Program extraction in a logical framework setting

Penny Anderson

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Program Extraction in a Logical Framework Setting

Penny Anderson

N° 2261

Mai 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

Program Extraction in a Logical Framework Setting

Penny Anderson

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

Rapport de recherche n° 2261 — Mai 1994 — 20 pages

Abstract: This paper demonstrates a method of extracting programs from formal deductions represented in the Edinburgh Logical Framework, using the Elf programming language. Deductive systems are given for the extraction of simple types from formulas of first-order arithmetic and of λ-calculus terms from natural deduction proofs. These systems are easily encoded in Elf, yielding an implementation of extraction that corresponds to modified realizability. Because extraction is itself implemented as a set of formal deductive systems, some of its correctness properties can be partially represented and mechanically checked in the Elf language.

Key-words: Program extraction, Realisability, Typed lambda calculus, Logical framework, Higher-order logic programming language, Elf, Edinburgh Logical Framework, Natural deduction, Natural semantics

(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

Extraction de programmes représentée dans un langage de programmation logique d'ordre supérieur

Résumé : Nous utilisons le langage de programmation Elf pour extraire des programmes à partir des preuves formelles représentées dans le *Logical Framework* d'Edinburgh. Des types simples sont extraits des formules de l'arithmétique du premier ordre, et des termes du λ-calcul sont extraits des preuves en déduction naturelle, en utilisant des systèmes de déduction que nous décrivons. Ces systèmes sont codés dans le langage Elf, ce qui donne une implémentation qui correspond à la *réalisabilité modifiée*. Grâce à cette implémentation quelques propriétés de correction de l'extraction peuvent être codées en Elf, et être partiellement vérifiées automatiquement.

Mots-clé : Extraction de programmes, Réalisabilité, Lambda-calcul typé, Langage de programmation logique d'ordre supérieur, Elf, Edinburgh Logical Framework, Métalogique, Déduction naturelle, Sémantique naturelle

1 Introduction

Research in the development of verified programs through theorem proving has traditionally relied on systems based on a fixed logic or type theory (e.g., [3], [5], [12]). Given the current interest in variations on logics and programming languages for proofs-as-programs (e.g. [14], [15], [21]), a more flexible approach may be useful. As argued in [2], the use of a type theory as a logical framework contributes to the implementation of metaprograms for logic in a flexible, declarative and verifiable way. This is the approach to program extraction examined in this paper. Combined with existing work on the implementation of theorem proving [6] and the syntax and semantics of programming languages [13], this approach lays the groundwork for complete support for programming by theorem proving in a logical framework setting. The declarative style of encodings in a logical framework affords flexibility in the choice of logic, programming language, and notions of extraction, as well as ease of programming and some mechanical support for verification of metaprograms.

The Edinburgh Logical Framework (LF) [10] is a dependent type theory designed to support the specification of a variety of formal deductive systems. An LF type represents a judgment, and objects of that type represent deductions; given a proper encoding, the well-typedness of such an object is equivalent to the validity of the deduction it represents. The decidability of LF's type system provides a proof checker for deductions represented in this way. LF has been given an operational interpretation by the Elf language [18], which supports a logic programming style for the specification of proof search. This paper investigates the use of Elf to implement the extraction of programs from proofs and to partially verify this implementation.

The approach we take is to view each element of the problem, from the specification of syntax to program extraction, as a deductive system that can be easily transcribed into an Elf signature. We represent a logical language, its proof system, and the syntax of a typed λ-calculus by static Elf signatures, i.e., collections of clauses that define a syntax but are not used for search by the Elf interpreter. The semantics of the programming language, and type and program extraction are represented as dynamic Elf signatures, i.e., programs used for search, yielding an executable type checker, interpreter, and extractor. Because extraction is implemented as a set of formal deductive systems, some of its correctness properties can be partially represented and mechanically checked in the Elf language. This approach allows a single framework to support the study of a variety of logics, programming languages, and notions of extraction, within the limitations of the kinds of deductive systems that can be faithfully represented in LF.

The paper is organized as follows. In Sect. 2 we describe the Elf implementation of extraction. This requires as a preliminary the definition of a logic and a typed λ-calculus. In Sect. 3 we discuss some correctness properties of the extraction and how their proofs can be partially represented in Elf. Sect. 4 provides a summary and discussion.

2 Program extraction as a deductive system

Logic.

We sketch an encoding in the style of Harper et al. [10] of natural-deduction style proofs for intuitionistic first-order arithmetic. For reasons of space we give details here only for implication and existential quantification. A full treatment is given in [1]. Here is a fragment of the abstract syntax for the logic with its encoding in Elf:

$$\begin{array}{llcl} \text{Individuals} & t & ::= & x \mid 0 \mid \mathrm{S}n \\ \text{Formulas} & F & ::= & \top \mid \bot \mid t_1 = t_2 \mid F_1 \supset F_2 \mid \exists x\,.\,F \end{array}$$

```
i : type.                    true : o.
o : type.                    false : o.
zero : i.                    eq :  i -> i -> o.
succ : i -> i.               => : o -> o -> o.
                             exists : (i -> o) -> o.
```

We will consider extraction from the following set of natural deduction rules, together with axioms for arithmetic:

$$\frac{}{\top}\,\top\mathrm{I} \qquad \frac{\bot}{C}\,\bot\mathrm{E} \qquad \frac{[t/x]A}{\exists x\,.\,A}\,\exists\mathrm{I}$$

$$\frac{\begin{array}{c}\overline{A}^{\,p}\\ \vdots\\ B\end{array}}{A \supset B}\,\supset\mathrm{I}^{p} \qquad \frac{A \supset B \quad A}{B}\,\supset\mathrm{E} \qquad \frac{\exists x\,.\,A \qquad \begin{array}{c}\overline{A}^{\,p}\\ \vdots\\ C\end{array}}{C}\,\exists\mathrm{E}^{p}\ *$$

We represent the discharge of an assumption A by placing an annotated bar over it. The use of letters rather than numbers for annotations is useful for specifying

extraction: we regard an annotation p as a *proof variable*, and write $p : \vdash A$ to express the association of p with A.

The rules are encoded in Elf as:

```
|- : o -> type.
truei : |- true.
falsee : {C:o} |- false -> |- C.
impliesi : (|- A -> |- B) -> |- (A => B).
impliese : |- (A => B) -> |- A -> |- B.
existsi : {A:i -> o} {T:i} |- (A T) -> |- (exists A).
existse : ({x:i} |- (A x) -> |- C) -> |- (exists A) -> |- C.
```

The use of LF's dependent types ensures that proof checking is LF type checking. The Elf syntax `{x:A} B` represents the LF dependent type construction $\Pi x : A \,.\, B$. The code omits many Π-quantifiers since Elf takes free variables in clauses to be implicitly Π-quantified, and types for them can often be inferred automatically[17]. Higher-order abstract syntax supports the representation of the introduction and discharge of assumptions, as well as the side conditions (indicated by the asterisk attached to the $\exists$-elimination rule) restricting the free occurrences of variables.

Programming language.

We define a simply typed λ-calculus for representing extracted programs. Syntax, evaluation and type inference have been implemented [1] in the style of Michaylov and Pfenning [13], which extends the higher-order representations developed in λProlog by Hannan and Miller in [8], [7]. In this paper we consider only syntax and type inference, as background for the metatheory of the next section.

We use a Nuprl-like syntax [3] for programs in order to emphasize the close relation to the logic. Here is the portion of the language definition necessary for extraction from the fragment of logic we consider:

e	::=	$x \mid \mathtt{0} \mid \mathbf{s}(e) \mid$	*Natural numbers*
		$\langle\, e_1, e_2 \,\rangle \mid \mathbf{spread}(e_1;\, x, y \,.\, e_2) \mid$	*Pairs*
		$\mathbf{lam}\, x \,.\, e \mid \mathbf{app}(e_1, e_2) \mid$	*Functions*
		$() \mid \mathbf{error}(e)$	*Unit, error*

This is straightforward to translate into Elf:

```
term : type.
```

```
0 : term.
s : term -> term.
pair : term -> term -> term.
spread : term -> (term -> term -> term) -> term.
lam : (term -> term) -> term.
app : term -> term -> term.
unity : term.
error : term -> term.
```

A natural operational semantics for the language is given in [1]. Most constructs are self-explanatory. The destructor for pairs may require some explanation: in $\mathbf{spread}(e_1;\ x, y\,.\,e_2)$, x and y are bound in e_2 to the components of a pair obtained by evaluating e_1. The construct $\mathbf{error}(e)$ signals an error and is extracted from proof by the intuitionistic absurdity rule $\bot$E.

Typing rules for this language can be formulated and implemented in Elf using the techniques of [13]. Types have the following syntax:

$$\tau \quad ::= \quad \mathsf{nat} \ \mid\ \tau_1 \times \tau_2 \ \mid\ \tau_1 \Rightarrow \tau_2 \ \mid\ \mathsf{unit}$$

with Elf encoding:

```
tp : type.                    ==> : tp -> tp -> tp.
nat : tp.                     unit : tp.
* : tp -> tp -> tp.
```

The typing judgment is of the form $\Gamma \vdash e \in \tau$, where Γ is a context of typing assumptions, and $\Gamma, x : \tau$ is the result of extending a given context Γ with the assumption that the variable x has type τ. We show only the fragment dealing with λ-abstraction and application:

$$\frac{\Gamma, x : \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \mathbf{lam}\, x\,.\,e \in \tau_1 \Rightarrow \tau_2}\ \mathsf{tp\text{-}lam} \qquad \frac{\Gamma \vdash e_1 \in \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash \mathbf{app}(e_1, e_2) \in \tau_2}\ \mathsf{tp\text{-}app}$$

Here is an Elf signature that implements the system:

```
of : term -> tp -> type.

tp_0 : of 0 nat.
tp_s : of (s M) nat <- of M nat.
```

```
tp_pair : of (pair M N) (A * B) <- of M A <- of N B.
tp_spread : of (spread Mpr N) C
            <- of Mpr (A * B)
            <- {x} of x A -> {y} of y B -> of (N x y) C.
tp_lam : of (lam M) (A ==> B)
         <- {x:term} of x A -> of (M x) B.
tp_app : of (app M N) B <- of M (A ==> B) <- of N A.
tp_unity : of unity unit.
tp_any : {A} of (error M) A <- of M B.
```

defined over higher-order syntax representations, the typing context is represented by Elf meta-level assumptions introduced during the search process (e.g., in the rule `tp_lam`).

Extraction.

The encodings of proofs and functional programs as LF objects facilitate the treatment of type and program extraction as deductive systems. In [1] we give naive formulations of these systems, which extract types that mimic closely the structure of propositions and programs that mimic closely the structure of proofs. This naive extraction results in a program containing many subterms that carry no computationally useful information. Intuitively speaking, this is because the only formulas whose proofs involve choice are existential quantifications (and disjunction in the full logic). We call formulas free of positive occurrences of $\exists$ *uninformative*. By extension we call their proofs, the object types extracted from them, and the programs extracted from the proofs uninformative as well. We define extraction procedures for types and programs that simplify the extracted terms to remove uninformative subterms, while retaining computationally useful information. The extracted programs are well-typed and the extracted types can be inferred for them.

This notion of extraction is an adaptation to the Elf setting of the basic ideas of *modified realizability* developed for the Calculus of Constructions by Paulin-Mohring [16] which in turn takes from the PX system [11] the idea of syntactically defining a class of content-free terms. Sasaki [22] develops these ideas for Nuprl. The negative formulas of Schwichtenberg [23], [24] are used in a similar way to decrease the complexity of realizing terms.

The uninformative formulas are the *Rasiowa-Harrop*[25] formulas:

$$U \quad ::= \quad \top \quad | \quad t_1 = t_2 \quad | \quad \bot \quad | \quad A \supset U$$

where A is any formula. From proofs of these formulas we extract the unit element ().

Extraction is defined by the following judgments:

1. Uninf A (A is an uninformative formula)
2. Inf A (A is an informative formula)
3. $A \Downarrow^t \tau$ (the type τ is extracted from the formula A)
4. $t \Downarrow^i e$ (the program e is extracted from the individual term t)
5. $\mathcal{P} \Downarrow e$ (the program e is extracted from the proof $\mathcal{P}$)

The first two are simple syntactic properties of formulas: (1) is a straightforward deductive formulation of the grammar of uninformative formulas given above, and (2) is the complement of (1). The main judgments are the extraction of types and programs, guided by the syntactic analysis given by the auxiliary judgments.

The following rules define type extraction:

$$\frac{\text{Uninf } A}{A \Downarrow^t \mathsf{unit}}\,\mathsf{xst\text{-}uninf} \qquad \frac{\text{Inf } A \quad A \Downarrow^t \tau_1 \quad \text{Inf } B \quad B \Downarrow^t \tau_2}{A \supset B \Downarrow^t \tau_1 \Rightarrow \tau_2}\,\mathsf{xst}_{\supset}$$

$$\frac{\text{Uninf } A \quad \text{Inf } B \quad B \Downarrow^t \tau}{A \supset B \Downarrow^t \tau}\,\mathsf{xst}_{\supset \mathrm{R}} \qquad \frac{\text{Inf } A \qquad A \Downarrow^t \tau}{\exists x\,.\,A \Downarrow^t \mathsf{nat} \times \tau}\,\mathsf{xst}_{\exists} \qquad \frac{\text{Uninf } A}{\exists x\,.\,A \Downarrow^t \mathsf{nat}}\,\mathsf{xst}_{\exists \mathrm{L}}$$

It is straightforward to transcribe them into Elf:

```
extract_tp : o -> tp -> type.

exts_u: extract_tp A unit <- uninf A.
exts_imp : extract_tp (A => B) (T1 ==> T2)
           <- inf A <- extract_tp A T1
           <- inf B <- extract_tp B T2.
exts_impr : extract_tp (A => B) T
            <- uninf A
            <- inf B <- extract_tp B T.
exts_ex : extract_tp (exists A) (nat * T)
```

```
          <- ({x:i} inf (A x))
          <- ({x:i} extract_tp (A x) T).
exts_exl : extract_tp (exists A) nat <- ({x:i} uninf (A x)).
```

Extraction from individual terms is trivial, mapping the natural numbers of the logic to the numerals of the programming language:

```
extract_tm : i -> term -> type.

ex_zero : extract_tm zero 0.
ex_succ : extract_tm (succ X) (s M) <- extract_tm X M.
```

Program extraction requires the use of a context of extraction assumptions. For an informative proof $\mathcal{P}$ the deduction of $\mathcal{P} \Downarrow e$ imitates the structure of $\mathcal{P}$ itself. In particular, wherever $\mathcal{P}$ introduces an assumption A with annotation p, the extraction deduction introduces an assumption $p \Downarrow p'$ for some fresh program variable p'. Similarly, wherever $\mathcal{P}$ introduces a parameter x, the extraction deduction introduces an assumption $x \Downarrow^i x'$ for a fresh program variable x'. Thus the extraction judgment is defined relative to a context of extraction assumptions Γ. All variables in a given context may be assumed to be distinct. We write $\Gamma, x \Downarrow^i x'$ for the result of extending Γ with the individual term extraction assumption $x \Downarrow^i x'$, and $\Gamma, p \Downarrow p'$ for the result of extending Γ with the proof extraction assumption $p \Downarrow p'$. The full form of the extraction judgment is thus $\Gamma \vdash \mathcal{P} \Downarrow e$.

The definition of program extraction is straightforward; the only complications that arise are due to the elimination rules of the logic. Extraction from proofs of uninformative formulas is trivial: we simply extract the unit term $()$ without analysing the structure of the proof. From a proof by the intuitionistic absurdity rule $\bot\mathrm{E}$ we extract a value $\mathbf{error}(e)$ where e is extracted from the proof of the premise; this is of course the unit value $()$ since $\bot$ is uninformative. We show the fragment of the deductive system that treats implication and existential quantification:

$$\dfrac{\mathrm{Inf}\, A \quad \mathrm{Inf}\, B \quad \Gamma, p \Downarrow p' \vdash \mathcal{P} \Downarrow e}{\Gamma \vdash \left[\begin{array}{c} \overline{}\, p \\ A \\ \mathcal{P} \\ B \\ \hline A \supset B \end{array}\right] \supset\mathrm{I}^p \Downarrow \mathbf{lam}\, p' \,.\, e}\ \mathsf{xs}{\supset}\mathrm{I} \qquad \dfrac{\mathrm{Uninf}\, A \quad \mathrm{Inf}\, B \quad \Gamma \vdash \mathcal{P} \Downarrow e}{\Gamma \vdash \left[\begin{array}{c} \overline{}\, p \\ A \\ \mathcal{P} \\ B \\ \hline A \supset B \end{array}\right] \supset\mathrm{I}^p \Downarrow e}\ \mathsf{xs}{\supset}\mathrm{IR}$$

$$\frac{\mathrm{Inf}\,A \quad \mathrm{Inf}\,B \quad \Gamma \vdash \mathcal{P}_1 \Downarrow e_1 \quad \Gamma \vdash \mathcal{P}_2 \Downarrow e_2}{\Gamma \vdash \left[\dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ A \supset B & A\end{array}}{B}\,{\supset}\mathrm{E}\right] \Downarrow \mathbf{app}(e_1, e_2)}\,\mathsf{xs}{\supset}\mathrm{E} \qquad \frac{\mathrm{Uninf}\,A \quad \mathrm{Inf}\,B \quad \Gamma \vdash \mathcal{P}_1 \Downarrow e}{\Gamma \vdash \left[\dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ A \supset B & A\end{array}}{B}\,{\supset}\mathrm{E}\right] \Downarrow e}\,\mathsf{xs}{\supset}\mathrm{ER}$$

$$\frac{\Gamma \vdash t \Downarrow^i e_1 \quad \mathrm{Inf}\,A \quad \Gamma \vdash \mathcal{P} \Downarrow e_2}{\Gamma \vdash \left[\dfrac{\begin{array}{c}\mathcal{P}\\ [t/x]A\end{array}}{\exists x\,.\,A}\,\exists\mathrm{I}\right] \Downarrow \langle\, e_1, e_2 \,\rangle}\,\mathsf{xs}\exists\mathrm{I} \qquad \frac{\mathrm{Uninf}\,A \quad \Gamma \vdash t \Downarrow^i e}{\Gamma \vdash \left[\dfrac{\begin{array}{c}\mathcal{P}\\ [t/x]A\end{array}}{\exists x\,.\,A}\,\exists\mathrm{I}\right] \Downarrow e}\,\mathsf{xs}\exists\mathrm{IL}$$

$$\frac{\mathrm{Inf}\,A \qquad \Gamma \vdash \mathcal{P} \Downarrow e_1 \qquad \Gamma, x \Downarrow^i x', p \Downarrow p' \vdash \mathcal{Q} \Downarrow e_2}{\Gamma \vdash \left[\dfrac{\begin{array}{cc} & \overline{\begin{array}{c}p\\ A\end{array}}\\ \mathcal{P} & \mathcal{Q}\\ \exists x\,.\,A & C\end{array}}{C}\,\exists\mathrm{E}^p\right] \Downarrow \mathbf{spread}(e_1;\ x', p'\,.\,e_2)}\,\mathsf{xs}\exists\mathrm{E}$$

$$\frac{\mathrm{Uninf}\,A \qquad \Gamma \vdash \mathcal{P} \Downarrow e_1 \qquad \Gamma, x \Downarrow^i x' \vdash \mathcal{Q} \Downarrow e_2}{\Gamma \vdash \left[\dfrac{\begin{array}{cc} & \overline{\begin{array}{c}p\\ A\end{array}}\\ \mathcal{P} & \mathcal{Q}\\ \exists x\,.\,A & C\end{array}}{C}\,\exists\mathrm{E}^p\right] \Downarrow \mathbf{app}((\mathbf{lam}\,x'\,.\,e_2), e_1)}\,\mathsf{xs}\exists\mathrm{EL}$$

The rule $\mathsf{xs}\exists\mathrm{EL}$ has the interesting feature that extraction is performed on subproofs containing proof variables for which no extraction assumption is introduced in the context. The deduction can succeed only if $\Gamma \vdash \mathcal{Q} \Downarrow e$ can be deduced with the proof variable p occurring free in $\mathcal{Q}$. At first glance it might seem that extraction could

fail without an assignment for p. But for uninformative formulas modified extraction does not need to examine the proof.

Again it is straightforward to translate this system into Elf; we show a fragment of the implementation:

```
uninf : o -> type.
inf : o -> type.

exs_un : extract (P: |- A) unity <- uninf A.
exs_falsee : extract (falsee C P) (error M)
             <- extract P M.

exs_impli2 : extract (impliesi (P: |- A -> |- B)) (lam M)
             <- inf A <- inf B
             <- {p: |- A} {p':term}
                 (extract p p' -> extract (P p) (M p')).

exs_impli1 : extract (impliesi (P: |- A -> |- B)) M
             <- uninf A <- inf B
             <- {p: |- A} extract (P p) M.

exs_imple2 : extract (impliese (P1: |- (A => B)) P2) (app M1 M2)
             <- inf A <- inf B
             <- extract P1 M1 <- extract P2 M2.

exs_imple1 : extract (impliese (P1: |- (A => B)) P2) M
             <- uninf A <- inf B <- extract P1 M.

exs_existsi2 : extract (existsi A T P) (pair M N)
               <- ({x:i} inf (A x))
               <- extract_tm T M <- extract P N.

exs_existsi1 : extract (existsi A T _) M
               <- ({x:i} uninf (A x)) <- extract_tm T M.

exs_existse2 : extract (existse Q P) (spread N M)
               <- ({x:i} inf (A x))
               <- ({x:i} {x':term} extract_tm x x'
                   -> {p:|- (A X)} {p':term} extract p p'
                   -> extract (Q x p) (M x' p'))
               <- extract P N.
```

```
exs_existse1 : extract (existse Q (P: |- (exists A))) (app (lam M) N)
               <- ({X:i} uninf (A X))
               <- ({X:i} {x:term} extract_tm X x
                   -> {p: |- (A X)} extract (Q X p) (M x))
               <- extract P N.
```

There is one feature of this program that has not appeared before: some dependently-typed terms are annotated with their types. For example in the rule `exs_existse1` the major premise of the existential elimination must be $\exists x\,.\,A$ where A is uninformative. We annotate `P` with its type in order to express this premise.

3 Correctness properties of extraction

Some correctness properties of extraction can be partially verified in Elf by formulating deduction transformations along the lines described in [19], [9]. The verification technique is essentially that of [4], but the dependent types of LF eliminate the need for explicit reasoning about the validity of the objects involved, and the term and type reconstruction of Elf mechanize the management of many details.

We consider in some detail the property of *type soundness*:

Proposition 1 (Type soundness of extraction) *For any* $\mathcal{P} : \vdash A$, e, *and* τ, *if* $\vdash \mathcal{P} \Downarrow e$ *and* $\vdash A \Downarrow^t \tau$ *then* $\vdash e \in \tau$.

This property can be proved by induction over the structure of extraction deductions. The proof constructs formal deductions of typing judgments from given deductions of extraction judgments. The dual nature of Elf signatures – which can be viewed as either logic programs or language definitions – supports the direct expression of the constructive parts of the proofs. Each case of the induction is expressed as an Elf clause that matches a deduction of a particular shape.

When discussing the proof, we will need to name the formal deductions of extraction and typing. To say that some judgment J is derivable, we often write simply "J"; we write $\mathcal{D} :: J$ when $\mathcal{D}$ is a deduction of the judgment J.

Since extraction deductions in general depend on a context of extraction assumptions, we generalize Proposition 1 accordingly. We define a function from contexts of extraction assumptions to typing contexts:

Definition 2 *Given a context of extraction assumptions* $\Gamma = \langle\ \Gamma_1, \ldots, \Gamma_n\ \rangle$ *with domain* $x_1 \ldots, x_n$, *define*

1. $\lceil \Gamma_i \rceil = x_i' \in \mathsf{nat}$, *if* Γ_i *is* $x_i \Downarrow^i x_i'$
2. $\lceil \Gamma_i \rceil = p_i' \in \tau_i$, *if* Γ_i *is* $p_i \Downarrow p_i'$, $p_i :\vdash A_i$, *and* $A_i \Downarrow^t \tau_i$.
3. $\lceil \Gamma \rceil = \langle\ \lceil \Gamma_1 \rceil, \ldots, \lceil \Gamma_n \rceil\ \rangle$

Now we can generalize Proposition 1 to the following:

Lemma 3 (Type soundness of extraction in arbitrary contexts)
Given a proof $\mathcal{P}$ *of a formula* A, *if* $\Gamma \vdash \mathcal{P} \Downarrow e$ *and* $\vdash A \Downarrow^t \tau$ *then* $\lceil \Gamma \rceil \vdash e \in \tau$.

Since the extraction judgment depends on the judgment $t \Downarrow^i e$ (extraction from individual terms) a type soundness property for individual term extraction is also needed:

Lemma 4 (Type soundness for individual extraction) *If* $\Gamma \vdash t \Downarrow^i e$ *then* $\lceil \Gamma \rceil \vdash e \in \mathsf{nat}$.

The proof is trivial since the only terms in the object logic are natural numbers. Its representation in Elf is a very simple example of our partial verification technique. Here is the Elf signature that represents the proof of Lemma 4:

```
tsetm : extract_tm T M -> of M nat -> type.

tsetm_zero : tsetm ex_zero tp_0.
tsetm_succ : tsetm (ex_succ E) (tp_s D) <- tsetm E D.
```

The clause `tsetm_zero` corresponds to the base case of the proof, which constructs the depth-one typing derivation for the program expression 0. The clause `tsetm_succ` contains a subgoal that represents an appeal to the induction hypothesis.

The encoding of the proof of Lemma 3 in Elf (Fig. 1) follows the same basic principles. The declaration of the judgment `tse` expresses a relation between a program extraction deduction, a type extraction deduction, and a typing assignment deduction. Proving Lemma 3 amounts to giving a total function from the first two deductions to the third. The encoding of the proof is partial in the following sense:

```
tse : extract (P: |- A) M -> extract_tp A T -> of M T -> type.

tse_un : tse (exs_un H) (exts_u H) tp_unity.

tse_falsee : tse (exs_falsee E) Et (tp_any _ D)
             <- tse E exts_false D.

tse_impliesi2 : tse (exs_impli2 E Ib Ia) (exts_imp Etb Ib Eta Ia) (tp_lam D)
                <- ({p} {p'} {e: extract p p'} {d} tse e Eta d
                    -> tse (E p p' e) Etb (D p' d)).
tse_impliesi1 : tse (exs_impli1 E Ib Ua) (exts_impr Etb Ib Ua) D
                <- ({p} tse (E p) Etb D).

tse_impliese2 : tse (exs_imple2 Ea Eab Ib Ia) Et (tp_app D2 D1)
                <- tse Eab (exts_imp Etb Ib Eta Ia) D1
                <- tse Ea Eta D2.
tse_impliese1 : tse (exs_imple1 Eab Ib Ua) Et D
                <- tse Eab (exts_impr Et Ib Ua) D.

tse_existsi2 : tse (exs_existsi2 E (Etm:extract_tm T M) Ia)
                   (exts_ex Et Ia) (tp_pair D2 D1)
               <- tsetm Etm D1
               <- tse E (Et T) D2.
tse_existsi1 : tse (exs_existsi1 (Etm:extract_tm T M) Ua) (exts_exl Ua) D
              <- tsetm Etm D.

tse_existse2 : tse (exs_existse2 Ee Ec I) Et (tp_spread D2 D1)
               <- tse Ee (exts_ex Et' I) D1
               <- ({x} {x'} {etm:extract_tm x x'} {p} {p'}
                   {e:extract_simp p p'} {d1} {d2}
                   tsetm etm d1 -> tse e (Et' x) d2 ->
                   tse (Ec x x' etm p p' e) Et (D2 x' d1 p' d2)).
tse_existse1 : tse (exs_existse1 Ee Ec U) Et
                    (tp_app D1 (tp_lam D2))
               <- tse Ee (exts_exl U) D1
               <- ({x} {x'} {etm:extract_tm x x'}
                   {d} tsetm etm d ->
                   {p} tse (Ec x x' etm p) Et (D2 x' d)).
```

Figure 1: Elf representation of type soundness

we can code this function as an Elf signature, and the well-typedness of each declaration in the signature guarantees that the construction carried out is correct. But there is no internal guarantee that the signature determines a total function (and clearly we cannot code the construction directly as an Elf function since it is not schematic). This guarantee could probably be obtained by applying the *schema checking* of [19]. We have partially schema-checked the proof by hand.

A simple lemma needed throughout the proof is:

Lemma 5 (Inversion for type extraction) *Given $\mathcal{E} :: A \Downarrow^t \tau$, the last inference rule of $\mathcal{E}$ is uniquely determined by the form of A.*

This is seen by inspection of the type extraction system.

The partial representation of the proof in Elf is shown in Fig. 1. We discuss the treatment of uninformative subproofs for extraction from the rules for existential quantification. If A is informative, to extract a program from a proof of $\exists x\,.\,A$ by $\exists$I we extract a pair $\langle\, e_1, e_2 \,\rangle$ representing a witness t and a proof of $[t/x]A$. But when A is uninformative, we extract just an expression e representing the witness t. In this case the extraction deduction must have the form:

$$\dfrac{\begin{array}{cc}\mathcal{U} & \mathcal{E}' \\ \text{Uninf}\, A & \Gamma \vdash t \Downarrow^i e\end{array}}{\Gamma \vdash \boxed{\dfrac{\begin{array}{c}\mathcal{P} \\ [t/x]A\end{array}}{\exists x\,.\,A}\,\exists\text{I}} \Downarrow e}\;\textsc{xs}\exists\text{IL}$$

Because A is uninformative the type extracted from $\exists x\,.\,A$ must be `nat`. Then we can apply Lemma 4 (type soundness for term extraction) to $\mathcal{E}'$ to obtain a typing derivation $\mathcal{D} :: \lceil\Gamma\rceil \vdash e \in \texttt{nat}$ as required. The clause `tse_existsi1` of Fig. 1 represents this case.

Next we consider the case of $\textsc{xs}\exists\text{EL}$, where the object proof discharges an uninformative assumption A (clause `tse_existse1` of Fig. 1). The extraction deduction must have the form:

$$\dfrac{\begin{array}{ccc}\mathcal{U} & \mathcal{E}_1 & \mathcal{E}_2 \\ \text{Uninf}\, A & \Gamma \vdash \mathcal{P} \Downarrow e_1 & \Gamma, x \Downarrow^i x' \vdash \mathcal{Q} \Downarrow e_2\end{array}}{\Gamma \vdash \boxed{\dfrac{\begin{array}{ccc} & & \overline{}^{\,p} \\ & & A \\ \mathcal{P} & & \mathcal{Q} \\ \exists x\,.\,A & & C\end{array}}{C}\,\exists\text{E}^p} \Downarrow \mathbf{app}((\mathbf{lam}\, x'\,.\,e_2), e_1)}\;\textsc{xs}\exists\text{EL}$$

Assume $\mathcal{T}_2 :: \vdash C \Downarrow^t \tau$. Since A is uninformative we may construct the type extraction derivation $\mathcal{T}_1 =$

$$\dfrac{\begin{array}{c}\mathcal{U}\\ \mathrm{Uninf}\, A\end{array}}{\exists x\,.\,A \Downarrow^t \mathsf{nat}}\,\mathsf{xst}\exists\mathrm{L}$$

By the induction hypothesis applied to $\mathcal{E}_1$ and $\mathcal{T}_1$, there is a typing deduction $\mathcal{D}_1 :: \lceil\Gamma\rceil \vdash e_1 \in \mathsf{nat}$; similarly by the induction hypothesis applied to $\mathcal{E}_2$ and $\mathcal{T}_2$, there is a deduction $\mathcal{D}_2 :: \lceil\Gamma, x \Downarrow^i x'\rceil \vdash e_2 \in \tau$. By definition $\lceil\Gamma, x \Downarrow^i x'\rceil = \lceil\Gamma\rceil, x' \in \mathsf{nat}$. Then construct the typing derivation

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_2\\ \lceil\Gamma\rceil, x' \in \mathsf{nat} \vdash e_2 \in \tau\end{array}}{\lceil\Gamma\rceil \vdash \mathbf{lam}\, x'\,.\, e_2 \in \mathsf{nat} \Rightarrow \tau}\,\mathsf{tp\text{-}lam} \qquad \begin{array}{c}\mathcal{D}_1\\ \lceil\Gamma\rceil \vdash e_1 \in \mathsf{nat}\end{array}}{\lceil\Gamma\rceil \vdash \mathbf{app}(\mathbf{lam}\, x'\,.\, e_2, e_1) \in \tau}\,\mathsf{tp\text{-}app}$$

The cases for implication are handled in essentially the same way, but without the need to treat individual parameters.

3.1 Discussion

In [1] we give a similar proof of an evaluation soundness property for a naive version of program extraction: given a proof $\mathcal{P}$ of a formula A, if $\vdash \mathcal{P} \Downarrow e$ and e evaluates to v then there is a proof $\mathcal{P}'$ of A such that $\vdash \mathcal{P}' \Downarrow v$. The proof of evaluation soundness for naive extraction is an induction on the structure of an evaluation deduction. Its partial representation in Elf constructs an object proof and its extraction deduction from the given extraction; since evaluation is closely related to proof reduction, this gives us a set of executable reductions for object proofs. Note that this is not a normalization theorem for the object logic, and because the programming language semantics does not evaluate under functional abstractions, the proof does not give even weak normal forms for object proofs in the sense of Prawitz [20].

An immediate consequence of type and evaluation soundness together is subject reduction for the set of programming language expressions that can be extracted from proofs.

The metatheory of the programming language can be encoded in the same way; see for example [19]. Type and evaluation soundness would then allow us to carry some metatheory over from the domain of programs to the domain of proofs; for

instance, we could get a very weak normal form for natural deduction proofs from a proof that evaluation of an extracted program produces a "value", for some appropriate definition of value.

4 Conclusion

We have given a deductive formulation of program extraction and shown how this style leads directly to an implementation in Elf. The approach relies on defining the underlying logic and programming language by a hierarchy of deductive systems, with deductions of syntactic well-formedness at the base of the hierarchy and extraction deductions at the top. The decidability of LF's type system leads to implementations that are guaranteed to construct valid deductions.

Extending the hierarchy to transformations defined on extraction deductions is a natural step that allows us to partially represent in Elf some of the metatheory of extraction. The mechanical checking of the type-correctness of this representation is equivalent to the automatic partial verification of the metatheory.

The implementation approach is robust with regard to changes in the object logic and programming language, affording a good basis for experimentation with variations.

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] David A. Basin and Robert L. Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993. Also available as Max-Planck-Institut für Informatik technical report MPI–I–92–205.

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

[4] Joëlle Despeyroux. Proof of translation in natural semantics. In A. R. Meyer, editor, *Symposium on Logic in Computer Science*, pages 193–205, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.

[5] 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.

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

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

[8] 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.

[9] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[10] 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.

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

[12] Lena Magnusson. The new implementation of ALF. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 265–282, Båstad, Sweden, June 1992. University of Göteborg.

[13] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory 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.

[14] Chetan Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, August 1990.

[15] Christine Paulin and Benjamin Werner. Extracting and executing programs developed in the inductive constructions system: a progress report. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, pages 377–390. Preliminary Version, May 1990.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Dag Prawitz. Ideas and results in proof theory. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307, Amsterdam, London, 1971. North-Holland Publishing Co.

[21] Christophe Raffalli. Machine deduction. To appear in the Proceedings of the Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands, May 1993.

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

[23] 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.

[24] 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.

[25] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1988.

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