fby \ I + 1 \in N \). The two-line Lucid program is extracted in the obvious way.
4 Conclusions

We have shown that CTT can be extended from intuitionistic predicate calculus to intuitionistic intensional predicate calculus. The complete extended system can be used to derive or verify Lucid programs. Although conventional CTT is retained, the induction axioms could be replaced by \( fby \) introduction as presented here. Since Lucid is explicitly intensional, the induction rule does not need the addition of a higher order operator like \( \text{rec} \). The Lucid rule is also more general than the corresponding CTT rule because it can be used for intensions of any type. In CTT it is necessary to introduce a new induction rule for each new type.

If systems such as Nuprl become the programming environments of the future, Lucid will still have its place. In fact, in cases where intensional reasoning is more natural than conventional reasoning, an extended system that allows the generation of Lucid programs will be easier to use than CTT.

References

