Journal of Universal Computer Science, vol. 7, no. 1 (2001), 71-88
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/1/01 © Springer Pub. Co.

`Fred`: An Approach to Generating Real, Correct, Reusable Programs from Proofs

John Crossley
School of Computer Science and Software Engineering
Monash University, Australia
jnc@csse.monash.edu.au

Iman Poernomo[1]
School of Computer Science and Software Engineering
Monash University, Australia
ihp@csse.monash.edu.au

Abstract: In this paper we describe our system for automatically extracting "correct" programs from proofs using a development of the Curry-Howard process.

Although program extraction has been developed by many authors (see, for example, [HN88], [Con97] and [HKPM97]), our system has a number of novel features designed to make it very easy to use and as close as possible to ordinary mathematical terminology and practice. These features include 1. the use of Henkin's technique [Hen50] to reduce higher-order logic to many-sorted (first-order) logic; 2. the free use of new rules for induction subject to certain conditions; 3. the extensive use of previously programmed (total, recursive) functions; 4. the use of *templates* to make the reasoning much closer to normal mathematical proofs and 5. a conceptual distinction between the computational type theory (for representing programs)and the logical type theory (for reasoning about programs).

As an example of our system we give a constructive proof of the well known theorem that every graph of even parity, which is non-trivial in the sense that it does not consist of isolated vertices, has a cycle. Given such a graph as input, the extracted program produces a cycle as promised.

Key Words: Program synthesis, proofs as programs, reusable software.
Category: F.3.1, D.2.4

1 Introduction and Overview

Our first aim is to produce correct *and practical* programs from mathematical proofs.[2] Our ultimate aim is to imitate mathematical practice *and* to produce *not* a nice formal mathematical theory *but* a *usable* system for program synthesis.

We start with a formal proof of a specification. Then the proof is encoded as a *Curry-Howard* or *proof term*, which is a term in a typed lambda calculus with dependent sums and products. The proof term can then be processed further to get a *program* in the Caml-light variant of *ML*. (This program is a term of a simply typed lambda calculus with disjoint unions). This program satisfies the original specification.

[1] Research partly supported by Australian Research Council grant A 49230989.
[2] The word "correct" in this paper means "meeting its specification" and we shall also require the user to guarantee that the specification is consistent.

Finally we give an example: extracting a cycle from a non-trivial, even parity graph.

We use a mixture of approaches (all to be explained later): the "traditional" Curry-Howard method, but using a strategy due to Leon Henkin [Hen50], templates, and using a protocol between two type theories: the logical type theory (LTT) of proof terms and the computational type theory (CTT) of *ML*.

1.1 The Curry-Howard Isomorphism

The well known Curry-Howard isomorphism (see e.g. [How80] or [CS93]), produces a term of a lambda calculus from a (constructive) proof of a formula. This can be used to give a program which computes the constructive content of the formula. Thus, in arithmetic a constructive proof of a formula of the form $\forall x \exists y \alpha(x, y)$ yields an algorithm for computing a function f such that $\alpha(\overline{n}, f(\overline{n}))$ holds for every natural number n. ($\overline{n}$ is the numeral for n.)

In this paper we present an extension of the Curry-Howard isomorphism to a first order, many-sorted, predicate calculus which also allows the use of previously programmed functions (and predicates) subject to guarantees of consistency. The extension to a many-sorted calculus allows us to extract programs over different sorts. This has previously been done successfully in various higher order systems. Our approach avoids the use of higher order logic. (Inevitably this theory is not as strong as e.g. Martin-Löf's, but our use of templates gives it some second order strength.)

Now it is well known that the programs extracted from full proofs in formal logic are immensely long both in size and in running time. We therefore introduce other features into our system to make it more manageable and the programs shorter. These features are designed to mirror, as far as possible, normal mathematical practice.

1.2 Extensions and the LTT/CTT Protocol

The Curry-Howard isomorphism uses a logical type theory with dependent sums and product types to encode proofs. Besides this logical type theory (the LTT) we also have a computational type theory (the CTT) for representing programs. We shall define a protocol between the LTT and CTT through which we can

1. (easily axiomatize and) use pre-programmed functions in our proofs,
2. extract correct, realistic programs (in the CTT) from proofs of specifications (in the LTT) and then reason about these programs in future proofs, thus reusing these programs in program synthesis and
3. retain a conceptual distinction between reasoning about programs and running programs.

Our motivation is that this leads to a practical, useful system requiring minimal training and specialist knowledge.

We have built a software system, written in C++ and currently called `Fred`, as an implementation of our system.[3] 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 taking a constructive proof that every non-trivial even parity graph contains a cycle and then using our system to extract a program that computes such a cycle from a given (non-trivial) graph.

2 Related Work

There have been a number of systems exploiting the Curry-Howard notion of formulae-as-types. In particular we mention: Hayashi's system *PX*, [HN88], the implementations of Martin-Löf's type theory [ML84], such as [NPS90] and [Con97], the *Coq* prover, see [HKPM97] (based on Coquand and Huet's *Calculus of Constructions*), Schwichtenberg's *Minlog* system, see [BS95], and Zhaohui Luo's *Extended Calculus of Constructions* (ECC) [Zha94].

These systems provide Curry-Howard style program synthesis via a single, unified framework for programming and doing logic. This distinguishes them from ours, where we emphasize retaining a strong distinction between the logic and programming languages (achieved via our distinction between the LTT and the CTT).

Martin-Löf [ML84] makes the point that his type theory is open, in the sense that new terms and new types may be added at any point in time (added via a computational definition). 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. In contrast, 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). The CTT/LTT protocol demands that a suitable new function or type has been correspondingly introduced in the CTT so that our extraction theorem still holds.

In the area of type theory, Zhaohui Luo's *Extended Calculus of Constructions* [Zha94] 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. So, 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). However, the ECC was not designed with program synthesis in mind, rather to provide a unified framework for the two (recognised) separate tasks of logical reasoning and program development. Consequently, the techniques we employ in this paper (in particular, an extraction protocol between the CTT and LTT) have not been used in the ECC.

We present the CTT/LTT protocol in an informal metalogic. In [And93], Anderson used the Edinburgh Logical Framework to achieve 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

[3] The name comes from "Frege dynamic system" because the proofs have an appearance similar to those in Frege's *Begriffschrift* [Fre79]. (See [Fig. 6].)

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 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 [Cal98], it was shown how to use *NuPRL*'s set type to view such verifications as a kind of program extraction. Similarly, *Coq* is able to directly define and synthesize *ML* programs from proofs (see [PMW93]). 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.

When we import a program with axioms, we assume that the program satisfies the axioms and that these axioms are consistent. It is up to the programmer to guarantee this. (There are many methods: model checking, by using Hoare logic or `Fred`). *NuPRL* and *Coq* allow for this guarantee to be constructed within the logic itself.

Both Martin-Löf and Coquand and Huet have directly integrated a programming language within higher order type theory. Hayashi's system uses a logical system using partially defined functions developed by Feferman, and Schwichtenberg uses minimal logic. Constable's uses higher-order logic. Although Martin-Löf's system is much liked by those who have used it for some time, all of these logical systems seem, for most people, much less familiar, and more complicated, than ordinary first-order logic.

3 The Logical and Computational Type Theories

We work in `Fred` in the same way as mathematicians: constantly introducing new functions and reusing previously proved theorems (that is to say, the logical aspect), or as computer scientists: constantly reusing (reliable) code (that is to say, the computational aspect).

3.1 The Logical Type Theory (LTT)

We present a logical type theory (LTT) of many-sorted intuitionistic logic. The types are many-sorted intuitionistic formulae and the proof terms are essentially terms in an extended typed lambda calculus which represent proofs. LTT is modular and extensional with respect to the operational meaning of its function terms. However the function terms may be programmed in a computational type theory. In this case we may introduce axioms for them in LTT. These function terms can be defined in whatever way we wish, as long as they satisfy the axioms of the LTT. However the *user* is required to guarantee that these programs are also "correct" and, in particular, that the axioms added for the functions from the new programs preserve consistency. Thus we retain a distinction between *extensional meaning* (given by the axioms they must satisfy) and *intensional meaning* (how they are coded in the computational type theory).

Each term t has an associated sort[4] s – we denote this relationship in the usual fashion, by $t : s$, read "t is of sort s". In constructing terms we shall always assume that the sort of the constructed term is appropriate. For example: If $t_1 : s_1 \times s_2$ and $t : (s_1 \times s_2 \rightarrow s_3)$, then $t(t_1) : s_3$.

The axioms (denoted by Ax) that we permit are the ones one would normally employ in (constructive) mathematics are Harrop formulae (defined below). The restriction is a natural one and also has a significant effect on reducing the size of our extracted programs. Harrop axioms are axioms that are Harrop formulae and Harrop formulae are defined as follows: 1. An atomic formula or $\perp$ is a Harrop formula. 2. If α and β are Harrop, then so is $(\alpha \wedge \beta)$. 3. If α is a Harrop formula and γ is any formula, then $(\gamma \rightarrow \alpha)$ is a Harrop formula. 4. If α is a Harrop formula, then $\forall x \alpha$ is a Harrop formula.

Harrop formulae play an important rôle in the program extraction process. These formulae contain no computational information and therefore allow the deletion of large parts of the proof terms (see below).

The rules for ordinary first order natural deduction are readily adapted to the many-sorted case. We associate with each many-sorted formula a proof term (essentially a term of our lambda calculus) representing the derivation of the formula. The terms are formed using λ, application, pairing $(\cdot, \cdot)$, the projections π_1 and π_2, (as usual we have the reduction rule: $\pi_i(x_1, x_2) = x_i$ for $i = 1, 2$) and two operations **select** and **case** which have reduction rules given in Albrecht and Crossley [AC96]. [Fig. 1] gives the natural deduction rules and the proof terms.

Proof normalization corresponds to lambda calculus reductions over the LTT as usual (see [GLT89] or [CS93]). Thus an implication introduction followed by an implication elimination can be dramatically reduced to a triviality. This kind of situation arises when a lemma is used before a new theorem and the lemma has already been proved previously. One such reduction can lead to others as introduction and elimination rules are brought into proximity.

In other earlier approaches to Curry-Howard program synthesis the LTT was treated as a programming language and proof normalization was treated as program compilation. However, in our implementation `Fred` we first extract a program from the proof term and the extracted program uses the optimized evaluation strategies of ML. (In fact the proof reductions are incorporated in this process. Details may be found in [CPW00].)

3.1.1 New Induction Rules

Adding a sort s with constructors often gives rise to a structural induction rule in the usual manner.[5] This may introduce a new proof term operation rec_s (where the subscript S indicates the sorts involved) with the usual fixed point semantics,

[4] It is convenient to call the entities "sorts" rather than "types" as there are many other "types" in this paper. In fact for our present purposes we could easily reduce everything to first order. To do this we should just use a predicate, $In(x, y)$, say, to represent "x is in the list y" and similarly for lists of lists. The technique is described in Henkin [Hen50]. However we write our expressions in the conventional way and they therefore sometimes appear as involving higher order expressions.

[5] Hayashi [HN88] has a very general rule for inductive definitions but we do not need such power for our present purposes.

Initial Rules

$$\frac{}{x : A \vdash x : A}\ (\text{Ass I}) \qquad \frac{}{\vdash () : A}\ (\text{Ax I})$$
when $A \in Ax$ for some sort s

Introduction Rules

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

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

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

Elimination Rules

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

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

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

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

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.

Figure 1: Logical Rules and Proof Terms.

and an obvious set of reduction rules. For example in [Fig. 2] we give the axioms, induction rule and definition of rec_N for the sort of natural numbers N.

An important sort for representing graphs is the *parametrized list*, *List* (α), the list of objects of sort α. The constructors of *List* (α) are: ϵ_α, the empty list in *List* (α) and $con_\alpha : \alpha \times List\ (\alpha) \to List\ (\alpha)$. We abbreviate the term $con(\alpha)(t, l)$ by $\langle t\rangle :: l$ and use $\langle t_0, t_1, \ldots, t_n\rangle$ as an abbreviation for the term $con_\alpha(t_0, con_\alpha(t_1, con_\alpha(\ldots con_\alpha(t_n, \epsilon_\alpha))))$.

Lists have the following induction rule for each sort α. Let l be a variable of sort *List* (α) and a a variable of sort α.

$$\frac{\vdash a : A(l/\epsilon_\alpha) \wedge \forall a\, \forall l\ (A \to A(l/(\langle a\rangle :: l)))}{\vdash rec_{List\ (\alpha)} \forall l\ A(l)} \qquad (List\ (\alpha)\ \text{induction})$$

This gives rise to a recursion operator $rec_{List\ \alpha}$ with the obvious operational

Axioms for the sort N include

$$\forall x : N(x = x) \quad \forall x : N \forall y : N(x = y \to y = x)$$
$$\forall x : N \forall y : N \forall z : N(x = y \wedge y = z \to x = z)$$
$$\forall x : N \forall y : N(x + y = y + x)$$
$$\forall x : N \forall y : N \forall z : N(x + (y + z) = (x + y) + z)$$
$$\forall x : N(x + 0 = x)$$
$$\forall x : N \forall y : N(x + s(y) = s(x + y))$$

Structural induction rule generatedby$N$:

$$\frac{\vdash a : P(0 : N) \wedge \forall x : N(P(x) \to P(s(x)))}{\vdash rec_N(A) : \forall x : NP(x)}$$

Associated reduction rules:

$$rec_N(A)0 : N \succ \pi_1(A)$$
$$rec_N(A)s(x) : N \succ \pi_2(A)rec_N(A)x$$

Figure 2: The sort, N, of natural numbers with some of its axioms, the associated induction rule and the operational meaning of the rec_N operator.

meaning:

$$rec_{List(\alpha)}\ \epsilon_\alpha AB \succ A$$
$$rec_{List(\alpha)}\ (h :: t)AB \succ Bh(rec_{List(\alpha)}\ tAB)$$

3.1.2 New Predicates and Functions

An important constructive proof idiom is that of predicate definition. 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 to use common "proof patterns". In `Fred`, we introduce a metalogical abbreviation $P(x)$for a formula $F(x)$ (with zero or more occurrences of the variable x) by:

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

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 proof term $q(z)$ where z is a proof term of type P:

The Metarule "Template": If $set\ P(x) \equiv F(x)$ then we add the rule:

$$\frac{\vdash f : F \quad z : P \vdash q(z : P) : Q(P)}{\vdash q(f : F/z : P) : Q(F/P)}$$

That is, if we have formula Q which contains occurrences of the formula P as sub-formulae, then we may substitute the formula F for P in Q at some or all occurrences. The converse is also a rule. Of course in doing this we must avoid clashes of variable.

Thus *Template* is a means of abstracting a proof over a "formula variable". Defining it as a structural rule is a means of avoiding higher order quantification of formula variables (as in Huet, Kahn and Paulin-Mohring [HKPM97]) – although this could be achieved by creating a new sort (logical formulae) with a universe hierarchy (as in Martin-Löf [ML84]).

3.2 The Computational Type Theory (CTT)

Our computational type theory is the programming language *ML*, although it might just as easily be LISP or C++. Any language $\mathcal{L}$ for which there is a mapping from terms of simply typed lambda calculus with products, disjoint unions and parametrized types into $\mathcal{L}$ will work.

We define an *extraction mapping* **extract** from proof terms in the LTT to terms of *ML*. Each sort is mapped to a corresponding *ML* type. For any sort[6] s, we assume that all the $f \in F_s$ (where the subscript S indicates the sorts involved) are mapped to programs for functions which satisfy the appropriate axioms Ax.

For instance, consider the sort of natural numbers. We assume that the *ML* program corresponding to $+$ satisfies axioms including those given in [Fig. 2] for the addition function. The predefined *ML* function for addition will suffice, with the sort N being mapped to the *ML* type `Int`.

Theorem 1. *Given a proof term* $p : \forall x : s_1 \exists y : s_2 A(x, y)$ *in the logical type theory, there is a program* f *in the computational type theory ML such that* $A(x : s_1, f(x) : s_2)$ *is a theorem and the extracted program,* $f = \mathbf{extract}(p)$*, has ML type* $\mathtt{s_1} -> \mathtt{s_2} * \mathtt{s_3}$ *where* s_3 *is the type of the computational content of* $A(x, y)$.

The proof (see [AC96], [Poe99], or [CPW00]) involves the map, **extract**, from proof terms to terms of the simply typed lambda calculus by first "deleting" computationally irrelevant proof terms: that is, by removing [true] Harrop formulae from deductions, and then extracting the value from the first part of the proof term.

3.3 Protocol between the CTT and the LTT

The protocol works as follows:

From *LTT* to *CTT* : We use the function **extract**.

Suppose d is the C-H term for a proof of a formula $\forall x : s_1 \exists y : s_2 A(x, y)$ for some sorts s_1 and s_2. Our process **extract** yields a program (in our CTT). $\mathbf{extract}(d)$ is the program for the function f_A in the CTT, such that $A(x : s_1, f_A(x) : s_2)$. This amounts to Skolemization.

From *CTT* to *LTT* : We have symmetrically: to use an *ML* program in the logical calculus, an axiom guaranteeing that the program does what it is supposed to do, i.e. is correct, must be added *consistently* to the axiom system.

[6] Note that each parametrized sort $s : (Sort_1 \to Sort_2)$ corresponds to a parametrized type.

Obviously, checking for consistency may be an onerous task. Here are two examples: 1. Model checking techniques could be used in some cases. 2. In earlier work we have used the protocol from the LTT to CTT, extracted a program from a proof term in the LTT and then added this program back into the LTT with an axiom which is a Skolemzied version of the theorem from whose proof it was originally extracted.

In any case if we are given a program $\mathbf{f\_}A$ for a function $f_A : s_1 \rightarrow s_2$, that satisfies a specification $A(x, f_A(x))$, then we add the axiom $\forall x : s_1 A(x, f_A(x))$ and, of course, interpret f_A by the action of $\mathbf{f\_}A$.

This protocol makes programs much shorter. For example, we do not have to prove the Peano axioms for addition and then generate a program from that proof. We can simply use our old (user-guaranteed reliable) *ML* program. Further, we can hide large parts of programs in other (smallish) programs that are just called when needed.

Here is an example. Suppose we have a proof that for all x there is a y greater than x such that y is prime:

$$t : \forall x : N \exists y : N(Prime(y) \wedge y > x)$$

By Skolemization, we have the Harrop formula

$$() : \forall x : N(Prime(f(x)) \wedge f(x) > x)$$

and we know that f is a unique function representing $\mathbf{extract}(t)$ in the CTT. f and its associated Harrop formula can be used in future proofs in exactly the same way as any other function constant and its Harrop axioms (for example, just like $+$ and the axioms for addition). Later we shall give much more complicated examples.

A related proof idiom is *Function definition*. This involves both the LTT and the CTT. For instance, the function $length_\alpha : List\ (\alpha) \rightarrow N$ is given by the following axioms

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

These axioms define a total function $length_\alpha$ in the LTT.

We are required to specify a corresponding program in the CTT. We associate the irreflexive CTT operation of *computing* with the reflexive LTT equality $=$.

The corresponding program in the CTT is obtained by taking this axiomatization which gives a (total) recursive definition and then: 1. UnSkolemizing to a form[7] $\forall x : List(\alpha) \exists y : N A(x, y)$, next 2. proving this by induction, and finally 3. extracting our *ML* program obtaining:

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

[7] $\forall x : List(\alpha) \exists y : N \forall f : List(\alpha) \rightarrow N$
$((x = \epsilon_\alpha \rightarrow f(x) = 0 \wedge \forall a : \alpha \forall l : List(\alpha)(x = \langle \alpha \rangle :: l \rightarrow \bar{1} + f(l))) \rightarrow f(x) = y)$

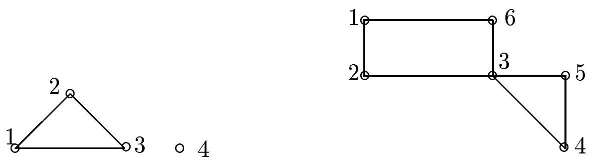

Figure 3: Two sample graphs

Note that, in larger proofs when we are anxious to reduce the size of the term (program), we may choose to implement the associated program in a manner different from that suggested by the axiomatization. This is an important feature of our approach – intensionally distinct programs in the CTT correspond to extensionally interchangeable functions in the LTT. Of course, the programs extracted from our system are only as correct with respect to the axiomatization as these defined programs are.

As noted above, axiomatizations of functions in the LTT and their associated computational definitions in the CTT are separate. In many constructive proofs, functions are not proved and extracted: instead, a total function is defined by an axiomatization.

4 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 this paper we take a concrete representation of graphs but it is generally desirable to use a more abstract representation. This can be done by using a parametrized specification but this is beyond the scope of this paper (see [PJM00]).

Here the vertices will be represented by positive integers. Consider the first graph in [Fig.3] with four vertices 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 $List(N)$. Not all lists of elements of sort $List(N)$ correspond to graphs: in a graph the edge relation is irreflexive and symmetric. The list above has the properties

1. The n^{th} member of the list is a list of numbers beginning with n.
2. (*Symmetry*) If the n^{th} member of the list is a list containing m and $m \neq n$, then the m^{th} member of the list is a list containing n.
3. Each member of the list is a repetition-free list of numbers.[8]

These properties are expressible in our formal system for $\mathcal{G}$ with the aid of certain extra function symbols, which we now define. Note that each function is provably total in the formal system.

Here is the list of some of the functions required in F_{List_α} and the associated axioms. All formulae are considered to be universally closed. We note that

[8] This ensures that the edge relation is irreflexive and that no pair of vertices are joined by more than one edge (i.e. the graph is a simple graph).

appropriate *ML* definitions can be generated automatically as in the previous section.

1. A binary function $member_N$ of two arguments: a natural number, n, and a list.[9] The function computes the nth member of the list. Since all functions are total we will need to use a "default value"[10] for cases where n is larger than the length of the list or where $n = 0$.
2. Position function, *listpos*. The 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.
3. Initial segment of size k of a list l: $initlist(k, l)$.
4. Tail segment of size k of a list l: $tail(l, n)$.

4.1 Cycles in Even Parity Graphs

We set a predicate $graph(l)$ to mean that a list of lists of natural numbers, l (therefore of sort $List(List(N))$), represents a graph.[11] The formula $graph(l)$ is defined in `Fred` by the conjunction of four Harrop formulae:

$$
\begin{aligned}
set\ graph(l) \equiv\ & length(l) \leq 1 \longrightarrow \perp \wedge \\
& \forall i : N(1 \leq i \leq length(l) \rightarrow member_N(1, member_{List(N)}(i, l)) = i) \wedge \\
& \forall i : N\ (1 \leq i \leq length(l) \rightarrow repfree(member_{List(N)}(i, l)) \wedge \\
& \forall i : N \forall j : N(((1 \leq i \leq length(l) \wedge (1 \leq j \leq length(l) \wedge j \neq i)) \longrightarrow \\
& listpos(j, member(i, l)) \neq \epsilon \longrightarrow listpos(i, member(j, l)) \neq \epsilon))
\end{aligned}
$$

where $repfree(l)$ is a predicate defined by

$$set\ repfree(l) \equiv \forall n : N(length((listpos(n, l)) > 1) \longrightarrow \perp)$$

A graph has *even parity* if the number of vertices adjacent to each vertex is even.[12] So each list in l must have an odd number length. We therefore define a predicate $evenpar(l)$.

For the proof we begin tracing a path with the first vertex, 1, say, till we find a different vertex and then scan the (tail of) its list of neighbours for the first vertex not equal to 1. Continuing in this manner we can construct a list of adjacent vertices $\langle 1, 2, 3, 1, \ldots \rangle$ of arbitrary length. Such a list defines a *walk* in the graph. The *first* occurrence of a repeated vertex yields a cycle represented by the sublist of the vertices between the repeated vertices. Note that the desired sublist does not necessarily begin at the vertex we start from.

---

[9] For lists of elements of sort α we use $member_\alpha$ as the function letter.

[10] Note that the default value for the first case below is 0. Because all our graphs contain only positive integers, it is always the case that when we apply our functions to lists of vertices we shall be able to decide whether we are getting a vertex or the default value.

[11] We shall exclude trivial graphs consisting of one or zero vertices.

[12] A referee has pointed out that the the actual predicate used could be weakened.

4.2 The Proof of the Main Theorem

In `Fred`, just as in mathematics practised by mathematicians, we can examine a proof at various levels of "granularity". In this section we examine the topmost level, where the required theorem is proved using several lemmas. We omit most of the details.

If c is a list of numbers then we define a predicate *cycle* in the obvious way.

To start the construction we also need a function, which we call *start*, that 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.

We need a function *genlist* that generates a list of adjacent vertices from the list l (that specifies the graph). First we define $gen(l, m)$ which is either identically zero (in the case that l has no edges) or the function is never zero and $gen(l, m)$ and $gen(l, m+1)$ are adjacent vertices for every m. Next we use *gen* to give $genlist(l, n)$ which gives the vertex for the n^{th} stage of construction (starting from 0). If l is a list corresponding to an even-parity graph, then $genlist(l, n)$ corresponds to $\langle gen(l, n), ...gen(l, 0)\rangle$.

The Main Theorem we want to prove is

$$\forall l : List(List(N))\ (evenpar(l) \wedge start(l) \neq 0 \rightarrow \exists c : List(N)(cycle(c, l)))$$

This says that if l represents a graph which does not consist entirely of isolated vertices, then l contains a (non-trivial) cycle. The predicate $genlistGivesWalk(l)$ stands for the statement that the function *genlist* generates walks in the graph l; from these walks we wish to extract a *cycle*:

$$\begin{aligned} &set\ genlistGivesWalk(l) \equiv \\ &\quad evenpar(l) \wedge start(l) \neq 0 \rightarrow \forall m : N\ (m > 0 \rightarrow walk(l, genlist(l, m))) \end{aligned}$$

Note that $genlistGivesWalk(l)$ is represented by a Harrop formula and therefore it does not contribute to the computation and nor does its proof. It therefore does not matter whether we establish this constructively or even classically (cf. the footnote on p. 101 in Kreisel [Kre59]). We can just take it as a new (computational-content-free) axiom.

The proof of the Main Theorem relies on the following lemma which states that it is provable that any list of numbers is either repetition free or the list contains an element (say a) such that for some tail segment of the list the element a occurs exactly twice in the segment and no other element occurs more than once in this tail segment.

$$\forall l : List(N)(repfree(l) \vee ListHasUniqueEltOccursTwiceInTail(l))$$

where $ListHasUniqueEltOccursTwiceInTail(l)$ is a new predicate that we define.

The *ML* program extracted for the lemma is called `Cgr21`, see [Fig. 5], and it calls two other programs `KSC158` and `Cgr20` corresponding to other lemmas used in the proof of the lemma.

We can prove

$$\vdash repfree(genlist(l, length(l) + 1)) \rightarrow \bot$$

```
let main = let fun96 l X =
 begin match ((Cgr21 (genlist l (s (length l ) )) )) with
     inl(g) ->  [ ]
     | inr(g) -> ((let fun97 X40 =
 (select (X40) (let fun98 b =
 let fun99 X43 =
 (select (X43) (let fun100 c =
 let fun101 X44 =
 (app ((pi1 X44)) (let fun102 y =
 (initlist (y+1) (tail (genlist l ((length l )+1 )) c))
 in
 fun102)) in fun101 in fun100)) in fun99 in fun98))
 in
 fun97)g)
 end in  fun96
 ;;
```

Figure 4: The *ML* program for the final program

but we observe that this (true) formula is Harrop and therefore has no computational content, so we can take this formula as an additional axiom.

This Harrop axiom together with the formula obtained from the lemma above by $\forall$-elimination give us

$$\begin{aligned} \mathit{evenpar}(l) \vdash\ & \mathit{ListHasUniqueEltOccursTwiceInTail}(\mathit{genlist}(l, \mathit{length}(l)+1)) \wedge \\ & \mathit{repfree}(\mathit{tail}(\mathit{genlist}(l, \mathit{length}(l)+1), k+1)) \end{aligned}$$

We can also obtain the two theorems

$$\begin{aligned} & \vdash \mathit{start}(l) \neq 0 \wedge \mathit{evenpar}(l), \text{ and} \\ & \mathit{ListHasUniqueEltOccursTwiceInTail}(\mathit{genlist}(l, \mathit{length}(l)+1)) \\ & \qquad\qquad\qquad\qquad \vdash \exists c : \mathit{List}(N)\ (\mathit{cycle}(c, l)) \end{aligned}$$

These two proofs give our theorem.

The use of the Harrop formulae allows us to make significant reductions in the size of the program we extract. Adding an automated theorem prover could reduce the work involved in obtaining the proof and tactics could be employed to reduce the size of the proof (see [Section 7]). The final program calls the program `Cgr21` (see [Fig. 5]) for the Main Lemma and is as in [Fig. 4].

Note that this function takes an input l for the graph we want to use and also an input X which *should* stand for a term mapped by **extract**() from a proof that $\mathit{evenpar}(t) \wedge \mathit{start}(t) \neq 0$. However, that statement is Harrop, so X can be anything (because it is not used in the computation).

This program is relatively easy to read because the code mirrors the structure of the proof. In particular, the calls to other functions correspond directly to references to lemmas in the main proof. However, the program for `Crg21` is more complicated but could be made more readable if comments were generated automatically from the proof. This however is work for the future.

```
let Cgr21 =
 let rec fun80 l =
   begin match l with
   [ ] -> inl(let fun100 x = (s 0 ) in fun100)
 | h::t -> let fun81 z =
  let fun82 l =
  let fun83 X217 =
  begin match (X217) with
inl(g) -> ((let fun92 X218 =
  begin match ((((Cgr20 l) X218) z)) with
inl(g) -> (inl(let fun98 x =
  begin match (((KSC158 x) z)) with
                inl(g) -> (0)
      | inr(g) -> (X218 x)
      end
  in fun98))
      | inr(g) -> ((let fun93 X221 =
  inr((z,(0,((X221),
let fun94 x =
  (app X221 (let fun95 y =
(X218 x) in fun95))
  in fun94))))
  in fun93) g)
    end
   in
   fun92) g)
     | inr(g) -> ((let fun84 X219 = (select X219 (let fun85 b =
  let fun86 X241 = inr((b,(select X241 (let fun87 c =
  let fun88 X242 = ((s c ),((pi1 X242), let fun89 x =
  ( (pi2 X242) x)
  in fun89)) in fun88 in fun87))))
  in fun86 in fun85))
  in fun84) g)
     end
  in fun83 in fun82 in
 fun81 h t (fun80 t)
 end
in fun80
 ;;
```

Figure 5: *ML* program (for `Cgr21`) extracted from proof of the Main Lemma: $\forall l : List(N)(repfree(l) \vee ListHasUniqueEltOccursTwiceInTail(l))$

5 The Fred Environment

Fred provides an advanced GUI for modular proof development,[13] see the screenshot in [Fig. 6].

The environment draws inspiration from visual programming techniques [BGL95]. Over the past 10 years, visual programming environments (for example, the Microsoft Visual Suite or Borland C++ Builder) have become ubiquitous

[13] Please note that Fred currently runs under Windows.

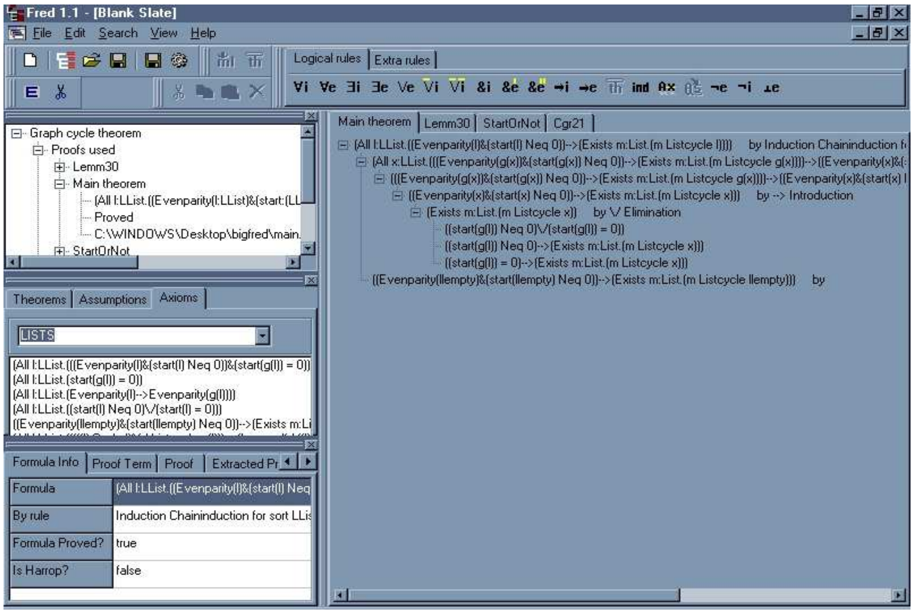

Figure 6: Screenshot of the `Fred` prover.

for rapid application development. This success may be attributed to the fact that such environments provide a visual representation of appropriate programming practices (for example, building a GUI by selection and placement of visual components). From our experience with `Fred`, the same advantages can be applied in the domain of logical proofs. This is essentially because, as Frege [Fre79] realized, logical processes are often better presented in two-dimensional visual representations.

We outline these processes and their visual representations in `Fred`:

- `Fred` uses a Multiple Document Interface allowing for many proofs to be edited at the same time in separate child windows. This is analogous to developing modules or object-oriented classes in an IDE (Integrated Development Environment), or editing several documents using a Word Processor.
- Proofs are presented in tree form, using a navigation directory tree visual component (commonly used for file explorer programs under Windows or MacOS operating systems). Proofs are made using backward reasoning – applying a valid rule to a node will generate several child nodes, each corresponding to a subgoal.
- Application of a rule is made by selection of a (sub)goal node, together with selection of a rule button from a menu. The menu has buttons for the

standard natural deduction rules of [Fig. 1] and new, domain-specific tactics.

- Formulae are validated as axioms, theorems or assumptions by a drag-and-drop action. For example, if a goal formula has been proved in another window, the user may validate the goal by selecting the proved formula, and then dragging and dropping it onto the goal formula. This is analogous to the drag-and-drop action used to construct a GUI in programming IDEs.
- Each proof node has associated information, including current status within a proof, specifications (see below), extracted program, and lambda calculus proof term, which can be displayed in table format. This information may be kept on-screen to aid the proof process.

Users construct collections of proofs via a *proof project*: this is analogous to the programming projects used in many IDEs, and provides a hierarchical means for the collection and maintenance of the structures associated with a proof. A proof project includes the following features:

- Known or new proofs may be imported or added into a proof project, and referenced in other proofs. So, a proof may use other proofs in the project (subject to circularity constraints). This graphically reflects the manner in which mathematicians solve a theorem by breaking it into separate related lemmas.
- A project in which everything is proved may be exported into a library of *theorems*. Such theorems can be imported for use into a proof project, but are read-only and cannot be edited further. In this sense, theorems are analogous to byte-coded components in an IDE.
- Sorts, function symbols, relation symbols and axioms can be imported into a project as *specifications*. Any number of user-designed specifications may be used in proofs. We attempt to make this appear, visually, as analogous to importing classes or modules in an IDE. The axioms and theorems are located on the screen according to their associated specifications.
- Given a goal formula $\forall x : s\, P(x)$, proved by induction, the structural induction rule required is generated automatically according to the definition of the sort s in its specification. Other inductions are generated according to user-designed induction schemata which may be imported into a proof project.

A proof project is represented visually via a *project explorer tree* in which all available structures are presented using a tree structure. This component serves as a enables the user to navigate effectively through the active proofs and theorems of the project. It is also used for the drag-and-drop instantiation of subgoals of a proof using the proofs and theorems already displayed in the explorer tree.

6 Demonstration Results

Finally we present some practical results. Here is the result for the left hand (even parity) graph in [Fig. 3], [Section 4].

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

Next we consider the right hand (even parity) graph in [Fig. 3] and extract a cycle 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 = [3; 5; 4; 3]
```

In the cases where the graph does not have any cycle we get a default result. Here are two examples.

```
#main [[1];[2];[3];[4];[5];[6]];;
- : int list = [0; 0]
```

```
#main [[];[];[]];;
- : int list = [0; 0]
```

7 Conclusions and Future Work

So far we can automatically produce programs of manageable size that follow the structure of the proof.

In the future we plan to classify mathematical techniques into standard forms, to add a tactic language and an automatic theorem prover front end to `Fred`, to automatically generate comments in the extracted code and also to include algebraic specifications in `Fred` (see [CPW00] for a description of the theory).

Acknowledgements

Thanks to Martin Wirsing for comments in preparation for the presentation at Schloss Reisensburg and later, and special thanks to John Jeavons and Bolis Basit for the proof of the theorem from both authors. Finally, thanks to three anonymous referees who provided incisive comments and helpful suggestions.

References

[AC96] D. Albrecht and J.N. Crossley. Program extraction, simplified proof-terms and realizability. Technical report 96/275, Monash University Department of Computer Science, 1996.

[And93] P. Anderson. *Program Derivation by Proof Transformation.* Phd thesis, Carnegie Mellon University, 1993.

[BGL95] M. Burnett, A. Goldberg, and T. Lewis. *Visual Object-Oriented Programming: Concepts and Environments.* Prentice-Hall, 1995.

[BS95] U. Berger and H. Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, IN, USA, October 1994*, volume 960 of *Lecture Notes in Computer Science*, 1995.

[Cal98] J. L. Caldwell. Moving Proofs-As-Programs into Practice. In *Automated Software Engineering, Proceedings 12th IEEE International Conference*, pages 10–17. IEEE Computer Society, 1998.

[Con97] R. L. Constable. The structure of nuprl's type theory. In *Logic of Computation (Marktoberdorf, 1995)*, volume 157 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 123–155. Springer, Berlin, 1997.

[CPW00] J. N. Crossley, I. Poernomo, and M. Wirsing. Extraction of structured programs from specification proofs. In D.Bert, C.Choppy, and P.Mosses, editors, *Recent Trends in Algebraic Development Techniques (WADT'99)*, volume 1827 of *Lecture Notes in Computer Science*, pages 419–437. Springer, Berlin, 2000.

[CS93] J. N. Crossley and J.C. Shepherdson. Extracting programs from proofs by an extension of the Curry-Howard process. In J. N. Crossley, J. B. Remmel, R. Shore, and M. Sweedler, editors, *Logical Methods: Essays in honor of A. Nerode*, pages 222–288. Birkhäuser, Boston, Mass., 1993.

[Fre79] G. Frege. *Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle, Berlin, 1879.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types.* Cambridge University Press, 1989.

[Hen50] L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[HKPM97] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof assistant Reference Manual: Version 6.1. Coq project research report rt-0203, Inria, 1997.

[HN88] S. Hayashi and H. Nakano. *PX, a computational logic.* MIT Press, Cambridge, Mass., 1988.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J.R.Seldin and R.J.Hindley, editors, *To H.B. Curry : Essays on combinatory logic, lambda calculus, and formalism*, pages 479–490. Academic Press, London, New York, 1980.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in mathematics*, pages 101–128. North Holland Publishing Co., 1959.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[PJM00] I. Poernomo, J.N.Crossley, and M.Wirsing. Programs, proofs and parametrized specifications. Submitted to *WADT 2001*, 2000.

[PMW93] C. Paulin-Mohring and B. Werner. Synthesis of ml programs in the system coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[Poe99] I. Poernomo. Extracting ML programs from Intuitionistic proofs. Technical report, Monash University School of Computer Science and Software Engineering, 1999.

[Zha94] Luo Zhaohui. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.