Protocols between Programs and Proofs

Iman Poernomo[⋆] and John N. Crossley[⋆⋆]

School of Computer Science and Software Engineering
Monash University, Clayton, Victoria, Australia 3168
{ihp,jnc}@csse.monash.edu.au

Abstract. In this paper we describe a new protocol that we call the *Curry-Howard protocol* between a theory and the programs extracted from it. This protocol leads to the expansion of the theory and the production of more powerful programs. The methodology we use for automatically extracting "correct" programs from proofs is a development of the well-known Curry-Howard process. Program extraction has been developed by many authors (see, for example, [9], [5] and [12]), but our presentation is ultimately aimed at a practical, usable system and has a number of novel features. These include

1. a very simple and natural mimicking of ordinary mathematical practice and likewise the use of established computer programs when we obtain programs from formal proofs, and
2. a conceptual distinction between *programs* on the one hand, and *proofs of theorems* that yield programs on the other.

An implementation of our methodology is the `Fred` system.[1] As an example of our protocol we describe a constructive proof of the well-known theorem that every graph of even parity can be decomposed into a list of disjoint cycles. Given such a graph as input, the extracted program produces a list of the (non-trivial) disjoint cycles as promised.

1 Introduction

In constructing new proofs, mathematicians usually use existing proofs and definitions by means of abbreviations, and also by giving names to objects (such as constants or functions) that have already been obtained or perhaps merely proved to exist. These are then incorporated into the (relatively simple) logical steps used for the next stage of the proof. In this paper we mimic this approach by re-using programs that have been guaranteed as "correct"[2] and giving names,

⋆ Research partly supported by ARC grant A 49230989.

⋆⋆ The authors are deeply indebted to John S. Jeavons and Bolis Basit who produced the graph-theoretic proof that we use.

[1] The name `Fred` stands for "Frege-style dynamic [system]". `Fred` is written in C++ and runs under Windows 95/98/NT only because this is a readily accessible platform. See `http://www.csse.monash.edu.au/fred`

[2] We write "correct" because the word is open to many interpretations. In this paper the interpretation is that the program meets its specifications.

say f, to any new functions whose programs we extract from proofs of closed formulae of the form $\forall x \exists y A(x, y)$. This is done by the well-known Curry-Howard isomorphism in order to make $A(\bar{n}, f(\bar{n}))$ true, where $\bar{n}$ denotes the numeral (name) for the number n. (See e.g. [11] or [7].) These new functions are therefore also "correct" – provided, of course, that the theory is consistent. The *Curry-Howard protocol* which we introduce is a means of preserving this correctness and seamlessly integrating the functions these programs compute into the logic.

There have been a number of systems exploiting the Curry-Howard notion of formulae-as-types. In particular we mention: Hayashi's system *PX* [9], Constable's *NuPRL* [5], [6] and Coquand and Huet's system *Coq* [12]. The first of these uses a logic different from traditional logics which is not widely known; the last two use a (higher-order) hybrid system of logic and type theory. We discuss their relation to our work in our conclusion in Section 6.

Our protocol works with a logical type theory and a computational type theory. We first describe a fairly general *logical* type theory (LTT) whose types are formulae of a first-order many-sorted calculus and whose terms represent proofs. We next describe our *computational* type theory (CTT) and then the Curry-Howard protocol between elements of the LTT and elements of the CTT. The protocol involves an extraction map between terms of the LTT and programs in the CTT, and a means of representing programs of the CTT in the LTT (including new programs that have been extracted). Our methodology differs from those mentioned above (but see also our conclusion, Section 6) in the following ways. It allows us to

1. first of all, both mimic ordinary mathematical practice in the construction of new mathematics and use established computer programs when we extract programs from formal proofs;
2. use a standard (first-order) many-sorted logic, so that our proofs are easy to follow;
3. establish a conceptual distinction between programs and proofs of theorems about programs; and
4. build a *dynamic* system that is "open" in the sense that new axioms and functions may constantly be added and re-used.

Our division of the labour of proving theorems and running programs between the LTT and the CTT means that

1. when a new function is added to the system, its full implementation details (in the CTT) are usually not available to the LTT,
2. programs that have been proved (say by a separate verification system) to possess a required property may be added to the system and reasoned about in the LTT.

In order to be quite specific, we describe the protocol over ordinary intuitionistic logic (as LTT) and the Caml-light variant of ML (as CTT). However, the protocol can be used over more complex constructive deductive systems and other computational type theories. We briefly describe some examples.

We have built a software system, currently called `Fred`, for “Frege-style dynamic [system]”, as an implementation of our system. At present `Fred` produces programs in *ML*. It has a LaTeX output feature, so that we can easily include proofs written in `Fred` in a document such as the present paper.

Remark 1. Of course there always remains the global question (raised at the conference by Alberto Pettorossi) of whether the proofs that we use are correct and whether the software we use preserves correctness. We are almost as vulnerable as any mathematician concerning the correctness of a proof. We say “almost” because our proofs are formalized and therefore checking the steps is a mechanical process. However the extraction process uses software that may be unreliable. We minimize the effects of this because our procedures are simple and simply syntactical. Ideally we would run the process on our own software to check it, but this is, at present, a daunting task.

Example. We demonstrate the system by means of a constructive proof that every even parity graph can be decomposed into a list of disjoint cycles and then extract a program that computes such a list of the non-trivial disjoint cycles from a given graph.

2 The Logical Type Theory and the Computational Type Theory

For definiteness, we take our logic to be a standard intuitionistic natural deduction system. However, the techniques we shall describe apply to other systems of natural deduction, including the labelled deductive systems of [8] and [17]. See Section 5 for further discussion. We work with a many-sorted, first order logic. Many higher-order concepts can be formulated quite adequately in such theories. For example we employ lists which simply comprise a new sort, *List*, with associated functions (for *append*, etc.). We are also able to simulate a limited form of quantification over predicates by using a template rule (see Section 4.1). One can go much further (as Henkin suggests in a footnote in [10]) and, using the protocol outlined here, we are able to reason about modules and structured specifications. (See our paper [8] and section 5 below).

Our reasons for using a many-sorted first order logic rather than higher types (as does, for example, Martin-Löf, [16]) include 1. the fact that we maintain a separation between the *LTT* and the *CTT* whereas others want to create a unified theory and 2. the desire to make our system very user friendly and use the fact that first order logic is familiar to many people.

A theory Th has a collection of sorts generated by a base set of sorts $\mathcal{S}_{Th}$. Its signature $\mathtt{sig}(Th)$ contains names for a set of function symbols (constructors) of the appropriate sorts and for a set of relation symbols. There is a set of Harrop axioms Ax_{Th}, built using only symbols from the signature.

Initial Rules

$$\frac{}{x : A \vdash x : A}\ (\text{Ass I}) \qquad \frac{}{\vdash () : A}\ (\text{Ax I})$$

when $A \in Ax_{Th}$

Introduction Rules

$$\frac{\vdash d : B}{\vdash \lambda x : A.d : (A \rightarrow B)}\ (\rightarrow \text{I}) \qquad \frac{\vdash d : A \quad \vdash e : B}{\vdash (d, e) : (A \wedge B)}\ (\wedge \text{ I})$$

$$\frac{\vdash d : A}{\vdash (\pi_1, d) : (A \vee B)}\ (\vee_1 \text{ I}) \qquad \frac{\vdash e : B}{\vdash (\pi_2, e) : (A \vee B)}\ (\vee_2 \text{ I})$$

$$\frac{\vdash d : A}{\vdash \lambda x : s.d : \forall x : sA}\ (\forall \text{ I}) \qquad \frac{\vdash d : A[t/x]}{\vdash (t, d) : \exists x : sA}\ (\exists \text{ I})$$

Elimination Rules

$$\frac{\vdash d : (A \rightarrow B) \quad \vdash r : A}{\vdash (dr) : B}\ (\rightarrow \text{E}) \qquad \frac{\vdash d : (A_1 \wedge A_2)}{\vdash \pi_i(d) : A_i}\ (\wedge \text{ E}_i)$$

$$\frac{\vdash d : \forall x : sA}{\vdash dt : A[t/x]}\ (\forall \text{ E}) \qquad \frac{\vdash d : \bot}{\vdash dA : A}\ (\bot \text{ E})$$

provided A is Harrop

$$\frac{\vdash d : C \quad \vdash e : C \quad \vdash f : (A \vee B)}{\vdash \mathsf{case}(x : A.d : C, y : B.e : C, f : (A \vee B)) : C}\ (\vee \text{ E})$$

$$\frac{d : \exists x : sA \quad \vdash e : C}{\vdash \mathsf{select}(z : s.y : A[z/x].e : C, d : \exists x : sA) : C}\ (\exists \text{ E})$$

Conventions: 1. The usual *eigenvariable* restrictions apply in ($\forall$ I) etc.
2. We assume that all undischarged hypotheses or assumptions are collected and listed to the left of the $\vdash$ sign although we shall usually not display them.

Fig. 1. Logical Rules and Curry-Howard terms. (We omit the types as much as possible for ease of reading.)

Remark 2. Harrop formulae are defined as follows:

1. An atomic formula or $\bot$ is a Harrop formula.
2. If A and B are Harrop, then so is $(A \wedge B)$.
3. If A is Harrop and C is any formula, then $(C \rightarrow A)$ is Harrop.
4. If A is a Harrop formula, then $\forall x A$ is Harrop.

The axioms one would normally employ in (constructive) mathematics are Harrop formulae so the restriction is a natural one. It also has a significant effect on reducing the size of our extracted programs (see, e.g. [7]).

Because we are interested in expanding theories as we prove more and more theorems, we view the proof process as the construction of a chain of inclusive *theories*: $Th_1 \subseteq Th_2 \subseteq \ldots$, where we write $Th \subseteq Th'$ when $\mathcal{S}_{Th} \subseteq \mathcal{S}_{Th'}$,

$\mathtt{sig}(Th) \subseteq \mathtt{sig}(Th')$ and $Ax_{Th} \subseteq Ax_{Th'}$. Throughout this paper, we assume that *current* is an integer and $Th_{current}$ is the theory we are using to prove theorems. We shall show how to add an axiom, relation, function or sort to the theory to form a new theory $Th_{current+1} \supseteq Th_{current}$. It is in this sense that our system is dynamic or open.

2.1 The Logical Type Theory

Our logical type theory (LTT) is a means of encoding proofs in the natural deduction calculus. The types are many-sorted formulae for a theory and "Curry-Howard terms" (or proof-terms) are essentially terms in a lambda calculus with dependent sum and product types that represent proofs. The rules of our LTT are presented in Fig. 1.

Because Curry-Howard terms are terms in a form of lambda calculus, they have *reduction rules* whose application corresponds to proof normalization. (See [7] and [1] for the full list of these reduction rules.)

Note that, unlike other systems for program extraction, terms from a theory that occur within a Curry-Howard term are not reduced by these reduction rules. So, for instance, the normalization chain

$$\begin{array}{ll} ((\lambda x.(x+x,()) : \forall x : Nat\ \exists y : Nat\ 2.x = y)3) : \exists y : Nat\ 2.3 = y \\ \text{reduces to} \qquad (3+3,()) : \exists y : Nat\ 2.3 = y \end{array}$$

but continues no further – the term $3+3$ is treated as a constant. The term $3+3$ can only be evaluated when mapped into a programming language – a computational type theory.

In general, a LTT is defined as follows.

Definition 1 **(***LTT***).** *A* Logical Type Theory *LTT*, $\mathsf{L} = \langle L, PT, :, \rhd\rangle$, *consists of*

- *a deductive system* $L = \langle Sig_L, \vdash_L\rangle$,
- *a lambda calculus* PT,
- *a typing relation, written " : ", defined between terms of* PT *and formulae of* L, *such that*

$$(\textit{there is a } p \in PT \textit{ such that } \vdash p : A) \Leftrightarrow \vdash_L A$$

- *a normalization relation* $\rhd$, *defined over terms of* PT, *which preserves the type, i.e.*

$$\vdash p : A \textit{ and } p \rhd p' \Rightarrow \vdash p' : A.$$

Remark 3. The typing relation and the normalization relation above are assumed to be constructive.

2.2 The Computational Type Theory

To obtain our computational type theory we take any typed programming language that contains the simply typed lambda calculus as a subset (possibly interpreted into it). Any modern functional language (e.g., Scheme, *ML*, Pizza-Java) qualifies. We have chosen (the Caml-light version of) *ML* as our *CTT*.

An important property for the *CTT* for our purposes is the notion of extensional equivalence of programs, written $\mathtt{f} \equiv_{ML} \mathtt{g}$, which holds when two programs always evaluate to the same value.

In general, a *CTT* should satisfy the following:

Definition 2 (*CTT*). *A* Computational Type Theory *CTT,* $\mathsf{C} = \langle C, :, Sig, \rhd_{\mathsf{C}}, \equiv_{\mathsf{C}} \rangle$*, is such that*

- *C is a typed programming language.*
- *the typing relation for C, written "$:$", is defined over the terms of C and the sorts of Sig.*
- *$\rhd_{\mathsf{C}}$ is the evaluation relation for C.*
- *the $\equiv_{\mathsf{C}}$ relation holds between terms $\mathtt{a}, \mathtt{b}$ of C such that $\mathtt{a} \rhd_{\mathsf{C}} \mathtt{c}$ and $\mathtt{b} \rhd_{\mathsf{C}} \mathtt{c}$ implies $\mathtt{a} \equiv_{\mathsf{C}} \mathtt{b}$.*

We shall, in general use teletype font for actual programs, e.g. `f`, while the interpretation will be in normal mathematical italics.

3 Protocol between the *CTT* and the *LTT*

The idea here is that, if the Curry-Howard protocol holds between the *CTT* and the *LTT*, then we can extract correct programs from proofs.

We now describe our interpretation of the extraction of correct programs. We define a map Value from programs of the *CTT* to terms of the theory in the *LTT*. This map is used to represent the values of programs in the *LTT*. We say that a program `f` is an *extended realizer* of a theorem, A, when the value of the program $\mathsf{Value}(\mathtt{f})$ satisfies the theorem A. The definition of extended realizability depends on the choice of *LTT* and *CTT*. For a definition for an intuitionistic *LTT*, see [3] or [8]. The most well-known example of an extended realizer is a program `f` extracted from a proof of $\forall x \exists y A(x, y)$ such that $A(\bar{n}, \mathsf{Value}(\mathtt{f})(\bar{n}))$ is true.

We also define a map program from terms of the theory to programs of the *CTT*. This map allows us to use functions of the theory in programs of the *CTT*. We define program so that $\mathsf{Value}(\mathsf{program}(c)) = c$.

Correct extraction is then defined *via* a map extract from the *LTT* to the *CTT* which, given a Curry-Howard term $p : A$ gives a program $\mathsf{extract}(p)$ which is an extended realizer of A. The definition of extract involves a type simplification map ϕ – we require that the type of $\mathsf{extract}(p)$ is $\phi(A)$.

The map program is used by extract because Curry-Howard terms involve functions from $Th_{current}$. So any extraction map (from Curry-Howard terms to *CTT* programs) must extend a map (such as program) from $\mathcal{F}_{Th_{current}}$ to the

$$\mathsf{extract}(x : A) = \begin{cases} () & \text{if } \mathfrak{H}(\mathrm{A}) \\ \texttt{x} & \text{otherwise} \end{cases}$$

$$\mathsf{extract}(() : A) = () \qquad \mathsf{extract}(dA : A) = ()$$

$$\mathsf{extract}((\lambda x : A.d) : (A \to B)) = \begin{cases} \mathsf{extract}(d) & \text{if } \mathfrak{H}(A) \\ () & \text{if } \mathfrak{H}(B) \\ \texttt{fun x} \ - > \mathsf{extract}(d) & \text{otherwise} \end{cases}$$

$$\mathsf{extract}((a, b) : (A \wedge B)) = \begin{cases} \mathsf{extract}(a) & \text{if } \mathfrak{H}(A) \\ & \text{and not } \mathfrak{H}(B) \\ \mathsf{extract}(b) & \text{if } \mathfrak{H}(B) \\ & \text{and not } \mathfrak{H}(A) \\ () & \text{if } \mathfrak{H}((A \wedge B)) \\ (\mathsf{extract}(a), \mathsf{extract}(b)) & \text{otherwise} \end{cases}$$

$$\mathsf{extract}((\lambda x : s.d) : \forall x : SA) = \begin{cases} () & \text{if } \mathfrak{H}(\forall x : S\ A) \\ \texttt{fun x} \ - > \mathsf{extract}(d) & \text{otherwise} \end{cases}$$

$$\mathsf{extract}((t, d) : \exists x : s\ A) = \begin{cases} \mathsf{program}(t) & \text{if } \mathfrak{H}(A) \\ (\mathsf{program}(t), \mathsf{extract}(d)) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathsf{extract}((\pi_1, d : A_1) : (A_1 \vee A_2)) &= \texttt{inl}(\mathsf{extract}(d)) \\ \mathsf{extract}((\pi_2, d : A_1) : (A_1 \vee A_2)) &= \texttt{inr}(\mathsf{extract}(d)) \end{aligned}$$

$$\begin{aligned} \mathsf{extract}((d : (A \to B)) \\ (r : A) : B) \end{aligned} = \begin{cases} \mathsf{extract}(d) & \text{if } \mathfrak{H}(A) \\ () & \text{if } \mathfrak{H}(B) \\ \mathsf{extract}(d)\mathsf{extract}(r) & \text{otherwise} \end{cases}$$

$$\mathsf{extract}(\pi_i(d : (A_1 \wedge A_2)) : A_i) = \begin{cases} () & \text{if } \mathfrak{H}(A_i) \\ \mathsf{extract}(d) & \text{if } \mathfrak{H}(A_j), j \neq i \\ & \text{and not } \mathfrak{H}(A_i) \\ \pi_{\mathbf{i}}(\mathsf{extract}(d)) & \text{otherwise,} \\ & \text{where } \pi_{\mathbf{1}} \text{ is } \texttt{fst} \\ & \text{and } \pi_{\mathbf{2}} \text{ is } \texttt{snd} \end{cases}$$

$$\mathsf{extract}((d : \forall x : s\ A(x))(r : s) : A[r/x]) = \begin{cases} () & \text{if } \mathfrak{H}(A[d/x]) \\ \mathsf{extract}(d)r & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\mathsf{extract}(\mathsf{case}(x : A.d : C, \\ &y : B.e : C, f : (A \vee B)) : C) \end{aligned} = \begin{cases} () & \text{if } \mathfrak{H}(C) \\ (\texttt{function inl(x)} \ - > \mathsf{extract}(d) & \\ |\texttt{inr(y)} \ - > \mathsf{extract}(e))\mathsf{extract}(f) & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\mathsf{extract}(\mathsf{select}(z : s.y : A[z/x]. \\ &e : C, d : \exists x : s.A) : C) \end{aligned} = \begin{cases} () & \text{if } \mathfrak{H}(C) \\ (\texttt{fun x}- > \mathsf{extract}(e))\mathsf{extract}(d) & \text{if } \mathfrak{H}(A) \\ (\texttt{function z y}- > \mathsf{extract}(e)) & \text{otherwise} \\ \qquad\qquad \mathsf{extract}(d) & \end{cases}$$

Fig. 2. The definition of extract except we have written $\mathfrak{H}(A)$ for "A is Harrop". The definition of ϕ for our *LTT* and *CTT* is given in Fig. 3

$$\phi(A) = () \qquad \text{if } Harrop(A)$$

$$\phi(A \wedge B) = \begin{cases} \phi(A) & \text{if } Harrop(B) \\ \phi(B) & \text{if } Harrop(A) \\ \phi(A)*\phi(\mathtt{B}) & \text{otherwise} \end{cases}$$

$$\phi(A \vee B) = (\phi(A), \phi(B))\ \mathtt{DisjointUnion}$$

$$\phi(A->B) = \begin{cases} \phi(B) & \text{if } Harrop(A) \\ \phi(A)->\phi(B) & \text{otherwise} \end{cases}$$

$$\phi(\forall x : s\ A) = s->\phi(A)$$

$$\phi(\exists x : s\ A) = \begin{cases} s & \text{if } Harrop(A) \\ s*\phi(\mathtt{A}) & \text{otherwise} \end{cases}$$

where we define

$$\mathtt{type}('\mathtt{a},'\mathtt{b})\ \mathtt{DisjointUnion} \;=\; \mathtt{inl\ of}\ '\mathtt{a} \mid \mathtt{inr\ of}\ '\mathtt{b}\ ;;$$

Fig. 3. The definition of the map ϕ for our *LTT* and *CTT* used in defining extract in Fig. 2. (H is the unit type of (). The formula $\perp$ is Harrop so comes under the first clause.)

CTT. The semantic restrictions placed on program ensure that the theorems we prove about functions are actually true of (the results of) their corresponding programs.

The details of the maps for our *LTT* and *CTT* are given in Figs 3 and 2.

The fact that extract produces extended realizers entails that Theorem 1 holds.

Theorem 1. *Given a proof $p : \forall x : s_1 \exists y : s_2 A(x, y)$ in the logical type theory, there is a program* extract(f) *of ML type* $\mathtt{s}_1 -> \mathtt{s}_2$ *in the computational type theory ML such that* $A(x : s_1, \mathsf{Value}(\mathsf{extract}(f))(x) : s_2)$.

The protocol between our *LTT* and *CTT* is generalized in the following way.

Definition 3. *A* Curry-Howard protocol *holds between the LTT and the CTT when:*

1. *Every sort in the LTT is a type name in the CTT.*
2. *The signature of the CTT is included within that of the logic of the LTT. That is to say:* $Th_{current} \subseteq Sig_{\mathsf{C}}$.
3. *Each term,* c*, of the CTT, is represented by a term* Value(c)*, of the Sig of the LTT.*

4. *There is an extraction map* $\mathsf{extract} : LTT \rightarrow CTT$ *such that, given a proof* $d : A$ *then* $\mathsf{extract}(d)$ *is in the* CTT, *is of type* $\phi(A)$ *and is an extended realizer for* A.
5. *There is a map* $\mathsf{program} : \mathtt{sig}(\mathcal{F}_{Th_{current}}) \rightarrow CTT$, *such that if* f *is a function of* $Th_{current}$ *and* A *is its sort, then* $\mathsf{program}(f)$ *is a program in the CTT with type* A.
6. $\mathsf{program}$ *can be extended to an interpretation* $\mathsf{program}^*$ *of* $\mathtt{sig}(Th_{current})$ *such that every axiom in* $Ax_{Th_{current}}$ *is satisfied.*
7. *We suppose that equality between elements of* A *is represented by* $=: A \times A \in \mathcal{R}_{Th_{current}}$. *Then the relations* $\equiv_{\mathsf{C}}$ *and* $=$ *are preserved by the interpretations* $\mathsf{program}^*$ *and* Value *in the following way*

$$\begin{aligned} \mathsf{c}_1 \equiv_{\mathsf{C}} \mathsf{c}_2 &\Leftrightarrow \vdash_L \mathsf{Value}(\mathsf{c}_1) = \mathsf{Value}(\mathsf{c}_2) \\ \mathsf{program}^*(t_1) \equiv_{\mathsf{C}} \mathsf{program}^*(t_2) &\Leftrightarrow \vdash_L t_1 = t_2 \end{aligned}$$

8. *The maps* $\triangleright$ *and* $\mathsf{extract}$ *are such that the following diagram commutes:*

$$\begin{tikzcd} LTT: & t : T \arrow[r, "\triangleright"] \arrow[d, "\mathsf{extract}"'] & t' : T \arrow[d, "\mathsf{extract}"'] \\ CTT: & p : \phi(T) \arrow[r, phantom, "\equiv_{\mathsf{C}}"] & p' : \phi(T) \end{tikzcd}$$

Remark 4. The diagram of 6. simply states that, if a proof t of T can be simplified to a proof t' of T, then both proofs should yield equivalent programs (in the sense that they give identical results).

Remark 5. It can be seen that the LTT and CTT we are working with satisfy this protocol. For example, 6. is satisfied because we take $\equiv_{ML}$ to be equivalence of the normal forms of the values of the elements of A in ML.

The Curry-Howard protocol guarantees that there will always be agreement between the theory's axioms for function terms and the CTT definitions of the corresponding programs. Note that the CTT programs can be defined in whatever way we wish. The user is only required to guarantee that these programs are correct in the sense of satisfying the Curry-Howard protocol.[3] Thus we retain a distinction between the *extensional meaning* of functions – given by the axioms they must satisfy, and their *intensional meaning* – how they are coded in the computational type theory.

[3] This means that the final program is only going to be as correct as the programs introduced by the user are. Of course the programs extracted from proofs are guaranteed by the existence of the proof to be correct – provided that the axioms are consistent.

4 Enlarging the Theories

The current logical theory, $Th_{current}$, being used in a proof may be enlarged by adding new function constant symbols or new sorts or even new axioms, provided, of course, that consistency is maintained, to a new theory $Th_{current+1}$.

If, into the *CTT*, we introduce a new program `f`, say, that is known to satisfy some property P, then we add a constant f of the same sort and the property $P(f)$ to the theory, $Th_{current}$, in the *LTT*, yielding a new theory $Th_{current+1}$, and with the map program extended by $\mathsf{program}(f) = \mathtt{f}$.

For example, the function for determining the length of a list (which gives a number of sort Nat from an element of sort $List(\alpha)$)

$$length_\alpha : List(\alpha) \to Nat$$

is given by the following axioms (where concatenating the element a on the front of the list l is written $\langle a \rangle :: l$ in our *LTT*)

$$\begin{aligned} &length_\alpha(\epsilon_\alpha) = 0 \\ &length_\alpha(\langle a \rangle :: l) = \overline{1} + length_\alpha(l) \end{aligned}$$

These axioms define a function $length_\alpha$ that is total on $List\ (\alpha)$. If we added them to the current theory of the *LTT*, then we would have to add some function `length` $= \mathsf{program}(length_\alpha)$ and guarantee that the Curry-Howard protocol is preserved. We might therefore add a corresponding program in the *CTT* (where concatenating the element `a` on the front of the list `l` is written `a :: l` in our *CTT*):

```
let rec length = function
                    [ ] -> 0
                    | a::l -> 1+length(l)
;;
```

and in this way the Curry-Howard protocol would be preserved.

Note that, in larger proofs, when we are anxious to increase efficiency and reduce the size of the program, we may choose to implement the program in a manner different from that suggested by the axiomatization.

4.1 New Predicates and Functions

In ordinary mathematics, we often abbreviate a formula by a predicate. This is a useful way of encapsulating information, aids readability and helps us to identify and use common "proof patterns". In `Fred`, we extend our logical calculus by means of a meta-rule of predicate abbreviation for a formula F (with zero or more occurrences of the variable x) by:

$$set\ P(x) \equiv F$$

Note that we do not allow predicates over predicates.

We introduce a new function letter f of type F and the following structural meta-rule, *Template*, for any Curry-Howard term $q(x)$ where x is a Curry-Howard term of type P:

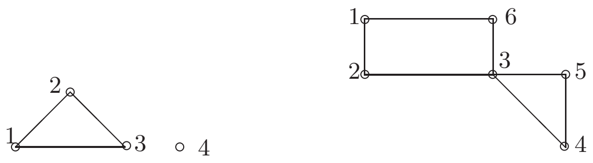

Fig. 4. Two sample graphs

$$\text{If} \qquad set\ P(x) \equiv F, \text{ then} \qquad \frac{f : F \quad q(x : P) : Q(P)}{q(f) : Q(F)}$$

That is, if we have formula Q that is dependent on the formula P, then we may substitute the formula F for P in Q. The converse is also a rule. Of course in doing this we must avoid all clashes of variable. *Template* is a means of abstracting a proof over a "formula variable". Defining it as a structural meta-rule is a means of avoiding higher order quantification of formula variables.[4]

Example (*cont.*). **Representing graphs in the formal system.** We consider a standard axiomatization of the theory of graphs, $\mathcal{G}$, in terms of vertices and edges. For the proof of Theorem 2 below, we assume that $Th_{current}$ contains $\mathcal{G}$. The vertices will be represented by positive integers and the graph encoded by a list of (repetition-free) lists of the neighbours of those vertices.(Of course, not all elements of the sort: lists of lists of natural numbers, correspond to graphs.)

Consider the left hand graph with four vertices in Fig. 4. This is represented by the four element list of lists of neighbours $\langle\langle 1,2,3\rangle, \langle 2,1,3\rangle, \langle 3,1,2\rangle, \langle 4\rangle\rangle$ where each element is of sort $List(Nat)$.

These properties are expressible in our formal system $\mathcal{G}$ with the aid of certain extra function symbols. Here is the list of required functions in $\mathcal{F}_{\mathcal{G}}$ and the associated axioms. (We easily arrange that each function is provably total in the formal system by introducing and sometimes using a "default value".) All formulae are considered to be universally closed. We note that appropriate CTT programs must be provided according to the Curry-Howard protocol in definition 3 in section 3.

1. For each sort α, a binary function, $member_\alpha$, of two arguments: a natural number n and a list of elements of sort α. The function computes the nth member of the list. The definitions for the cases $\alpha = Nat$, $List(Nat)$ are given in [13].
2. List successor, S. This function takes a list of natural numbers as argument, adds one to each number in the list and returns the revised list.
3. Position function: $listpos(n, l)$ gives a list of all the *positions* the number n takes in the list l. If the list l does not contain n then the empty list is returned. We take the head position as 0, so position k corresponds to the $k + 1^{st}$ member of the list.

[4] As in *Coq*, see [12], this could also be achieved by creating a new sort (for logical formulae), or with a universe hierarchy as in Martin-Löf [16].

4. Initial segment of a list: $\mathit{initlist}(k, l)$ computes the list consisting of the first $k+1$ elements of the list l; if $k+1 > \mathit{length}(l)$ then the list l is returned.
5. Tail segment of a list: $\mathit{tail}(l, n)$ takes a list l (of natural numbers) and a number n as arguments and computes the list obtained by deleting the first n members of l.

We now use new predicates and functions to prove our graph-theoretic result. We set a predicate $\mathit{Graph}(l)$ to mean that a list $l : \mathit{List}(\mathit{List}(\mathit{Nat}))$ represents a graph. The formula $\mathit{Graph}(l)$ is defined in `Fred` by the conjunction of four Harrop formulae (see [13]). A graph has *even parity*, which is represented by the predicate $\mathit{Evenparity}(l)$, if the number of vertices adjacent to each vertex is even.

We next sketch a proof that every even parity graph can be decomposed into a *list of disjoint cycles*. (More details of the proof may be found in [13] and [14].) From this proof we obtain a program that gives a list of the non-trivial disjoint cycles. The theorem to be proved is

Theorem 2.

$$\begin{aligned} H \vdash \forall l : \mathit{List}(\mathit{List}(\mathit{Nat}))(\mathit{Evenparity}(l)\ \&\ \mathit{start}(l) \neq 0 \\ \rightarrow \exists m : \mathit{List}(\mathit{List}(\mathit{Nat}))\ (\mathit{Listcycle}(m, l)) \end{aligned}$$

where l, m are lists (of lists of natural numbers) and $\mathit{Listcycle}(m, l)$ holds if m is a maximal list of the non-trivial disjoint cycles in the graph represented by the list l. The assumption formula H is a conjunction of Harrop formulae that describe the predicate Listcycle.

Remark 6. See [13] for details of the function *start* which takes as its argument a list, l, of lists of numbers and returns the head of the first list in l that has length greater than 1. If there is no list in l (with length > 1) then the default 0 is returned.

4.2 Skolemization

We have explained that new programs may be represented in the LTT by enlarging the current theory. Theorem 1 above shows that we can extract a new program that satisfies the Skolemized version of the formula from whose proof we extracted the program. So the Curry-Howard protocol is satisfied if we add a new function symbol (standing for the new program) and a new axiom (for the property satisfied by the new program) to the current theory. From the perspective of the associated Curry-Howard terms, this means that if we have a *proof*, with Curry-Howard term t, of $\forall x \exists y A(x, y)$, then (the universal closure of) $A(x, f_A(x))$ can be treated as a new *axiom*, with f_A a constant identified with a program in the CTT representing $\mathsf{extract}(t)$. Formally, we introduce the following structural meta-rule, *Skolemization*, to allow us to introduce new functions.

If $A(x, y)$ is a Harrop formula and t is a Curry-Howard term of type $\forall x \exists y A(x, y)$, then

$$\frac{Th_{current} \vdash t : \forall x \exists y A(x, y)}{Th_{current+1} \vdash () : \forall x A(x, f_A(x))}$$

where f_A is a new function constant, $Th_{current+1}$ is $Th_{current}$ extended by the function letter f_A and the axiom $\forall x A(x, f_A(x))$, and $\mathsf{program}$ is extended by setting $\mathsf{program}(f_A) = \mathsf{extract}(t)$.

The importance of the *Skolemization* rule to our system is that it makes the proof process much more like the ordinary practice of (constructive) mathematics. While it is possible to use a formula, $\forall x \exists y A(x, y)$, in a proof, it is more natural to give the y a name $f_A(x)$ by using a new function letter, f_A, and provide, either from the proof or otherwise, an associated program, $\mathsf{f} = \mathsf{program}(f_A)$ in the CTT, and then use $P(f_A(x))$ in further proofs. Notice that the sort of f_A corresponds to a higher type object in the CTT but is still a first order element in our LTT. See the end of our example in section 5 for a good illustration.

In [13] we showed how to extract a program, from a Curry-Howard term t, for finding a cycle (represented by the predicate $Cycle$) in an even parity graph (represented by the predicate $Evenparity$):

$$t : \forall x : List(List(Nat))\ \exists y : List(Nat)(Graph(x) \wedge Evenparity(x) \rightarrow Cycle(y, x))$$

Once this has been done, we can use the *Skolemization* meta-rule to add a new function symbol $F : List(List(Nat)) \rightarrow List(Nat)$ to our current theory, together with the new (Harrop) axiom

$$() : \forall x : List(List(Nat))\ (Evenparity(x) \rightarrow Cycle(F(x), x))$$

The Curry-Howard protocol is preserved by adding $\mathsf{program}(F) = \mathsf{extract}(t)$, the extracted program in the CTT. As will be seen, the function F is pivotal in our proof.

4.3 New Induction Rules

Adding a sort s with constructors often gives rise to a structural induction rule in the usual manner.[5] This requires introducing a new recursion operator rec and a new kind of Curry-Howard term. However, unlike other systems, we do not provide a reduction rule for this operator (apart from reducing subterms). Instead, we ensure that our extraction map, $\mathsf{extract}$, maps occurrences of this operator to a recursion in the CTT so that the extraction theorem still holds. We give two examples.

A definition of the sort Nat (natural numbers) will give rise to the induction rule

$$\frac{a : A(0) \quad R : \forall x : Nat\ (A(x) \rightarrow A(s(x)))}{\mathsf{rec}(y : Nat, a, R) : \forall y : Nat\ A(y : Nat)}$$

(Induction rule generated for natural numbers)

and the program $\mathsf{extract}(\mathsf{rec}(x : Nat, a, R))$:

[5] Hayashi [9] has a very general rule for inductive definitions but we do not need such power for our present purposes.

```
let rec rho (y,a,R) =
begin match y with
0       ->      a
|_      -> R (y-1) (rho ((y-1),a,R))
```

Note that this applies in reverse as well. Given a form of recursion in the *CTT* (for example, in Fig. 5: `rho_ChaininductionLList`) we may generate a corresponding induction rule and recursion term in the *LTT*. Of course this means that we are expanding the *LTT* and therefore, in the same way as when we add any axiom or deduction rule to a logical theory, we have to be careful to ensure that consistency is preserved. The case of the recursion in Fig. 5 corresponds to an example of descending chain induction over lists:

$$\frac{A(l_0),\ldots,A(l_r) \quad \forall l\ ((A(g(l)) \rightarrow A(l))}{\forall l\ A(l)}$$
(Modified List Induction)

where g is some function giving a list $g(l)$ that is a "simpler" list than l, or else a base case list among $l_0,\ldots,l_r$.

Example (*cont.*). The idea behind the proof of the main theorem is as follows. We start with an even parity graph l and apply the function F obtained by *Skolemization.* this yields a cycle. By deleting the edges of this cycle from L we are left with another graph, $g(l)$ that is again of even parity. We then repeat the process, starting with $g(l)$ to form $g(g(l))$, etc., until we are left with an empty graph. The list of non-trivial disjoint cycles of l is then given by $\langle F(l), F(g(l)), F(g(g(l))),\ldots\rangle$. The base case for a list l occurs when the l has been obtained by deleting all the edges. Because $g(l)$ either gives a "simpler" list than l or else a base case list, we can achieve this proof by using the modified list induction mentioned above. We therefore add g to the current theory and a set of axioms describing its properties (and, as usual, the corresponding program in the *CTT*).

The *ML* program extracted is displayed in Fig. 5 where `Cgrmain` is a program corresponding to other lemmas in the proof of the theorem and `g` and `F` are the pre-programmed functions that satisfy the Curry-Howard protocol.

We present some practical results. Here is the result for the graph with four vertices in Fig. 4 (section 4) returned as a list of lists (`llist`). (The graph has one non-trivial and one trivial cycle.)

```
#main [[1;2;3];[2;1;3];[3;1;2];[4]];;
- :  int list list  = [[1;3;2;1]]
```

Next we consider the even parity graph on the right in Fig. 4 (Section 4) with vertices $1,\ldots,6$ and extract the list of non-trivial disjoint cycles in it.

```
#main [[1;2;6];[2;1;3];[3;2;4;5;6];[4;3;5];[5;4;3];[6;1;3]];;
- : int list list = [[1;6;3;2;1];[3;5;4;3]]
```

```
let rec rho_Chaininduction_LList (ll,A,BASE) =
begin match ll with
[[]] -> BASE
| _ -> A ll (rho_Chaininduction_LList((g ll),A,BASE))
end;;

let lappend x y = x@[y]
;;

let main =
(let recFunX33 l =
rho_Chaininduction_LList(l,
(fun x ->
(fun X10 -> (function inl(m) ->
                  ((let funX32 d = ((lappend d (F x))) in funX32)X10)
                     | inr(n) ->
                       ([(F x)]) ) (Cgrmain (g x))
)),[[]]) in recFunX33) ;;
```

Fig. 5. *ML* program extracted from our proof of the theorem: $\forall l(Evenparity(l) \ \& \ Start(l) \neq 0 \rightarrow \exists m(Listcycle(m, l)))$

5 The Protocol over other Logical and Computational Systems

We have described the Curry-Howard protocol, focusing on intuitionistic logic as *LTT*, and *ML* and *CTT*. One of the advantages the Curry-Howard protocol is that it provides a framework for program extraction over other kinds of logical system.

5.1 Structured Specifications

In [8], we employed the protocol to define a program extraction methodology over a logic for reasoning about, and with, structured algebraic specifications. An algebraic specification is essentially a many-sorted theory consisting of a signature and axioms about terms of the signature. For example, part of an algebraic specification for lists of natural numbers might be given as:

spec LISTNAT =
sorts $Nat, ListNat$
ops $0 : Nat;$
$\quad suc\_\_ : Nat \rightarrow Nat;$
$\quad \_\_ + \_\_ : Nat \times Nat \rightarrow Nat;$
$\quad empty : ListNat;$
$\quad [\_\_] : Nat \rightarrow ListNat;$
$\quad \_\_ :: \_\_ : ListNat \times ListNat \rightarrow ListNat;$
$\quad size\_\_ : ListNat \rightarrow Nat;$

axioms
$size(empty) = 0;$
$size([a] :: b) = 1 + size([a]);$

A structured algebraic specification is formed by applying structuring operations over flat (algebraic) specifications. For instance,

– if Sp_1 and Sp_2 are specifications, then Sp_1 **and** Sp_2 is a structured specification, denoting the pushout union of the two specifications.
– if Sp is a specification, then, given map ρ which renames sorts and function symbols, $\rho \bullet$ Sp denotes the specification in all symbols in the domain of ρ are renamed.
– if Sp is a specification and *Sig* is a signature, then Sp **hide** *Sig* denotes a specification whose models are those of Sp, but for which the symbols and axioms which use the symbols are not visible to the user of the specification.

In [8], we dealt with these three types of structuring operations: union, translation and hiding.

The logic we developed involved theorems of the form $\vdash_{\text{Sp}} B$, where Sp is a structured specification and B is a first order formula that is true of the specification Sp. The rules of the calculus adapt the usual intuitionistic rules to reasoning structured specifications.

For instance, ($\wedge$ I) is modified to be of the form

$$\frac{\vdash_{\text{Sp\_1}} d : A \quad \vdash_{\text{Sp\_2}} e : B}{\vdash_{\text{Sp\_1 } \mathbf{and} \text{ Sp\_2}} (d, e) : A \wedge B}$$

This rule means that, if A is true about Sp_1 and B is true about Sp_2, then $A \wedge B$ must be true about the union Sp_1 **and** Sp_2. Other, new, rules were introduced to accommodate purely structural inferences – that is, inferences when we conclude a formula is true about a new specification, based on the fact that it is true about an old specification. For example,

$$\frac{\vdash_{\text{Sp\_1}} d : A}{\vdash_{\text{Sp\_1 } \mathbf{and} \text{ Sp\_2}} union_1(d, \text{Sp\_2}) : A}$$

states that, if A is true about Sp_1, then A must also be true of the union Sp_1 **and** Sp_2.

In [8], we describe a *LTT* for this logic. Its Curry-Howard terms extend those given here for intuitionistic logic: new kinds of Curry-Howard terms are generated because of the structural rules and axiom introduction. The normalization relation satisfies some interesting constraints, imposed in order to satisfy point *6*. of Definition 3. We then use the same *CTT* as used here (namely, *ML*). The extract map is similar, apart from its treatment of Curry-Howard terms corresponding to structural rules.

5.2 Imperative Programs

In [17], we describe an adaptation of Curry-Howard methods to extracting imperative programs from proofs. We give an *LTT* for reasoning about side-effects and return values of imperative programs. The underlying calculus is similar to Hoare Logic, but with a constructive, natural deduction presentation. We develop new Curry-Howard terms for representing proofs in this calculus: these terms extend those given here in order to deal with rules that are specific to reasoning about imperative programs. Again, the normalization relation is defined so as to preserve the protocol. We then define an extract map from the LTT to an imperative CTT, which preserves the protocol, and allows us to extract imperative programs from proofs of specifications.

The LTT/CTT demarcation is particularly natural in this context. The CTT (like most imperative languages, such as C++ or Java) obeys a particular (call-by-value) evaluation strategy. On the other hand, it is most natural for the LTT to be indifferent to the order in which Curry-Howard terms are normalized. This is because we chose an LTT with a familiar underlying logic where simplifications of logical proofs can be carried out independent of order.

In contrast, a unified approach to representing imperative programs, and reasoning about programs, with a single type theory representing both proofs and programs, would require that the call-by-value evaluation strategy be imposed on logical proofs as well as programs. This would require an underlying logic with which most users would be unfamiliar. Such a naïve approach to adapting Curry-Howard methods to imperative program synthesis would not be simple to devise, and would be difficult to use. Thus the Curry-Howard protocol appears to be necessary in order to define a simple approach to adapting Curry-Howard methods to imperative program synthesis.

6 Related Work and Conclusions

The advantages of the Curry-Howard protocol are practical. We have chosen to use a LTT based on a first-order, many-sorted logic, because this is an easy type of deductive system to use. Additionally, adopting a loose coupling between terms and sorts of the LTT and programs and types of the CTT promotes a natural conceptual distinction: the logic is used for reasoning about programs, while the CTT is used for representing programs.

In some cases the protocol, or something like it, appears to be necessary in order to be able to define a simple Curry-Howard style extraction mechanism for more complicated logical systems. For example, without the protocol, Curry-Howard style synthesis would be difficult to achieve over the proof system for reasoning about algebraic specifications.

Martin-Löf [16] makes the point that his type theory is *open* in the sense that new terms and new types may be added (*via* a computational definition) at any point in time. Because logic and computational types have the same status, any axioms concerning a new term or elements of a new type have to be proved from such a computational definition. On the other hand, the introduction of a new function symbol or sort is accompanied by a set of axioms that are *taken as* true (just as in ordinary mathematics). Our Curry-Howard protocol demands that a suitable new function or type has been correspondingly introduced in the CTT

so that our extraction theorem still holds. While it is true that Martin-Löf's and Constable's systems can re-use programs, those programs have to be encoded into the standard syntax. With ours they remain in the *CTT* and do not need rewriting.

In the area of type theory, Zhaohui Luo's *Extended Calculus of Constructions* (*ECC*, [15]) is similar in motivation to our framework. The *ECC* provides a predicative universe *Prop* to represent logical propositions and a Martin-Löf-style impredicative universe hierarchy to represent programs. As in Martin-Löf, the impredicative universes are open, so the same comparison holds. Like our system, the *ECC* has a similar division of labour between proving properties of programs (in *Prop*) and creating new programs and types (in the universe hierarchy). The *ECC* was designed to provide a unified framework for the two (recognised) separate tasks of logical reasoning and program development but not with program synthesis in mind. This means that in the *ECC* there is no notion of a simplifying extraction map between terms that represent proofs and program terms – they are identified. Consequently it would not make sense to add to it a rule such as our *Skolemization*.

We have presented the Curry-Howard protocol in an informal metalogic. Anderson [2] used the Edinburgh Logical Framework to provide a similar relationship between proofs in a logical type theory and programs in a computational type theory. That work was primarily concerned with defining that relationship so as to obtain optimized programs. However, representations of optimized programs are not added to the logical type theory. Our metalogical results might benefit from a similar formal representation.

Constable's *NuPRL* system [6] contains an untyped lambda calculus for program definitions. Untyped lambda programs may then be reasoned about at the type level. (One of the main purposes of *NuPRL* is to do verification proofs of programs in this manner). In [4], it was shown how to use *NuPRL*'s set type to view such verifications as a kind of program extraction. Similarly, Coquand and Huet's *Coq* [12] is able to define *ML* programs directly and prove properties of them. However, it seems that little work has been done on the possibility of integrating this type of verification with program extraction along the lines we have described: rather, they are treated as separate applications of the system.

There are several extensions that could be made to our system and its implementation `Fred`:

– Currently, `Fred` is a purely interactive theorem prover. However the underlying logic is very standard, so it should be easy to integrate a theorem prover for many-sorted logic automatic as a front-end to `Fred`.
– Clearly it is better to have more structure in the theory being used by the *LTT*. As it gets larger, it might contain many different sorts, functions and axioms. This paper is really a "prequel" to our paper [8], where we defined how a *LTT* could be given with respect to a set of structured algebraic specifications in the language *CASL*. The results presented there obey the protocol we have outlined here and demonstrate its utility.

When we import a program from the *CTT* as a constant in the *LTT* (with appropriate axioms), we assume that the program satisfies the axioms and that these axioms are consistent. It is up to the programmer to guarantee this. We assume that the programmer has used some means of guaranteeing this (for example by using Hoare logic or *Coq*). Both *NuPRL* and *Coq* allow for this guarantee to be constructed within the logic itself. We *could* create a theory of *CTT* programs for use in the *LTT* for the purposes of such verification but our present approach admits all sorts of different verification procedures. In particular, although we have not dealt with the issue here, we are able to use certain *classically* proved results.[6]

However, with the rise of component-based software development and the acceptance of many different verification/synthesis methodologies, it seems that, in the future, any component-based synthesis/development will be done in a situation where the prover does *not* have access to source code enabling his/her own verification and will have to rely on the supplier for a guarantee. In this case interoperability will demand that this is (or can be) expressed in a standard logical language. Viewed in this light, methodologies such as the one we have outlined may be a sensible and practical means of integrating Curry-Howard-style program synthesis with other techniques.

References

1. David Albrecht and John N. Crossley. Program extraction, simplified proof-terms and realizability. Technical Report 96/275, Department of Computer Science, Monash University, 1996. 22
2. Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie Mellon University, 1993. 35
3. Ulrich Berger and Helmut Schwichtenberg, Program development by Proof Transformation. Pp. 1–45 in *Proceedings of the NATO Advanced Study Institute on Proof and Computation*, Marktoberdorf, Germany, 1993, published in cooperation with the NATO Scientific Affairs Division. 23
4. James L. Caldwell. Moving Proofs-As-Programs into Practice. Pp. 10–17 in *Proceedings 12th IEEE International Conference Automated Software Engineering,* IEEE Computer Society, 1998. 35
5. Robert L. Constable. The structure of Nuprl's type theory. Pp. 123–155 in H. Schwichtenberg, editor, *Logic of computation (Marktoberdorf, 1995)*, NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci., **157**, Springer, Berlin, 1997. 18, 19
6. Robert L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. Harper, D. J. Howe, T. B. Knoblock, N. P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986. 19, 35
7. John N. Crossley and John C. Shepherdson. Extracting programs from proofs by an extension of the Curry-Howard process. Pp. 222–288 in John N. Crossley, J. B. Remmel, R. Shore, and M. Sweedler, editors. *Logical Methods: Essays in honor of A. Nerode.* Birkhäuser, Boston, Mass., 1993. 19, 21, 22

[6] This includes some results requiring the law of the excluded middle.

8. John N. Crossley, Iman Poernomo and M. Wirsing. Extraction of Structured Programs from Specification Proofs. Pp. 419-437 in D. Bert, C. Choppy and P. Mosses (eds), *Recent Trends in Algebraic Development Techniques (WADT'99)*, Lecture Notes in Computer Science, **1827**, Berlin: Springer, 2000. 20, 23, 32, 33, 35
9. Susumu Hayashi and Hiroshi Nakano. *PX, a computational logic.* MIT Press, Cambridge, Mass., 1988. 18, 19, 30
10. Leon Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, **15** (1950) 81–91. 20
11. William A. Howard. The formulae-as-types notion of construction. Pp. 479–490 in John R. Seldin and R. J. Hindley, editors, *To H.B. Curry : essays on combinatory logic, lambda calculus, and formalism.* Academic Press, London, New York, 1980. 19
12. Gerard Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof assistant Reference Manual: Version 6.1.* Coq project research report RT-0203, Inria, 1997. 18, 19, 28, 35
13. John S. Jeavons, I. Poernomo , B. Basit and J. N. Crossley. A layered approach to extracting programs from proofs with an application in Graph Theory. Paper presented at the Seventh Asian Logic Conference, Hsi-Tou, Taiwan, June 1999. Available as T. R. 2000/55, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia. 28, 29, 30
14. John S. Jeavons, I. Poernomo, J. N. Crossley and B. Basit. `Fred`: An implementation of a layered approach to extracting programs from proofs. Part I: an application in Graph Theory. *AWCL (Australian Workshop on Computational Logic), Proceedings*, Canberra, Australia, February 2000, pp. 57–66. 29
15. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994. 35
16. Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984. 20, 28, 34
17. Iman Poernomo. PhD thesis, School of Computer Science and Software Engineering, Monash University, in preparation. 20, 33