

# What is the difference between proofs and programs?\*

John N. Crossley<sup>1</sup>

School of Computer Science and Software Engineering,  
Monash University, Clayton, Victoria, Australia 3800  
`John.Crossley@infotech.monash.edu.au`

**Abstract.** Curry and Howard observed that ordinary propositional logic can also be viewed as a functional (programming) language. Thus programs are contained, in a certain sense, in proofs in mathematical logic. The underlying reason (in the present author's view) is because of the formal, that is to say, purely syntactic, similarities between logical rules and those of the lambda calculus. This has led to the viewing of proofs (originally, just in formal logic) as computer programs. The advantage of these techniques is that the task of programming a function is reduced to reasoning with domain knowledge.

In this paper we sketch the development of the Curry-Howard correspondence, first of all into predicate calculus, then into arithmetic. After that we look at different applications of the idea of the Curry-Howard process into two very different applications: algebraic specifications and imperative programming. The full details may be found in our forthcoming book with Iman Poernomo and Martin Wirsing.

Finally, having seen how proofs may be said to contain programs, we discuss the question of whether there are, or should be, proofs in programs.

## 1 Introduction

About the year 1900 there was just “one true logic”: classical logic. In such a logic one would expect that everything was clear. Certainly, in that logic, any statement was either true or false: there was the law of the excluded middle,  $(A \vee \neg A)$ . But how do we check an infinite number of instances? What does it mean to say that there is no largest pair of twin primes, that is to say that there is an end to such pairs such as 5 and 7; 11 and 13 or even 202 289 and 202 291?

On the other hand, saying there are infinitely many pairs of twin primes has a clear meaning if we can show that, for every pair, there is a larger pair. In this context compare the way that Euclid established that there were infinitely many prime numbers (although he did not phrase it like that). He gave a method for constructing a larger prime from a given (finite) set of prime numbers.

---

\* This paper was presented at the First Indian Conference on Logic and its relationship with other disciplines, Mumbai, January 2005.

It was because of questioning by Brouwer, a Dutch mathematician, indeed a topologist, of the “one true logic”, that “constructive logic” or “intuitionist logic” arose.

When you look at this logic it is not evident, at first glance, that the logic actually gives you the way of performing the necessary construction. However, that is perhaps the wrong way to look at it. Brouwer was concerned only with constructing mathematical objects that were claimed to exist. He did not like mathematical logic and did not consider it relevant. However, when his approach is formalized, the details are buried inside the proof. Are they perhaps buried in the way that algorithms are buried inside computer programs?

Thus arose “constructive logic” or “intuitionist(ic) logic”. In the 1960s I went to a course on Intuitionism taught by Michael Dummett. The lectures have since become a book [6]. I was a student then and I must admit that I found it very odd. Odd, but also very interesting, indeed, fascinating. The subject matter seemed strange to us in the audience because it did not use the law of the excluded middle. Nowadays intuitionist logic has become a very standard subject as you can see from Michael Dummett’s book.

The most important thing about constructive proofs is that they contain the information that allows one to construct the objects considered. For example if we prove  $\forall x \exists y A(x, y)$  constructively then, given an  $x$ , we can actually construct a  $y$  such that  $A(x, y)$  is true, indeed, is provable. The information required for the construction is embedded in the proof.

For further details of the actual logical system that I use please see the tutorial at this conference [3]. The system is outlined in Fig. 1. Here  $\Delta, A \vdash_{\text{Int}} B$  means “ $B$  can be inferred from  $A$  and the formulae in  $\Delta$ ”. The restriction to Harrop formulae is a technical one.<sup>1</sup>

## 2 The Lambda Calculus and the Curry-Howard correspondence

How do we extract the information from a proof in mathematical logic? Curry started, and Bill Howard [11] developed, the basic idea. To show this we need some notation. We begin with the lambda calculus, but this calculus will slowly get extended. Here is the formal definition of lambda terms.

**Definition 1 (Lambda terms).** *The alphabet comprises variables  $x_1, y_1, \dots$ , together with  $\lambda$  and “.”, and the brackets ( and ).*

*The lambda terms,  $A$ , are formed as follows (in Backus-Naur notation):*

---

<sup>1</sup> A formula  $F$  is a *Harrop formula* if it is 1. an atomic formula, 2. of the form  $(A \wedge B)$  where  $A$  and  $B$  are Harrop formulae, 3. of the form  $(A \rightarrow B)$  where  $B$  (but not necessarily  $A$ ) is a Harrop formula, or 4. of the form  $\forall x.A$  where  $A$  is a Harrop formula. Harrop formulae, in a sense, contribute no information for the program. However, the rule ( $\perp$ -E) easily extends, through a proof by induction, to provide a proof of any formula  $A$  from the false formula  $\perp$  since atomic formulae are Harrop and so are their negations  $(A \rightarrow \perp)$ .

$$T = x|\lambda x.T|(T_1T_2)$$

The lambda calculus has two major constructions: *abstraction* and *application*.

Assume that  $x, y$  are individual variables, and that  $t$  and  $t'$  are individual terms.

$$\frac{}{A \vdash_{\text{Int}} A} \text{ (Ass-I)}$$

$$\frac{\Delta, A \vdash_{\text{Int}} B}{\Delta \vdash_{\text{Int}} (A \rightarrow B)} \text{ (}\rightarrow\text{-I)} \quad \frac{\Delta \vdash_{\text{Int}} A \quad \Delta' \vdash_{\text{Int}} (A \rightarrow B)}{\Delta, \Delta' \vdash_{\text{Int}} B} \text{ (}\rightarrow\text{-E)}$$

$$\frac{\Delta \vdash_{\text{Int}} A}{\Delta \vdash_{\text{Int}} \forall x.A} \text{ (}\forall\text{-I)} \quad \frac{\Delta \vdash_{\text{Int}} \forall x.A}{\Delta \vdash_{\text{Int}} A[t/x]} \text{ (}\forall\text{-E)}$$

$x$  is free in  $A$ , not free in  $\Delta$

$$\frac{\Delta \vdash_{\text{Int}} P[t'/y]}{\Delta \vdash_{\text{Int}} \exists y.P} \text{ (}\exists\text{-I)} \quad \frac{\Delta_1 \vdash_{\text{Int}} \exists y.P \quad \Delta_2, P[x/y] \vdash_{\text{Int}} C}{\Delta_1, \Delta_2 \vdash_{\text{Int}} C} \text{ (}\exists\text{-E)}$$

where  $x$  is not free in  $C$

$$\frac{\Delta \vdash_{\text{Int}} A \quad \Delta' \vdash_{\text{Int}} B}{\Delta, \Delta' \vdash_{\text{Int}} (A \wedge B)} \text{ (}\wedge\text{-I)}$$

$$\frac{\Delta \vdash_{\text{Int}} (A_1 \wedge A_2)}{\Delta \vdash_{\text{Int}} A_1} \text{ (}\wedge\text{-E}_1) \quad \frac{\Delta \vdash_{\text{Int}} (A_1 \wedge A_2)}{\Delta \vdash_{\text{Int}} A_2} \text{ (}\wedge\text{-E}_2)$$

$$\frac{\Delta \vdash_{\text{Int}} A_1}{\Delta \vdash_{\text{Int}} (A_1 \vee A_2)} \text{ (}\vee\text{-I}_1) \quad \frac{\Delta \vdash_{\text{Int}} A_2}{\Delta \vdash_{\text{Int}} (A_1 \vee A_2)} \text{ (}\vee\text{-I}_2)$$

for any Harrop formula  $A$

$$\frac{\Delta \vdash_{\text{Int}} A \vee B \quad \Delta_1, A \vdash_{\text{Int}} C \quad \Delta_2, B \vdash_{\text{Int}} C}{\Delta_1, \Delta_2, \Delta \vdash_{\text{Int}} C} \text{ (}\vee\text{-E)}$$

$$\frac{\Delta \vdash_{\text{Int}} \perp}{\Delta \vdash_{\text{Int}} A} \text{ (}\perp\text{-E)}$$

provided  $A$  is Harrop

$A[a/x]$  is read “ $A$  with  $a$  for  $x$ ” and denotes the formula  $A$  with  $a$  substituted for the variable  $x$ .

**Fig. 1.** The basic rules of intuitionistic logic, Int.

Where does the lambda calculus come from? Consider the following:

What is the function denoted by  $x^y$ ? We have several choices: as a function of two variables, as a function of  $x$  only with  $y$  held constant and as a function of  $y$  only with  $x$  held constant. These are usually denoted by

$$\lambda x \lambda y . x^y \text{ (often written as } \lambda x y . x^y \text{), } \lambda x . x^y \text{ and } \lambda y . x^y .$$

This is *abstraction*.

*Application* is written in a familiar way: thus  $(T_1 T_2)$  denotes the application of the lambda term  $T_1$  to the lambda term  $T_2$ . In particular  $fa$  is the application of the lambda term  $f$  to the lambda term  $a$ . (We omit brackets where there is no ambiguity.)

These notations have the obvious interpretations. (Try them on  $x^y$  and specific values of  $x$  and  $y$ .)

In ordinary mathematics if we apply the function  $\lambda x . f$  to  $a$  then we get  $f[a/x]$ , which is read “ $f$  with  $a$  for  $x$ ”. In the lambda calculus however this is *not* the same as the (application) term  $\lambda x . fa$ , i.e.  $\lambda x . f$  applied to  $a$ . That is to say they are *syntactically* different. We therefore have to introduce the notion of  *$\beta$ -reduction*<sup>2</sup>

$$\lambda x . fa \triangleright f[a/x]$$

(Here  $\triangleright$  is read “reduces to”.)

Now note the similarities between  $\rightarrow$ -introduction, the rule ( $\rightarrow$ -I), and  $\rightarrow$ -elimination ( $\rightarrow$ -E) (in Fig. 1) on the one hand, and  $\lambda$ -introduction and  $\lambda$ -elimination on the other, the  $\beta$ -rule.

Next consider a proof of  $B$  from  $A$  from which we get a proof<sup>3</sup> of  $A \rightarrow B$  (by the rule ( $\rightarrow$ -I)):

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{(A \rightarrow B)}$$

and lambda abstraction (which abstracts a function from the process where  $a \in A$  gives us  $f(a) \in B$ ): that is  $\lambda x . f$ . Consider the figure:<sup>4</sup>

$$\frac{\begin{array}{c} a \\ \vdots \\ f(a) \end{array}}{\lambda x . f}$$

What is the connexion?

The most obvious thing, I hope, is that the *shapes* are the same!

<sup>2</sup>  $\alpha$ -reduction refers to the simple renaming of one variable by another (without clashes).

<sup>3</sup> The square brackets indicate that  $A$  can be discharged, i.e. is not needed for the proof of  $B$ , though it is for the proof of  $B$ , of course.

<sup>4</sup> Here we have written  $f(a)$  to show that we think of  $a$  as being involved in  $f$ .

The *typed* lambda calculus that we shall consider, that is to say lambda calculus with each term having a *type* assigned to it, can be regarded as the amalgam of the two systems: logic, or more precisely, systems of predicate calculus, and the lambda calculus.

A special kind of typed lambda calculus involves taking formulae of logic as the types. Now this is a strange idea to accept but it is easier to work with it if you just think of a type (formula) as the set of proofs of that formula. Instead, therefore, of variables, we use typed variables of the form  $a : A$ .

The rule of *modus ponens* then becomes:

$$\frac{a : A \quad g : (A \rightarrow B)}{(ga) : B} \quad (1)$$

where we have changed  $f$  to  $g$  to avoid confusion in what follows.

If we had a proof of  $B$  from  $A$  then we would get an expression  $\lambda x : A. f : B$  by the rule of ( $\rightarrow$ -I) which has type  $(A \rightarrow B)$ . If the  $g$  in the expression (1) is actually of the form  $(\lambda x : A. f : B) : (A \rightarrow B)$ , then we get

$$\frac{a : A \quad (\lambda x : A. f : B) : (A \rightarrow B)}{((\lambda x : A. f : B) : (A \rightarrow B))a : A) : B}$$

which is somewhat hard to read. However the bottom line has the formula  $B$  as its type, and the expression reduces to

$$f : B[a : A/x : A] \quad (2)$$

where the substitution of  $a : A$  for  $x : A$  takes place throughout the term  $f : B$ .

If we translate this back into proofs it means that the corresponding proofs look as follows. On the one hand we have the complicated proof:

$$\frac{\begin{array}{c} \vdots \\ A \end{array} \quad \frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{(A \rightarrow B)}}{B} \quad (3)$$

and on the other hand, by putting the proof of  $A$  from the left on top of the proof of  $B$ , and *not* introducing the  $\rightarrow$ , we no longer need the hypothesis  $[A]$  in the proof on the right in order to get a proof of  $B$ .

That is to say, we *reduce* the proof in (3) to a simple proof of  $B$  of the form

$$\begin{array}{c} \vdots \\ A \\ \vdots \\ B \end{array}$$

This corresponds in the lambda calculus to the reduction<sup>5</sup> that resulted in (2). So we have a direct correspondence between proofs and terms of our typed lambda calculus. This is called the *Curry-Howard correspondence*.<sup>6</sup>

### 3 Strong Normalization and Program Extraction

Now it is obvious that a long and complicated formal proof has an even longer typed lambda calculus expression associated with it. If, however, all the possible reductions are carried out it may become considerably simpler. Indeed, in the cases with which we are concerned, we can usually omit all the types. (They will have served their purpose of ensuring that we get a result of the correct type when the proof is complete. This is related to the use of types in computer programming languages.)

The maximum benefit is when we have a *Strong Normalization Theorem* for the system. Such a theorem says that, whatever the order of the reductions (and there may be many possible different reductions for a long lambda term) the process always stops. (One reason the process might be expected *not* to stop is clear when you look at substituting  $x + x$  for  $x$ : the number of  $x$ s goes up each time and the expression gets longer!)

The Curry-Howard correspondence can be extended to the other logical connectives by modifying the lambda calculus. Surprisingly, in addition to the above operations involving lambdas, we only need the formation of ordered pairs and the projections onto the first and second elements of those pairs in order to capture all first order logic.<sup>7</sup> We give only a few examples; the full details can be found in [5]. The Curry-Howard term for a conjunction ( $A \wedge B$ ) obtained by the rule ( $\wedge$ )-I is the ordered pair  $(p : A, q : B)$  where  $p : A$  is the Curry-Howard term for the proof of  $A$ , and similarly for  $B$ . Conversely we use the projections `fst` and `snd` for the rules ( $\wedge$ )-E<sub>1</sub> and ( $\wedge$ )-E<sub>2</sub>. For the rule ( $\exists$ )-I we get the term  $(t, p : A(t))$  where the premise has the Curry-Howard term  $p : A(t)$ . Thus the Curry-Howard term contains the term  $t$  that was proved to exist.

The major consequence of the Strong Normalization theorem is then that, if we prove a formula of the form  $\exists x A(x)$ , we can actually extract, from the normalized proof (i.e. the lambda, or Curry-Howard, term in which no more reductions are possible), an  $x$  such that  $A(x)$ . Further, if we can prove  $\forall x \exists y A(x, y)$  then we can actually get a program such that, given an  $x$ , it will compute a corresponding  $y$ . Moreover, we have a proof of  $A(x, y)$  for this  $x$  and  $y$  so the program is “correct” in the sense that it meets its specification.<sup>8</sup>

*Curry-Howard terms* are, in general, a generalization of the idea known variously as formulae-as-types or, better, as proofs-as-types: the terms code up a

<sup>5</sup> This process of reduction is also called *cut elimination*.

<sup>6</sup> Some people use the term *isomorphism* but there are technical difficulties involved in making the correspondence one to one, so I prefer the weaker terminology.

<sup>7</sup> The process can also be extended to higher order logic.

<sup>8</sup> Intuitively speaking, the specification is the statement about the result of the program. See also below Section 4.1.

whole proof by successively encoding the applications of the logical rules in a proof.

Not surprisingly, not all rules of logic allow us to prove a strong normalization theorem. One major obstacle is the law of double negation: From  $\neg\neg A$  infer  $A$ . If we had a rule that would allow us to prove  $\exists xA(x)$  from  $\neg\neg\exists xA(x)$ , how do we obtain such an  $x$ ? So we generally restrict ourselves to constructive logic and all is well.

Changing to other systems, e.g. arithmetic, may bring in other axioms. Here the most dramatic is the rule of induction. Fortunately the induction axiom

$$\frac{A(0) \quad \forall x(A(x) \rightarrow A(x + 1))}{\forall xA(x)}$$

gives rise to a reduction *exactly* corresponding to the recursion

$$\begin{aligned} f(\bar{a}, 0) &= g(\bar{a}) \\ f(\bar{a}, x + 1) &= h(\bar{a}, x, f(\bar{a}, x)) \end{aligned}$$

Happily we can prove a strong normalization theorem for arithmetic (see [5]).

## 4 Beyond traditional logic

### 4.1 Algebraic Specifications

We now turn to an application of the above ideas to software engineering. Producing programs that satisfy their specifications is a primary goal of software engineering.

What is an algebraic specification? It is a description in formal logic of a structure, for example, the natural numbers.

We use the *Common Algebraic Specification Language* (*CASL*, see [2]) but the technique could be employed in other specification languages, indeed originally we ourselves used a different language.

Structured specifications in *CASL* are built from *basic* (or *flat*) specifications by means of *translation* (or *renaming*), written **with**, taking *unions* of specifications, written **and**, *hiding* signatures, and the *extension of specifications*. A typical example of a flat specification, this one is for natural numbers, is given in Fig. 2.

When we change a specification, then what is true changes – even if simply because we use new names, e.g. “car” instead of “auto”, “boot” instead of “trunk”, etc. but we may also add new predicates (relations).

We have developed logical systems to reflect the interaction between such changes and the logic statements.

Originally Martin Wirsing studied a logical calculus for structured specifications (see [15]). This was subsequently extended by Wirsing and his student Peterreins (see [13]). Next Wirsing and the present author extended the idea to

```

spec NAT_0 =
sorts
  Nat
ops 0 : Nat; s : Nat → Nat; + : Nat × Nat → Nat
preds
  ≥ : Nat × Nat
axioms
  ∀x : Nat • x + 0 = x                                %(Nat_0.1)%
  ∀x; y : Nat • x + s(y) = s(x + y)                  %(Nat_0.2)%
  ∀x : Nat • x ≥ 0                                     %(Nat_0.3)%
  ∀x; y : Nat • x + y = y + x                         %(Nat_0.4)%
  ∀x : Nat • s(x) ≥ x                                  %(Nat_0.5)%
  ∀x; y; v; w : Nat • x ≥ v ∧ y ≥ w → x + y ≥ v + w %(Nat_0.6)%
end

```

**Fig. 2.** The specification NAT\_0.

algebraic specifications, and then we went even further with Iman Poernomo to include even parametrized specifications in the language *CASL*.

Abstractly speaking we have an annotated or labelled deductive system.<sup>9</sup> The basic form of a rule in such a logic can be written in the form

$$\frac{p : A \quad q : B}{s(p, q) : \sigma(A, B)}$$

It is convenient to use “contexts” also. That is to say, the actual hypotheses with which we are working. These will be written in the standard logical style using the “turnstile” symbol *vdash*. We shall use  $\Gamma$ , possibly with subscripts, to denote a set of logical formulae. Thus we write  $\Gamma \vdash A$  to indicate that  $A$  is provable in the context  $\Gamma$  (or equivalently, from the hypotheses  $\Gamma$ ).

When we wish to extract programs from proofs from algebraic specifications the Curry-Howard terms that we use are now more complicated for two reasons. In addition to the information from, for example, the logical rule being used, the Curry-Howard term also has to “remember” the specification. We have a similar situation for the structural rules. However, the message is as before: the Curry-Howard term carries *all* the information as to how we have constructed the proof so far.

The annotations we shall use will also involve Curry-Howard terms, specification names and the logical connectives. We have two kinds of rules: those for the logical connectives, *logical rules*; and those for the structural changes in the specifications, *structural rules*. Even with the purely logical rules, the specification of the conclusion depends on those in the premises. Thus the full rule for *modus ponens* we have is

<sup>9</sup> The logical system that we then have is therefore related to the *labelled deduction systems* of Gabbay [7].

$$\frac{\Gamma_1 \vdash a.SP : A \quad \Gamma_2 \vdash d.SP : (A \rightarrow B)}{\Gamma_1 \cup \Gamma_2 \vdash (da).SP : B} (\rightarrow \text{E})$$

Let me explain what is going on in the rule for implication elimination ( $\rightarrow$  -E). We are working within a single specification SP. We have Curry-Howard terms  $d$  for the proof of  $(A \rightarrow B)$  and  $a$  for the proof of  $A$ . Therefore we have, just as in (1),  $(da)$  as the Curry-Howard term for the resulting proof of  $B$ . As usual the contexts  $\Gamma_1$  and  $\Gamma_2$  are added together. The specification SP is unchanged throughout.

Now here is the rule for implication introduction:

$$\frac{\Gamma, x : A \vdash d.SP : B}{\Gamma \vdash \lambda x : A.d.SP : (A \rightarrow B)} (\rightarrow \text{I})$$

$\frac{}{A \vdash SP : A} (\text{Ass-I})$ <p>where <math>\text{Sig}A \subseteq \text{Sig}(SP)</math></p>	$\frac{}{\emptyset \vdash \langle \Sigma, Ax \rangle : A} (\text{Ax-I})$ <p>where <math>Ax</math> are the axioms</p>
--	--

**Fig. 3.** Two new logical rules for Structured Specification Logic. SP is any structured specification expression. Sig indicates taking the signature.

For the structural rules, the change in the structure is reflected in the specification of the conclusion. The full rule for translations, including the lambda calculus elements and the specifications, is as follows:

$$\frac{\Gamma \vdash d.SP : A}{\rho'(I) \vdash \rho \bullet d.SP \text{ with } \rho : \rho \bullet A} (\text{trans})$$

In the structural rule for translation the formula  $A$  is unchanged in meaning but the language is changed, therefore  $A$  has to be changed into its translation  $\rho \bullet A$ . Similarly the context  $\Gamma$  has to be translated. This is written (by us) as  $\rho'(I)$ . In addition, the Curry-Howard term for  $A$  has to be changed (translated) into the new language. So the new Curry-Howard term is  $\rho \bullet d$ . Finally the new specification is the translated one: SP **with**  $\rho$ .

The logical rules for our system *Structured Specification Logic* are very similar to those in Fig. 1 with two exceptions that we give in Fig. 3. In order to make the figures less complicated the rules are presented without their Curry-Howard terms. Likewise the structural rules may be found in Fig. 4. The complete set of rules we have for *CASL*, with their Curry-Howard terms, may be found in [4] or [14].

In this situation we are again able to prove strong normalization.

$$\begin{array}{c}
\frac{\Gamma \vdash \text{SP} : A}{\rho'(\Gamma) \vdash \text{SP} \textbf{ with } \rho : \rho \bullet (A)} \text{ (trans)} \\
\frac{\Gamma \vdash \text{SP} : A}{\Gamma \vdash \text{SP} \textbf{ hide } SL : A} \text{ (hide)} \\
\text{where } \Gamma \cup \{A\} \subseteq WFF(\text{Sig}(\text{SP})/SL, \text{Var}) \\
\frac{\Gamma \vdash \text{SP}_{-1} : A}{\Gamma \vdash \text{SP}_{-1} \ \& \ \text{SP}_{-2} : \text{inl}(A)} \text{ (union}_1\text{)} \\
\frac{\Gamma \vdash \text{SP}_{-2} : A}{\Gamma \vdash \text{SP}_{-1} \ \& \ \text{SP}_{-2} : \text{inr}(A)} \text{ (union}_2\text{)} \\
\frac{\Gamma \vdash \text{SP}_{-1} : A}{\Gamma \vdash \text{SP}_{-1} \ \textbf{ then } \ \text{SP}_{\text{-EXT}} : \text{inl}(A)} \text{ (ext}_1\text{)} \\
\frac{\Gamma \vdash \text{SP}_{\text{-EXT}} : A}{\Gamma \vdash \text{SP}_{-1} \ \textbf{ then } \ \text{SP}_{\text{-EXT}} : \text{inr}(A)} \text{ (ext}_2\text{)}
\end{array}$$

**Fig. 4.** The structural rules of Structured Specification Logic. Here the condition  $\Gamma \cup \{A\} \subseteq WFF(\text{Sig}(\text{SP})/SL, \text{Var})$  means that none of the well-formed formulae in  $\Gamma$  and  $A$  contains any letter from the hidden signature  $SL$ . We have omitted the lambda calculus parts of the Curry-Howard terms for clarity.

From this strong normalization theorem we are then able to give an *extraction map*, that is to say, we give a formal process that, given a Curry-Howard term for a proof of  $\forall x \exists y A(x, y)$  for a given specification, the extraction map returns a suitable  $y$  for a given  $x$ . Indeed it gives a program in the programming language  $ML$ . The extraction map works recursively and, in particular, the cases for  $\rightarrow$ -introduction and elimination correspond directly to the procedure we have outlined above.

## 4.2 Imperative programming

My most recent PhD student, Iman Poernomo, has developed a protocol for integrating ordinary computer programs into the kind of deductive system we have been discussing. This protocol he calls the *Curry-Howard* protocol. The logical system for such a situation includes the state of the system (i.e. the contents of registers in the machine) and accounts for the changes that take place when a program is run. Despite the complications this produces it is still possible to produce a constructive version of a Hoare logic (cf. [10]) for reasoning about imperative programs to which the Curry-Howard isomorphism may be adapted.

Note that a theorem in the Hoare logic consists of an imperative program and a truth about the program. Because of this the logic can be used to synthesize programs.

However we are also concerned to use programs already in the programming language that we regard as “reliable”. We do not use the word “correct” here,

reserving that word for programs that have been formally proved to meet their specifications. Here we simply mean that we have programs that we are satisfied will give the correct answers. Such programs include very simple ones such as programs for the multiplication of natural numbers. This achieves a significant saving in the length of the programs extracted. Otherwise we would have to prove a formula in formal arithmetic that allows us to extract a program, for example for the multiplication function. The proof would be inordinately long, involving several applications of induction and its corresponding program would then involve the same number of recursions. This is obviously very uneconomical, because we know it is possible to write a relatively simple program for multiplication (if one is not built into the computer already).

Imperative computer programs have *side-effects*: they change the *state* of the machine and, in particular, the values in various registers. The presence of side-effects is what distinguishes the imperative programming paradigm from the functional one. However, side-effect-free functions are also important in imperative programs because they enable access to data, obtaining views of state and producing return values. Imperative programs involve both side-effects and side-effect-free return values. Consider, for instance, a program that triples the number in the register  $s$  and returns the value twice the value in  $s$ . In *Standard ML* the program is

$$s := !s * 3; !s * 2$$

It has a side-effect producing assignment statement,  $s := !s * 3$ , followed by the return value  $!s * 2$ . In many popular imperative languages such return values are potentially complex, involving higher-order functional aspects that are difficult to program correctly.

Our goal is to specify, reason about and synthesize both aspects of imperative programs – side-effects and functional return values.

Our approach is as follows. We use a version of Hoare logic to synthesize the side-effect producing aspect of a program, specified in terms of pre- and post-conditions. Hoare logic involves considering triples of the form

$$\{pre-condition\} \text{program step} \{post-condition\}$$

The *pre-condition* is true before the program step commences and the *post-condition* is true after the step.

The formula

$$s_f > s_i$$

specifies a side-effect where the final value of state  $s$ , denoted by  $s_f$ , is greater than the initial value, denoted by  $s_i$ . We can use Hoare logic to synthesize a *Standard ML* program that satisfies this specification, by producing, for example, a theorem of the form

$$\vdash s := !s * 3 : s_f > s_i$$

where the left-hand-side of the  $:$  symbol is the required *Standard ML* program (written in teletype font), and the right-hand-side is a true statement about the program.

The structural rules may be found in Fig. 5. The operation  $\mathbf{tologic}_i$  indicates the operation of the Curry-Howard protocol taking the name of the variable in the programming language into the logical language. Thus  $\mathbf{tologic}_i(\mathbf{b})$  yields the variable coming from the register  $\mathbf{b}$ . The remaining items in teletype font are standard imperative programming constructs. Thus  $\mathbf{p}; \mathbf{q}$  means the program  $\mathbf{p}$  is pipelined into the program  $\mathbf{q}$  and  $\mathbf{while } b \mathbf{ do } \mathbf{q}$  is the usual while-loop that repeats the program  $\mathbf{q}$  while the assertion  $q$  is satisfied.

$$\begin{array}{c}
\frac{}{\vdash_{\mathbf{K}} \mathbf{s} := \mathbf{v} : s_f = \mathbf{tologic}_i(\mathbf{v})} \text{ (assign)} \\
\text{where } s \text{ is a state reference.} \\
\frac{\vdash_{\mathbf{K}} \mathbf{p} : (\mathbf{tologic}_i(\mathbf{b}) = \mathit{true} \rightarrow C) \quad \vdash_{\mathbf{K}} \mathbf{q} : (\mathbf{tologic}_i(\mathbf{b}) = \mathit{false} \rightarrow C)}{\vdash_{\mathbf{K}} \mathbf{if } \mathbf{b} \mathbf{ then } \mathbf{p} \mathbf{ else } \mathbf{q} : C} \text{ (ite)} \\
\frac{\vdash_{\mathbf{K}} \mathbf{p} : (A[\bar{s}_i/\bar{v}] \rightarrow B[\bar{s}_f/\bar{v}]) \quad \vdash_{\mathbf{K}} \mathbf{q} : (B[\bar{s}_i/\bar{v}] \rightarrow C[\bar{s}_f/\bar{v}])}{\vdash_{\mathbf{K}} \mathbf{p}; \mathbf{q} : (A[\bar{s}_i/\bar{v}] \rightarrow C[\bar{s}_f/\bar{v}])} \text{ (seq)} \\
\text{where } A \text{ and } B \text{ are free of state identifiers.} \\
\frac{\vdash_{\mathbf{K}} \mathbf{q} : (\mathbf{tologic}_i(\mathbf{b}) = \mathit{true} \wedge A[\bar{s}_i/\bar{v}]) \rightarrow A[\bar{s}_f/\bar{v}]}{\vdash_{\mathbf{K}} \mathbf{while } b \mathbf{ do } \mathbf{q}; \mathbf{done} : A[\bar{s}_i/\bar{v}] \rightarrow (A[\bar{s}_f/\bar{v}] \wedge \mathbf{tologic}_i(\mathbf{b}) = \mathit{false})} \text{ (loop)} \\
\text{where } A \text{ is free of state identifiers.} \\
\frac{\vdash_{\mathbf{K}} \mathbf{p} : P \quad \vdash_{\text{Int}} (P \rightarrow A)}{\vdash_{\mathbf{K}} \mathbf{p} : A} \text{ (cons)}
\end{array}$$

**Fig. 5.** The structural rules for our logic for imperative programming. Intuitionistic deduction  $\vdash_{\text{Int}}$  is given in Fig. 1.

The logical rules are the same as we had much earlier in Fig. 1, but with the state information added. This is because the intuitionistic rules are concerned with truths that are universal to all programs. That is to say, they can be used to infer properties that hold over any side-effect.

*Example 1.* For instance, an application of the logical ( $\wedge$ -I) rule

$$\frac{s_f = s_i * 2 \vdash_{\text{Int}} s_f \geq s_i \quad s_f = s_i * 2 \vdash_{\text{Int}} \mathit{Even}(s_f)}{s_f = s_i * 2 \vdash_{\text{Int}} s_f \geq s_i \wedge \mathit{Even}(s_f)} \text{ (\wedge-I)}$$

tells us that, any program that makes  $s_f = s_i * 2$  true, makes  $s_f \geq s_i$  and  $\mathit{Even}(s_f)$  true, and therefore the statement:  $s_f \geq s_i \wedge \mathit{Even}(s_f)$  must also be true of the program.

To specify and synthesize return values of a program we adapt *realizability* and the extraction of programs from proofs. We have already treated the latter, so now we consider realizability.

When we extract a program we wish to demonstrate that it is “correct”. This requires the notion of *realizing*. This is a different way of verifying proofs in intuitionistic logic by means of computable functions. It was first developed by Kleene, see the last chapter of [12]. The basic idea is that we produce a program for a (partial) recursive function that is a *witness* to the proof of an assertion. Such witnesses can be produced recursively by going down through the proof. Such a program can be regarded as a number (for example, the binary string that encodes the program). For example if we have partial recursive functions with programs  $p, q$  realizing  $A, B$ , then we take  $(p, q)$  as the realizer of  $(A \wedge B)$ . The full details, which may be found in Kleene [12] for the basic system of intuitionist logic and in our book [14] for the systems we discuss here.

Here is an example. Given the theorem

$$\mathbf{s} := \mathbf{s} * \mathbf{3} : s_f > s_i \wedge (\exists x : \text{int.} \text{Even}(x) \wedge x > s_i)$$

we can synthesize a program of the form

$$\mathbf{s} := \mathbf{s} * \mathbf{3}; \mathbf{f}$$

where the function  $f$  is a side-effect-free function (such as  $!s * 2$ ) that realizes the existential statement of the post-condition  $(\exists x : \text{int.} \text{Even}(x) \wedge x > s_i)$ , by providing a witness for the  $x$ .

With our program extraction users will have no need to manually code the return value, instead they can work within the Hoare logic. There they prove a theorem from which the return value is then synthesized.

Here is an outline of a specific example about an electronic banking system. In the logic here we are using a *many-sorted* system, that is to say, individual variables have their own sorts or varieties. This is a small and natural modification of the logic.<sup>10</sup>

Consider an Automatic Bank Teller machine (ATM) example with the following domain conditions:

1. The ATM permits the user to enter a Personal Identification Number (PIN) and to withdraw money. In order to withdraw money, the user must enter their PIN and a database connection to the bank’s server must be made. The machine has a screen on which it displays messages to the user.
2. The integer state reference `pin` stores the PIN number entered by the user, the boolean state reference `canWithdraw` stores a flag to determine whether or not the user may withdraw money from the machine, and the boolean state reference `isConnected` stores a flag to determine whether or not there is a connection to the bank’s server.
3. We use the predicate `appMessage(m)` to assert that a string  $m$  is an appropriate message to display on the screen for the user, given that the ATM is in some particular state.

<sup>10</sup> It can be avoided by adding extra predicates, one for each sort. In this case instead of  $x : s$  meaning “ $x$  is of sort  $s$ ” we have a predicate  $s$  and we write  $s(x) \rightarrow \dots$ , which can be read as “If  $x$  is of sort  $s$ , then  $\dots$ ”.

- There is a program  $p$  satisfying the following property. Given the user has entered their Personal Identification Number (PIN) correctly, the program allows the user to withdraw money. This property is formally given by an axiom

$$\vdash_{\mathcal{K}} p : PINCorrect(pin_i) \rightarrow canWithdraw_f = true$$

- There is a program  $q$  such that, if the user is permitted to withdraw money, then a database connection is established, and also it is the case that there is an appropriate message that can be displayed. These properties are formally given by the axiom

$$\vdash_{\mathcal{K}} q : canWithdraw_i = true \rightarrow (isConnected_f = true \wedge \exists x : string \bullet appMessage(x))$$

For the sake of argument, we simplify our domain with the following assumptions:

- We assume two *Standard ML* record datatypes have been defined, **user** and **account**. Instances of the former contain information to represent a user in the system, while instances of the latter represent bank accounts. We do not detail the full definition of these types.

However, we assume that an **account** record type contains a **user** element in the **owner** field to represent the owner of the account. So the owner of the account element **myAccount** : **account** is accessed by **myAccount.owner**.

We also assume that **user** is an equivalence type in *Standard ML*, so that its elements may be compared using the boolean valued comparison function  $=$ .

We assume a constant **currentUser** : **user** that represents the current user who is the subject of the account search.

- The database is represented in *Standard ML* as an array of accounts,

$$db : \text{account array}$$

Following the *Standard ML* API, the array is 0-indexed, with the  $i$ th element accessed as

$$sub(db, i)$$

and the size of the array given as

$$length\ db$$

Assume we have an array of size **Size**, called **accounts**. Although *Standard ML* arrays are mutable, for the purposes of this example, we will consider **db** to be an immutable value. Consequently, it will be represented in our logic as a constant.

- We assume a state reference **counter** : **int ref**, to be used as a counter in searches through the database.

We take a predicate

$$\text{allAccountsAt}(u : \text{user}, x : \text{account list}, y : \text{int})$$

whose meaning is that  $x$  is a list of all accounts found to be owned by the user  $u$ , up to the point  $y$  in the database  $\text{db}$ . The predicate is defined by the following axioms in  $\mathcal{AX}$

$$\begin{aligned} \forall u : \text{user} \bullet \forall x : (\text{account list}) \bullet \forall y : \text{int} \bullet (\text{allAccountsAt}(u, x, y) \rightarrow \\ (\forall z : \text{int} \bullet z \leq y \rightarrow \text{sub}(\text{db}, z).\text{owner} = u)) \end{aligned} \quad (4)$$

$$\begin{aligned} \forall u : \text{user} \bullet \forall x : (\text{account list}) \bullet \forall y : \text{int} \bullet \\ ((y < (\text{length db}) - 1) \wedge \text{sub}(\text{db}, y + 1).\text{user} = u \wedge \text{allAccountsAt}(u, x, y)) \rightarrow \\ \text{allAccountsAt}(u, \text{sub}(\text{db}, y + 1) :: x, y + 1) \end{aligned} \quad (5)$$

$$\begin{aligned} \forall u : \text{user} \bullet \forall x : (\text{account list}) \bullet \forall y : \text{int} \bullet \\ (y < (\text{length db}) - 1 \wedge \neg \text{sub}(l, y + 1).\text{user} = u \wedge \text{allAccountsAt}(u, x, y)) \rightarrow \\ \text{allAccountsAt}(u, x, y + 1) \end{aligned} \quad (6)$$

$$\forall u : \text{user} \bullet \forall y : \text{int} \bullet y = 0 \rightarrow \text{allAccountsAt}(u, [], y) \quad (7)$$

Observe that these are intuitionistic axioms that will be used in Hoare logic.

We develop a program that satisfies the following property: Given a user's details, it is *possible* to obtain a list of all accounts held at the bank by the user, by searching through the database. This is formally stated as the following **Specification**:

$$\begin{aligned} \exists y : (\text{account list}) \bullet \text{listallAccounts}(\text{currentUser}, y, \text{counter}_f) \wedge \\ (\text{counter}_f < (\text{length db}) - 1) = \text{false} \end{aligned} \quad (8)$$

The post-condition requirement of  $\text{counter}_f$  signifies that a complete search of the database should be completed by the program.

We also have an axiom that is needed because, since we are working in intuitionist logic, we do not automatically have the law of the excluded middle.

$$y < (\text{length db}) - 1 \rightarrow \text{sub}(l, y + 1).\text{owner} = u \vee \neg \text{sub}(l, y + 1).\text{owner} = u \quad (9)$$

From these we can eventually prove the theorem

$$\begin{aligned} \vdash \text{counter} := !\text{counter} + 1; \\ \text{while } !\text{counter} < (\text{length db}) - 1 \text{ do } \text{counter} := !\text{counter} + 1; \text{done} : \\ \exists y : (\text{account list}) \bullet \text{allAccountsAt}(\text{currentUser}, y, \text{counter}_f) \wedge \\ (\text{counter}_f < (\text{length db}) - 1) = \text{false} \end{aligned} \quad (10)$$

The program that we extract by a special *extraction function* satisfies the specification (8). We omit details of the extraction function. Suffice it to say that it works recursively in the same spirit as we extracted our programs in Section 4.1.

Essentially, when viewed as a specification of a return value, the specification (8) requires a program that, given a user’s details, will search through a database to obtain all accounts held at the bank by the user, and then return this list.

## 5 Programs and proofs

So far we have seen how to obtain programs from proofs in constructive systems of logic. Therefore we could conclude that all proofs are already programs, or at least, that every proof in (constructive) logic contains a program.<sup>11</sup>

What if we were to write the program first? Would we automatically have a proof? The answer is obviously “No!” if we simply write computer programs as many people do. However, a thoughtful computer programmer would wish to know that the program written would do what it was expected to do, that is to say, would meet its specification.<sup>12</sup> Therefore, as part of the task of writing the program, a proof should be produced at the same time.

The approach that we have presented shows how to accomplish both of these tasks at the same time. It does not require a separate investigation to produce a proof that the program will be correct.

From a practical point of view it is sometimes obvious how to write the proof. I studied a program for quicksort.<sup>13</sup> Then I wrote a proof corresponding to the program and extracted a program from it. The resulting program was essentially the quicksort program from which I had started. However I have not yet been able to formalize the procedure that I used in producing the proof from the program. It would appear that one needs to know the *algorithm* rather than the program in order to construct the proof. This in itself indicates that one also needs to know that the program is a *correct* implementation of the algorithm. But this is work for the future.

## 6 Conclusion

The techniques we have presented here are based on a variant of Gabbay’s labelled deductive systems [7]. Our logical rules are of the form

$$\frac{\text{Logical context, State, Curry-Howard term } \vdash \text{ Formula}}{\text{New Logical context, New State, New Curry-Howard term } \vdash \text{ New Formula}}$$

<sup>11</sup> The restriction to constructive systems of logic is essential for us.

<sup>12</sup> This is a very serious issue when it comes to the control of powerful systems, in particular, the control of nuclear weapons.

<sup>13</sup> This was inspired by looking at work of Helmut Schwichtenberg on program extraction in [1].

although the actual order may vary. Further, each of the items on the lower line may depend on, that is to say, be functions of, any or all of those on the top line, and of course there may be two or more sequences on the top line.

The semantics of these rules will depend on the structures that we are using. Also the interpretation of the informal terms: Logical context, State, etc. will also vary.

What seems to be most important is that we have extended the notion of logic in two ways. First of all we now have programs or other constructions (for example, specifications) interacting with the standard logical connectives. Secondly, the context of the logic may change in the course of a proof. This certainly happens in the context of algebraic specifications. Thirdly, we are now discussing logics (plural) and we arrive at such logics by an analysis of a technical setting. This seems to me to be following Aristotle's approach of looking at the real world, or a small part of it, and then abstracting the logical principles that work in that arena. But we have come a long way from the "one true logic" that I mentioned at the beginning of the introduction!

## Acknowledgements

Many years ago Georg Kreisel said that I should not work in proof theory, which is the setting of the work described here. I have not taken that advice but am grateful to him for his enthusiasm and stimulation over many decades. I was originally introduced to the area of program extraction when visiting my old friend Anil Nerode in Cornell in the 1990s where we studied Girard's thesis [8], essentially published as [9]. Later John Shepherdson (Bristol) and I extended the Curry-Howard terms to cover all the standard logical connectives directly, that is to say, without going through what I regard as the tortuous translations of Girard into higher order logic. Martin Wirsing of Ludwig-Maximilians Universität, Munich, and I, inspired by work of Martin and his student Hannes Peterreins, began to produce elegant rules for the context of algebraic specifications. From then on, with my former student Iman Poernomo, now at King's College, London we have extended the logical systems even further to structured specifications and then to imperative programming. The idea of the Curry-Howard protocol is due to Iman Poernomo. I am extremely grateful to all of these colleagues for their friendship, ideas and stimulation.

## References

1. Ulrich Berger and Helmut Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, IN, USA, October 1994*, pages 77–97, 1995.
2. CoFI Language Design Task Group on Language Design. *CASL, The Common Algebraic Specification Language, Summary, 25 March 2001*, March 2001. Available at <http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/> (accessed 3.01.2005).

3. John Newsome Crossley. What is mathematical logic? A survey. Tutorial at this conference.
4. John Newsome Crossley, Iman Poernomo, and Martin Wirsing. Extraction of structured programs from specification proofs. In Didier Bert, Christine Choppy, and Peter Mosses, editors, *Workshop on Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 419–437, 1999.
5. John Newsome Crossley and John Cedric Shepherdson. Extracting programs from proofs by an extension of the Curry-Howard process. In John Newsome Crossley, Jeffrey B. Remmel, Richard A. Shore, and Moss Eisenberg Sweedler, editors, *Logical Methods: In honor of Anil Nerode's Sixtieth Birthday*, pages 222–288. Birkhäuser, Boston, MA, 1993.
6. Michael Dummett. *Elements of Intuitionism*. Oxford University Press, 1977.
7. Dov Gabbay. *Labelled deductive systems*. Oxford University Press, 1996.
8. Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, Paris, 1972.
9. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge University Press, Cambridge, 1989.
10. Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–80, 1969.
11. William Howard. The formulae-as-types notion of construction. In John Roger Hindley and Jonathan Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1969.
12. Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.
13. Hannes Peterreins. *A natural-deduction-like calculus for structured specifications*. PhD thesis, Ludwig-Maximilians-Universität, München, 1996.
14. Iman Hafiz Poernomo, John Newsome Crossley, and Martin Wirsing. *Adapting proofs-as-programs*. Springer, New York, 2005 (forthcoming).
15. Martin Wirsing. Structured specifications: Syntax, semantics and proof calculus. In Manfred Broy, editor, *Informatik und Mathematik, Festschrift für F. L. Bauer*, pages 269–283. Springer, Berlin, 1991.