Representing proof transformations for program optimization

Penny Anderson

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Representing Proof Transformations for Program Optimization

Penny Anderson

N° 2229

Mai 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

Representing Proof Transformations for Program Optimization

Penny Anderson

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet CROAP

Rapport de recherche n° 2229 — Mai 1994 — 21 pages

Abstract: In the *proofs as programs* methodology a program is derived from a formal constructive proof. Because of the close relationship between proof and program structure, transformations can be applied to proofs rather than to programs in order to improve performance. We describe a method for implementing transformations of formal proofs and show that it is applicable to the optimization of extracted programs. The method is based on the representation of derived logical rules in Elf, a logic programming language that gives an operational interpretation to the Edinburgh Logical Framework. It results in declarative implementations with a general correctness property that is verified automatically by the Elf type checking algorithm. We illustrate the technique by applying it to the problem of transforming a recursive function definition to obtain a tail-recursive form.

Key-words: Program development, Specification, Proof transformation, Logics of programs, Program verification, Recursion schema, Program extraction, Realisability, Typed lambda calculus, Logical framework, Higher-order logic programming language, Elf, Edinburgh Logical Framework, Natural deduction

(Résumé : tsvp)

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Représentation des Transformations de Preuve pour L'optimisation des Programmes

Résumé : La construction d'une preuve dans une logique constructive peut être vue comme le développement d'un programme certifié. La correspondance entre la structure d'une preuve et la structure d'un programme permet, en transformant une preuve, d'obtenir un programme optimisé. Nous décrivons une méthode d'implémentation de transformations de preuves. Cette méthode permet d'optimiser les programmes extraits. Elle est basée sur la représentation des règles logiques dérivées dans le langage Elf. Elf est un langage de programmation logique qui donne une interprétation exécutable au *Logical Framework* d'Edinburgh. Nous obtenons une implémentation déclarative avec une propriété de correction qui est vérifiée automatiquement par l'algorithme de typage d'Elf. Comme exemple d'application de notre technique nous donnons la transformation d'une fonction récursive en fonction récursive terminale.

Mots-clé : Développement de programmes, Spécification, Transformation de preuve, Logique de programmes, Vérification de programmes, Schéma de récursion, Extraction de programmes, Réalisabilité, Lambda-calcul typé, Langage de programmation logique d'ordre supérieur, Elf, Edinburgh Logical Framework, Métalogique, Déduction naturelle

1 Introduction

Research in constructive logics and type theories has resulted in a good understanding of the relation of formal constructive proof to computation, and has produced a number of systems (e.g., Nuprl [6], Coq [7]) capable of supporting the development of programs by constructive theorem-proving (the *proofs as programs* strategy). However, proof development presents some of the same difficulties that program development does: notably problems of adaptation and re-use. This observation has stimulated work on extending the strategy with proof transformations; the idea is to attempt to optimize the program by transforming the proof, rather than extracting the program from the proof and transforming it afterwards. Research in this area suggests that this approach can address problems in program development that are not easily solved either by syntactic program transformation techniques or through theorem-proving support alone. Such problems include the specialization of a program to its input [10], [17] and the optimization of the call structure of a recursive function definition [26], [17], [18].

One standard approach to metaprogramming tasks for formal logic like proof transformation has been to use a separate programming language, such as ML, as a metalanguage for a type theory considered as an object logic. A more recently developed strategy, which has been applied to programming language semantics [13], [12], [19], theorem provers [8], [9], and the metatheory of deductive systems [29], is the use of a higher-order logic programming language as a logical framework. This is the basis of the approach taken here. The Elf programming language [27] supports this strategy by providing an operational interpretation to the Edinburgh Logical Framework (LF) [14]. LF is a dependently-typed λ-calculus designed to serve as a framework for the encoding of formal deductive systems. Its type system is expressive enough to support straightforward encodings of many logics used in reasoning about programming languages, formal logic, and the like. The decidability of the LF type system gives practical force to the *judgments as types* and the corresponding *proof-checking as type-checking* principles. Since a deduction represented in LF is valid exactly when its representation is well-typed, the Elf implementation of LF type-checking provides a proof checker for encoded deductive systems.

In this paper we describe a methodology for implementing proof transformations in Elf that yields a declarative formulation with easily verified correctness properties. It is closely related to the deduction transformations of [29]. The method exploits the higher-order unification capabilities of Elf to express a transformation by means

of higher-order patterns in the spirit of Huet and Lang [16]. Although the technique is restricted to transformations based on derived rules of the encoded logic, it has been applied successfully [1] to program optimizations such as the introduction of tail recursive form, finite differencing [23], and modification of the domain of a recursive function definition. Experiments to date have been limited to a simple system of natural deduction for first-order logic, but the approach is applicable to any logic that can be encoded in LF. In combination with support for theorem proving and tools for analysis and heuristics like those of [17] it could provide an easily verifiable and extensible basis for interactive or automated systems for the development and maintenance of verified programs.

The paper is organized as follows: Sect. 2 gives some necessary background on an Elf encoding of natural deduction proofs and implementation of program extraction. In Sect. 3 we describe the basics of the transformation encoding methodology by means of a small example. Sect. 4 shows how the method can be used to implement a transformation to obtain a tail-recursive function definition. We apply the transformation to an example program and demonstrate that transforming the proof can result in more optimization than is obtained by syntactic program transformation. We conclude with a summary and assessment in Sect. 5.

2 Representation of Proofs and Program Extraction

We implemented basic support for the proofs as programs strategy in Elf. Program extraction, program execution, proof transformations, and other forms of term manipulation were implemented as Elf programs. The use of LF and Elf permits concise encodings that directly reflect the inference systems they represent. This is often sufficient for proof search – for instance, it yields implementations of a programming language interpreter and type assignment system, and of program extraction. Although the representation of natural deduction does not give a theorem-prover, the equivalence of proof checking and Elf type checking is the basis for the verification of our encodings of proof transformations.

Elf gives an operational semantics to LF type declarations: an Elf program is a collection of type declarations and an Elf query is a type. The query may contain free variables, which are treated like Prolog logic variables; it succeeds if the system can construct an inhabitant of the query type from the declarations that constitute the program.

We sketch an encoding in the style of Harper et al. [14] of natural-deduction style proofs for intuitionistic many-sorted first-order predicate calculus. Individual terms, propositions and proofs are represented as object-level Elf terms.

Terms are either natural numbers or lists of natural numbers:

```
i : type.
zero : i.
succ : i -> i.
ilist : type.
nl : ilist.                %empty list
cns : i -> ilist -> ilist. %cons
```

There is no declaration corresponding to variables; they are represented as Elf bound variables through the use of higher-order abstract syntax.

Logical constants and connectives are encoded as follows (we omit the declarations that permit the use of the binary connectives as infix operators):

```
o : type.                      %formulas
true : o.                      %truth
false : o.                     %absurdity
~ : o -> o.                    %negation
& : o -> o -> o.               %conjunction
\/ : o -> o -> o.              %disjunction
=> : o -> o -> o.              %implication
forall : (i -> o) -> o.        %universal quantification
exists : (i -> o) -> o.        %existential quantification
lforall : (ilist -> o) -> o.   %list quantifiers
lexists : (ilist -> o) -> o.
```

Elf permits declared constants to begin with a non-letter; thus `=>` is the constant representing implication. LF non-dependent function types are notated using `->`. Higher-order abstract syntax is used to represent the binding properties of the quantifiers by meta-level abstraction.

The encoding of proofs (Fig. 1) uses the dependent types of LF to obtain a proof checker. The dependent type constructor `|- : o -> type` provides an association between a proof and its end-formula, which permits the encoding of inference rules

```
|- : o -> type.

truei : |- true.
falsee : {C:o} |- false -> |- C.
andi : |- A -> |- B -> |- (A & B).
andel : |- (A & B) -> |- A.
ander : |- (A & B) -> |- B.
oril : {B:o} |- A -> |- (A \/ B).
orir : {A:o} |- B -> |- (A \/ B).
ore : (|- A -> |- C) -> (|- B -> |- C) -> |- (A \/ B) -> |- C.
impliesi : (|- A -> |- B) -> |- (A => B).
impliese : |- (A => B) -> |- A -> |- B.
noti : (|- A -> |- false) -> |- (~ A).
note : |- (~ A) ->  |- A  -> |- false.
foralli : ({x:i} |- (A x)) -> |- (forall A).
foralle : {T:i} |- (forall A) -> |- (A T).
existsi : {A:i -> o} {T:i} |- (A T) -> |- (exists A).
existse : ({x:i} |- (A x) -> |- C) -> |- (exists A) -> |- C.
lforalli : ({x:ilist} |- (A x)) -> |- (lforall A).
lforalle : {L:ilist} |- (lforall A) -> |- (A L).
lexistsi : {A:ilist -> o} {L:ilist} |- (A L) -> |- (lexists A).
lexistse : ({x:ilist} |- (A x) -> |- C) -> |- (lexists A) -> |- C.
lind :    {A:ilist -> o}
          |- (A nl)
          -> ({l:ilist} |- (A l) -> {x:i} |- (A (cns x l)))
          -> |- (lforall A).
```

Figure 1: Elf encoding of inference rules

to enforce the constraint that any well-typed proof term is a valid proof. The Elf syntax `{x:A} B` represents the LF dependent type construction $\Pi x : A \,.\, B$. In the figure many Π-quantifiers are omitted, since Elf takes free variables in clauses to be implicitly Π-quantified, and types for them can often be inferred automatically[25]. Higher-order abstract syntax supports the representation of the introduction and discharge of assumptions, as well as the side conditions restricting the free occurrences of variables in assumptions.

We defined a simply typed λ-calculus with primitive recursion for representing extracted programs. A typing discipline and operational semantics for the language were given in the form of deductive systems encoded as Elf programs. The programming language syntax, evaluation and type inference were implemented in the style of Michaylov and Pfenning [19], which extends the higher-order representations developed in λProlog by Hannan and Miller in [13], [12]. We omit details of the language description and give example programs in the syntax of Standard ML [22].

Our implementations of type and program extraction are an adaptation to the Elf setting of *modified realizability* [24], [15], [30], [31], [32], in which extracted types and programs are simplified to remove uninformative subterms while retaining computationally useful information. The simplification is performed during extraction, based on the fact that program values of interest are generated only by proofs of disjunctions and existentially quantified formulas. In particular, a proof of $\neg A$ is always uninformative; we exploit this fact in the example of Sect. 4.

Type and program extraction can be expressed as deductive systems inductively defined over formulas and proof objects respectively; these are straightforward to transcribe into executable Elf programs. The interested reader is referred to [2] for details. For an understanding of the rest of this paper, it is sufficient to keep in mind that the use of induction corresponds to primitive recursion in the extracted program, and that a proof of $\exists x \,.\, A(x)$ with x a list and $A(x)$ uninformative corresponds to a list expression in the program.

3 Proof Transformations as Derived Rules

What do we mean by a transformation based on a derived rule? A small example is the following transformation to swap the members of a pair. For any proof $\mathcal{P}$ of

$A \wedge B$ there is a proof of the following form:

$$\dfrac{\dfrac{\dfrac{\mathcal{P}}{A \wedge B}}{B}\wedge\mathrm{E}_R \qquad \dfrac{\dfrac{\mathcal{P}}{A \wedge B}}{A}\wedge\mathrm{E}_L}{B \wedge A}\wedge\mathrm{I}$$

Thus we may conclude $B \wedge A$ from $A \wedge B$ by a derived rule of the object logic. A derived rule involving metavariables for formulas (as A and B above) can be directly encoded in Elf by putting Π-quantified variables for the metavariables, as in the following encoding of the derivation.

```
%%% Rule statement
flipand : |- A & B -> |- B & A -> type.
%%% Rule derivation
fliptrans : {A:o} {B:o} {P: |- A & B}
            flipand P (andi (ander P) (andel P)).
```

This also implements a proof transformation: given to the Elf interpreter, it can be used to transform any proof of a formula $A \wedge B$ to one of $B \wedge A$. Here is an example query, giving as input a proof of $0 = 0 \wedge 1 = 1$:

```
?- flipand (andi (eq_refl zero) (eq_refl (succ zero)))
           Output_proof.
```

In response to the query Elf succeeds with the substitution:

```
Output_proof =
 andi (ander (andi (eq_refl zero)
                   (eq_refl (succ zero))))
      (andel (andi (eq_refl zero)
                   (eq_refl (succ zero)))).
```

The transformation acts indirectly as a program transformation: from a proof of $A \wedge B$ the system extracts a program expression e with a pair type $\alpha \times \beta$; from the

transformed proof of $B \ \wedge \ A$ it extracts the expression $\langle\ \mathbf{snd}(e), \mathbf{fst}(e)\ \rangle$ with the type $\beta \times \alpha$.

The type-correctness of the clause `fliptrans` provides strong guarantees of some properties of the transformation. If the transformation succeeds then `Output_proof` is bound to a term $\mathcal{P}'$ that encodes a valid proof of $B \ \wedge \ A$. This property is distinct from the guarantee given by program extraction, which ensures that the program satisfies its specification regardless of how the transformation is encoded. Properties specific to this transformation can also be easily proved. From properties of extraction proved in [1] it follows that if the program expression extracted from $\mathcal{P}$ has type $\alpha \times \beta$, where α is the type extracted from A and β the type extracted from B, then the program expression extracted from $\mathcal{P}'$ has type $\beta \times \alpha$. It is not hard to show that in fact the values are exchanged. It is also easy to see that the transformation is total.

4 A Proof Transformation for the Introduction of Tail-recursion

A well-known programming strategy for the improvement of a recursive function definition is the introduction of an accumulator argument to achieve a tail-recursive form, which can be compiled to a loop. This strategy has long been studied by researchers in transformational programming, for example in [5], [4], and many others. Here we describe a transformation that performs the analogous optimization at the level of proofs; its application to a small example program shows that in some cases the proof transformation can perform more optimization than techniques restricted to the program level.

A Simple Proof and Program.

We present the problem in terms of a program to select all elements of a list that are at least 2. Following is a specification for the program, with a simple proof.

Specification 1

$$\forall l\,.\,\exists r\,.\,[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge \ y \geq 2)]$$

Proof 2 The proof is by induction on the list l. If l is empty, choose r to be empty as well. Otherwise $l = x :: l'$ and, by the induction hypothesis, there is an r' such

that $\forall y \,.\, y \in r' \Leftrightarrow (y \in l' \ \wedge \ y \geq 2)$. Then if $x \geq 2$ let r be $x :: r'$; otherwise let r be r'. $\Box$

The following program is extracted from a formalized version of the proof:

Program 3

```
fun select [] = []
  | select (x::l) =
     let val r = (select l) in if x >= 2 then x::r else r end
```

This can easily be transformed at the program level to tail-recursive form using the fold-unfold system [5].

Program 4

```
fun sel [] k = (reverse k)
  | sel (x::l) k = if x >= 2 then (sel l (x::k)) else (sel l k)
fun select l = sel l []
```

The transformed definition is tail-recursive, but contains a call to `reverse` which could be eliminated since the specification says nothing about the order of the elements in the list to be computed. But since this change does not preserve functional equivalence, it is not straightforward to accomplish via purely syntactic program transformation. One could avoid the difficulty by observing that the input is treated here as a set, not a list, and working in a more appropriate algebra. Depending on the context in which the function is used, this may or may not be easy to do. The use of a proof transformation yields a similar tail-recursive program that computes with lists but does not use the reverse function.

A Proof Structure that Guarantees Tail-recursive Form.

The view of proofs as programs rests on the correspondence between proof structure and program structure. Recursion in the program corresponds to induction in the proof. In a constructive proof of $\forall l \,.\, \exists r \,.\, \Phi(l, r)$ by list induction, the inductive case constructs a witness: some term t such that for arbitrary x, $\Phi(x :: l, t)$. The witness is usually some function of a parameter r' obtained by eliminations from the induction

hypothesis $\exists r' \,.\, \Phi(l, r')$. It corresponds to the value computed in the body of the extracted recursive function definition.

A program is tail-recursive when it does no computation with the result of its recursive call. Correspondingly, an inductive proof is realized by a tail-recursive program when the witness t is identical to the parameter r' of the induction hypothesis. That is, if the proof appeals to the induction hypothesis to obtain a parameter r' satisfying $\Phi(l, r')$, and shows that *the same value* r' also satisfies $\Phi(x :: l, r')$ for any x, the extracted function is tail-recursive.

To obtain a proof of this form, it is necessary to modify the induction hypothesis of the original proof. This corresponds to the well-known technique of generalizing a loop invariant [11]. Generalization to a particular form is the basis for our proof transformation.

A Derived Rule for Tail-recursion Introduction.

Given an initial specification of the form $\forall l \,.\, \exists r \,.\, \Phi(l, r)$, a useful generalization is a formula

$$\forall l \,.\, \forall k \,.\, \exists s \,.\, \exists r \,.\, \Phi(l, r) \;\wedge\; \Psi(k, r, s),$$

where k corresponds to the accumulator argument in the extracted program. The transformation produces an inductive subproof of this formula, which corresponds to the auxiliary function definition `sel` of the fold-unfold derivation. The existentially quantified variable s corresponds to the result computed by the auxiliary tail-recursive function. However, the extracted program will compute a pair of lists corresponding to $\langle\, s, r \,\rangle$. To eliminate the unwanted computation of r, we insert a double negation in the new formula, obtaining

$$\forall l \,.\, \forall k \,.\, \exists s \,.\, \neg\neg \exists r \,.\, \Phi(l, r) \;\wedge\; \Psi(k, r, s).$$

This technique exploits the use of modified realizability: negations are uninformative and do not induce computation in the extracted program.

The derived rule that represents the transformation expresses what additional proof obligations must be met for the construction of the output proof. The base and inductive cases of the input supply the first two premises of the rule. Premises (3), (4), and (5) represent proof obligations. Their proofs and the formula Ψ

must be supplied to the transformation in addition to the input proof.

$$\frac{\begin{array}{l}(1)\ \ \Phi([], \mathbf{r}_0)\\ (2)\ \ \forall x, l, r\,.\,\Phi(l, r) \supset \Phi(x :: l, \mathbf{f}(x, r))\\ (3)\ \ \forall k\,.\,\Psi(k, \mathbf{r}_0, \mathbf{s}_0(k))\\ (4)\ \ \forall x, k, r, s\,.\,\Psi(\mathbf{h}(x, k), r, s) \supset \Psi(k, \mathbf{f}(x, r), s)\\ (5)\ \ \forall l, r, s\,.\,\Phi(l, r) \wedge \Psi(\mathbf{k}_0, r, s) \supset \Phi(l, s)\end{array}}{\forall l\,.\,\exists s\,.\,\neg\neg\Phi(l, s)}\ \text{TR}$$

The premises of the rule are presented on separate lines and numbered for ease of reference. Bold-face identifiers, e.g. $\mathbf{r}_0$, are parameters of the inference rule. When applied, e.g. $\mathbf{f}(x, r)$, they represent expressions in which the terms to which they are applied may occur free.

The structure of the output proof is specified by the derivation of the rule. For simplicity's sake we ignore the double negation in the conclusion, which is trivial to derive, and informally give the derivation of an intermediate output proof $\mathcal{P}'$ of $\forall l\,.\,\exists s\,.\,\Phi(l, s)$.

Proof 5 We first show $\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l, r) \wedge \Psi(k, r, s)$ by induction on the list l. If $l = []$ then for arbitrary k by premises (1) and (3) we can choose $\mathbf{r}_0$ for r and $\mathbf{s}_0(k)$ for s. Otherwise $l = x :: l'$. Let k' be an arbitrary list. We instantiate the induction hypothesis with $\mathbf{h}(x, k')$ for k to obtain s', r' such that $\Phi(l', r') \wedge \Psi(\mathbf{h}(x, k'), r', s')$. By premise (2) it follows that $\Phi(x :: l', \mathbf{f}(x, r'))$, and by premise (4) we have $\Psi(k', \mathbf{f}(x, r'), s')$. Then we can choose $\mathbf{f}(x, r')$ for r and s' for s. Since k' is arbitrary we have $\forall k\,.\,\exists s\,.\,\exists r\,.\,\Phi(l, r) \wedge \Psi(k, r, s)$.

Then for any list l we have s, r such that $\Phi(l, r) \wedge \Psi(\mathbf{k}_0, r, s)$, and from premise (5) $\Phi(l, s)$. □

Unlike the example of Sect. 3 the rule TR does not have the conclusion $\forall l\,.\,\exists r\,.\,\Phi(l, r)$ of the input proof as a premise. Thus it cannot be used directly to transform Proof 2; first the input proof must be analyzed to obtain premises (1) and (2) of the rule. The next section shows how higher-order unification can be used to do the analysis.

Representation in Elf.

We describe a representation that produces the intermediate proof $\mathcal{P}'$; [1] gives details of how the double negation is obtained. The proof transformation is encoded

```
tail_rec :
 {R0} {S0} {K0} {F} {H}
 {Phi: ilist -> ilist -> o} {Psi: ilist -> ilist -> ilist -> o}
 %% Input proof:
 |- (lforall ([l] lexists ([r] (Phi l r))))
 -> ({k:ilist} |- (Psi k R0 (S0 k)))                    %% Premise 3
 -> ({x:i} {k:ilist} {r:ilist} {s:ilist}                %% Premise 4
      |- (Psi (H x k) r s) -> |- (Psi k (F x r) s))
 -> ({l} {r} {s}                                        %% Premise 5
     |- ((Phi l r) & (Psi K0 r s)) -> |- (Phi l s))
 %% Output proof:
 -> (|- (lforall ([l] (lexists ([r] Phi l r)))))
 -> type.
```

Figure 2: Transformation judgment

```
tl_rec_clause :
 {R0} {S0} {K0} {F} {H} {Phi} {Psi}
 {Prem1} {Prem2} {Prem3} {Prem4} {Prem5}
 {Output_proof}
 tail_rec R0 S0 K0 F H Phi Psi
 (lind ([l] lexists ([r] Phi l r))
       (lexistsi ([r] Phi nl r) R0 Prem1)            %% Premise 1
       ([l] [p] [x]
        (lexistse ([r] [q]
         lexistsi _ (F x r) (Prem2 x l r q))         %% Premise 2
         p)))
 Prem3 Prem4 Prem5 Output_proof
 <- tr_drule R0 S0 K0 F H Phi Psi Prem1 Prem2 Prem3 Prem4 Prem5
             Output_proof.
```

Figure 3: Transformation clause

```
tr_drule :
 {R0} {S0} {K0} {F} {H}
 {Phi: ilist -> ilist -> o} {Psi: ilist -> ilist -> ilist -> o}
 %% Input base case (premise 1):
 |- Phi nl R0
 %% Input inductive case (premise 2):
 -> ({x:i} {l} {r} |- Phi l r -> |- Phi (cns x l) (F x r))
 %% Premise 3
 -> ({k:ilist} |- Psi k R0 (S0 k))
 %% Premise 4
 -> ({x:i} {k:ilist} {r:ilist} {s:ilist}
      |- Psi (H x k) r s -> |- Psi k (F x r) s)
 %% Premise 5
 -> ({l} {r} {s} |- (Phi l r) & (Psi K0 r s) -> |- Phi l s)
 %% Output proof:
 -> (|- lforall [l] (lexists [s] (Phi l s)))
 -> type.
```

Figure 4: The derived rule TR

in two stages: the first analyzes the input to obtain the premises (1) and (2) of the derived rule; the second encodes the rule itself.

To express the transformation in terms of the derived rule we define a judgment (Fig. 2) relating the input proof, the proof obligations, and the output proof. The judgment is implemented by a single clause (Fig. 3) that specifies how to obtain premises (1) and (2) of the derived rule TR by unification with the input. Higher-order unification is essential here; it enables us to pick out premise (2) as a (metalevel) function from terms and proofs to proofs. The instantiations for `Prem1` and `Prem2` are passed to the derived rule, which is invoked by the subgoal `<- tr_drule R0 ...`. It is in solving this subgoal that the Elf interpreter constructs the output proof.

The encoding (Fig. 4) of the rule TR follows the same methodology as the example of Sect. 3. Meta-level Π-quantification represents the metavariables for formulas and individual expressions. Comparison of the encodings of the premises with the statement of the rule TR shows the use of a technique called lifting: object-level universal quantifiers and implications have been replaced by meta-level Π-quantification. The

```
tr_rule_derive :
 tr_drule R0 S0 K0 F H Phi Psi Prem1 Prem2 Prem3 Prem4 Prem5
 (lforalli [l]
  %% Recovery of original specification:
  lexistse ([s] [p']
   lexistse ([r] [p]
    (lexistsi _ s (Prem5 l r s p)))
    p')
   (lforalle K0 (lforalle l
     %% New inductive sub-proof:
     (lind ([l] lforall [k] lexists [s] lexists [r]
             (Phi l r) & (Psi k r s))
       %% Base case:
       (lforalli [k]
         lexistsi _ (S0 k) (lexistsi _ R0 (andi Prem1 (Prem3 k))))
       %% Step case:
       ([l] [p] [x] lforalli [k]
         (lexistse  ([s] [q'] (lexistse ([r] [q]
            lexistsi _ s
             (lexistsi _ (F x r)
               (andi (Prem2 x l r (andel q)) (Prem4 x k r s (ander q)))))
            q'))
          (lforalle (H x k) p)))))).
```

Figure 5: Derivation of TR

lifted representation is faithful to the object logic; this is a consequence of the fact that constructive minimal many-sorted predicate logic is representable in LF via the propositions-as-types interpretation, as described in, e.g., [3]. (For details of the correspondence between the lifted representation and a representation using object-level quantification the interested reader is referred to [1].) When the lemma proofs end in introductions (the most likely case) lifting has the benefit of avoiding the generation of detours in the output proof, which correspond to unwanted redices in the extracted program.

As in the example of Sect. 3, the clause (Fig. 5) implementing this judgment is a straightforward formalization of the rule derivation (Proof 5) and specifies the structure of the output proof.

Some care must be taken when coding transformations this way to avoid encountering unification problems that are not solved by the Elf implementation. If an encoding restricts its use of logic variables for pattern matching to *higher-order patterns* in the sense of [28], only deterministic unification problems will arise during execution. These problems fall within a decidable subcase of higher-order unification discovered by Miller [20] for the simply typed lambda calculus and extended to LF by Pfenning. Unification problems that cannot be solved by the Elf implementation can still be solved by an Elf program for unification, using a technique adapted from Miller [21]. This complicates the encoding but presents no essential difficulty.

With this clause and formalizations Psi, P_3, P_4, P_5 of Ψ and the proof obligations, we can transform a formalization P of Proof 2 by submitting the following query to Elf:

`?- tail_rec R0 S0 F H K0 Phi` Psi P P_3 P_4 P_5 `Q.`

Here the italic variables (e.g., P) stand for closed LF object terms provided by the user. The text in typewriter font (e.g., `R0`) is input as-is by the user; thus the variables `R0...Phi` and `Q` are Elf logic variables. If the query succeeds the Elf interpreter displays the terms bound to them in the course of the search. The term bound to `Q` is a representation of the transformed proof.

The resulting proof can be further transformed by the same methods (which impose no further proof obligations) to obtain a proof of $\forall l\,.\,\exists s\,.\,\neg\neg\Phi(l,s)$ in which unwanted computation is suppressed.

The following is an informal statement of the new proof of Specification 1 obtained by applying this transformation to the `select` example:

Proof 6 We first show

$$\forall l\,.\,\forall k\,.\,\exists s\,.\,\exists r\,.\,[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge\ y \geq 2)] \ \wedge\ [\forall y\,.\,y \in s \Leftrightarrow (y \in r \ \vee\ y \in k)]$$

by induction on l. If $l = []$ then choose $s = k$ and $r = []$. Otherwise $l = x :: l'$. If $x \geq 2$, then by instantiating the induction hypothesis with l' and $x :: k$, we obtain a list r' containing all elements ≥ 2 of l' and a list s' containing all the elements of r' and $x :: k$. Then choose $r = x :: r'$ and $s = s'$. Otherwise $\neg x \geq 2$; again by the induction hypothesis there is a list r' as before, and a list s' containing all the elements of r' and k. Let $r = r'$ and $s = s'$.

Now let k be the empty list, so that s contains exactly the elements of r. Then $\forall y\,.\,y \in s \Leftrightarrow (y \in l \ \wedge\ y \geq 2)$ and $\forall l\,.\,\exists r\,.\,[\forall y\,.\,y \in r \Leftrightarrow (y \in l \ \wedge\ y \geq 2)]$ as required. □

Applied to the example the implementation yields the following extracted program:

Program 7

```
fun sel l =
 let fun sel' nil = (fn k => k)
       | sel' (x::l') = let val p = (sel' l') in
          fn k => (p (if x >= 2 then (x::k) else k)) end
 in (sel' l nil) end;
```

This program is tail-recursive, and is an improvement over Program 4, as it does not use the `reverse` function. The difference is a consequence of working with the specification and its proof throughout the development rather than restricting the reasoning to the properties of the program alone. It is an example of the effect noted by Goad [10]: using proof transformation a program can be optimized by exploiting the fact that there is no need to preserve functional equivalence of the successive stages of program development.

Correctness Properties.

The encoding provides two sorts of correctness guarantees for the transformation. Because type-checking is proof-checking, the type-correctness of the transformation ensures that the transformed proof is a valid proof of a particular known end-formula. This is a general property that holds for any proof transformation encoded this way. The technique also supports reasoning about properties specific to this transformation. For example, in [1] we give an Elf program to recognize tail-recursive object programs. Using the techniques of [29] we give a partial representation in Elf of a proof that the tail-recursion transformation yields a tail-recursive program if it succeeds.

5 Conclusion

We have a demonstrated a methodology for encoding proof transformations that is applicable to program optimization problems. By limiting the scope of the technique to transformations based on derived rules of the encoded logic, we obtain encodings with correctness properties that are easy to establish. Although this is a strong

limitation it appears that many techniques from the field of program transformation may be amenable to this kind of encoding. We should point out that the choice of Elf as a basis for the encoding does not limit the implementable transformations to those based on derived rules. Elf programs can be written to implement more complex transformations, for example, proof normalization. Such transformations share the general correctness property that the resulting proof is valid and its end-formula is known, but other properties are typically more difficult to establish.

This style of representation is largely independent of the choice of theorem-proving techniques and heuristic issues. Thus it could perhaps be usefully combined with some of the work of Madden [17] [18] on the use of automatic theorem proving techniques and analysis tools tailored for fully automated program development by proof transformation.

Acknowledgement

I would like to thank Frank Pfenning for providing the Elf system and for supervising my research. Thanks also to the referees for their helpful comments.

References

[1] Penny Anderson. *Program Derivation by Proof Transformation.* PhD thesis, Department of Computer Science, Carnegie Mellon University, October 1993. Available as Technical Report CMU-CS-93-206.

[2] Penny Anderson. Program extraction in a logical framework setting. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, July 1994. To appear.

[3] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[4] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

[5] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

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

[7] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Technique 134, INRIA, Rocquencourt, France, December 1991. Version 5.6.

[8] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989. Available as Technical Report MS-CIS-89-53.

[9] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11:43–81, 1993.

[10] Christopher A. Goad. Computational uses of the manipulation of formal proofs. Technical Report Stan-CS-80-819, Stanford University, August 1980.

[11] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[12] John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, January 1991. Available as MS-CIS-91-09.

[13] John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, pages 453–476. MIT Press, 1989.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[15] Susumu Hayashi. An introduction to PX. In Gerard Huet, editor, *Logical Foundations of Functional Programming.* Addison-Wesley, 1990.

[16] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[17] Peter Madden. *Automated Program Transformation Through Proof Transformation.* PhD thesis, University of Edinburgh, 1991.

[18] Peter Madden. Automatic program optimization through proof transformation. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 446–460, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[19] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its metatheory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[20] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 253–281. Springer-Verlag LNCS 475, 1991.

[21] Dale Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 255–269. MIT Press, July 1991.

[22] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[23] Robert Paige and Shaye Koening. Finite differencing of computable expressions. Technical Report LCSR-TR-8, Laboratory for Computer Science Research, Rutgers University, August 1980.

[24] Christine Paulin-Mohring. Extracting F_ω programs from proofs in the calculus of constructions. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, January 1989.

[25] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 199? To appear. Preliminary version available as Technical Report CMU–CS–92–105, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1992.

[26] Frank Pfenning. Program development through proof transformation. *Contemporary Mathematics*, 106:251–262, 1990.

[27] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[28] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[29] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[30] James T. Sasaki. *Extracting Efficient Code from Constructive Proofs*. PhD thesis, Cornell University, May 1986. Available as Technical Report TR 86–757.

[31] Helmut Schwichtenberg. On Martin-Löf's theory of types. In *Atti Degli Incontri di Logica Mathematica*, pages 299–325. Dipartimento di Matematica, Università di Siena, 1982.

[32] Helmut Schwichtenberg. A normal form for natural deductions in a type theory with realizing terms. In Ettore Casari et al., editors, *Atti del Congresso Logica e Filosfia della Scienza, oggi. San Gimignano, December 7–11, 1983*, pages 95–138, Bologna, Italy, 1985. CLUEB.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399