Protocols between programs and proofs*

Iman Poernomo** and John N. Crossley***

School of Computer Science and Software Engineering,
Monash University, Clayton, Victoria, Australia 3168.
{jnc, ihp}@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, [8],[4] and [11]), but our presentation is ultimately aimed at a practical, usable system and has a number of novel features. These include

1. first of all, 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 and proofs of theorems that give programs.

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 such 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, f, to new functions that we extract from proofs of closed formulae of the form $\forall x \exists y \alpha(x, y)$ by the well-known Curry-Howard isomorphism in order to make $\alpha(\bar{n}, f(\bar{n}))$ true. (See e.g. [10] or [6]). 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* [8], Constable's *NuPRL* [4],[5] and the *Coq* system [11]. The first of these uses a logic different from the traditional logic and not so 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.

We first describe a *logical* type theory (LTT) whose types are formulae of this calculus and whose terms represent proofs. We next describe a *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

* Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

** 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.

map between the 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) in the following ways. It allows us to

1. first of all, we 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.
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.
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 extracting a program that computes such a list of disjoint cycles from a given graph.

2 The Logical Type Theory and the Computational Type Theory

Theories. We work with standard, many-sorted, first order intuitionistic logic so that we can extract programs. (For the details see [7] or [6].) 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.). One can go much further (as Henkin suggests in a footnote in [9]) and, using the protocol outlined here, we are able, in our paper [7], to reason about modules and structured specifications.
A theory Th has a collection of sorts generated by a base set of sorts $\mathcal{S}_{Th}$. Its signature $Sig(Th)$ 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.[3]
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 $\mathcal{S}_{Th} \subseteq \mathcal{S}_{Th'}$, $Sig(Th) \subseteq Sig(Th')$ 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.

The Logical Type Theory. Our logical type theory (LTT) is a means of encoding proofs in this natural deduction calculus. The types are many-sorted formulae for a theory and ("Curry-Howard" or proof) terms are essentially terms in a lambda calculus with dependent sum and product types that represent proofs. (Full details may be found in [6] or [7].)
Because Curry-Howard terms are terms in a form of lambda calculus, they have *reduction rules* whose application corresponds to proof normalization. (See [6] and [1] for the full list of rules.) However, unlike other systems for program extraction, some terms involving functions of a theory that occur within a Curry-Howard term are not reduced by these reduction rules. So, for instance the reduction chain

$$((\lambda x.(x+x,()))^{\forall x:Nat\ \exists y:Nat\ 2.x=y} 3)^{\exists y:Nat\ 2.3=y} \longrightarrow (3+3,())^{\exists y:Nat\ 2.3=y}$$

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.

[3] 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. [6].

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*.

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.
There is an *extraction map* from terms of the *LTT* to terms of the *CTT*, such that Theorem 1 holds. (For a detailed example of such a map see [7].)

Theorem 1. *Given a proof* $p^{\forall x:s_1 \exists y:s_2 \alpha(x,y)}$ *in the logical type theory, there is a program* f *of ML type* $\mathtt{s}_1 -> \mathtt{s}_2$ *in the computational type theory ML such that* $\alpha(x : s_1, f(x) : s_2)$ *is true under the interpretation of symbols given by the Curry-Howard protocol. The extracted program is given by* $f =$ **extract** (p).

Definition 1. *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. *There is an extraction map* **extract** : $LTT \rightarrow CTT$ *that gives a function* f *from a theorem* $\forall x \exists y \alpha(x, y)$ *such that* $\alpha(\bar{n}, f(\bar{n}))$ *is true.*
3. *There is a map* **program**:$\mathcal{F}_{Th_{current}} \rightarrow CTT$*, such that if* f *is a function of* $Th_{current}$ *and* A *is its sort, then* $\mathbf{program}_A(f)$ *is a program in CTT with type* A.
4. *The map* **program** *can be extended to an interpretation of* $Th_{current}$ *such that every axiom in* $Ax_{Th_{current}}$ *is satisfied. (If* $=: A \times A \in \mathcal{R}_{Th_{current}}$ *is equality between elements of* A*, then the interpretation should map* $=$ *to equivalence of the normal forms of the values of the elements of* A *in the CTT.)*

The map **program** is used by **extract** because proof terms involve functions from $Th_{current}$. So any extraction map (from proof terms to *CTT* programs) must extend a map (such as **program**) from $\mathcal{F}_{Th_{current}}$ to the *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 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.[4] 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.

3.1 Enlarging the theory.

The current theory $Th_{current}$ being used in a proof may be enlarged to $Th_{current+1}$ by adding new function constant symbols or new sorts and axioms, consistently, of course.

Adding new programs. If we *introduce* a new program `f` in the *CTT* that is known to satisfy some property P, then we add a constant f of the appropriate sort and the property $P(f)$ to the *LTT*'s theory, $Th_{current}$, to give a new theory $Th_{current+1}$, with the **program** extended by $\mathbf{program}(f) =$`f`. For example, the function for determining the length (a number of sort Nat) of a list (an element of sort $List$) $length_\alpha : List(\alpha) \rightarrow Nat$ 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*)

$$length_\alpha(\epsilon_\alpha) = 0$$
$$length_\alpha(\langle a \rangle :: l) = \bar{1} + length_\alpha(l)$$

[4] 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.

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` $=$ **program**$(length_\alpha)$ and guarantee that the Curry-Howard protocol is preserved. We might 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 then 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.

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:

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

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.[5]

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. In the proof of the theorem, 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 lists of elements of sort $List(Nat)$ correspond to graphs.) Consider the graph with four vertices[6] 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. We easily arrange that each function is provably total in the formal system. Here is the list of required functions in $\mathcal{F}_\mathcal{G}$ and the associated axioms. All formulae are considered to be universally closed. We note that appropriate CTT programs must be provided according to the Curry-Howard protocol outlined in section 3.
1. Binary functions, $member_\alpha$, of two arguments: a natural number n and a list of elements of sort α. The function computes the nth member of the list.[7] The definitions for the cases $\alpha = Nat,\ List(Nat)$ are given in [12].
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. Initial segment of a list: $initlist(k,l)$ computes the list consisting of the first $k+1$ elements of the list l, if $k+1 > length(l)$ then the list l is returned.
5. Tail segment of a list: $tail(l,n)$ 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.

[5] As in *Coq*, see [11], this could also be achieved by creating a new sort (for logical formulae), or with a universe hierarchy as in Martin-Löf [15].
[6] The graph comprises a triangle with vertices $1,2,3$ and a single unconnected point 4.
[7] Since all functions are total we shall sometimes use a "default value".

We now use new predicates and functions to prove our graph-theoretic result. We set a predicate $Graph(l)$ to mean that a list $l : List(List(Nat))$ represents a graph. The formula $Graph(l)$ is defined in `Fred` by the conjunction of four Harrop formulae (see [12]). A graph has *even parity* (represented by the predicate $Evenparity(l)$) 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 [12] and [13]). From this proof we obtain a program that gives the decomposition into a list of disjoint cycles. The theorem to be proved is

$$H \vdash \forall l : List(List(Nat))(Evenparity(l) \,\&\, start(l) \neq 0 \rightarrow \exists m : List(List(Nat))\; Listcycyle(m, l))$$

where l, m are lists (of lists of natural numbers) and $Listcycle(m, l)$ holds if m is a maximal list of disjoint cycles in the graph represented by the list l. The assumption formula H is a conjunction of Harrop formulae that describe the predicate $Listcycle$. (See [12] for the function $start$.)

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* t of $\forall x \exists y \alpha(x, y)$, then (the universal closure of) $\alpha(x, f_\alpha(x))$ can be treated as a new *axiom*, with f_α a constant identified with a program in the CTT representing $\mathbf{extract}(t)$. Formally, we introduce the following structural meta-rule, *Skolemization*, to allow us to introduce new functions. If $\alpha(x, y)$ is a Harrop formula and t is a Curry-Howard term of type $\forall x \exists y \alpha(x, y)$, then

$$\frac{Th_{current} \vdash t^{\forall x \exists y \alpha(x,y)}}{Th_{current+1} \vdash ()^{\forall x \alpha(x, f_\alpha(x))}}$$

where f_α is a new function constant, $Th_{current+1}$ is $Th_{current}$ extended by the function letter f_α and the axiom $\forall x \alpha(x, f_\alpha(x))$, and **program** is extended by setting $\mathbf{program}(f_\alpha) = \mathbf{extract}(t)$.
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, $\exists y P(y)$, in a proof, it is more natural to give the "y" a name $f_\alpha(x)$ by using a new function letter, f_α, and provide, either from the proof or otherwise, an associated program, $\mathtt{f} = \mathbf{program}(f_\alpha)$ in the CTT, and then use $P(f_\alpha(x))$ in further proofs. Notice that the sort of f_α 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 3.2 for a good illustration.
In [12] we showed how to extract a program (from a proof) 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 axiom

$$()^{\forall x : List(List(Nat))\; (Evenparity(x) \rightarrow Cycle(F(x), x))}$$

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

3.2 New induction rules

Adding a sort s with constructors often gives rise to a structural induction rule in the usual manner.[8] This requires introducing a new recursion operator ρ 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).

[8] Hayashi [8] has a very general rule for inductive definitions but we do not need such power for our present purposes.

Instead, we ensure that our extraction map **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

$$\frac{a^{A(0)} \quad R^{\forall x:Nat\ (A(x) \to A(s(x)))}}{\rho(y : Nat, a, R)^{\forall y:Nat\ A(y:Nat)}} \quad \text{(Induction rule generated for natural numbers)}$$

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

```
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. 1: `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. 1 corresponds to an example of descending chain induction over lists:

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

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 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, adding a corresponding program in the CTT).
The ML program extracted is displayed in Fig. 1 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 – see Appendix C of [12] for a full listing of the programs for `g` and `F`.
We present some practical results. Here is the result for the graph with four vertices in section 3.1 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]];;
- :  llist = [[1;3;2;1];[4]]
```

Next we consider an even parity graph with vertices $1, \ldots, 6$ and extract the list of disjoint cycles in it. (Here the graph comprises a triangle {`3,4,5`} and a square {`1, 2, 3, 6`}.)

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

4 Related work and Conclusions

Martin-Löf [15] 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 occupy 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 a 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.

```
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,
(let funX30 x =
(let funX31 X10 = (case3 (Cgrmain (g x))
((let funX32 d = ((lappend d (F x))) in funX32) X10) ([(F x)]) )
in funX31) in funX30),[[]]) in recFunX33)
;;
```

Fig. 1 *ML* program extracted from proof of the theorem:
$\forall l(Evenparity(l) \;\&\; Start(l) \neq 0 \rightarrow \exists m(Listcycyle(m, l)))$

In the area of type theory, Zhaohui Luo's *Extended Calculus of Constructions* (*ECC*) ([14]) 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 primary concerned with defining the 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.
The *NuPRL* system [5] 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 [3], it was shown how to use *NuPRL*'s set type to view such verifications as a kind of program extraction. Similarly, *Coq* [11] 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 many-sorted logic automatic theorem prover 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 [7], 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 axioms, we assume that the program satisfies the axioms and that these axioms are consistent. It 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*). *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, though we have not dealt with the issue here, we are able to use certain *classically* proved results.[9]
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, Monash University Department of Computer Science, 1996.
2. Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Carnegie Mellon University, 1993.
3. 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.
4. 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.
5. 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.
6. John N. Crossley and J. 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.
7. John N. Crossley, Iman Poernomo and M. Wirsing. Extraction of Structured Programs from Specification Proofs. To appear in WADT99, Springer Lecture Notes in Computer Science, vol. 1827.
8. Susumu Hayashi and Hiroshi Nakano. *PX, a computational logic.* MIT Press, Cambridge, Mass., 1988.
9. Leon Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, **15** (1950) 81–91.
10. 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.
11. 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.
12. 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 School of Computer Science and Software Engineering, Monash University, Melbourne, Australia. Technical report 2000/55, 17pp.
13. 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.
14. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.
15. Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

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