Formalization and Reasoning in a Reflective Architecture

H. Rueß, H. Pfeifer, F.W. von Henke

Künstliche Intelligenz
Fakultät für Informatik
Universität Ulm
D-89069 Ulm/Donau, Germany

Abstract

This paper is concerned with developing a reflective architecture for formalizing and reasoning about entities that occur in the process of software development, such as specifications, theorems, programs, and proofs. The starting point is a syntactic extension of the type theory *ECC*. An encoding of this object calculus within itself comprises the meta-level, and reflection principles are provided for switching between different levels. These reflection principles are used to mix object- and meta-level reasoning, to generate "standard" units by executing meta-operators, and to apply formal tactics that allow for abstraction from the basic inference rules.

1 Introduction

Formalizing artifacts of software development and software engineering activities that produce these artifacts is, according to [2], a central issue of *knowledge-based software engineering*. Here we propose a reflective architecture based on a type-theoretic calculus that is capable of expressing most units of the software development process like theorems, specifications, proofs, programs, and relative implementations between specifications in order to formalize schematic developments as operators on the meta-level. These meta-operators are applied to specific problems by means of reflection principles that connect object- and meta-level. Formalizing software development steps as executable (meta-) operators supports several aspects of the by now almost universally accepted goal of *reusability*, which involves not only reuse of program fragments but also of designs and developments, and, in the context of fully formal approaches, proofs.

The work reported here is mainly related to reflexive systems for safely extending mechanical theorem provers [20, 3, 5, 9, 1, 8]. Work reported in [1] and [8] also uses a type theory as the base calculus, but their meta-level encoding consists of encoding of proof trees, while in our object calculus proofs are already first-class entities. Moreover, our main interest lies in applying reflective architectures in formalizing software development steps [19]; these may also include operators modification.

Section 2 includes a brief description of the programming and specification calculus TYPELAB, while an outline of the encoding of TYPELAB within TYPELAB together with reflection principles is considered in Sections 3 and 4. This completes the description of the reflective architecture. An in-depth discussion on reflective architectures for type theories can be found in a forth-coming thesis [17] and in [15]. Here we concentrate on describing some experiments that have been carried out with this reflective system. Sections 5, 6, and 7 provide examples for mixing object and meta-level deduction, for generating standard developments for inductive datatypes, and for applying tactics in such reflective architecture, respectively.

2 The Object Calculus

The object calculus TYPELAB basically is a syntactic extension of the type theory *ECC* [10] with inductive datatypes, and the design of these constructs is influenced mainly by the PVS specification language [13]. The resulting language is quite expressive in the sense that many entities of the software development process – programs, proofs, implementations, logical formulae, and (parameterized) specifications – can be formally expressed very directly and naturally.

Type constructors are introduced to form Cartesian products, (dependent) record types, semantic subtypes, and specifications. All these constructs are special forms of strong sum types in ECC; they are, however, handled differently by the typing system and therefore require special syntax. A semantic subtype $\{x : A \mid P\}$ comprises those members of type A which satisfy predicate P, while specifications consist, as usual, of a signature part and an axiom part. A specification *Nat_Spec* of natural numbers by means of formation, introduction, elimination, and equality rules is given, for example, in Figure 1.

Nat_Spec :=
SPEC
$nat : Type(0)$,
$zero : nat$,
$succ : nat \to nat$,
$elim : \Pi C : nat \to Type(0).$
$C(zero) \to$
$(\Pi x : nat.\ C(x) \to C(succ(x)))$
$\to \Pi n : nat.\ C(n)$
WITH
$\forall C : nat \to Type(0), f_1 : C(zero),$
$f_2 : (\Pi x : nat.\ C(x) \to C(succ(x))),$
$n : nat.$
$elim\ C\ f_1\ f_2\ zero = f_1,$
$elim\ C\ f_1\ f_2\ succ(n)$
$= f_2\ n\ (elim\ C\ f_1\ f_2\ n)$
END

Figure 1: Specification of Natural Numbers

A distinctive feature of the typing system is a mechanism for converting members of one type to members of a different type automatically; this feature is mainly used to generate so-called *type correctness conditions* [13] or *proof obligations*. A proof obligation is a placeholder for a term which will be filled in later by the prover. Discharging these proof obligations can be postponed because the type checker only requires type information.

The mechanisms to form *inductive datatypes* follow Ore's extension of *ECC* [12]. Since

all objects in TYPELAB are first-class entities, names of constructors for inductive datatypes have to be introduced explicitly. Inductive datatypes representing Booleans, natural numbers, and polymorphic lists together with appropriate operators are predefined. Expressions of TYPELAB constitute an embedded functional language with basic expressions corresponding to predefined types, *let*-statements, conditionals, higher-order structural recursion (induction), and well-founded recursion (induction). An informal but comprehensive introduction to the TYPELAB language can be found in [16].

3 The Meta-Level

A first step towards a reflective architecture consists in a representation of the object layer; here, the object calculus TYPELAB is also used as the meta-language, and the encoding itself amounts to a *definitional extension* to TYPELAB. This encoding constitutes the *meta-level*.

Syntactic categories of TYPELAB are represented as objects of an inductive datatype *trm*. Since many applications require to explicitly examine and manipulate both free and bound variables, we choose to distinguish between object and meta-level variables. This is in contrast to the *higher-order abstract syntax* approach mainly used for defining logics in *Logical Frameworks* [6]. Moreover, the representation type *trm* of TYPELAB terms closely models the object-oriented implementation of TYPELAB in that hierarchically ordered classes are encoded by means of layered constructors. Contexts are represented as lists of both declarations and definitions resulting in some representation type *cxt*.

Quoting associates entities of the object-layer with their representations in the meta-layer, and thereby allows the meta-layer to refer to, and express properties of the elements of the object-layer. Quoting $\ulcorner . \urcorner$ is external to both object- and meta-level and associates objects of TYPELAB with corresponding representations, i.e. elements in normal form of representation types *var*, *trm*, and *cxt*. Definition of the quoting mechanism is straightforward and proceeds on the structure of syntactic categories. Efficient execution of quoting is a necessity for practical application of reflective architectures. Thus, quoting of both terms and contexts is implemented efficiently in that (parts of) objects of respective representation types are computed only when accessed; the methods used are reminiscent of techniques for implementing *lazy lists*. Sometimes it is convenient to mix object-level syntax with meta-level representations. This can be accomplished using *Backquoting* '.' that is reminiscent of the *Lisp* comma operator within backquotes. For example,

$$\forall\, c : cxt, A, B : trm.\ inh(c, \ulcorner `A` \to `B` \to `B` \urcorner)$$

is a more readable notation for

$$\forall\, c : cxt, A, B : trm.\ inh(c, mk\_impl(A, mk\_impl(B, A)))$$

The backquote feature is especially useful when dealing with "large" representations. The inverse of quoting is called *unquoting* and is denoted by $\llcorner . \lrcorner$. Unquoting associates a (normal form) representation with its object entity. Note, that the result of unquoting is not necessarily well-typed.

Next, functions on representation types for substitution (*subst*), one-step reduction (*reduce*1), weak-head-normalform (*whnf*), and cumulativity

subst	:	$trm \times var \times trm \to trm$
*reduce*1	:	$cxt \times trm \to trm$
whnf	:	$cxt \times trm \to trm$
cum	:	$trm \times trm \to bool$
deriv	:	$cxt \times trm \times trm \to bool$

Figure 2: Meta-functions

(*cum*), that generalizes convertibility, are provided. Derivability of type judgments is expressed as a predicate *deriv* of type $cxt \times trm \times trm \to Prop$. Instead of coding all these functions on representation type *trm*, one simply declares corresponding constants, see Figure 2 and attaches them to the underlying *Lisp* implementation of the TYPELAB system. More precisely, whenever evaluating a function application $f(\ulcorner M \urcorner)$ one does so by using the *Lisp* representation of M and evaluating the corresponding *Lisp* function on this representation; finally the result of this computation is expressed as an object of the TYPELAB language. Following Weyhrauch [20] we refer to this mechanism of evaluating meta-functions as *semantic attachments*. Using semantic attachments has several advantages over explicit encoding of TYPELAB within TYPELAB: one avoids duplicating the *Lisp* implementation in TYPELAB and inherits from *Lisp* efficient execution of meta-operators; a small overhead results only from transforming (quote/unquote) between *Lisp* representations of objects and entities of representation types. Besides practical considerations, there are also theoretical limitations in representing evaluation function *whnf* and deciding derivability by means of type-theoretic functions. These functions, however, can be approximated by respective families of functions that are indexed with an upper bound for the number of evaluation steps [17].

In order to be able to express and prove properties on semantically attached meta-functions one provides axioms on declared constants. These axioms describe the operational nature of the term functions and closely follow the underlying *Lisp* implementation. Thus, in effect, these axioms are not only part of the meta-level description but can be regarded as an operational specification of the TYPELAB system. Moreover, using these axioms it is possible to reason about the underlying *Lisp* code within the TYPELAB system. The correctness of such reasoning depends, of course, on the *Lisp* implementation, that is assumed to "fulfill" the specified axioms. In order to substantiate this claim, structures of specifying axioms reflect the structure of the underlying *Lisp* functions. The axioms that describe operational behavior of the *whnf* function closely follows the structure of the implementation: In this way many axioms directly reflect methods of the underlying *Lisp* code. On the other hand, axioms for *deriv* are not operational but rather describe the typing rules (together with some meta-theoretic results) of the underlying calculus. Moreover, the TYPELAB system provides for advanced features such as *anonymous universes* [7] and *hidden applications* that drastically complicate the implementation, and the formalizations provided in [15] abstract from these features.

Finally, the set of well-typed terms of type A in context c is represented by means of semantic subtypes and derivability:

$$wtt := \lambda(c, A) : cxt \times trm. \\ \{M : trm \mid deriv(c, M, A) = true\}$$

A related notion is that of inhabitedness (prova-

bility) of some type (formula) A in context c:

$$\begin{array}{l} inh \;:=\; \lambda(c, A) : cxt \times trm. \\ \quad \exists\, M : trm.\ (deriv(c, M, A) = true) \end{array}$$

4 Reflection Principles

The encodings given so far constitute the meta-level encodings of TYPELAB, but no connections between object-level and meta-level deductions have yet been provided. The following *reflection rules* provide such connections:

$$\frac{\Gamma \vdash M : A}{\vdash up(\Gamma, M, A) \;:\; inh(\ulcorner\Gamma\urcorner, \ulcorner A\urcorner)}$$

$$\frac{\vdash p \;:\; inh(\ulcorner\Gamma\urcorner, \ulcorner A\urcorner)}{\Gamma \vdash down(p) \;:\; A}$$

Note that both *up* and *down* can not be encoded as functions of TYPELAB. In the context of the *pure calculus of construction* these rules have been shown to be admissible, and the object-level system together with the encodings and reflection rules is a conservative extension of the original system [17].

Corresponding to the reflection rules *reflect up* and *reflect down* the basic TYPELAB system has been extended with prover commands that allow one to switch between different levels. For example, goal $\Gamma \vdash ?_0 : A$ may be rewritten as $\vdash ?_1 : inh(\ulcorner\Gamma\urcorner, \ulcorner A\urcorner)$. This process of reification can be iterated. Next reification, for example, yields $\vdash ?_2 : inh(\ulcorner\urcorner, \ulcorner inh(\ulcorner\Gamma\urcorner, \ulcorner A\urcorner)\urcorner)$.

Another reflection rule involves computing of object-level terms from representations thereof:

$$\frac{\vdash \ulcorner M\urcorner \;:\; wtt(\ulcorner\Gamma\urcorner, \ulcorner A\urcorner)}{\Gamma \vdash M : A}$$

We refer to this transition as *safe unquoting*, since the unquoted term is known to be type-correct in the unquoted context. On the other hand, one may not always want to construct proofs of well-typedness for results of meta-operations. For this reason, we also allow for *unsafe unquoting*. In this case, transitions from meta-level to object-level involve type checking of the reflected term in the current context.

5 Mixing Reasoning on Different Levels

Reflection rules can be utilized to solve object-level goals by reifying the problem, applying a meta-theorem, and reflecting the constructed representation of a proof object. This procedure is similar to that of Weyhrauch's FOL system, with the addition that proof objects are constructed explicitly. Consider, for example, the (toy) problem of solving

$$A : Prop \;\vdash\; ?_0 \;:\; A \to A$$

One possibility to solve such a goal could involve a meta-theorem such as

$$\begin{array}{l} obvious \;:\; \Pi c : cxt, A : trm. \\ \quad deriv(c, A, \ulcorner Prop\urcorner) = true \to \\ \quad\quad inh(c, \ulcorner `A` \to `A`\urcorner) \end{array}$$

In a first step one reifies the goal at hand:

$$\vdash ?_1 \;:\; inh(\ulcorner A : Prop\urcorner, \ulcorner A \to A\urcorner)$$

Now, one applies the *refine* command of the TYPELAB prover that matches the current goal with the conclusion of meta-theorem *obvious*. Justification of $?_0$ relative to $?_1$ is computed from meta-proof *obvious* and downward reflection:

$$?_0 \;:=\; down(obvious(\ulcorner A : Prop\urcorner, \ulcorner A\urcorner, ?_2))$$

Here, $?_2$ is a proof obligation

$$\vdash ?_2 \ : \ deriv(\ulcorner A : Prop \urcorner, \ulcorner A \urcorner, \ulcorner Prop \urcorner) = true$$

that can be easily shown to hold using semantic attachment for derivability. This process of reifying a goal followed by application of refinement and discharging of "trivial" goals is condensed in a proof command called *reflect*. Above toy problem, for example, could be solved in one step by issuing the prover command *reflect obvious*. Note that this command, in addition, tries to instantiate meta-theorems with the representation of the current context.

6 Meta-Operators as Generators

Large parts of proof libraries of Coq [4] and Lego [11] consist of rather trivial developments – comprising both theorems and programs and proofs – that have to be carried out separately for each datatype. A better idea seems to capture the general scheme of such developments once-and-for-all and apply these schemes for each instance. Thus, developments on inductive datatypes are a particularly rich source for formalizing meta-operators.

For example, specifications of inductive datatypes that are given by means of introduction, formation, elimination, and equality rules follow a certain pattern, and the form of elimination and equality can already be computed from formation and introduction rules by means of the *inversion principle* (see [18]). This knowledge can be poured into a meta-operator named *gen_dt_spec*. Consider, for example, the inductive datatype

$$\begin{array}{lll} nat & := & \text{DATATYPE } X : Type(0). \\ & & \quad zero : X \ \mid \ succ : X \to X \\ & & \text{END} \end{array}$$

Applying *gen_dt_spec* together with *unsafe unquoting* yields the specification *Nat_Spec* of Figure 1:

$$Nat\_Spec \ := \ \llcorner gen\_dt\_spec(\ulcorner nat \urcorner) \lrcorner;$$

Note also that such meta-functions need explicit access to variables in order to distinguish between recursive and non-recursive arguments.

The generated specifications can easily be extended to support further notions related to inductive datatypes. Meta function *gen_disj_thm*, for example, generates for a given inductive datatype theorems that state disjointness of constructors together with proof objects. For lack of space we do not elaborate on how to implement

$$\begin{array}{l} zero\_not\_succ \ := \\ \quad \forall\, n : nat.\ zero \neq succ(n) \\ \\ zero\_not\_succ\_prf \ : zero\_not\_succ \ := \\ \lambda x : nat, H : (zero = succ(x)).\ eq\_sym \\ (H\ (\lambda y : nat. \\ \quad (elim\ (\lambda\_ : nat.\ Prop)\ true \\ \qquad (\lambda\_ : nat, \_ : Prop.\ false) \\ \qquad y) \\ \qquad\quad = \ true) \\ \quad (eq\_refl\ true)) \\ (\lambda z : Prop.\ z) \\ (\lambda A : Prop, a : A.\ a) \end{array}$$

Figure 3: Disjointness of Constructor Terms

the meta-functions ($gen\_disj\_thm\ i\ j\ \ulcorner D \urcorner$) and ($gen\_disj\_prf\ i\ j\ \ulcorner D \urcorner$) that respectively generate the disjointness theorem for the i-th and j-th constructors of datatype D and a proof thereof; see [15] for a detailed description of these functions. These functions can be applied – again using *unsafe unquoting* – to specific datatypes like natural numbers:

$$zero\_not\_succ\ :=\ \llcorner gen\_disj\_thm\ zero\ succ\ \ulcorner nat \urcorner \lrcorner$$

$$zero\_not\_succ\_prf\ :zero\_not\_succ\ :=\ \llcorner gen\_disj\_prf\ zero\ succ\ \ulcorner nat \urcorner \lrcorner$$

This yields the same terms as shown in Figure 3.

7 Formal Tactics

Another application of the reflective architecture presented here involves tactics that are capable of abstracting from the basic inference rules. A tactic is a (meta-) function that maps, in case of success, a goal $(c, g) : cxt \times trm$ to a list of subgoals, or it fails. Unlike *LCF* tactics [14] these tactics do not have to compute justifications in terms of primitive inference rules. Instead, a certain correctness result that states *existence* of such a proof object is established once and for all. Since correctness of tactics only requires existence of a proof, one may easily integrate decision procedures in a sound way. Also, higher-order tactics can be defined freely but have to be proven correct.

An example may help to clarify some points. Assume given the problem

$$A, B : Prop\ \vdash ?_0\ :\ (A \wedge B) \rightarrow (B \wedge A)$$

The prover command *tac*, that is responsible for applying tactics, expects as argument a meta-function together with a correctness proof of this tactic. Issuing, for instance, the command

$$tac\ flatten\ flatten\_corr$$

causes application of tactic *flatten*. This tactic repeatedly applies conjunction and implication elimination, and *flatten_corr* is a correctness proof of this tactic. More precisely, in a first step the current goal is represented on the meta-level as

$$(\ulcorner A, B : Prop \urcorner, \ulcorner (A \wedge B) \rightarrow (B \wedge A) \urcorner)$$

and meta-operator *flatten* is applied to this goal. This yields the result

$$yes([(\ulcorner A, B : Prop, H_1 : A, H_2 : B \urcorner, \ulcorner B \urcorner), (\ulcorner A, B : Prop, H_1 : A, H_2 : B \urcorner, \ulcorner A \urcorner)])$$

In a last step, tactic command *tac* reflects the result down to the object-level and one is left to show the two subgoals

$$A, B : Prop, H_1 : A, H_2 : B\ \vdash ?_1\ :\ B,\ ?_2\ :\ A$$

Moreover, the relationship between meta-variables $?_0$ and $?_1$, $?_2$ is computed using correctness result *flatten_corr* of tactic *flatten*.

8 Conclusions

We have described a reflective architecture that is capable of applying meta-operators and meta-theorems to object-level problems, and performed a number of experiments related to formal program construction using this architecture.
This reflective architecture is open in the sense

that new knowledge can be added. On the other hand, such knowledge can not be arbitrarily added to the system; this process possibly involves formal proof. Altogether, starting with a relatively small kernel our software development tool can be adjusted and extended in a safe way to meet new requirements.

A prototypical implementation of this reflective architecture has been developed on top of the TYPELAB system, and the experiments reported herein have been carried out in this system. In order to make the approach practical, however, several improvements have to be made. While speed of executing quoting/unquoting and application of reflection principles is already satisfactory, reduction of meta-operator application is currently rather slow. Consequently, some mechanism has to be introduced to the system in order to speed up reduction of the underlying type theory. It is mainly this lack of execution speed that prevented us from encoding more meta-operators and applying these operators to the *trm* representation itself in order to bootstrap the given encoding and enhance "knowledge" about this encoding.

Our short term goal in this respect is to develop a formal "theory of datatypes" that includes most standard proofs and theorems and standard functions on datatypes like (decidable) equality and *map*-functions together with characteristic theorems on these functions. In the long run, we believe that a large collection of formalized meta-operators that encode common knowledge on programming (and proving) tasks greatly facilitates formal software development, and allows one to develop and maintain large program systems in a fully formal way.

References

[1] S.F. Allen, R.L. Constable, D.J. Howe, and W.E. Aitken. The Semantics of Reflected Proof. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, pages 95–105. IEEE CS Press, 1990.

[2] R. Balzer, T. E. Cheatham, and C. Green. Software Technology in the 1990's: Using a new Paradigm. *IEEE Computer*, 16(11):39–45, Nov. 1983.

[3] R.S. Boyer and J.S. Moore. Metafunctions: Proving them Correct and Using them Efficiently as New Proof Procedures. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, chapter 3. Academic Press, 1981.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Murthy, C. Parent, Chr. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide (Version 5.8)*. INRIA-Rocquencourt – CNRS - ENS Lyon. Projet Formel.

[5] F. Giunchiglia and P. Traverso. A Formalized Metatheory for Reflective Reasoning. Technical Report IRST-9111-01, Istituto per la Ricerca Scientifica e Tecnologica, 1991.

[6] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[7] R. Harper and R. Pollack. Type Checking, Universal Polymorphism, and Type Ambiguity in the Calculus of Constructions. In

TAPSOFT'89, volume II, Lecture Notes in Computer Science, pages 240–256. Springer-Verlag, 1989.

[8] D. Howe. Reflecting the Semantics of Reflected Proof. In P. Aczel, H. Simmons, and S. Wainer, editors, *Proof Theory*, pages 227–250. Cambridge University Press, 1992.

[9] D.J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988. Available as technical report TR 88-925 from the Department of Computer Science, Cornell University.

[10] Z. Luo. ECC, an Extended Calculus of Constructions. In *Proc. of the Fourth Ann. Symp. on Logic in Computer Science*, pages 386–395, Asilomar, California, June 1989.

[11] Z. Luo and R. Pollack. The Lego Proof Development System: A User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[12] Ch.E. Ore. The Extended Calculus of Constructions (ECC) with Inductive Types. *Information and Computation*, 99, Nr. 2:231–264, 1992.

[13] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language.* Computer Science Lab, SRI International, Menlo Park CA 94025, March 1993.

[14] Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Number 2 in Cambride Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[15] H. Pfeifer. Eine reflexive Architektur zur Darstellung von Beweis- und Softwareentwicklungsschritten in Typtheorie. Master's thesis, Universität Ulm, March 1995.

[16] H. Rueß. Report on the Specification Language *qed.* Korso Working Paper 93–1, Universität Ulm, 1993.

[17] H. Rueß. *Meta-Programming in the Calculus of Constructions.* PhD thesis, Universität Ulm, 1995. Forthcoming.

[18] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[19] F.W. von Henke, A. Dold, H. Rueß, D. Schwier, and M. Strecker. Construction and Deduction Methods for the Formal Development of Software. Ulmer Informatik-Berichte 94-09, Universität Ulm, June 1994.

[20] R. W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence*, 13(1):133–170, 1980.