Lectures on: Classical Proofs as Programs

Robert L. Constable*
Department of Computer Science
Cornell University
Ithaca, NY 14853
rc@cs.cornell.edu

1 Introduction

Background

In the 1970's the idea of using constructive proofs of theorems as programs was advocated and studied theoretically [5, 23]. In the 1980's the computer science community started to experiment with this concept [4, 28], and today there are several prototype systems that enable us to write programs in this way [9, 21, 6, 28, 11]. So for instance, from a constructive proof of the Intermediate Value Theorem, a system like Nuprl [9] can extract a program which computes the intermediate value to any precision required [7].

I claim that we now know that this *proofs-as-programs* paradigm for programming really works for a certain class of *specifiable* problems whose solutions are *sequential functional* programs. Moreover, we know that this programming paradigm offers advantages over all others in cases where the explanation of the algorithm is very important and is being offered to people with a mathematical orientation. In fact informal mathematics over the centuries abounds with algorithms presented in just this way.

The method is noteworthy because it suggests ways of tackling certain problems basic to a mature *theory of programming*. For instance, the ordinary laws of a logic also provide those of a *programming logic* is built on it directly and simply. Furthermore the

*This research supported by NSF grant CCR86-16552 and ONR grant N00014-88K-0490.

technology of automating mathematical reasoning applies immediately to reasoning about programs. Perhaps most critically, the methodology of mathematical problem solving which has been so much studied is applicable *holus pollus* to programming. This includes knowing the foundations of the enterprize, knowing when and how to calculate both symbollically and numerically, and knowing how to present complex arguments in intelligible ways.

In this article, I will assume the statements made above and ask how we might improve this valuable programming paradigm. Certainly we can improve it by expanding its applicability to a wider class of specifiable problems, to a richer class of programs as solutions and to a larger scale of programming (say thinking of *systems as theories*). This paper is concerned with just one of these topics, namely allowing a richer class of solutions. Infact we will examine the idea that *classical proofs* can be regarded as programs as well as constructive ones. This is not a very intuitive idea, it is *counter-intuitive* in fact. It is made possible by expanding the programming constructs allowed in the underlying functional language.

The fact that classical proofs can be interpreted as programs in some cases is a surprizing discovery that did not arise by simply exploring the engineering issues systematically. It is the result of a scientific discoveries made by Griffin and Murthy [19, 25, 26, 10]. It has led the community to a line of research that was unknown before, a line which may have ties to the several other investigations which aim to gradually broaden the scope of the paradigm [17].

Overview of the paper

The paper begins with a discussion of type theory. This is the logical system in which the proofs-as-programs paradigm will be explored. We could have used a simpler logic such as higher-order number theory. But the amount of work involved in explaining that theory is not appreciably less than what we will do here, and it is one of my secondary goals to present the elements of constructive type theory. So the details of how the type theory is presented will be of independent interest. The next section discusses programming in type theory. We will see that the programming part looks a lot like a functional programming language with rich types such as Standard ML [20, 24]; already at this point we can appreciate the value of using type theory as opposed to other formalisms.

In section 4 we examine how logic is defined on top of the type theory. Here we invoke

the well-known *propositions-as-types* principle, and we define the sublogics of constructive and classical arithmetic (so-called Heyting and Peano arithmetics respectively). In section 5 we examine program derivation from specification. Then in section 6 we expand the programming language to include the control operators of Felliesen [12]. This is the programming basis for the extension of the logic in section 7 to interpret classical proofs computationally.

2 Core Type Theory

Informal basis

We are attempting to speak of *mathematical objects*. We think of them as mental constructs created by the process of imagination. A particular method of constructing objects corresponds to a class or *type*. Objects do not exist before hand in some Platonic world, instead they are created by carrying out constructions in accordance with a method. On this view the types are specified before the objects in them. Among the objects we consider are methods of construction themselves or algorithms.

Mathematical constructions can take place in the mind of a single individual, but we are concerned only with sharable experiences and their communication. (Perhaps there are some kinds of mathematical object that come into existence only *jointly* through communication among creating subjects.) Notation for objects is essential to communication; so notation is central to this account of mathematics.

My colleagues and I have noticed a systematic way to present a notation for objects that is adequate for the type theories we know. We can limit these forms without apparently limiting the algorithms for manipulating objects. The basic form of a expression in this theory is the *term*. Every term is of the form $op(\bar{v}_1.t_1, \ldots, \bar{v}_n.t_n)$. We say that *op* is the *operator name*, $\bar{v}_i$ is a list of *binding variables*, and t_i is the $i-th$ subterm which is exactly the *scope* of the binding variables the precede the dot separating them from the term.

2.1 A Simple Type Theory

Here is a small type theory that serves for the basic results needed in the rest of the paper. The basic syntactic concept is a *term*.

Definition 1 *The terms are divided into two groups, the* canonical *ones and the* non-canonical *ones. Here are all of the canonical terms:* $prod(A; x.B)$, $fun(A; x.B)$, $set(A; x.B)$, $union(A; B)$, $void$, N, $Type$, $pair(a; b)$, $\lambda(x.b)$, $inl(a)$, $inr(b)$, $any(b)$, 0, $succ(t)$ *where* A, B, a, b *are terms,* x *is a variable.*

Here are the noncanonical ones:

$$spread(p; u, v.t),\ ap(f; a),\ decide(d; u.t1; v.t2),\ ind(e; b; n, i.t).$$

The notions of *binding variable*, *bound variable*, *free variable*, *scope* and *closed term* are carried over to these terms from the lambda calculus in a natural way. For instance for the lambda term $\lambda(x.b)$ we say that x before the dot is a binding occurrence of x whose scope is b. This occurrence of x along with the dot forms a binding operator which binds all free occurrences of x in b. A variable is free in an expression if it does not occur in the scope of a binding operator. These concepts are defined precisely for the reflected concept of terms, denoted $Term$, to come later.

There are *computation rules* telling how to *reduce* the noncanonical terms; the canonical ones reduce to themselves. In these rules the notation $exp[a/x]$ denotes the term obtained from exp by substituting a for each occurrence of x in exp.

Definition 2 *To reduce* $spread(p; u, v.t)$, *first reduce* p. *If the result is* $pair(a; b)$, *then continue by reducing* $t[a/u, b/v]$.

To reduce $ap(f; a)$, *first reduce* f; *if the result is* $\lambda(x.b)$, *then continue by reducing* $b[a/x]$.

To reduce $decide(d; u.t1; v.t2)$, *first reduce* d; *if the result is* $inl(a)$, *then continue by reducing* $t1[a/u]$; *if the result is* $inr(b)$, *then continue by reducing* $t2[b/v]$.

To reduce $ind(e; b; n, i.t)$, *first reduce* e, *and if the result is* 0, *then continue by reducing* b. *If the result is* $succ(a)$, *then continue by reducing* $t[a/n, ind(a; b; n, i.t)/i]$.

Abbreviations

We sometimes write $ap(f; a)$ as $f(a)$ and $\lambda(x.b)$ as $\lambda x.b$. This is the "standard notation" for lambda terms [3], and we use it in any long examples or in standard ones. We write $f(< x, y >)$ sometimes as $f(x, y)$ and more generally $f(<< x, y > z, >)$ as $f(x, y, z)$. We also write $n + 1$ for $succ(n)$.

2.2 Typing rules

These terms are the basis of a simple type theory. A type will be defined when we give it a canonical name and say which canonical terms are elements of it. One type is determined by the term *void*. It is a type with no members. We generally speak of the term which is the canonical name of a type as a type, so we say *void* is a type.

Another base type is N which represents the natural numbers. The canonical members are 0 and $succ(n)$ for n an element of N.

Another type is determined by the name $Type$.(This is the simple name we are using for the first universe, $U1$.) To define its canonical members we proceed inductively, and we introduce the auxiliary concept of a family of types over a type A. If B is a term and A is a type, and if for every element a of A the term $B[a/x]$ is a type, then B is a family of types indexed by A (using index variable x).

If A and C are types and if B is a type indexed by A (with index variable x) then $prod(A; x.B)$, $fun(A; x.B)$ and $union(A; C)$ and $set(A; x.B)$ are types. If x does not occur in B, then we omitt the $x.$ binding operator.

In general to specify types, we must define their canonical elements. The canonical elements of the types $union(A; C)$ are those terms $inl(a)$ and $inr(c)$ where a is a term of type A and c is one of type C; neither a nor c need be canonical. So we must also to define what it means generally for a term to be of some type T. We do this below.

The canonical elements of $prod(A, x.B)$ are the terms $pair(a; b)$ where a is of type A and b is of type $B[a/x]$. We know that these $B[a/x]$ are types since $prod(A; x.B)$ is.

The canonical elements of $fun(A; x.B)$ are the terms $\lambda(x.b)$ such that for each a of type A, $b[a/x]$ is of type A.

The canonical elements of $set(A; x.B)$ are those of A.

A term with no free variable (i.e. a *closed term*) is of type A if it reduces to a canonical term of type A. Thus if $t[a/x, b/y]$ reduces to a canonical term type T, then $spread(pair(a; b); x, y.t)$ is of type T.

A term with free variables can be contingently of type A depending on assumptions about the types of the free variables. To define this, let $x_1, \ldots, x_n$ be the free variables of t assume A_1, is a type, and say that A_2 is a type, possibly depending on A_1, if for each a_1 of type A_1, $A_2[a/x]$ is a type. Likewise for A_n depending on $A_1, \ldots, A_{n-1}$, and T depending on $x_1, \ldots, x_n$. Now t is of type T if for each a_1 of type A_1, A_2

of type $A_2[a_1/x], \ldots, a_n$ of type $A_n[a_1/x_1, \ldots, a_{nil}/x_{nil}]$, $t[a_1/x_1, \ldots, a_n/x_n]$ is of type $T[a_1/x_1, \ldots, a_n/x_n]$. For example, $inr(x)$ is of type $union(A; B)$ if x is assumed to be of type A.

2.3 Semantics

Just as in the lectures we adopted a semantic approach to type theory, so at this point we do not give an enumeration of the rules which embody the ideas just described concerning reduction of terms (which become *computation rules*), the formation of types (which become *formation rules*) and the conditions for membership in types (which for the canonical terms become *introduction rules* and in the case of the noncanonical terms become *elimination rules*). A complete set of rules can be found in [9].

In the lectures at Marktoberdorf the semantic methods of Stuart Allen [1] were used as a rigorous basis for this semantic approach. But due to space limitations here we merely cite Allen's work and proceed informally since there is no need to use the rigorous approach for the results stated in this article.

3 Programming in Type Theory

3.1 Conditionals and while loops

Most programming languages use some notion of Boolean values and conditional expressions over them to express choices. Instead of a special type of Booleans we can use N. The conditional

$$\textbf{if } bexp \textbf{ then } a \textbf{ else } b \textbf{ fi}$$

can be expressed as

$$ind(bexp; b; u, i.a).$$

so that $bexp \neq 0$ corresponds to a true boolean and $bexp = 0$ to a false boolean. (The bound variables u, i are completely meaningless in this use of ind, but we require that they *not occur* in a.)

The natural programming language determined by these terms is *functional*. However we can understand the concept of a state in the style of denotational semantics, by manipulating n-tuples of values as if they were the values stored in a state. What is

missing is the procedural interpretation that values are up-dated in place. (Ways of dealing with this in Nuprl are discussed in [2]). So if the state contains values x, y for example, then the operation $y := y + 1$ is represented by the function $f(< x, y >) = < x, y + 1 >$. Notice that $y + 1$ is an abbreviation for $succ(y)$.

The terms introduced in Definition 1 provide all the expressive power of the untyped lambda calculus. We are especially interested in the Y combinator or the fixed point operator. For readability we use the term fix which is defined to be $\lambda f.(\lambda x.\ f.(xx))(\lambda x.\ f(xx)))$

If F is an expression involving values of f, then a general recursive function definition of the form $f(x) = F$ can be expressed as a fixed point of the functional $\lambda f.F$. The term $fix(\lambda f.F)$ computes to $F[fix(\lambda f.F)/f]$.

With this notation we can write the conventional while-loop, **while** *bexp* **do** *body* **od** over a state s as $fix(\lambda w.\lambda s.$ **if** $bexp(s)$ **then** s **else** $w(body(s))$**fi**.

$$\lambda s.\ while(s; bexp; body) \ = \ fix(\lambda w.\ \lambda s.\ \textbf{if}\ bexp(s)\ \textbf{then}\ s\ \textbf{else}\ w(body(s))\textbf{fi})$$

Here is how we express an iterative program to find the integer square root r of n.

$$\lambda x.\lambda r.\ while(r; \lambda y.(y^2 \leq x); \lambda y.y + 1)(n)(0)$$

This is essentially the program $r := 0$; **while**$(r + 1)^2 \leq x$ **do** $r := r + 1$ **od** which is one way to compute the root. Next we look at another program for this.

3.2 Primitive recursion

In general if $n+1$ denotes the successor of n, $succ(n)$ then primitive recursive functions are defined as

$f(0, y) = g(y)$
$f(n + 1, y) = h(n, f(n, y), y)$.

It is more convenient in a higher order setting to write this as

$f(0) = \lambda y.g(y)$
$f(n + 1) = \lambda y.h(n, f(n)(y), y)$.

A further simplification is to write both clauses in a single "linear" expression and recognize that the recursive call is always to $f(n)$ so that expression can be written

as a single letter, say r below. Then all of the components are summarized in the expression

$$ind(n; \lambda y.g(y);\ u, r.\ \lambda y.h(u, r(y), y)).$$

(We need to distinguish the specific argument, n, from the symbolic argument u as in computing $ind(17; - - - : u, r. - -)$.)

The reduction relation is obvious.

$ind(0;\ \lambda y.g(y); \_\_)ev\mapsto \lambda y.g(y)$
$ind(n+1; \_\_; u, r.\ \lambda y.h(u, r(y), y))ev\mapsto$
$\lambda y.\ h(n, ind(n; \_\_; \_\_)(y), y).$

This expression can be generalized to

$$ind(n+1; b;\ u, r.h)$$

and the reduction rule on successors is

$$ind(n+1; b;\ u, r.h)ev\mapsto h[n/u, ind(n; \_\_; \_\_)/r].$$

A primitive recursive function for root is

$root(0) = 0$
$root(n+1) =$ **if** $n < (root(n)+1)^2$ **then** $root(n)$ **else** $root(n)+1$ **fi**

which is written linearly as

$$ind(n; 0; x, r.\ ind(test(x, r); r;\ u, v.\ r+1))$$

where $test(x, r)$ produces 0 if $x < (r+1)^2$ and 1 otherwise. We assume test is a primitive for this example.

4 Logic in Type Theory

4.1 Propositions-as-Types Principle

Propositions are the material of logic. They can be defined in type theory. One precise mechanism for doing this was pieced together by Curry, Howard, and deBruijn (and perhaps others as well as, say Lauchli). It is sometimes called the *Curry-Howard isomorphism* or the *propositions-as-types principle* and is related to Brouwer's notion

that logic is not a subject prior to mathematics but is part of it. [1] Although these ideas were formulated for constructive mathematics, we now know that they are interesting for classical mathematics as well [10, 26, 16, 8], and indeed this paper is concerned with exploring this aspect.

The basic idea is this. For a large class of propositions P there corresponds a type, say $[P]$, which is inhabited if and only if P is true. Thus,

Correspondence 1 *P is true* iff *$p \epsilon [P]$ for some p.*

We can think of the inhabiting element, p, as *evidence* for the truth of P. One interpretation of this notion of evidence will be developed below as we define the types corresponding to atomic arithmetic propositions.

We begin exploring the correspondence by building it up for the atomic propositions of number theory, starting with those parametized by a variable x which are true if and only if x is zero. The types corresponding to this proposition we denote by $Zero(x)$. We want a type expression in x which is inhabited if and only if x is zero. Notice that

$$ind(x; N;\ u, i.\ void).$$

is a noncanonical term which denotes a type for each value of x.

So $ind(0; N;\ u, i.void) ev\mapsto N$
$ind(1; N;\ u, i.void) ev\mapsto void$
$ind(2; N;\ u, i.void) ev\mapsto void.$

We see the *correspondence*

$$0 = 0 \textit{ iff } n \epsilon N \text{ for some } n$$
$$1 = 0 \textit{ iff } k \epsilon \text{ void for some } k.$$

Since there is some element of N, $0 = 0$ corresponds to an inhabited type. Since there is no element of *void*, $1 = 0$ corresponds to an empty type.

It might be more elegant to have some unit type, say *unit*, inhabited by some token like $*$. Then

$$ind(x; unit;\ u, i.void)$$

would be a simpler type corresponding to $x = 0$. We would have

[1] The principle can be seen as a precise formulation of one of Brouwer's tenents of *Intuitionism.*

$$(0 = 0) \ \textit{iff} \ (* \ \epsilon \ \text{unit}).$$

But there is no other need for this extra type. So we settle for using N as our standard inhabited type. This type theory builds logic on a principle that *identifies* propositions and types.

Principle The propositions $x = 0$ are defined by the type $ind(x; N;\ u, i.void)$. We abbreviate these as $Zero(x)$.

So for each natural number x, $Zero(x)$ is the proposition that x is zero. This is the *definition* of the proposition. We say that $Zero(x)$ is a *predicate* in the variable x, and $\lambda(x.Zero(x))$ is a *propositional function.*

Let us next define the propositions that x equals y in the integers. First we need the predecessor function $x \dot{-} 1 = 0$. The function is

$$\lambda(x.ind(x; 0;\ u, i.u)).$$

Call this function *monus*. Then $monus(0) = 0,\ monus(1) = 0,\ monus(2) = 1$, and generally $monus(succ(x)) = x$.

Now define $Equal(x, y)$ based on the primitive recursion

$$\begin{array}{lll} equal(0) & = & \lambda y.\ Zero(y) \\ equal(x+1) = & & \lambda y.\ \textbf{if}\ Zero(y)\ \textbf{then}\ \text{void} \\ & & \quad \textbf{else}\ equal(x)(monus(y))\textbf{fi} \end{array}$$

The official definition of $Equal(x, y)$ is

$$ind(x; \lambda(z.\ Zero(z); u, eq.\ \lambda(z.eq(monus(z)))(y).$$

We will be able to prove these simple properties of the defined equality.

Fact

1. Equal(x,0) *iff* Zero(x)
2. Equal(x,x)
3. Equal(x,y) *iff* Equal(y,x)
4. Equal(x,y) and Equal(y,z) *implies* Equal(x,z)

5. Equal(x,y) *iff* Equal(succ(x), succ(y))

6. For no x is Zero(succ(x)) true.

These are enough properties to give us Peano's axioms for equality.

In Nuprl and Martin-Löf type theory certain atomic propositions are built in. For example $x = y$ in N is a primitive predicate and $0 = 0$ on N is a primitive proposition. Its inhabitant is called *axiom*. Likewise $x < y$ is a primitive in Nuprl, and its true instances are inhabited by *axiom* as well. Sometimes in the text we will use *axiom* in this way when we do not care to look into the structure of these simple propositions.

4.2 Compound Propositions

The proceeding discussion might make it appear that the correspondence between propositions and types is a highly artificial matter depending on properties of the natural numbers. In fact the correspondence is far more general and profound. It was discovered first by Curry in the realm of the propositional calculus. We will look at this next. Notice first the situation for the false proposition, say *false*, or as it is sometimes written nowadays, bottom, $\bot$. The right type corresponding to $\bot$ is *void*. False is not true, and void in not inhabited.

What Curry noticed is a correspondence between the function space $fun(A; B)$ and implication $A \Rightarrow B$. If we write $fun(A; B)$ as is customary, $A \to B$, then we immediately see a *typographical similarity*. This may be suggestive but it is *not* the idea. To see the idea, notice the following application of a function and the rule *modus ponens*.

Typing	Logic
$\dfrac{f : A \to B \quad a : A}{f(a) : B}$	$\dfrac{A \Rightarrow B \quad A}{B}$

In each case the form of the rule is that if the relations above the line hold, then so does the one below. Call the typing rule AP and the logical rule MP (for *modus ponens*).

Let us now compare the rules for typing a function $\lambda(x.b)$ with those for proving an implication (in the sequent calculus). To state these we need the notion of an *environment* E (in the case of types) or of a hypothesis *list* H (in the case of logic).

An environment is a list of variable declarations $x_1 : T_1, \ldots, x_n : T_n$ saying that variable x_i has type T_i. An hypothesis list has the form $A_1, \ldots, A_k$ where A_i are propositions. Now compare the rules

Typing	Logic
$\dfrac{E, x : A \vdash b : B}{E \vdash \lambda(x.b) : A \to B}$	$\dfrac{H, A \vdash B}{H \vdash A \Rightarrow B}$

In these rules the expression on top of the line relates the *assumptions*, written to the left of the turnstyle, $\vdash$, and the *conclusion*, written to the right. These relations are called *sequents*.

Curry had noticed that the types of the combinators I, K, and S were valid formulas in the Intuitionistic propositional calculus. That is

$$I : A \to A$$
$$K : A \to (B \to A)$$
$$S : (A \to (B \to C)) \to (A \to B) \to (A \to C).$$

If we write these combinators as lambda terms and type them, we will see that the typing derivation corresponds exactly to a proof of the corresponding propositions. Let us do that for S.

Type: $\lambda f.\lambda g.\lambda x.\ f(x)(g(x)) : (A \to (B \to C)) \to (A \to B) \to (A \to C)$.

Check: Given $f : A \to (B \to C), g : (A \to B),\ x : A$

Type: $f(x)(g(x))$

$f(x) : B \to C$	by $AP\ f, x$
$g(x) : B$	by $AP\ g, x$
$f(x)(g(x)) : C$	$AP\ f(x), g(x)$

End

Now compare this type checking to the following proof.

Theorem 1 $(A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$.

Proof: Assume

1. $A \Rightarrow (B \Rightarrow C)$
2. $(A \Rightarrow B)$
3. A

Show C

4. $(B \Rightarrow C)$ by MP 1,3
5. B by MP 2,3
6. C by MP 4,5

Qed.

This Curry-Howard correspondence is seen to involve not only propositions and types, but rules of inference and typing rules. Also proofs of propositions correspond to derivations of the relationship between types and their elements. But does the correspondence make sense? How should we think if the elements of types when these types represent propositions?

In the case of implication $A \Rightarrow B$ and the function space $A \rightarrow B$, the correspondence makes good sense. According to the Intuitionistic interpretation of $A \Rightarrow B$, the proposition is true exactly when there is a method that takes evidence for A to evidence for B. In general a proposition P is Intuitionistically true exactly when we can find evidence for the truth of the proposition.

We will accept this interpretation of implication and say that for example $\lambda(x.x)$ is a proof expression which denotes evidence for the truth of the proposition $A \Rightarrow A$. Let us see how to extend this interpretation to other connectives. A more detailed account if these ideas can be found in my 1989 Marktoberdorf lecture notes [8] and in several books [15, 28, 18].

Notice that & is like product in Boolean algebra, so let us compare A & B with $A \times B$. (Recall that $A \times B$ abbreviates $prod(A : B)$).

Logic	Types
$\dfrac{A \quad B}{B \ \& \ B}$	$\dfrac{a : A \quad b : B}{< a, b >: A \times B}$

Notice that $A \Rightarrow B \Rightarrow A \ \& \ B$ is true $\lambda(x.\ \lambda(y. < x, y >)) : A \rightarrow B \rightarrow A \times B$.

Logic	Types
$\dfrac{H, A, B \vdash G}{H, A \ \& \ B \vdash G}$	$\dfrac{E, x : A, y : B \vdash g : G}{E, p : A \times B \vdash spread(p; x, y.g) : G}$

Proving $A \ \& \ B \Rightarrow A$ is like building $\lambda p.spread(p; x, y.x)$ in the type $A \times B \rightarrow A$.

The Boolean analogy also suggests that $A \vee B$ is like $union(A; B)$. Let's see how it works. We abbreviate $union(A; B)$ as $A + B$.

Logic	Types
$\dfrac{A}{A \vee B} \ \dfrac{B}{A \vee B}$	$\dfrac{a : A}{inl(a) : A + B} \ \dfrac{b : B}{inr(b) : A + B}$
$\dfrac{H, A \vdash G \ \ H, B \vdash G}{H, A \vee B \vdash G}$	$\dfrac{E, x : A \vdash g : G \ E, y : B \vdash g_2 : G}{E, z : A + B \ \vdash \ decide(z : x.g; y.g_2) : G}$

Proving $A \Rightarrow A \vee B$ is like building $\lambda(x.inl(x))$ in $A \rightarrow A + B$.

If we think of $\exists x : A.B$ as a generalized $+$, it suggests these rules:

Logic	Types
$\dfrac{a \epsilon A \ \ B[a/x]}{\exists x : A.B}$	$\dfrac{a : A \ \ b : B[a/x]}{< a, b >: prod(A; x.B)}$

Proving $\forall x : A.(B(x) \Rightarrow \exists y : A.B(y))$ is like building $\lambda(x.\ \lambda(y.\ < x, y >))$ in $fun(A; x.fun(B(x); prod(A; y.B(y))))$

Logic	Types
$\dfrac{H, x : A, B(x) \vdash G}{H, \exists x : A.B \vdash G}$	$\dfrac{E, x : A, y : B(x) \vdash g : G}{E, z : prod(A; x.B(x)) \vdash \ spread(z : x, y.g) : G}$

If we think of $\forall x : A.B$ constructively then

Logic	Types
$\dfrac{H, x : A \vdash B(x)}{H \vdash \forall x : A.B}$	$\dfrac{E, x : A \vdash b : B(x)}{E \vdash \lambda(x.b) : fun(A; x.B(x))}$
$\dfrac{\forall x : A.B \ a \epsilon A}{B[a/x]}$	$\dfrac{f : fun(A; x.B) \ a : A}{f(a) : B[a/x]}$

4.3 Defining Classical Logic

The embedding of logic into type theory given in the previous section applies only to constructive or Intuitionistic logic, but it can be extended to classical logic using the methods of Kolmogorov and Gödel. Here is Gödel's definition of the classical connectives.

Definition 3 *If P and Q are formulas, and A is an atomic formula, then*

$$
\begin{aligned}
(A)^G &= A \\
(P \ \& \ Q)^G &= P^G \ \& \ Q^G \\
(P \ \Rightarrow \ Q)^G &= P^G \ \Rightarrow \ Q^G \\
(\forall x : N.P)^G &= \forall x : N.P^G \\
(P \vee \ G)^G &= \neg(\neg P^G \ \& \ \neg Q^G) \\
(\exists x : N.P)^G &= \neg\forall x : N.\neg P^G
\end{aligned}
$$

If $H = (P_1, \ldots, P_n)$ then $H^G = (P_1^G, \ldots, P_n^G)$.

Gödel proved the following theorem which gives a translation of Peano arithmetic (PA) into Intuitionistic arithmetic (called Heyting arithmetic, HA).

Theorem 2 *(Gödel) If formula F is provable from hypotheses H in Peano arithmetic, then F^G is provable form H^G in Heyting arithmetic.*

This theorem shows that we can express all of the concepts of Peano arithmetic in our type theory. We can be explicit about using the classical connectives if we introduce the following definitions.

Definition 4 $A \bigcirc\!\!\!\!\vee B == \neg(\neg A \ \& \ \neg B)$.
$\bigwedge x : N.B == \neg\forall x : N.\neg B$

We can prove the classical axioms such as

$$P \bigcirc\!\!\!\!\vee \neg P$$

But notice that proving a proposition like

$$\forall x : N.prime(x) \bigcirc\!\!\!\!\vee \neg prime(x)$$

does not give the same information as proving

$$\forall x : N.prime(x) \vee \neg prime(x)$$

In the later case the proof provides a decision procedure for primeality; in the former case the proof provides evidence for a contradiction (a function from $\neg prime(x)$ & $\neg\neg prime(x)$ to *void*). Consider also the difference between

$$\exists x : N.(0 < x) \text{ and } \bigwedge x : N.(0 < x).$$

Proving the first fact provides explicitly a witness for x whereas proving the second might not give the witness explicitly. (We will see in section 7 how to recover the witness from proofs of the second formula.)

5 Program Derivation

As an aside before getting to the main topic, notice that the decide operator provides a more flexible way to define while loops. Given an expression $Bexp(s)$ for which we know $Bexp(s) \vee \neg Bexp(s)$, say this is decided by $bexp(s)$, then we could define a while loop as

$$\lambda(s.\ while(s; bexp; body))=$$
$$fix(\lambda w.\lambda s.\ decide(bexp(s); t.w(body(s)); f.s)).$$

We now have sufficient expressiveness to precisely state the integer root problem. Given any n of type N we seek a number r such that

$$r^2 \leq n < (r+1)^2.$$

That is, we are looking for an additional inhabitant of

Root Theorem: $\forall n : N.\ \exists r : N.\ r^2 \leq n < (r+1)^2$.

We can prove this theorem in several ways. One of them corresponds to building the while loop and another to building the primitive recursive program. We sketch both of these. The first one requires that we develop in type theory a notion of iteration and iterative construction of proofs.

One way to prove any proposition $R(n, r)$ that requires finding r is to build up r interatively from some initial value; say in the example above starting r at 0 and increasing it by one until $R(n, r)$ holds.

Sometimes the proposition $R(n, r)$ can be factored into some decidable part, say $Bexp$, and some part, called the *invariant*, say $I(n, r)$ such that:

$\neg Bexp(n, r)$ & $I(n, r) \Rightarrow R(n, r)$ and
$I(n, r) \Rightarrow Bexp(n, r) \Rightarrow I(b(n, r))$
where $b(n, r)$ produces a pair of values (say n', r').

We might need to assume something about the initial conditions on n and r expressed as a *precondition*, say $Pre(n, r)$. We expect to have

$$Pre(n, r) \Rightarrow I(n, r).$$

The process of producing new values from n and r, say it is the function whose value is $b(n, r)$, must be known to finally achieve the desired end. This can be guaranteed by *termination conditions*. For example, if we know

$Pre(n, r) \Rightarrow \exists t : N.\ T(t, n, r)$,
$Bexp(n, r) \Rightarrow \neg\ T(0, n, r)$,
$I(n, r)$ & $Bexp(n, r)$ & $T(t, n, r) \Rightarrow T(t - 1, b(n, r))$,

then we know that eventually repeated application of b will result in $Bexp(b(n, r))$ becoming false.

The while induction theorem will be formulated in terms of "states." The simplest state is just an n-tuple, say $s \in A_1 \times \cdots \times A_n$. The goal formula is written $G(s)$, the precondition is $Pre(s)$, the invariant is $I(s)$, the termination condition is $T(n, s)$, and the body is $body(s)$.

Let us define the *invariant condition* as:

$$\begin{aligned} \forall s.State.\quad & ((Pre(s) \Rightarrow I(s))\ \& \\ & (I(s) \& \neg Bexp(s) \Rightarrow G(s))\ \& \\ & I(s) \Rightarrow Bexp(s) \Rightarrow I(body(s))) \end{aligned}$$

and the *termination condition* as:

$$\begin{aligned} \forall s : State.\quad & ((Pre(s) \Rightarrow \exists n : N.\ T(n.s))\ \& \\ & (Bexp(s) \Rightarrow \neg T(0, s))\ \& \\ & \forall n : N.\ (I(s)\ \&\ T(n, s)\ \&\ Bexp(s) \Rightarrow \\ & \quad T(n - 1, body(s))). \end{aligned}$$

We need to mention pieces of the invariant condition, so let $initial(s)$ be $Pre(s) \Rightarrow I(s)$ and let $post(s)$ be $I(s)\&\neg Bexp(s) \Rightarrow G(s)$ and let proof-body be the function $\lambda(s.\ I(s) \Rightarrow Bexp(s) \Rightarrow I(body(s)))$. These can be defined from the invariant condition.

We basically want to prove that under the above two conditions

$$\forall s : State.\ (Pre(s) \Rightarrow \exists s' : State.\ G(s')).$$

The essence of the proof is that given s and a proof, pre of $Pre(s)$, s' is given by

$$while(s; bexp; body).$$

where $bexp(s)\ \epsilon\ Bexp(s) \vee \neg\ Bexp(s)$.

In order to iteratively build up the proof term for $G(s')$ we define an iteration form on proofs called whilep

$whilep(s; i; bexp; b; p) =$
if $bexp(s)$ **then** i **else** $whilep(b(s); p(i)(bexp(s)); b, p)$ **fi**.

This can be defined straight forwardly with the fix operator. We could have incorporated while and whilep into one form if we included proof information in the state.

Here is the theorem.

While-induction:

For Pre, I, $Bexp$, $bexp$,$body$, T, G, as above,
if the invariant condition holds and the termination condition holds,
then $\forall s : State.\ (Pre(s) \Rightarrow \exists s' : State.\ G(s'))$ is inhabited by
$\lambda(inv.\ \lambda(term.\ \lambda(s.\ \lambda(pre.\ pair(while(s; bexp; body);$
$whilep(s; initial(s)(pre); bexp; body;\ proof\text{–}body))))))$

We can also derive the primitive recursive program for root from the following inductive proof that a root exists.

$\forall x : N.\ \exists r : N.\ Root(r, x)$
Pf : Let x be an arbitrary element of N, proceed induction on x.

Base case: $\exists r : N.\ Root(r, 0)$ by introducing
0 for r; using arithmetic to finish

Induction case:
Assume h : $\exists r : N.\ Root(r, x - 1)$.
Show $\exists r : N.\ Root(r, x)$.
Choose r_o where $Root(r_o, x - 1)$ and note that
by arithmetic reasoning we have
$(r_o + 1)^2 \leq x$ or $(r_o + 1)^2 > x$.
In case $(r_o + 1)^2 > x$ take $r = r_o$, and notice $Root(r_o, x)$.

In case $(r_o + 1)^2 \leq x$ take $r = r_o + 1$,
and show $x < ((r_o + 1) + 1)^2$ by arithmetic
from $x - 1 < (r_o + 1)^2$; so $Root(r_o, x)$.

Qed

The construction in the proof is this:

$$\begin{array}{l} \lambda x.\ ind(x;\ <0, arith>; u, h. \\ \quad spread(h; r_o, order. \\ \quad decide(test(u, r_o); \quad \text{right.} <r_o, arith(order)> \\ \qquad\qquad\qquad\qquad\quad \text{left.} <\ r_o + 1, arith(order)>))) \end{array}$$

where $test(u, r_o)$ produces an element of the type $(r_o + 1)^2 \leq x$ or $(r_o + 1)^2 > x$ and the term $arith(order)$ provides a proof that the root value satisfies the specification. It is not a simple term, but we are not concerned with this part of the proof so we leave the term undefined.

Notice that this is essentially the primitive recursive program of section 3.2

6 Control Operators

6.1 Evaluation

Purely functional programming is not as efficient as we would like. For example, suppose we are multiplying an array of numbers. A typical functional program for this, expressed iteratively (as defined in section 3) is:

$$p := 1; i := 1; \textbf{ while } i \leq n \textbf{ do } p := p * a[i];\ i := i + 1 \textbf{ od}.$$

where $a : \{x : N | 1 \leq x \leq n\} \rightarrow N$ is a definition of an array. Given a tail recursive evaluation strategy, this evaluates as efficiently as an imperative program on a state. But if some of the array elements are zero, then we would like to stop the computation. A first try might be the loop

$$\textbf{while } i \leq n \textbf{ do if } a[i] = 0 \textbf{ then } p := 0 \textbf{ else } p := p * a[i] \textbf{ fi } i := i + 1 \textbf{ od}.$$

But this does not actually save steps unless we can stop the execution early. In an imperative language we could write the loop as

$$\textbf{while } i \leq n \textbf{ do if} a[i] = 0 \textbf{ then } (p := 0;\ \textbf{exit}) \textbf{ else } p := p * a[i] \textbf{fi}; i = i + 1;\ \textbf{od}.$$

The **exit** statement stops evaluation of the loop. The exit statement is a *control statement*, like a *goto*. In the functional setting the role of such statements is taken over by *control operators*. These are operations which appear as functions in expressions, but then evaluation involves a change in the normal order of evaluation. These are discussed in [25, 12, 15].

Here is a version of the program with a control operator:

$$\begin{aligned}\lambda(a.\ \lambda(n.\ ap(\lambda(s.\mathcal{C}(\lambda(k.\ &\textbf{while}(i \leq n)\textbf{do}\\ &\text{if } a[i] = 0 \text{ then } ap(k; < 0, i >)\\ &\textbf{else } < p, i >:=< p * a[i], i + 1 >\ \textbf{fi od})); < 1, 1 >)))\end{aligned}$$

We use $< p, i >:=< p * a[i], i + 1 >$ to indicate that the pair s is being modified by a function, $B(s)$, which increments the second element of the pair and multiplies the first by $a[i]$. The initial value of the pair s is $< 1, 1 >$. The test $(i \leq n)$ is really a test on s, checking that its second element is less equal n. When we translate the

while into its definition in terms of fix, the result is a functional program with the additional operator $\mathcal{C}$.

To specify the evaluation of a control operator, we can use reduction rules as in Definition 2, but now it is necessary to be precise about the context in which a reduction takes place because the control operator $\mathcal{C}$ can change that context.

In general a fixed evaluation strategy, such as lazy evaluation, can be presented in two parts. The first is to locate the redex to be contracted (the subterm to be rewritten). The second is to carry out the reduction of redex to its contractum. The first part can be specified precisely by noticing that it partitions any term into the redex and the rest of the term, or its context. For example, given the term

$$ap(ap(\lambda(x.\ \lambda(y.\ y + x)); 2); 3),$$

and given a lazy evaluation strategy, the term can be written $E[ap(\lambda(x.\lambda(y.\ x+2)); 2)]$ where $E[\ ]$ is called a *context*. It can be thought of as a term with a hole in it. For the above example $E[\ ]$ is $ap(\_; 3)$.

Control operators can manipulate the context. To define $\mathcal{C}$ it is convenient to use an operator called *abort* (although abort can be defined from $\mathcal{C}$ see [26, 19]).

Reduction Equation 1 $E[abort(m)]ev\!\!\rightarrow m$.

So an abort operation executes by throwing away its surrounding context and returning it argument as a value.

Now the evaluation rule for control can be defined.

Reduction Equation 2 $E[C(m)]ev\!\!\rightarrow ap(m;\ \lambda(x.\ abort(E[x])))$.

This operator packages up the surrounding context and saves it for future use. The argument to $\mathcal{C}$ must be a function.

We can now use this rule to evaluate the program for multiplying the elements of an array. Suppose a specific array a and a specific value n is provided, then the outer level of the program is essentially

$$\mathcal{C}(\lambda(k.\ \textbf{while}\ bexp1(s)\ \text{do}\ \textbf{if}\ bexp2(s)\ \textbf{then}\ k(<0, i>)\\ \textbf{else}\ B(s)\ \textbf{fi}\ \textbf{od})).$$

where $bexp1$, $bexp2$, and B are the obvious operations on s. The context of this control operator is the outer context. That is, if we write this as the expression $E[\mathrm{C}(m)]$ then $E[x]$ is just x.

So the evaluation of $\mathcal{C}$ takes us to

$$ap(\lambda(k.\ \textbf{while\_\_od});\ \lambda(x.\ abort(x)))$$

which reduces to

$$\begin{array}{c}\textbf{while}\ bexp1(s)\ \text{do}\quad \textbf{if}\ bexp2(s)\ \textbf{then}\ ap(\lambda(x.\ abort(x)); < 0, i >)\\ \textbf{else}\ B(s)\ \textbf{od}.\end{array}$$

This causes the while loop to begin executing. If at some point $bexp2(s')$ becomes true for some s', then there will be a context F at which the reduction is

$$F[ap(\lambda(x.\ abort(x));\ < 0, i >)].$$

When this reduces the result is just $< 0, i >$.

6.2 Typing

Let us now try to assign a type to the control operator. Suppose that the context $E[\ ]$ defines a value of type B and that the hole, $[\ ]$, accepts values of type A. So $\lambda(x.E[x])$ has type $A \to B$.

Recall that $E[\mathcal{C}(m)]$ reduces to $ap(m; \lambda(x.abort(E[x])))$. Since $E[\mathcal{C}(m)]$ has type B, we know that m must return values of type B. Moreover the input to m is of type $A \to B$. Thus we see that m has the type $(A \to B) \to B$. Since the hole has type A; this means that $\mathcal{C}$ has the type

$$\mathcal{C} : ((A \to B) \to B) \to A.$$

It does not make sense to think of $\mathcal{C}$ as the inhabitant of the propositions $((A \Rightarrow B) \Rightarrow A)$ since this is not logically true (we could prove any A with this). But it is a sensible inhabitant of the classically valid proposition

$$\begin{array}{l}((A \to \bot) \to \bot) \to A, \text{ that is}\\ ((A \to void) \to void) \to A\end{array}$$

This interesting observation was made by Griffin [19]. But the computational value is unclear since it seems to apply only to values of type *void*, of which there are none!

Chet Murthy [26] observed that Harvey Friedman's "trick" [14] for getting computational meaning from classical proofs is just what is needed to make computational sense out of the typing of $\mathcal{C}$. Friedman's result is that in order to prove certain kinds of statements, S, namely of the form $\exists n\!:\!N.\ R$ where R is decidable (i.e., a $\sum_1^0$ formula), it suffices to prove $(S \to \mathit{void}) \to \mathit{void}$. In computing terms, this means that for some types S, in order to compute values of type S, it suffices to write functions that return values in *void*. This is the significance of Friedman's "top level trick."

Friedman's result suggests more since he shows how to use a classical law of logic to produce constructive results. The import of this is that operators like $\mathcal{C}$ make sense as part of the computation of values. In the case of proofs, it is a matter of investigation to know just which constructive results can be proved by classical means; so in the case of programs it is a matter of investigation to learn which computations using $\mathcal{C}$ produce the results we expect.

Friedman's result

To state Friedman's result we need to talk about the form of proofs and rules. For this we use *sequent calculus* in which proofs and rules are written in a top-down fashion, as in tableau. We write sequents as

$$A_1, \ldots, A_n >> G$$

where $A_i,\ G$ are formulas. The A_i are *hypotheses* and G is the *goal* or conclusion. Sometimes we abbreviate these as

$$H >> G$$

where H denotes a list of formulas as above.

A typical rule is *modus ponens* or implication elimination. We write it as

$$\begin{array}{lll} H, (A \Rightarrow B), H' & >> G & \text{by elim on } A \Rightarrow B \\ H, (A \Rightarrow B), H', B & >> G & \\ H, (A \Rightarrow B), H' & >> A & \end{array}$$

This says that to prove G from hypotheses list $H, (A \Rightarrow B), H'$ (where H, H' can be formula lists), it suffices prove the subgoal that G follows from the additional hypotheses.

We need to call attention to the rule for treating a false hypothesis. For ease of writing, let $\perp$ denote the proposition *false* (the type *void* is the same). Now the rule $\perp$-elimination is just

$$H, \perp, H' >> G.$$

That is, from $\perp$ we conclude *anything*.

The negation of a formula B is defined as $\neg B == (B \Rightarrow \perp)$, and the A-negation of B, $\neg_A B$ is $(B \Rightarrow A)$. The rule of double-negation elimination is

$$\begin{array}{l} H >> B \text{ by } \neg\neg elim \\ \quad H >> \neg\neg B. \end{array}$$

The system of Peano Arithmetic, PA, is obtained from Heyting Arithmetic, HA (essentially the logic of section(4)) by adding the above double-negation elimination rule. We sometimes write

$$H >>_{HA} G \quad \text{or} \quad H >>_{HA} g : G$$

$$H >>_{PA} G \quad \text{or} \quad H >>_{PA} g : G$$

to mean that G is provable from hypotheses H by the rules of HA or PA respectively, and if g is mentioned, that g is the evidence for G or the *proof expression* inhabiting the proposition viewed as a type.

Theorem 3 *Given a PA proof P of a $\sum_1^0$ sentence $\exists y : N. Spec(x, y)$,*

1. *We can find P' an HA proof of $\neg\neg\exists y : N.\ \neg\neg\ Spec(x, y)$ without use of $\perp$ elimination.*
2. *If we substitute a formula A for each occurrence of $\perp$ in P' the result is a proof, P'_A, of $\neg_A\neg_A\ \exists y{:}N.\ \neg_A\neg_A\ Spec(x, y)$.*
3. *From P'_A we easily get a proof of $\exists y : N.\ Spec(x, y)$.*

Here is an example of the result first proved informally. Although this may seem too trivial to consider, the example makes interesting points. We show simply that $\exists x : N.(0 < x)$. Assume not, then we show a contradiction. So we are assuming $\neg\exists x : N.(0 < x)$. (Call this assumption ne.) This means we assume $\exists x : N.(0 < x) \Rightarrow \bot$ and we are trying to prove $\bot$. To do this we use *modus ponens* and show $\exists x : N.(0 < x)$. To show this, take 0 for x and show $0 < 0$. To show $0 < 0$, assume the converse and prove a contradiction, that is, assume $\neg(0 < 0)$. (Call this assumption nz.) Next to show false, again use *modus ponens* on $\exists x : N.(0 < x) \Rightarrow \bot$. This time we show $\exists x : N.(0 < x)$ by choosing 1 for x. So we must show $0 < 1$, but this could be regarded as an axiom. (In the logic of this paper we must define $x < y$ as $\exists y : N.(\neg Zero(y)\ \&\ x + y = z)$ but we do not need to carry the proof to this level of detail; in Nuprl $0 < 1$ is just an axiom.)

Here is the proof obtained from the above argument which satisfies clause 1 of the theorem. We look at a formal proof for this which follows the informal argument above very closely. From this formal proof it is easy to build the inhabiting proof term. The goal is the formula $((\exists x.\neg\neg(0 < x)) \rightarrow \bot) \rightarrow \bot$ (as required by clause 1 of the theorem). This is proved by assuming $(\exists x.\neg\neg(0 < x)) \rightarrow \bot >> \bot$ is a subgoal. Next the *modus ponens* rule generates the subgoal of showing $>>\ \exists x.\neg\neg(0 < x)$ (the subgoal that $\bot$ follows form $\exists x.\neg\neg(0 < x) \rightarrow \bot$, $\bot$ is omitted to save space).

$>> ((\exists x.\neg\neg(0 < x)) \rightarrow \bot) \rightarrow \bot$

$ne : (\exists x.\neg\neg(0 < x)) \rightarrow \bot$ $>> \bot$

$>> \exists x \neg\neg(0 < x)$

$>> \neg\neg(0 < 0)$

$ne : (\exists x.\neg\neg(0 < x)) \rightarrow \bot,\ nz : \neg(0 < 0) >> \bot$

$>> \exists x.\neg\neg(0 < 1)$

$>> \neg\neg(0 < 1)$

$ne : (\exists x.\neg\neg(0 < x)) \rightarrow \bot,\ nz : \neg(0 < 0),\ x : \neg(0 < 1)$ $>> \bot$

$>> 0 < 1$

by axiom

The term for this proof is $\lambda ne.ap(ne; < 0, \lambda nz.ap(ne; < 1, \lambda x.x(axiom)\ >)\ >)$. How

do we use the term

$$\lambda ne.ap(ne; < 0, \lambda nz.ap(ne; < 1, \lambda x.x(axiom >) >)$$

The method is to use Friedman's "top-level trick" which we now explain.

Notice that according to clause 2 of Friedman's theorem we have a proof of $(\exists x.(((0 < x) \Rightarrow A) \Rightarrow A) \Rightarrow A) \Rightarrow A$ in other words of $((\exists x.\neg_A \neg_A (0 < x)) \Rightarrow A) \Rightarrow A$. Now Friedman's top level trick allows us to prove $\exists x.(0 < x)$ from this. The first step is to take A to be the formula $\exists x.(0 < x)$. Then we notice that the above formula has the form

$$(F_A \Rightarrow A)$$

where F_A is $\exists x.((0 < x) \Rightarrow A) \Rightarrow A$.

For the particular choice of A as $\exists x.(0 < x)$ we can prove F_A. First we notice easily that it is true based on our intuition about quantifiers. But indeed we see that the following proof term gives the proof, where $0 < n + \neg(0 < n)$ stands for an arithmetic expression which returns either $inl(axiom)$ or $inr(axiom)$ depending on whether $0 < n$ or not. The proof term for

$$\exists x.(((0 < x) \to A) \to A) \to A$$

is

$$\begin{array}{l} \lambda p.spread(p; n; imp. \\ \quad decide(0 < n + \neg(0 < n); \\ \qquad t. < n, t >;\ f.\ imp(\lambda x.any(x))). \end{array}$$

Call this term *top*. Now apply the proof term for the theorem to it; we should get a proof of $\exists x.(0 < x)$. After substituting *top* for *ne* we get

$ap(top; < 0, \lambda\ nz.ap(top; < 1, \lambda x.x(axiom) >) >)$
Let $n = 0$ and $imp = \lambda nz.ap(top; \_\_)$
then $decide((0 < 0 + \neg(0 < 0 + \neg(0 < 1)); t. < n.t >; f.imp(\lambda x.any(x)))$, reduces to
$\lambda\ nz.ap(top; < 1, \lambda x.x(axiom) >)$ which reduces to
$decide((0 < 1 + \neg(0 < 1)); t. < n, t >; \_\_)$.
On evaluation of *decide* we get
$n = 1$ and $t = axiom$ and the evaluation of $0 < 1 + \neg(0 < 1)$ results in $inl(axiom)$ so that $< 1, axiom > \epsilon\ \exists x.(0 < x)$.

We can see from this simple example how the Friedman theorem works. It translates the classical proof into a constructive proof of a more elaborate formula, the so called *A-transform* of the original. Then Friedman supplies a top level proof term which brings out the right computational meaning.

The Friedman theorem is easy to prove. Essentially the Gödel style translation of classical logic into constructive from section 4.3 is used to translate the classical proof locally. Murthy noticed that this method of translation corresponded exactly to the *continuation passing style* (CPS) translation of call by value semantics into call by name semantics that Plotkin [30] used. This led him to see that it is possible to give a direct computational semantics to classical proofs. We turn to this observation next and apply it to the simple example of $\exists x.(0 < x)$.

7 Programming with Classical Proofs

We now consider programming in a language with the control operator $\mathcal{C}$. We extend the evaluation rules for the language to include the ones for $\mathcal{C}$ and *abort*.

Definition 5 *A* $ev_K\!\!\rightarrow a'$ *if a evaluates to* a' *using the reduction rules which include those for* $\mathcal{C}$ *and abort.*

Murthy's results [26] relate computing with $ev\!\!\rightarrow_K$ and proving properties in PA. The main theorem we use is

Theorem 4 *(Murthy '90): If* $>>_{PA}\ \ a : S$ *for* S *a* $\sum_1^0$ *formula,*

then we can find a canonical a' *such that* $a\ ev\!\!\rightarrow_K a'$ *and* $>>_{PA}\ \ a' : S$.

Now we apply this theorem to the example in the previous section.

Look at the classical proof term for $\exists x.(0 < x)$. It is

$$\mathcal{C}(\lambda ne.ap(ne; < 0, \mathcal{C}(\lambda nz.ap(ne; < 1, axiom >)) >))$$

This term is built from the classical proof. How does $\mathcal{C}$ get at the right evidence?

Let's reduce the term and see.

Recall $E[\mathcal{C}(t)]\ ev\!\!\rightarrow t(\lambda z.abort(E(z)))$. In this case, $E[z] = z$. The first reduction step substitutes $\lambda z.abort(z)$ for ne, we remember the binding but delay the substitution.

$ap(ne; < 0, \mathcal{C}(--) >)$ reduces to
$< 0, \mathcal{C}(\lambda nz.ap(ne; < 1, axiom >)) >$ reduces to
$< 0, ap(\lambda nz.ap(ne; < 1, axiom >); \lambda x.\ abort(< 0, x >))$reduces to
$< 0, ap(ne; < 1, axiom >) >$ reduces to
$< 0, abort(< 1, axiom >) >$ reduces to
$< 1, axiom >$ which is the right evidence. Notice that
$\lambda x.abort(< 0, x >)$, is not needed as a function, just as data.

For more results along these lines see [10, 26, 27]. As a final comment on this line of work let us see what the computational interpretation of $P \vee \neg P$ is. This can be proved from the law of $\neg\neg elim$ as follows.

$>> P \vee \neg P$ by $\neg\neg elim$
  $>> \neg\neg(P \vee \neg P)$
$np : \neg(P \vee \neg P) >> \perp$ by $elim$ on np
  $\neg(P \vee \neg P) >> P \vee \neg P$ by inr
    $>> \neg P$ by $intro$
    $P >> \perp$ by $\neg elim$
      $>> P \vee P$ by inl
      $P >> P$ hyp

The proof term for this proof is

$$\mathcal{C}(\lambda(np.\ ap(np; inr(\lambda(p.\ ap(np; inl(p))))))).$$

This reduces to

$$inr(\lambda(p.\ ap(\lambda z.\ abort(z); inl(p)))).$$

This term acts like a proof of $\neg P$. It has the right type and is in normal form. But, if this term is ever applied in a proof, rather than just used as a value, then its input must be an object of type P, call it p_o. When $\lambda(p.\_\_)$ is applied to p_o the result is $ap(\lambda z.\ abort(z); p_o)$ which reduces immediately to p_o throwing away the surrounding context.

This is quite an interesting meaning for $P \vee \neg P$. It is computational and it interacts with the other computational forms to preserve types in the sense of Murthy's theorem. We need to know more general conditions on when this form preserves types and gives standard content. That is an excellent research topic.

References

[1] S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[2] S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the 5^{th} Symposium on Logic in Computer Science*, pages 95–197, Philadelphia, Pennsylvania, June 1990. IEEE, IEEE Computer Society Press.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Symantics*, volume 103 of *Studies in Logic*. North-Holland, Amsterdam, 1981.

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

[5] E. Bishop. Mathematics as a numerical language. In *Intuitionism and Proof Theory*, pages 53–71. North-Holland, NY, 1970.

[6] A. Bundy. A broader interpretation of logic in logic programming. In *Proceedings Fifth Symposium on Logic Programming*, pages 1624–1648, 1988.

[7] J. Chirimar and D. J. Howe. Implementing constructive real analysis: a preliminary report. In *Symposium on Constructivity in Computer Science*, pages 165–178. Springer-Verlag, 1991.

[8] R. L. Constable. Assigning meaning to proofs: a semantic basis for problem solving environments. *Constructive Methods of Computing Science*, F55:63–91, 1989.

[9] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[10] R. L. Constable and C. Murthy. Finding computational content from classical proofs. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 341–362. Cambridge University Press, 1991.

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

[12] M. Felleisen and D. Freidman. Control operators, the SECD machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–41. North-Holland, 1986.

[13] A. Felty and D. A. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 61–80. Springer-Verlag, NY, 1988.

[14] H. Friedman. Classically and intuitionistically provably recursive functions. In D. S. Scott and G. H. Muller, editors, *Higher Set Theory*, volume 699 of *Lecture Notes in Mathematics*, pages 21–28. Springer-Verlag, 1978.

[15] J.-Y. Girard. *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, Napoli, 1987.

[16] J.-Y. Girard. A new constructive logic: classical logic. *Math. Struct. in Comp. Science*, 1:255–296, 1991.

[17] J.-Y. Girard. On the unity of logic. In *Proceedings of Computer and Systems Sciences*, volume F?? of *NATO ASI Series*, page ??, 1991.

[18] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1989.

[19] T. Griffin. A formulas-as-types notion of control. In *Proceedings of the Seventeeth Annual Symposium on Principles of Programming Languages*, pages 47–58, 1990.

[20] R. Harper. Constructing type systems over an operational semantics. *J. Symbolic Computing*, 14(1):71–84, 1992.

[21] L. Helmink. The **constructor** Proof Development System. Technical Report RWR-113-lh-91247-ih, Philips Reseach Laboratories, Eindhoven, 1991.

[22] D. MacQueen, B. Duba, and R. Harper. Typing first-class continuations in ML. In *Principles of Programming Languages*, pages 163–173, Orlando, FL, 1990.

[23] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[24] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1991.

[25] C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Department of Computer Science, 1990. (TR 90-1151).

[26] C. Murthy. An evaluation semantics for classical proofs. In *Proceedings of Sixth Symposium on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

[27] C. Murthy. On the spectrum of value-hood in classical control calculi: A computational explanation of the answers in classical proofs. (preprint), 1991.

[28] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[29] F. Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science*, pages 313–322, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE, IEEE Computer Society Press.

[30] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Journal of Theoretical Computer Science*, pages 125–59, 1975.