PROGRAM DEVELOPMENT THROUGH PROOF TRANSFORMATION

Frank Pfenning[1]

Abstract. We present a methodology for deriving verified programs that combines theorem proving and proof transformation steps. It extends the paradigm employed in systems like NuPrl where a program is developed and verified through the proof of the specification in a constructive type theory. We illustrate our methodology through an extended example — a derivation of Warshall's algorithm for graph reachability. We also outline how our framework supports the definition, implementation, and use of abstract data types.

1 Introduction

Program development through theorem proving in a constructive logic or type theory has been suggested in many places in the literature (see, for example, [13,11,1,5]). Example programs illustrating this approach have been derived in [12,17,1,18,15]. NuPrl [4] provides sophisticated machine support for program development using this *proofs as programs* paradigm. It allows extraction of verified programs from completed proofs, unverified programs from partial proofs, and the writing of tactics to automate part of the program synthesis and verification process [3,8].

Goad in [6] showed that proof transformations can improve the efficiency of extracted programs in the context of *specialization*, that is, in situations where a very general program is applied to inputs satisfying some given constraints. The use of proof transformations in his work is essential, since the functionality of the specialized program in general may be different from the functionality of the original program, but they both satisfy the same specification.

In this paper we suggest an enrichment of the NuPrl methodology for developing programs by incorporating proof transformations. This enrichment also generalizes Goad's work and addresses two commonly made criticisms of the *proofs as programs* approach to programming.

The first criticism relates to the efficiency of the code extracted from proofs. It seems that one could derive only purely functional programs, and that one could therefore not expect efficient implementations of the specifications. More seriously perhaps, one is trained

1980 *Mathematics Subject Classification* (1985 *Revision*). 68N05.

[1]This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404.

to find short, elegant proofs rather than long, opaque, and laborious ones that may yield efficient programs. As in everyday programming, elegance and efficiency often conflict — a point illustrated if one compares the initial and final version of Warshall's algorithm (see Proofs 8 and 20 or Programs 9 and 21).

With proof transformations the programmer has the luxury of writing a short and elegant proof and then reshaping it in small steps into an opaque proof that corresponds to an efficient program. Of course, proof transformation is not a cure-all. Selection of the steps often requires considerable ingenuity — the programmer has to rely on his analysis of the problem and of the program's efficiency. We hope that the extensive example in this paper will help to illustrate that proof transformation provides a practical tool for deriving verified efficient implementations of very abstract specifications.

Destructive operations can in many cases be introduced during compilation of the extracted program. Advanced compilation techniques for functional programs as described in [9] yield a very efficient, destructive implementation of the functional program extracted from our final proof.

The second criticism relates to the ability to maintain and modify proof objects. It seems difficult to react to changing specifications when programs are defined through proofs. This problem is of course not unique to this approach; traditional programming suffers from a similar problem. However, in the realm of completely verified programming this problem is compounded by the inability to make local changes to a proof structure.

Proof transformations provide a way of making those local changes and propagating them throughout the proof. This process very clearly identifies places where additional theorem proving is required in order to meet the changed specification or verify the modified program. This is in sharp contrast to traditional programming where maintenance under changing requirements is one of the most difficult and costly phases in the software life-cycle and relies entirely on the programmer's understanding of the code he has to modify.

We illustrate proof transformation techniques and their applications by means of an extended example. We derive Warshall's algorithm for testing whether there is a path connecting two nodes in a directed graph.

The basic mechanisms for proof transformation are well-known from proof theory. They introduce a lemma followed by a number of *proof reductions*. Proof reductions take somewhat different but closely related forms in different proof systems. In sequent calculi they are simply cut-elimination steps. In natural deduction systems they appear as normalization steps, in the λ-calculus they are β-reductions. Surprisingly, the meta-programs that control how these steps are applied are *proofs* of simple theorems (see Section 2.3).

The class of transformations that can be described this way is limited by the expressive power of the logic under consideration. We are therefore now considering using LF [7] as a formal system for describing a logic. This allows the formal statement and proof of meta-theorems. The proof of a meta-theorem then may be used to transform proofs in the object logic.

These basic transformations also allow the use of abstract data types and different implementations of them in the program development process. We illustrate this in the example of Warshall's algorithm where the initial specification mentions the set of nodes, while the final implementation assumes that nodes are represented as integers.

Because of the universality of the the transformations, we have formulated our examples in a general way which concentrates on the structure of the proofs and not on the underlying

logic or type theory. The proofs involved could easily be implemented in a system like NuPrl.

However, to implement the *proof transformations* one would need explicit support for management of different generations of a proof. Another problem is that proof steps accomplished by tactics (and almost all are!) must be transparent, that is, one should be able to either (a) expand them into their more primitive components, or (b) derive composite transformations from the definition of tactics. NuPrl provides for solution (a). We are currently investigating if the more economical (b) could be achieved through formal justification of tactics by meta-proofs in a meta-theory. We are currently implementing a system that explicitly provides formally proven tactics through a meta-logical formalism based on LF [7]. Their operational interpretation is based on ideas from λProlog [14].

2 Warshall's Algorithm

In this section we will derive a version of Warshall's algorithm for determining whether there is a path from a node x to a node y in a finite, directed graph.

We chose Warshall's algorithm for several reasons. Firstly, it starts from a specification which is easy to write down and understand. Secondly, the resulting program is relatively complex and relies on destructive operations. This means that the conceptual distance between final implementation and initial specification is so wide that it is hard to see how the whole development could have been carried out without using any proof transformations, relying solely on theorem proving. Of course, theoretically this is possible, since our final result is no more than a proof of the specification. Finally, a formal derivation of Warshall's algorithm is presented by Broy and Pepper in [2]. This allows the reader to compare our approach to the wide-spectrum derivation approach as taken by the CIP group [16] which was used in [2].

We start by deriving the original function in [2]. It can be extracted from a very natural proof of the specification. In several steps we then transform this initial proof into one describing Warshall's algorithm.

2.1 Definitions

The logic in which we would mechanize this development would most likely be a strongly typed, higher-order logic. The strong typing aids the mechanization. The higher-order nature of the logic is convenient for expressing general facts, like Lemma 13. Thus we can use the logic itself to some extent as a meta-logic for describing proof transformation in the guise of higher-order theorems.

In the presentation below we will often omit types for the sake of brevity. $\mathcal{P}$ stands for the type of propositions. We use ":" for types of bound variables, "." to separate a quantifier from its scope, and "$\rightarrow$" to form function types.

Definition 1 *Given a finite set V. Then a* graph *is a pair (V,E) such that $E \subseteq V \times V$. Let $\mathcal{G}$ be the type of graphs.*

Definition 2 *Given a graph $G = (V,E)$. For $v, w \in V$,*

$$\mathsf{path}^{(V,E)}(v,w) = \{\langle u_1,\ldots,u_n\rangle : v = u_1, u_n = w, \langle u_i, u_{i+1}\rangle \in E \text{ for } 1 \leq i < n \text{ and } 2 \leq n\}$$

Theorem 3 (Graph Reachability). *Given a graph (V,E) and $x,y \in V$. Then either there is a path from x to y, or not.*

Definition 4 (Induction for a finite set V).

$$I_V \equiv \forall P\,.\,P\{\} \wedge \forall W{\subseteq}V\ \forall z{:}V\ (PW \wedge z \notin W \supset P(\{z\} \cup W)) \supset \forall W{\subseteq}V\,.\,PW$$

In order to have a constructive meaning of a proof using this induction principle, we will assume the decidability of $W = \{\}$ and the existence of a choice function "choose" such that $W \neq \{\} \supset \text{choose}\, W \in W$ for all $W \subseteq V$. We will call such a set an *enumerated set.*

2.2 Deriving an Initial Algorithm

Let us first informally describe some of the basic ideas of Warshall's algorithm. Given an enumeration $x_1, \ldots, x_n$ of the nodes in a graph G, and a pair $\langle x,y\rangle$. In the first iteration, we check if x and y are directly connected by an edge. During the i^{th} iteration we check if there is a path from x to y whose interior nodes (excluding x and y) are contained completely within $x_1, \ldots x_{i-1}$, making use of the results computed during the $i-1^{\text{st}}$ iteration.

We will now try to capture this idea formally and simultaneously make it more precise. Our specification is the following theorem.

Theorem 5

$$\forall (V,E){:}\mathcal{G}\ \forall x,y{:}V\,.\,(\exists p\,.\,p \in \mathsf{path}^{(V,E)}(x,y) \vee \neg\exists p\,.\,p \in \mathsf{path}^{(V,E)}(x,y))$$

The basic idea is to prove the theorem by induction over V (but not $|V|$, which could eventually lead to a breadth-first search of the graph). The set will be the vertices which we allow the path to contain in addition to x and y. In order to prove this we need to find the right induction hypothesis. This generalized theorem will then be a lemma.

Definition 6 *Given a graph (V,E) and $W \subseteq V$. Then $\mathsf{path}_W^{(V,E)}(x,y)$ is the set of paths from x to y all of whose interior nodes are contained in W.*

In the following, we will often omit the superscripts (V,E) to path.

Lemma 7 *Given (V,E). Then*

$$\forall W \subseteq V\ \forall x,y \in V\ (\exists p\,.\,p \in \mathsf{path}_W(x,y) \vee \neg\exists p\,.\,p \in \mathsf{path}_W(x,y))$$

Proof 8 *The proof is by induction on W. We use $W = z \uplus W'$ to mean $z \notin W' \wedge W = \{z\} \cup W$.*

Basis $W = \{\}$. *For any x,y:*

1. *If $\langle x,y\rangle \in E$ then $\langle x,y\rangle \in \mathsf{path}_{\{\}}(x,y)$. Hence $\exists p\,.\,p \in \mathsf{path}_{\{\}}(x,y)$.*
2. *If $\langle x,y\rangle \notin E$ then $\neg\exists p\,.\,p \in \mathsf{path}_{\{\}}(x,y)$.*

Step $W = z \uplus W'$. *We proceed by cases, using the induction hypothesis for x as x, y as y and W as W.*

1. $\exists p \,.\, p \in \mathsf{path}_{W'}(x,y)$. *Then for that path* p, $p \in \mathsf{path}_W(x,y)$.
2. $\neg\exists p \,.\, p \in \mathsf{path}_{W'}(x,y)$. *Again we use cases, using the induction hypothesis twice, once for* x *as* x *and* z *as* y, *and once for* z *as* x *and* y *as* y.
   (a) $\exists p \,.\, p \in \mathsf{path}_{W'}(x,z)$ *and* $\exists p' \,.\, p' \in \mathsf{path}_{W'}(z,y)$. *Then* $p{\cdot}p' \in \mathsf{path}_W(x,y)$.
   (b) $\neg\exists p \,.\, p \in \mathsf{path}_{W'}(x,z)$ *or* $\neg\exists p' \,.\, p' \in \mathsf{path}_{W'}(z,y)$. *Then* $\neg\exists p \,.\, p \in \mathsf{path}_W(x,y)$. *To see this, assume not and call the path* p. *If* p *does not contain* z, *then* $p \in \mathsf{path}_{W'}(x,y)$ *which is a contradiction. If* p *does contain* z, *then the two sections of* p *are in* $\mathsf{path}_{W'}(x,z)$ *and* $\mathsf{path}_{W'}(z,y)$, *which is a contradiction, too.*

□

Now we consider the program that is extracted from this proof. In order to make the program more readable we have made a few stylistic changes while leaving the basic computational properties intact. However, one major change should be noted: instead of returning the path in case there is one, this simplified program will only return a truth value.

Program 9 *The algorithm corresponding to this proof is almost exactly like the original algorithm in [2].*

```
ispath ⇐ λV λE λx λy . rp V E V x y
rp ⇐
  λV λE λW λx λy .
    if W = {}
      then ⟨x,y⟩ ∈ E
      else let z ⊎ W* = W in
             rp V E W* x y or
               (rp V E W* x z and rp V E W* z y)
```

We use pattern binders to avoid explicit destructors for enumerated sets. Thus "**let** $z \uplus W^* = W$ **in** ..." means "**let** $z = \text{choose}\, W$ **and** $W^* = W - \text{choose}\, W$ **in** ...".

The function above has a glaring inefficiency: because of the multiple recursive calls, function values may be recomputed many times. This is a common situation in program derivation and can often be resolved by introducing so-called accumulator arguments. Here it turns out to be appropriate to maintain an array of previously computed values.

2.3 Using Proofs to Transform Proofs

What is the basic computational mechanism that allows us to transform proofs? Not surprisingly, proof reduction (cut-elimination, normalization) steps and their inverses are crucial. Reduction is employed as the computational workhorse, but the main transformation that allows us to set up the reductions is *lemma insertion.* We use script letter $\mathcal{D}$ and $\mathcal{E}$ to stand for proofs. They are placed above the formula they justify.

Transformation 10 (Lemma Insertion). *Given a proof* $\mathcal{E}$ *of* $\phi \equiv \psi$. *Then we can transform*

$$\begin{array}{c}\mathcal{D}\\ \phi\end{array}$$

to

$$\dfrac{\dfrac{\mathcal{D}}{\phi} \qquad \dfrac{\mathcal{E}'}{\phi \supset \psi}}{\psi} \qquad \dfrac{\mathcal{E}''}{\psi \supset \phi}}{\phi}$$

where $\mathcal{E}'$ and $\mathcal{E}''$ follow from the lemma $\phi \equiv \psi$.

Lemma insertion has the property that it does not change the theorem we are trying to prove, but of course it creates a new intermediate theorem, ψ, that will sometimes be of interest.

The logic in which we formalize the deductions will typically not allow a normalization theorem due to the presence of a (finitary) induction rule. However, if we have an assumption $\forall n \,.\, \phi(n) \equiv \psi(n)$, an induction proof for $\forall n \,.\, \phi(n)$ can be transformed into an induction proof of $\forall n \,.\, \psi(n)$. The following illustrates this for induction over the natural numbers.

Transformation 11 (Lemma Insertion in Induction). *Given a proof $\mathcal{E}$ of $\forall n \,.\, \phi(n) \equiv \psi(n)$. Then transform*

$$\dfrac{\dfrac{\mathcal{D}_B}{\phi(0)} \qquad \dfrac{\mathcal{D}_S}{\forall n \,.\, \phi(n) \supset \phi(n+1)}}{\forall n \,.\, \phi(n)}$$

to

$$\dfrac{\dfrac{\dfrac{\mathcal{D}_B}{\phi(0)} \qquad \dfrac{[\psi(n)]^1 \qquad \dfrac{\mathcal{E}'}{\psi(n) \supset \phi(n)}}{\phi(n)} \qquad \dfrac{\mathcal{D}_S}{\phi(n) \supset \phi(n+1)}}{\phi(n+1)} \qquad \dfrac{\mathcal{E}''}{\phi(n+1) \supset \psi(n+1)}}{\psi(n+1)}}{\psi(0)} \qquad \dfrac{\dfrac{\psi(n+1)}{\psi(n) \supset \psi(n+1)}\,{}^1}{\forall n \,.\, \psi(n) \supset \psi(n+1)}}{\forall n \,.\, \psi(n)}$$

where $\mathcal{E}'$ and $\mathcal{E}''$ follow from the lemma $\forall n \,.\, \phi(n) \equiv \psi(n)$.

We will use an analogous transformation for induction over finite sets. At first glance, Lemma Insertion may seem counterintuitive. We are perturbing a given proof of a specification ϕ by first proving ψ and then recovering a proof of ϕ. This step is analogous to strengthening an induction hypothesis, even though the induction was already successful. One has to remember that the objects we are focusing on are the *proofs*, and not the fact that a given formula is true (which is usually trivial). The example and its transformation in the following sections should help to clarify Lemma Insertion and its utility.

2.4 Transforming the Initial Proof

It is clear that we need some notion of array in our language in order to derive Warshall's algorithm. Actually, one may say that we will never be able to derive Warshall's algorithm, since it uses destructive operations on an array, and destructive operations can not be

derived in our development paradigm. This is only partly true. We can bring the program into a form where advanced compilation techniques as described in [9] may be used to recognize that destructive operations may be used effectively. This seems to be possible in many cases and is a subject of ongoing research. One possibility we are exploring is to extract a program and then leave the proofs-as-programs paradigm in favor of correctness preserving program transformation techniques as described in [10] to derive a destructive implementation.

Definition 12 *An* array *is a function with finite, enumerated domain. We write* $[A \to B]$ *for the type arrays with index set* A *and element type* B*. Given* A *and* B*, there is an everywhere unspecified array* $[]{:}[A \to B]$ *and for* $f{:}[A \to B]$*, we write* $f \oplus [c \mapsto b]$ *for the (new) array which maps* c *to* b*, i.e.* $(f \oplus [c \mapsto b])[c] = b$*, and otherwise behaves like* f*.*

In this context we do not need to open Pandora's box by considering partial functions. Rather, we treat $[]{:}[A \to B]$ as a total function in the sense that $[][a]$ will yield a value for an $a \in A$. However, we cannot prove anything about $[][a]$ except that it will be in B.

The following lemma indirectly relates arrays to functions over finite domains. This is similar to a theorem stating the existence of choice functions. A choice function, however, is typically built by abstraction — here it must be built by (re)defining array values with "$\oplus$".

Lemma 13 *Given a finite, enumerated set* A*.*

$$\forall P\,.\,\forall x{:}A\;\exists y{:}B\;Pxy \iff \exists f{:}[A \to B]\;\forall x{:}A\;Px(f[x])$$

Proof 14 *Assume* $\forall x{:}A\;\exists y{:}B\;Pxy$*. We generalize the conclusion to* $\forall I \subseteq A\;\exists f{:}[A \to B]\;\forall x{:}A\;.\;x \in I \Longrightarrow P\;x\;(f\;[x])$*. This can be proved by induction over* I*. If* $I = \{\}$ *we let* f *be the everywhere unspecified array* $[]{:}[A \to B]$*. If* $I = z \uplus I'$ *we use the assumption for* $x = z$ *to obtain a* y *satisfying* Pxy*. Let* f' *be the array which exists according to induction hypothesis. Then let* f *be* $f' \oplus [z \mapsto y]$*. The other implication also follows easily.*

□

In our example the situation is slightly different, since we are proving a statement of the form $\forall x\;.\;Px \vee \neg Px$. Thus we also need the following analogue of Theorem 13 with a similar induction proof.

Lemma 15 *Given a finite, enumerated set* A*.* $\forall x{:}A\;\;(Px \vee \neg Px) \iff \exists f{:}[A \to \mathcal{P}]\;\forall x{:}A\;\;(f[x] \iff Px)$*.*

Proof 16 *Similar to Proof 14.*

□

Now we use Lemma Insertion twice, first with Lemma 15 and then with Lemma 13. What do these Lemma Insertions "mean"? Lemma 15 expresses that if we can decide a property P for every element of a finite set A, we can generate an array indexed by A such that the property P can simply be looked up. This may not seem much of an improvement, since computing a single value is replaced by the computation of an array, but in case there are many recursive calls the cost of computing the array is more than offset by the savings of looking up a previously computed value, rather than recomputing it. After the first transformation using Lemma 15 the main induction will prove the following lemma:

Lemma 17 *Given (V, E). Then*

$$\forall W \subseteq V\ \forall x\ \exists f_W{:}V \to [V \to \mathcal{P}]\ \forall y\ .\ f(x)[y] \iff \exists p\ .\ p \in \mathsf{path}_W(x,y)$$

Proof 18 *of Lemma 17. The proof is by induction on W.*

Basis $W = \{\}$. *Let x be given. We prove by induction over Y that $\forall Y \subseteq W\ \exists f_{\{\}}{:}V \to [V \to \mathcal{P}]\ \forall y \in Y\ .\ f_{\{\}}(x)[y] \iff \exists p\ .\ p \in \mathsf{path}_W(x,y)$.*

Basis $Y = \{\}$. *Then let $f_{\{\}}(x) = []$.*

Step $Y = y \uplus Y'$. *By inductive hypothesis there is an f' such that $f'_{\{\}}(x)[y] \iff \exists p\ .\ p \in \mathsf{path}_{\{\}}(x,y)$ for any $y \in Y'$. If $\langle x, y\rangle \in E$ then let $f(x) = f'(x) \oplus [y \mapsto \text{true}]$, otherwise map y to* "false".

Step $W = z \uplus W'$. *Let x be given. The proof again proceeds by induction over Y.*

Basis $Y = \{\}$. *Then let $f_W(x) = []$.*

Step $Y = y \uplus Y'$. *By the innermost inductive hypothesis, there is an f'_W such that $f'_W(x)\,[y] \iff \exists p\ .\ p \in \mathsf{path}_W(x,y)$ for any $y \in Y'$. We now extend f'_W to f_W, where f_W will be correct on y, too.*

1. $f_{W'}(x)[y]$. *Then* $f_W(x)[y]$.
2. $\neg f_{W'}(x)[y]$. *Again we use cases.*
   (a) $f_{W'}(x)[z]$ *and* $f_{W'}(z)[y]$. *Then* $f_W(x)[y]$.
   (b) $\neg f_{W'}(x)[z]$ *or* $\neg f_{W'}(z)[y]$. *Then* $\neg f_W(x)[y]$.

□

Now we can insert Lemma 13 and perform reductions to obtain a proof of the following Lemma.

Lemma 19 *Given (V, E). Then*

$$\forall W \subseteq V\ \exists f_W{:}[V \to [V \to \mathcal{P}]]\ \forall x, y \in V\ .\ f[x][y] \iff \exists p\ .\ p \in \mathsf{path}_W(x,y)$$

We show a selective view of the resulting deduction. It should be noted, however, that the best way of describing this deduction, is not through its final proof, but through the initial deduction and its transformation. The proof will be opaque, just like the corresponding program will be harder to understand than the initial program.

Proof 20 *of Lemma 19. The proof is by induction on W.*

Basis $W = \{\}$. *By nested inductions over V achieving that $f_{\{\}}$ satisfies $f_{\{\}}[x][y] \iff \langle x, y\rangle \in E$.*

Step $W' = z \uplus W$. *By nested inductions over V. Omitting the interior inductions and focusing on the cases inside them we have:*

1. $f_{W'}[x][y]$. *Then* $f_W[x][y]$.
2. $\neg f_{W'}[x][y]$. *Again we use cases.*

(a) $f_{W'}[x][z]$ *and* $f_{W'}[z][y]$. *Then* $f_W[x][y]$.
(b) $\neg f_{W'}[x][z]$ *or* $\neg f_{W'}[z][y]$. *Then* $\neg f_W[x][y]$.

□

In the first program, recursive calls were used to determine directly whether two points are connected by a path whose interior nodes are contained in a strictly smaller set than the one currently under consideration. In this new program, the recursive calls are made to build up a two-dimensional array that remembers this information for any pair of nodes. When the program needs to know if two points are connected at an earlier stage, this information is looked up in the array (in the last line of the program). This by itself is not an efficiency improvement, but now the (still functional) program may be implemented destructively.

Program 21 *The associated program:*

```
ispath ⇐ λV λE λx λy . (rp V E V V V [] [])[x][y]
rp ⇐
   λV λE λW λX λY λf λg .
      if W = {}
         then if X = {}
                 then f
                 else let x ⊎ X* = X in
                         if Y = {}
                            then rp V E W X* V (f ⊕ [x ↦ g]) []
                            else let y ⊎ Y* = y in
                               rp V E W X Y* f (g ⊕ [y ↦ (⟨x, y⟩ ∈ E)])
         else let z ⊎ W* = W in
                 let f* = (rp V E W* V V [] []) in
                    if X = {}
                       then f
                       else let x ⊎ X* = X in
                               if Y = {}
                                  then rp V E W X* V f ⊕ [x ↦ g] []
                                  else let y ⊎ Y* = Y in
                                          rp V E W X Y* f
                       (g ⊕ [y ↦ (f*[x][y] or (f*[x][z] and f*[z][y]))])
```

Even though the array operation $\oplus$ is non-destructive, it is clear that we need "only" $|V|$ arrays, one for each value taken by W during the recursion.

Techniques described in [9] can eliminate this source of inefficiency in the recursive program and generate a destructive version of the function above which needs only one array. Except for the representational details, this destructive version is Warshall's algorithm as described in [2].

3 Warshall's Algorithm Using Integer Representation

In this section we outline how abstract data types and their implementations can be incorporated into the proofs-as-program paradigm. We then illustrate the concepts by outlining how to eliminate finite sets from the formulation of Warshall's algorithm in favor integers, thus further improving the efficiency of the implementation.

The version of Warshall's algorithm we have developed so far uses sets and assumptions about the existence of certain operations on sets (like checking whether a set is empty). In a machine-checked derivation of the style presented above, these assumptions could occur in different guises. It could be that the sets and operations on them are *defined* in terms of other concepts, say lists, or predicates. The other possibility is that sets and operations are axiomatized and the existence of *realizing constants* for the axioms is assumed. This view was proposed by Goad [6]. In the latter case, programs extracted from proofs will contain these realizing constants, thus extending the language of programs (usually some form of the λ-calculus). These programs can only be executed, if the interpreter or compiler for the programming language in question knows how to treat these constants.

Here we propose a slightly different method for dealing with these assumptions. In the setting of verified programming, we view a data type as corresponding to a *theory*, that is, a language extension and a set of sentences axiomatizing the theory. For our purposes, we will assume that this is done through a single sentence. Axiom schemata are outside of this framework, but are usually unnecessary if we use a higher-order logic in which one can quantify over predicates.

An *abstract data type* is simply a sentence derived from the theory that existentially quantifies over the primitive symbols of the theory. If we are in a typed logic, and the theory introduce new types, we abstract and existentially quantify over those types as well. We call this the *defining sentence* for the abstract data type. This generalizes the usual notion of algebraic data type found in some programming languages, since it allows arbitrary axioms, and not simply equations or conditional equations.

An *implementation* of an abstract data type is given simply by a proof of its defining sentence. Alternative implementations are given through alternative proofs.

We now illustrate abstract data types and their use in program development in our example. Let us assume that we have a sentence ϕ defining an abstract data type of finite sets that gives the constants $\{\}, \uplus, \in, \doteq$ their usual meaning, that is, $\uplus$ adjoins an element to a set and $\doteq$ is equality between finite sets. Here V is the (finite) base set, and S stands for the type of finite subsets of V.

Theorem 22 (Abstract Data Type Warshall).

$$\phi = \exists V\ \exists S\ .\ \exists\{\}{:}S\ \exists\uplus{:}V \times S \to S\ \exists\in{:}V \times S \to \mathcal{P}\ \exists\doteq{:}S \times S \to \mathcal{P}\ .\ \phi'(V, S, \{\}, \cup, \in, \doteq)$$

Here ϕ' is intended to contain for example I_V, the induction principle for finite sets (see Definition 4).

Given the abstract data type ϕ, the original specification is of the form

$$\begin{array}{c} \forall V\ \forall S\ .\ \forall\{\}\ \forall\uplus\ \forall\in\ \forall\doteq\ .\ \phi'(V, S, \{\}, \cup, \in, \doteq) \\ \supset \forall E\ \forall x, y{:}V\ .\ (\exists p\ .\ p \in \mathsf{path}^{(V,E)}(x, y) \vee \neg\exists p\ .\ p \in \mathsf{path}^{(V,E)}(x, y)) \end{array}$$

Our derivation of Warshall's algorithm did not make use of all properties of finite sets, and therefore using an implementation of finite sets (for example through lists) would be unnecessarily inefficient. In other presentations of Warshall's algorithm (such as [2]), nodes in V are represented as integers, and subsets of V are also represented as single integers. This is possible, since the only subsets of V we need to consider are initial segments of the form $\{0, 1, \ldots k-1\}$.

We can therefore create an intermediate *ad hoc* abstract data type $\exists\vec{x}\ .\ \psi(\vec{x})$ that contains only those properties of finite sets that were actually used during the derivation ($\vec{x}$ abbreviates the variables bound in the definition of ϕ) . Clearly, ψ is not uniquely determined, and it is not clear how it could be generated *automatically*, but the information in the proof of the specification of Warshall's algorithm is a very helpful guide.

Through the use of *proof expansions* (the inverse of proof reductions), we can transform the given proof of the graph reachability theorem into one that has subproofs of $\forall\vec{x}\ .\ (\phi'(\vec{x}) \supset \psi'(\vec{x}))$ and $\forall\vec{x}\ .\ (\psi'(\vec{x}) \supset \mathrm{War}(\vec{x}))$, where War is the original specification.

Because of the second implication, an alternative proof of $\exists\vec{x}\ .\ \psi'(\vec{x})$, yields an alternative implementation of Warshall's algorithm.

Proof 23 *(Integer Implementation of* Warshall*). We represent V and S each by a single natural number. $\{\}$ is represented by 0, $\in$ by $<$, $\uplus$ by* max *and $\doteq$ by equality between natural numbers.*

□

We can thus obtain a proof of $\exists\vec{x}\ .\ \mathrm{War}(\vec{x})$ that contains a subproof of the following specification:

Theorem 24

$$\forall n \forall E{:}\{1,\ldots,n\} \times \{1,\ldots,n\} \to \mathcal{P}\ .\ \forall x,y \leq n\ .\\ (\exists p\ .\ p \in \mathsf{path}^{(n,E)}(x,y) \vee \neg\exists p\ .\ p \in \mathsf{path}^{(n,E)}(x,y))$$

Together with the remarks at the end of the previous section, it should be clear that the program extracted from this proof can be compiled into very efficient machine code.

Bibliography

[1] Joseph Bates and Robert Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[2] Manfred Broy and Peter Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, SE-7(1):44–67, 1977.

[3] Robert L. Constable, Todd Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3):285–326, 1984.

[4] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL85*, Springer-Verlag LNCS 203, 1985.

[6] Christopher A. Goad. *Computational Uses of the Manipulation of Formal Proofs.* Technical Report Stan-CS-80-819, Stanford University, August 1980.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, IEEE, June 1987.

[8] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Computer Science Department, Cornell University, 1987.

[9] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 300–314, ACM, January 1985.

[10] Ulrik Jørring and William L. Scherlis. Deriving and using destructive data types. In *IFIP TC2 Working Conference on Program Specification and Transformation*, North-Holland, 1986.

[11] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):92–121, 1980.

[12] Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[13] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175, North-Holland, 1980.

[14] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105, IEEE, June 1987.

[15] Christine Mohring. Algorithm development in the calculus of constructions. In *Symposium on Logic in Computer Science*, pages 84–91, IEEE Computer Society Press, Washington, D. C., June 1986.

[16] B. Möller. *A survey of the project CIP: Computer-aided, intuition-guided programming.* Technical Report TUM–18406, Institut für Informatik der TU München, Munich, West Germany, 1984.

[17] B. Nordström. Programming in constructive set theory. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ACM, Portsmouth, 1981.

[18] Kent Petersson and Jan Smith. *Program Derivation in Type Theory: The Polish Flag Problem.* Technical Report, University of Göteborg, Chalmers, Göteborg, Sweden, 1985.

DEPARTMENT OF COMPUTER SCIENCE
CARNEGIE MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213-3890