Valid Extensions of Introspective Systems: A Foundation for Reflective Theorem Provers

Arno Schönegge*
Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe
D-76128 Karlsruhe, Germany
email: `schoeneg@ira.uka.de`

Abstract

Introspective systems have been proved useful in several applications, especially in the area of automated reasoning. In this paper we propose to use structured algebraic specifications to describe the embedded account of introspective systems. Our main result is that extending such an introspective system in a valid manner can be reduced to development of correct software. Since sound extension of automated reasoning systems again can be reduced to valid extension of introspective systems, our work can be seen as a foundation for extensible introspective reasoning systems, and in particular for reflective provers. We prove correctness of our mechanism and report on first experiences we have made with its realization in the `KIV` system (`K`arlsruhe `I`nteractive `V`erifier).

Contents

*This work was supported by the Deutsche Forschungsgemeinschaft as part of the focus program "Deduktion".

1 Introduction

An *introspective system* is a software systems that has a partial description of itself embedded in itself [36, 51]. Such systems have been proved useful in areas like programming languages [7, 50, 32], knowledge representation [35], and deduction. The latter is the concern for this paper. We believe that introspection provides a solution to two main problems of powerful reasoning systems:

- soundness: How can we guarantee that the formulas proved by a theorem prover are actually theorems? This question arises as modern reasoning systems (e.g. `Nqthm` [9], `Nuprl` [15], `KIV` [45], `Never` [16], `HOL` [21]) are quite complex. One approach — taken e.g. in `Never` [34] and Ω-`MKRP` [29, 30] — is to transmit the proofs constructed by a complex prover to a proof checker which is so simple that it can be trusted. However this method leads to inefficiency. Another approach is to guarantee the soundness of the (complex) reasoning system itself. This can be done by starting out with a simple (and sound) prover and soundly extending it step by step.

- flexibility: It is a well-known fact that depending on the application different provers are well suitable. So it is desirable to tune a reasoning system for intended applications. This tuning can be done by extending the prover with kinds of rules and procedures that humans have found effective in constructing proofs. Understandably enough the soundness of such extensions has to be guaranteed by some mechanism.

So both, soundness problem and flexibility problem, can be reduced to the problem of sound extensions. The primary traditional solution to this problem is *tactics* [22, 15, 41, 27]. Tactic mechanisms are sound as each tactic application amounts to constructing a justification using primitive inference rules. However, (as has often been pointed out, e.g. in [49, 19, 39, 5]) this may be very time-consuming. One alternative that avoids this inefficiency is to explicitly prove the soundness of extensions. This requires the possibility to reason about (extensions of) the reasoning system. In so-called *reflective* provers this reasoning is done within the system itself. In order to reason about itself such a system must have an embedded declarative description of (a part of) itself. Therefore, reflective provers are introspective systems. However, though the reflective approach has been pursued several times [17, 52, 10, 33, 28, 38, 26], it appears that up to now it has not been used in significant applications. We believe that the reason for this is in the particular mechanisms taken so far: reflection requires to prove complex obligations, and so techniques are necessary to handle these tasks. Our approach reduces sound extensions of a (reflective) theorem prover to development of correct software. As a consequence, there are no new techniques

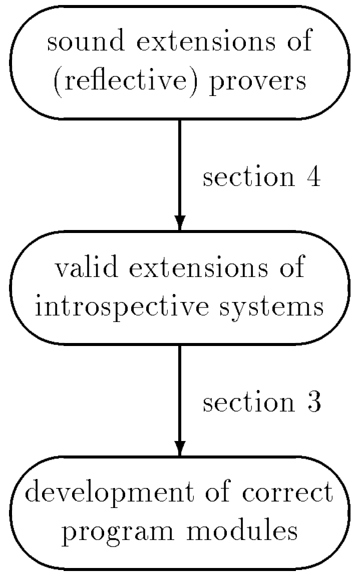

Figure 1: Reducing sound extensions of (reflective) provers to development of correct program modules

to be developed but we can directly employ advanced techniques known for correct software development (e.g. modularization). First experiments with the realization of our mechanism in the `KIV` system look quite promising.

This paper is organized as follows. Section 2 recalls some basic notions. In section 3 we introduce the notion of introspective systems and prove a theorem which states that validity preserving extension of introspective systems can be reduced to development of correct program modules. In section 4 we show how soundly extending a reflective prover can be reduced to extending an introspective system in a validity preserving manner. Figure 1 illustrates this situation. In section 5 related work is considered, and in the final section we draw conclusions and report on first experiences we have made with the realization in the `KIV` system.

2 Basic notions

Signatures, formulas. We consider many-sorted *signatures* $\Sigma = (S, F)$ with a set of sorts S and a set F of function[1] symbols equipped with a

[1]Without loss of generality we assume to have no predicate symbols. Instead we use functions with a (predefined) sort $bool = \{tt, ff\}$ as target sort.

mapping[2] $sort : F \to S^* \times S$. If $\Sigma' = (S', F')$ with $S' \subseteq S$ and $F' \subseteq F$ is again a signature, we call Σ' a *subsignature* of Σ. For a family $X := \{X_s \mid s \in S\}$ of variable sets $L(\Sigma, X)$ denotes the set of first-order formulas over Σ and X.

Algebras. Formulas are interpreted over Σ-algebras. For $\Sigma = (S, F)$ a Σ-*algebra* $\mathcal{A} = ((A_s)_{s \in S}, (f_{\mathcal{A}})_{f \in F})$ consists of non-empty carrier sets A_s and interpretations $f_{\mathcal{A}}$ for the symbols from F. For $f \in F$ with $sort(f) = (s_1 \dots s_n, s)$ the interpretation $f_{\mathcal{A}}$ is a total function from $A_{s_1} \times \dots \times A_{s_n}$ to A_s. For a subsignature $\Sigma' = (S', F')$ of Σ we call $\mathcal{A} \mid_{\Sigma'} := ((A_s)_{s \in S'}, (f_{\mathcal{A}})_{f \in F'})$ the Σ'-*reduct* of $\mathcal{A}$.

A Σ-algebra $\mathcal{A} = ((A_s)_{s \in S}, (f_{\mathcal{A}})_{f \in F})$ is called *generated* if for each sort $s \in S$ every carrier element $a \in A_s$ can be denoted by a ground term over Σ. For a formula $\varphi \in L(\Sigma, X)$ and a Σ-algebra $\mathcal{A}$ we write $\mathcal{A} \models \varphi$ if $\mathcal{A}$ is a model of φ. By $Gen(\Sigma, \Phi)$ we denote the set of generated Σ-algebras which are models of a formula set $\Phi \subset L(\Sigma, X)$.

Algebraic specifications. As motivated in [44] we use full first order specifications and consider the class of all generated models as its semantics (so-called *loose semantics*). A *specification* $SP = (\Sigma, X, \phi)$ consists of a signature $\Sigma = (S, F)$, a family $X = \{X_s \mid s \in S\}$ of countably infinite variable sets, and a set $\phi \in L(\Sigma, X)$ of first-order formulas over Σ and X. By $sig(SP) := \Sigma$ we denote the signature of SP, by $op(SP) := F$ its function symbols, by $vars(SP) := X$ its variable sets, and by $ax(SP) := \phi$ its axioms. For the semantics of a specification we adopt an approach of Giarratana et al. [18] and the Munich CIP-group [55, 54]: We define the *semantics* of $SP = (\Sigma, X, \phi)$ by $Sem_S(SP) := Gen(\Sigma, \phi)$.

A specification $SP_1 = (\Sigma_1, X_1, \phi_1)$ with $\Sigma_1 = (S_1, F_1)$ is an *enlargement* of the specification $SP_2 = (\Sigma_2, X_2, \phi_2)$ with $\Sigma_2 = (S_2, F_2)$ if $S_1 = S_2$, $F_1 \supseteq F_2$, $X_1 \supseteq X_2$, and $\phi_1 \supseteq \phi_2$. The enlargement operation can be used to add new function symbols to the signature of a specification SP and new axioms to those of SP which describe the new functions. For a specification $SP = (\Sigma, X, \phi)$, a subsignature $\Sigma' = (S', F')$ of Σ, and $X' := \{X_s \mid s \in S'\}$ we call $SP \mid_{\Sigma'} := (\Sigma', X', \phi \cap L(\Sigma', X'))$ the Σ'-*reduct* of SP.

Programs. We assume a typed programming language and an algebraic semantics defined for programs in this language (cf. [3]). A *program* $PRG = (TD, PD)$ consists of type declarations TD and procedure declarations PD and is built over a set of type identifiers $TIDS$ and a set of procedure identifiers $PIDS$ equipped with a mapping $type : PIDS \to TIDS^* \times TIDS$. We only consider well-formed programs; especially we demand that all type and procedure identifiers used in PRG are declared in PRG and that all

[2] S^* are the finite words over S.

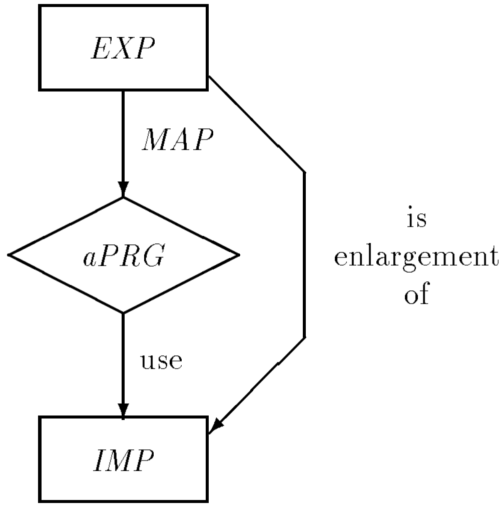

Figure 2: An (enlargement) module

procedure declarations in PD exhibit the typing indicated by their procedure identifiers. Furthermore we restrict ourselves to procedures that are functional[3] and never fail to terminate.[4] For (a part of) a program PRG, $dt(PRG)$ and $dp(PRG)$ denote the set of type identifiers and procedure identifiers declared in it, respectively. The *semantics* of the program PRG is the algebra induced by the type declarations TD (which has exactly one carrier set A_t for each $t \in dt(PRG)$) enlarged by functions $F_p := Sem_P(p, PRG)$, $p \in dp(PRG)$, computed by the corresponding declarations in PD. So if $type(p) = (t_1 \ldots t_n, t)$ then F_p is a total function from $A_{t_1} \times \cdots \times A_{t_n}$ to A_t.

Abstract programs. For a signature $\Sigma = (S, F)$ an *abstract program aPRG over* Σ is a set of procedure declarations which use the function symbols from F as elementary operations. No type declarations are required as the procedures operate on the sorts S. Wellformedness is defined as above and implicitly assumed. Again we restrict on functional procedures but do not demand termination. The *semantics* $Sem_{AP}(p, aPRG)$ of an abstract procedure in $aPRG$ with identifier p is a total function that maps Σ-algebras $\mathcal{A} = ((A_s)_{s \in S}, (f_{\mathcal{A}})_{f \in F})$ into the partial function over the carrier of $\mathcal{A}$ that is computed when calling p where the symbols $f \in F$ occurring in $aPRG$ are interpreted by $f_{\mathcal{A}}$.

[3] *Functional* procedures do not use global variables and use reference parameters only as result parameters.

[4] The demand for termination can be dropped if one uses *partial* algebras as semantics.

Program modules. As proposed in [2] the notion of program modules can be used for vertical refinement of specifications. In order to refine a specification SP_1 it is implemented in terms of a (more elementary) specification SP_2. In this paper we restrict ourselves to modules where SP_1 is an enlargement of SP_2. Formally, a *module* $M = (EXP, IMP, aPRG, MAP)$ consists of two specifications $EXP = ((S_{EXP}, F_{EXP}), X_{EXP}, \phi_{EXP})$ and $IMP = ((S_{IMP}, F_{IMP}), X_{IMP}, \phi_{IMP})$, a set $aPRG$ of abstract procedures over $\Sigma_{IMP} = (S_{IMP}, OP_{IMP})$, and a *representation function* MAP. We demand EXP to be an enlargement of IMP, so $S_{EXP} = S_{IMP}$ and $F_{EXP} \supseteq F_{IMP}$. EXP and IMP are called the *export* and the *import* of the module M, respectively. MAP is a total, injective function that maps function symbols from $F_{EXP} \setminus F_{IMP}$ into procedure identifiers of $aPRG$ with the same typing. Roughly speaking, the *semantics* $Sem_M(M)$ *of a module* $M = (EXP, IMP, aPRG, MAP)$ is a partial function induced by $aPRG$. It maps generated models of the import specification IMP to generated algebras of the export signature $\Sigma_{EXP} = (S_{EXP}, F_{EXP})$ as follows: For $\mathcal{A} = \Big((A_s)_{s \in S_{IMP}}, (f_{\mathcal{A}})_{f \in F_{IMP}}\Big) \in Sem_S(IMP)$ we set

$$Sem_M(M)(\mathcal{A}) \;:=$$
$$\left(\left(A_s\right)_{s \in S_{EXP}}, \left(f_{\mathcal{A}}\right)_{f \in F_{IMP}} \cup \left(Sem_{AP}\Big(MAP(f), aPRG\Big)(\mathcal{A})\right)_{f \in F_{EXP} \setminus F_{IMP}}\right)$$

if all functions $Sem_{AP}\Big(MAP(f), aPRG\Big)(\mathcal{A})$ are total. Otherwise, the value of $Sem_M(M)(\mathcal{A})$ is undefined. A module M is called *correct* if the function $Sem_M(M)$ is total and its values are models of the export specification EXP. Informally, this means that the procedures of the implementation terminate and exhibit the behavior specified in the export.

Signature representations. For a signature $\Sigma = (S, F)$ and a program PRG a Σ*-representation* REP *in* PRG is a total, injective function that maps sorts from S into type identifiers from $dt(PRG)$, and function symbols from F into procedure identifiers from $dp(PRG)$, so that[5] $REP_T(sort(f)) = type(REP(f))$. We call

$$\mathcal{A}_{PRG,REP} := \left(\left(A_{REP(s)}\right)_{s \in S}, \left(Sem_P(REP(f), PRG)\right)_{f \in F}\right)$$

the *algebra induced by* PRG *and* REP. $\mathcal{A}_{PRG,REP}$ is a Σ-algebra. By REP_P we denote the total function that maps abstract procedure declarations PRC over Σ into non-abstract procedure declarations:[6] $REP_P(PRC)$ is essentially

[5]The function REP_T is defined to map sorts of function symbols f in Σ into types of procedures by $REP_T(s_1 \ldots s_n, s) := (REP(s_1) \ldots REP(s_n), REP(s))$.

[6]We assume separate identifiers for abstract procedures and allow to use the same ones as identifiers for non-abstract procedures (with a different typing).

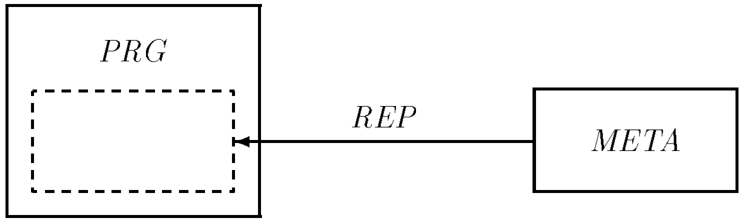

Figure 3: An introspective system

the same as PRC itself calling the procedures $REP(f)$ whenever a symbol $f \in F$ occurs in PRC.

In the next section we make use of the following connection between abstract programs and non-abstract programs:

Fact 2.1 *Let aPRG be an abstract program over a signature Σ and REP a Σ-representation in a program PRG with $dp(aPRG) \cap dp(PRG) = \emptyset$. Then for each procedure identifier p in aPRG holds:*

$$Sem_{AP}(p, aPRG)(\mathcal{A}_{PRG,REP}) = Sem_P(p, PRG \cup REP_P(aPRG)).$$

3 Introspective systems and their extensions

An introspective system is a software system that has an embedded account of itself (cf. [50, 36]), i.e. a partial description of itself in itself. We propose to use (first-order) specifications to represent such descriptions.

Definition 3.1 *An* introspective system $IS = (PRG, META, REP)$ *consists of a program PRG, a specification META, and a sig(META)-representation REP in PRG.*

The representation REP explicitly establishes a relation between the embedded account $META$ and the program PRG. REP can be described as a table or implemented as a procedure in the programming language too. Notice that introspection is restricted: $META$ represents components from PRG only and not from REP or $META$ itself. Notice further that $META$ may be merely a partial description of PRG: there can be procedures in PRG that are not in the range of REP. We want $META$ to represent PRG "adequately[7]", i.e. that the meaning of the represented part of PRG is in fact

[7] Besides adequacy, for some applications a kind of *faithfulness* of $META$ with respect to PRG is required. This can be achieved by demanding $META$ to be monomorph. We will not discuss this aspect here but refer to [44, 47].

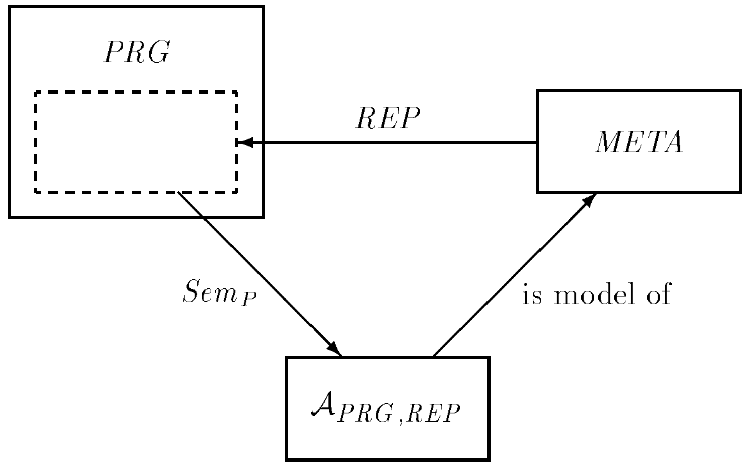

Figure 4: Validity of an introspective system

modeled by *META*. If *IS* has this property we call it a *valid* introspective system.

Definition 3.2 *An introspective system $IS = (PRG, META, REP)$ is* valid *if the algebra induced by PRG and REP is a model of META, i.e.*

$$\mathcal{A}_{PRG,REP} \in Sem_S(META).$$

This situation which is illustrated in figure 4 looks very similar to the work that has been done in connection with the reflective theorem prover `GETFOL` [26, 23]. However, in all what follows we significantly differ from the approach taken in `GETFOL`. For more details see section about related work.

In the rest of this section we deal with the question on how an introspective system can be extended in a validity preserving manner. We have restricted ourselves to extensions of an introspective system where no new sorts can be added to the signature of *META*. This restriction is not absolutely compelling but it simplifies presentation.

Definition 3.3 *An introspective system $IS_1 = (PRG_1, META_1, REP_1)$ is an* extension *of an introspective system $IS_2 = (PRG_2, META_2, REP_2)$ if $PRG_1 \supseteq PRG_2$ holds, $META_1$ is an enlargement of $META_2$, and furthermore*[8] *$REP_1 \mid_{sig(META_2)} = REP_2$.*

The following theorem is our main result. Informally, it states that (some kinds of) validity preserving extensions can be reduced to construction of correct program modules.

[8] For a function $g : D \to R$ and $D' \subseteq D$ we write $g \mid_{D'}$ to denote the restriction of g on D', i.e. $g \mid_{D'}: D' \to R$ with $g \mid_{D'} (x) := g(x)$ on D'.

Theorem 3.1 *Let* $IS = (PRG, META, REP)$ *be an introspective system and* $M = (META^+, META, aPRG, MAP)$ *a module with* $dp(aPRG)$ *and* $dp(PRG)$ *disjoint. Then for* $IS^+ := (PRG^+, META^+, REP^+)$ *with* $PRG^+ := PRG \cup REP_P(aPRG)$ *and*[9] $REP^+ := REP \cup MAP$ *holds:*

1. IS^+ *is an introspective system,*

2. IS^+ *is an extension of* IS*, and*

3. *if* IS *is valid and* M *is correct then* IS^+ *too is valid.*

Proof. The proof is almost straightforward.

1. It is easy to see that REP^+ is a $sig(META^+)$-representation in PRG^+. Notice that REP^+ is defined at all since all function symbols added by enlargment have to be new.

2. This is an immediate consequence of definition 3.3. Remember that $META^+$ is an enlargement of $META$ because of our restricted notion of modules.

3. From validity of IS follows that $\mathcal{A}_{PRG,REP} \in Sem_S(META)$. So, because of the correctness of M, $Sem_M(M)(\mathcal{A}_{PRG,REP})$ is defined and in $Sem_S(META^+)$. Furthermore it holds (for $sig(META) =: (S, F)$ and $sig(META^+) =: (S^+, F^+)$):

$$\begin{aligned}
&\mathcal{A}_{PRG^+,REP^+} \\
&= \left(\left(A_{REP^+(s)}\right)_{s\in S^+}, \left(Sem_P\left(REP^+(f), PRG^+\right)\right)_{f\in F^+}\right) \\
&= \left(\left(A_{REP^+(s)}\right)_{s\in S}, \left(Sem_P\left(REP^+(f), PRG^+\right)\right)_{f\in F} \bigcup \right. \\
&\qquad \left. \left(Sem_P\left(REP^+(f), PRG^+\right)\right)_{f\in F^+\setminus F}\right) \\
&= \left(\left(A_{REP(s)}\right)_{s\in S}, \left(Sem_P\left(REP(f), PRG\right)\right)_{f\in F} \bigcup \right. \\
&\qquad \left. \left(Sem_P\left(MAP(f), PRG \cup REP_P(aPRG)\right)\right)_{f\in F^+\setminus F}\right) \\
&= \left(\left(A_{REP(s)}\right)_{s\in S^+}, \left(Sem_P\left(REP(f), PRG\right)\right)_{f\in F} \bigcup \right.
\end{aligned}$$

[9] For two functions $g_1 : D_1 \to R_1$ and $g_2 : D_2 \to R_2$ with $D_1 \cap D_2 = \emptyset$ we define $g_1 \cup g_2 : D_1 \cup D_2 \to R_1 \cup R_2$ by $(g_1 \cup g_2)(x) := \begin{cases} g_1(x) \text{ , if } x \in D_1 \\ g_2(x) \text{ , if } x \in D_2 \end{cases}$

$$\Big(Sem_{AP}\big(MAP(f), aPRG\big)(\mathcal{A}_{PRG,REP})\Big)_{f \in F^+ \setminus F}\Big)$$

$$= \; Sem_M(M)(\mathcal{A}_{PRG,REP})$$

The fourth equation is an application of fact 2.1. In summary we have

$$\mathcal{A}_{PRG^+,REP^+} \;=\; Sem_M(M)(\mathcal{A}_{PRG,REP}) \;\in\; Sem_S(META^+)$$

which states the validity of IS^+.

■

Theorem 3.1 suggests the following instruction how to extend an introspective system $IS = (PRG, META, REP)$ so that its validity is preserved:

1. build $META^+$: <u>specify</u> the procedures $F_1, \ldots, F_n$ to be added to the program of IS. That is, enlarge $META$ by new function symbols $f_1, \ldots, f_n$ representing the procedures and by axioms describing their effect.
2. build $aPRG$: <u>implement</u> $F_1, \ldots, F_n$ as abstract procedures over $sig(META)$.
3. build MAP: <u>establish the relationship</u> $f_i \to F_i$, $i \in \{1, \ldots, n\}$ explicitly.
4. <u>prove</u> the correctness of the module $M = (META^+, META, aPRG, MAP)$.
5. <u>update</u> IS with $IS^+ := (PRG^+, META^+, REP^+)$ where $PRG^+ := PRG \cup REP_P(aPRG)$ and $REP^+ := REP \cup MAP$.

Carrying out these five steps results in an (extended) valid introspective system again and the whole process can be arbitrarily iterated. Notice that steps 1 – 4 are exactly the same that are required in development of correct modular software systems [46]. So techniques and tools developed for this task can be directly applied.

The definition of an introspective system (and validity preserving extensions of it) as presented here does not fit for all applications where introspection is required. However, in the next section we demonstrate that our notion is useful in building powerful theorem provers.

4 Application to reflective reasoning systems

Assume we want to build a powerful prover. Then on the one hand soundness preserving extensibility is required (as motivated in the introduction). However, on the other hand, if the underlying logic is powerful enough (to express and prove module correctness) the reasoning system itself can be used as a tool to support sound extensions of itself. To get into such a situation we suggest to proceed as follows:

(A) Design the reasoning system as an introspective system, so that sound extensions of it can be reduced to valid extensions of introspective systems, and therefore to construction of correct modules (by Theorem 3.1).

(B) Define a uniform mapping VC from modules into formulas of the underlying logic, so that a module M is correct if $VC(M)$ can be proved valid.

As illustrated in figure 5, it is now allowed to reduce the task of soundly extending the prover to a proof task which can be performed using the prover itself. Following the definition in [36] we have built a *reflective* prover since it is "about itself in a causally connected way", i.e. it is able to reason about itself in order to modify itself (see figure 6).

How to do **(B)** has already been treated in [46] (where it has been carried out in the setting of dynamic logic). Informally, to show correctness of a module $M = (EXP, IMP, aPRG, MAP)$ it is sufficient to prove that all procedures in $aPRG$ terminate and exhibit the behavior specified in EXP. So it remains the question on how to perform **(A)**, and the rest of this section is about it.

First of all we define some further notions. For a given logic (syntax, semantics) an *(inference) rule* is a computable function which is of a specific type $RULETYPE$. We do not specify this type any further here because it depends on the kind of reasoning system under consideration.[10] A rule is said to be *sound* if it is semantically correct. A *calculus* is a finite set of rules which we call *sound* if all rules in it are sound. A *reasoning system* $RS = (PRG, IM)$ is a program $PRG \cup IM$ divided up into PRG and an *inference machinery* IM which implements a calculus, i.e. $type(p) = RULETYPE$ for all $p \in dp(IM)$. RS is called *sound* if IM implements a sound calculus.

[10] For example in the case of proof checkers, rules are best represented as predicates, i.e. $RULETYPE = (formulalist \times formula, bool)$ may be a good choice.

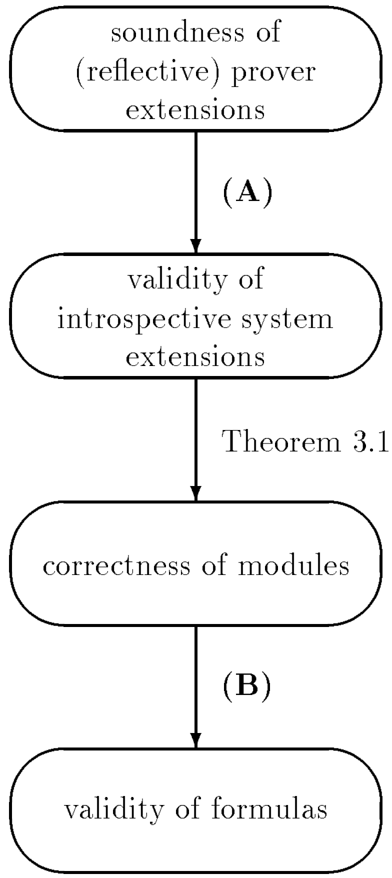

Figure 5: Reducing soundness of (reflective) prover extensions to validity of formulas

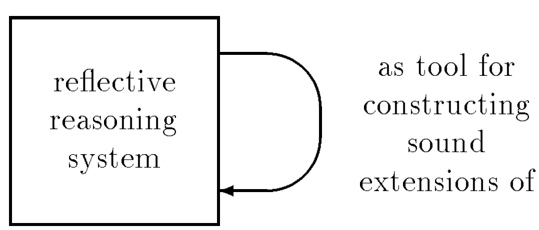

Figure 6: A reflective reasoning system

Definition 4.1 *$IRS = (PRG, IM, META, REP)$ is an* introspective reasoning system *if:*

(a) *$IS := (PRG \cup IM, META, REP)$ is an introspective system.*

(b) *$RS := (PRG, IM)$ is a reasoning system.*

(c) *$dp(IM) \subseteq REP(sig(META))$.*

(d) *"META formalizes a soundness criterion":*
for any enlargement $META^+$ of $META$ and any function symbol $f \in op(META^+)$ with $REP_T(sort(f)) = RULETYPE$ there is a formula $SOUNDRULE(f) \in L(sig(META^+), vars(META))$ so that for any $\mathcal{A} \in Sem_S(META^+)$ holds: if $\mathcal{A} \mid_{sig(META)} = \mathcal{A}_{PRG \cup IM, REP}$ and $\mathcal{A} \models SOUNDRULE(f)$ then $f_{\mathcal{A}}$ is a sound rule.

(e) *"META formalizes soundness of IM":*
for all $f \in op(META)$ with $REP(f) \in dp(IM)$ holds[11] $META \models SOUNDRULE(f)$.

Via **(a)** and **(b)** the notions of *validity* and *extensions* (of introspective systems) and *soundness* (of reasoning systems) are defined for introspective reasoning systems too.

Our notion of introspective reasoning systems is very restricted, especially **(c)** – **(e)**. For some applications a relaxation may be reasonable. However, we have found this definition appropriate for representation in this paper.

The following theorem states that soundness of an introspective reasoning systems can be reduced to its validity.

Theorem 4.1 *Any valid introspective reasoning system is sound.*

Proof. Let $IRS = (PRG, IM, META, REP)$ be an introspective reasoning system. If it is valid then holds $\mathcal{A} := \mathcal{A}_{PRG \cup IM, REP} \in Sem_S(META)$. Let $f \in op(META)$ a function symbol that represents an inference rule, i.e. $REP(f) \in dp(IM)$. Then by **(e)** we have $META \models SOUNDRULE(f)$ and therefore $\mathcal{A} \models SOUNDRULE(f)$. Using **(d)** (for the degenerated case $META^+ = META$) we get that $f_{\mathcal{A}} = Sem_P(REP(f), PRG \cup IM)$ is a sound rule. So by **(c)** all procedures in IM implement sound rules. ∎

We use this theorem to prove a corollary about sound extensions of introspective reasoning systems.

[11]For a specification SP and a formula $\varphi \in L(sig(SP), vars(SP))$ we write $SP \models \varphi$ if $\mathcal{A} \models \varphi$ for all $\mathcal{A} \in Sem_S(SP)$.

Corollary 4.1 *Let* $IRS^+ = (PRG^+, IM^+, META^+, REP^+)$ *be an extension of an introspective reasoning system* $IRS = (PRG, IM, META, REP)$ *with:* $dp(IM^+ \backslash IM) \subseteq REP^+(sig(META^+))$ *and for all* $f \in op(META^+) \backslash op(META)$ *with* $REP^+(f) \in dp(IM^+)$ *is* $REP_T(sort(f)) = RULETYPE$ *and*[12] $META^+ \models SOUNDRULE(f)$*. Then holds:*

(1) IRS^+ *is again an introspective reasoning system.*

(2) *if* IRS^+ *is valid then it is sound too.*

Proof. To prove **(1)** we go through the points **(a)** – **(e)** of definition 4.1.

(a) is an assumption in the corollary.

(b) for all $p \in dp(IM^+)$ it is:

$$\begin{aligned} type(p) &= type(REP^+(REP^{+^{-1}}(p))) \\ &= REP^+_T(sort(REP^{+^{-1}}(p))) \\ &= REP_T(sort(REP^{+^{-1}}(p))) \\ &= RULETYPE \end{aligned}$$

(c) trivial.

(d) because of transitivity of enlargement.

(e) trivial.

Assertion **(2)** follows from **(1)** by theorem 4.1. ∎

This corollary suggests how to specialize the algorithm for valid extensions of introspective systems to fit for sound (and valid) extensions of an introspective reasoning system $IRS := (PRG, IM, META, REP)$:

[12]Notice that $SOUNDRULE(f)$ actually exists since IRS is an introspective system.

1. build $META^+$: specify the procedures $F_1, \ldots, F_m$ to be added to PRG and the procedures $F_{m+1}, \ldots, F_n$ to be added to IM. That is, enlarge $META$ by new function symbols $f_1, \ldots, f_n$ representing the procedures and by axioms describing their effect. For $i \in \{m+1, \ldots, n\}$ it must be $REP_T(sort(f_i)) = RULETYPE$.
2. prove $META^+ \models SOUNDRULE(f_i)$ for $i \in \{m+1, \ldots, n\}$.
3. build $aPRG = aPRG_1 \cup aPRG_2$: implement $F_1, \ldots, F_m$ and $F_{m+1}, \ldots, F_n$ as abstract procedures over $sig(META)$.
4. build MAP: establish the relationship $f_i \rightarrow F_i$, $i \in \{1, \ldots, n\}$ explicitly.
5. prove the correctness of the module $M = (META^+, META, aPRG, MAP)$.
6. update IRS with $IRS^+ := (PRG^+, IM^+, META^+, REP^+)$ where $PRG^+ := PRG \cup REP_P(aPRG_1)$, $IM^+ := IM \cup REP_P(aPRG_2)$, and $REP^+ := REP \cup MAP$.

Step 2 becomes trivial by demanding $SOUNDRULE(f) \in ax(META^+)$ in step 1.

5 Related Work

We share the goal of self-reflection with a lot of work in the programming language community (e.g. [7, 50, 37, 32]). The substantial difference is that in our approach the introspection is performed by deduction instead of by computation. In all what follows only the related work in the area of automated reasoning systems is considered. Here introspection is mainly used to solve the problem of sound extensions. The relation to *tactic mechanisms*, which embody the traditional solution to this problem, is already discussed in the introduction. As argued in [25] the approach to *proof planning* in the sense of Bundy [12], which has been realized in the `Oyster/Clam` system [14], can be regarded as a specific tactic mechanism (very similar to the one proposed by Brown [11]). In particular the inefficiency problem of tactic mechanisms appears in a similar way (cf. [13]).

We now concentrate on work concerning *metatheoretical extensibility* of proving systems. The pioneers are Davis & Schwartz [17], Weyhrauch [52, 53], and Boyer & Moore [10]. Very close to the approach of Boyer & Moore is the one taken by Howe in the `Nuprl` system [28]. Another line of research at Cornell was that of Constable & Knoblock [33] in which they formalized the structure of proofs within (an extension of) `Nuprl`. Later it has been suggested to use a single proof type that refers to itself and can formally reasoned about [1]. The work of Weyhrauch has been continued and significantly extended by Giunchiglia, Traverso, and others [25, 26, 6, 24, 23]: they developed the reflective theorem prover `GETFOL` on top of a reimplementation of the `FOL` system.

Though quite different, the reflection mechanisms listed above have the common feature that the user has to provide only *one* description of a new inference rule. This description has to be *declarative* because one has to reason about it, but it has to be *procedural* as well because one wants to execute it (possibly after some compilation). In this point our approach differs from all other reflection mechanisms known to us. We strictly separate the declarative description (first-order specification $META^{+}$) from the procedural description (program PRG) — the connection has to be established by deduction (i.e. by proving module correctness) and not by fully automated compiling. At the first sight this feature may be seen to be a little bit clumsy since it takes a greater effort to extend a reasoning system: describing a new rule declaratively *and* procedurally, and proving that these descriptions are not contradictory. However, we believe that it is worth the additional expenditure, and that separating declarative and procedural description is the key for keeping reflection mechanisms manageable in large applications for the following reasons:

- On the one hand, using an expressive logic as specification language allows natural descriptions. In particular, some functions are best axiomatized using quantifiers. On the other hand, using (a part of) a common programming language makes implementing to a widely mastered job (which is not the case e.g. in the approach proposed in [17]).

- Separating declarative and procedural description allows separating implementational issues from correctness issues. A sophisticated, but very efficient implementation can be "hidden" by referring to the corresponding specification, which should be more accessible to deduction. Moreover the specification can abstract from details of the implementation: only the aspects of interest have to be formalized.

- In the course of software maintenance it may be desirable to optimize the code, i.e. to change the implementation but keep the specification. Then, as illustrated in figure 7, only the module containing the optimized procedure has to be proven correct again. Procedures using the changed procedure remain correct without any further verification effort, because their correctness has been shown with respect to the specification only.

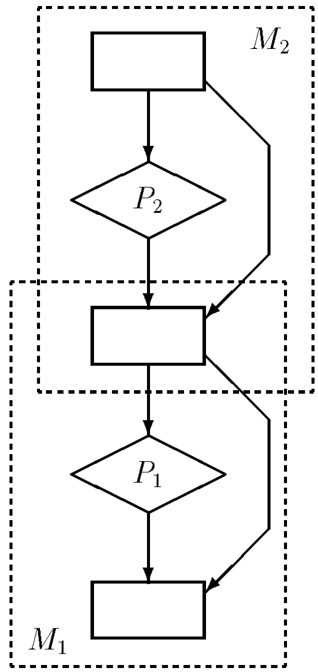

Figure 7: if P_1 changes only M_1 has to be proven correct again.

Just these arguments contribute to the fact that constructing both, procedural and declarative description, has been established and proved a good investment in the area of software engineering. Actually, our approach is quite natural because soundly extending a reasoning system amounts to construction of correct software. This allows us to directly employ the techniques and tools known for this task. For instance, the `KIV` system supports specification and modularization of large software systems as well as verification of individual program modules.

There is another point distinguishing our work. Most approaches to reflective proving make full use of quoting[13] or dequoting while applying (new) inference rules (e.g. [17, 52]) or while doing metareasoning (e.g. [10]). As pointed out in [8] this may lead to inefficiency. Our approach avoids this problem since changing of the representation is only done while performing the update operation (which is not critical with respect to efficiency).

About portability of our mechanism it can be said that we do not use special features of an underlying logic (e.g. the method presented in [10] is not directly applicable in typed logics). Moreover we do not restrict ourselves to a certain class of new inference rules we can reason about. In particular

[13]Quoting means switching from a logical object to its representation in the metatheory; dequoting is the inverse operation.

the realization in KIV allows induction over formulas and proofs; therefore also so-called admissible rules can be proven sound (which is not the case e.g. in [52, 26]).

Finally mention should be made of *logical frameworks* like, for instance, ELF [43], Isabelle [42] or λProlog [40] since they provide a meta-logic to encode syntax and inference rules of object-logics (so that proving at the object level is done by reasoning at the meta-level). This encoding of a logic in a logic constitutes an overlap between the concern of metatheoretic extensibility and the concern of logical frameworks. However, the concerns are different: logical frameworks are designed to make encoding of object-logics and proving at the object level as simple as possible. This is done by identifying some object-logic structures with corresponding framework logic structures, e.g. the representation of variable binding by using lambda abstraction. However, this *internalization* severely restricts the metareasoning facilities of logical frameworks. For this reason Basin and Constable [4] advocate to use *externalized* encodings — especially, they suggest to specify syntax and inference rules of an object-logic by means of (higher-order) abstract data-types. This paradigm was adopted e.g. in the 2OBJ system [19]. Another logical framework that is especially designed for doing metareasoning is $\mathtt{FS}_0$ [39]. In all these so-called *metalogical frameworks* only the object-logic can be extended and not the meta-logic. However, in our opinion extension of the framework logic is desirable as well since reasoning about provability is (in general) a quite complex task that calls for a quite complex (meta-)reasoning system (see the discussion in the introduction).

6 Conclusion

In this paper we have attempted to extract the features required for a system in order to introspect, and fixed them in the notion of introspective systems. Though our focus is on reasoning systems, the proposed mechanism of valid extensions is not restricted to ensuring the soundness of new inference rules. Properties of any (new) procedures are accessible to reasoning. This enables one to use formal methods in building (or at least in extending) a system for correct software development (which has often been called for by critics).

A main feature of our approach is that soundly extending a reflective prover is reduced to construction of correct program modules. (In particular we advocate separating procedural and declarative description of new procedures for reasons explained in the previous section.) Therefore advanced techniques known for correct software development can be directly employed. Our hope is that this is the key for keeping reflection mechanisms manageable in large applications. First experiences we have made with the realization of our ideas in the KIV system give some positive evidence.

A well-known example, which we have adopted from[14] [9, 49], is the tautology-checker [48]. Here the possibility to use quantors permits specification in a very natural manner. We have made use of modularization: the overall code was divided up into 5 modules — the correctness of each of them provable independently from all the others. Especially, this turned out to be very advantageous whenever the code of one module changes (which is the normal case in realistic software development), e.g. because of error correction or because of optimization. The tautology checker example in the KIV system embraces about 250 lines of code (in a PASCAL-like programming language) and about 150 lines of specification. 84 proof obligations (ensuring correctness of the modules) were generated by the system, 17 lemmas were formulated; most proofs worked by induction on formulas.

Another case-study we have carried out in the reflective version of the KIV system is the soundness proof for the "determinism-rule" [56]:

$$\frac{<\alpha>\varphi}{[\,\alpha\,]\,\varphi} \quad \text{if } \alpha \text{ is deterministic}$$

This is a rule in dynamic logic; it states that total correctness of a deterministic program α implies its partial correctness. This example is remarkable since the rule itself can be expressed schematically and applied in constant time (if only deterministic programs are considered), but (assuming a basic calculus as in [20]) a tactic (in the sense of LCF) with the same effect takes an amount of time linear in the size of α. This phenomenon also appears in the soundness proof: it works by induction on α.

Though we believe that our approach is a significant step towards reflective mechanisms which can be brought into action in a big way, the question concerning practicability cannot yet be fully answered. Especially, it has to be found out whether restrictions imposed by presently available software verification techniques prevent a rigorous use of reflection.

Acknowledgments

I am indebted to Thomas Fuchß, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel for valuable discussions during the course of this research. Especially, I would like to thank Gerhard Schellhorn who has made very helpful comments on an earlier draft of this paper.

[14] However, unlike [9, 49], we are able to add the code of new proof procedures verified with KIV to the KIV system *itself* and use them to do further proofs.

References

[1] S.F. Allen, R.L. Constable, D.J. Howe, and W.E. Aitken. The semantics of reflected proof. In *Symposium on Logic in Computer Science*. IEEE Press, 1990.

[2] G. Antoniou and V. Sperschneider. On the verification of modules. In *Conference on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 16–35. Springer Verlag, 1989.

[3] M.A. Arbib and E.G. Manes. *Algebraic Approaches to Program Semantics*. Springer Verlag, 1986.

[4] D. Basin and R. Constable. Metalogical frameworks. In [31].

[5] D. Basin, F. Giunchiglia, and M. Kaufmann, editors. *Proceedings of the Workshop on Correctness and Metatheoretic Extensibility of Automated Reasoning Systems*, held in conjunction with CADE-12, Nancy, June 1994.

[6] D. Basin, F. Guinchiglia, and P. Traverso. Automating meta-theory creation and system extension. In *AI*IA-91 (Italian Association for Artificial Intelligence), Palermo, Italy*, 1991.

[7] K.A. Bowen and R.A. Kowalski. *Amalgamating language and metalanguage in logic programming*, pages 153–172. Academic Press, 1982.

[8] R.S. Boyer. A few minor remarks. In [5].

[9] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[10] R.S. Boyer and J.S. Moore. Metafunctions: proving them correct and using them efficiently as proof procedures. In *The Correctness Problem in Computer Science*, chapter 3, pages 103–184. Academic Press, 1981.

[11] F.M. Brown. An investigation into the goals of research in automatic theorem proving as related to mathematical reasoning. *Artificial Intelligence*, 14:221–242, 1980.

[12] A. Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120. Springer Verlag, 1988.

[13] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[14] A. Bundy, F. von Harmelen, C. Horn, and A. Smaill. The Oyster–Clam system. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 647–648. Springer Verlag, 1990.

[15] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.f. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[16] D. Craigen, S. Kromodimoeljo, I. Meisels, W. Pase, and M. Saaltink. EVES: An overview. In S. Prehn and W.J. Toelenel, editors, *VDM'91, Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 389–405. Springer Verlag, 1991.

[17] M. Davis and J. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[18] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specifications. In *5th Symposium on Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*. Springer Verlag, 1976.

[19] J. Goguen, A. Stevens, H. Hilberdink, and K. Hobley. 2OBJ: a metalogical framework theorem prover based on equational logic. In *Phil. Trans. Roy. Soc. Lond.*, volume 339, pages 69–86, 1992.

[20] R. Goldblatt. *Axiomatizing the Logic of Computer Programming*, volume 130 of *Lecture Notes in Computer Science*. Springer Verlag, 1982.

[21] M. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[22] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[23] F. Guinchiglia and A. Cimatti. Introspective metatheoretic reasoning. In *Proceedings of the 4th International Workshop on Meta-Programming in Logic*, Pisa, Italy, June 1994.

[24] F. Guinchiglia, L. Serafini, and A. Simpson. Hierachical meta-logics: Intuitions, proof theory and semantics. In A. Petterossi, editor, *Proceedings of the 3rd International Workshop on Meta-Programming in Logic*, volume 649 of *Lecture Notes in Computer Science*, pages 235–249. Springer Verlag, 1992.

[25] F. Guinchiglia and P. Traverso. Plan formation and execution in an uniform architecture of declarative metatheories. In M. Bruynooghe, editor, *Proceedings of the 2nd International Workshop on Meta-Programming in Logic*, pages 306–322. Dep. of Computer Science, Leuven, Belgium, 1990.

[26] F. Guinchiglia and P. Traverso. Reflective reasoning with and between a declarative metatheory and the implementation code. In J. Mylopoulos and

R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 111–117. Morgan Kaufmann, 1991.

[27] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 117–131. Springer Verlag, 1990.

[28] D.J. Howe. Computational metatheory in Nurpl. In *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 238–257. Springer Verlag, 1988.

[29] X. Huang, M. Kerber, M. Kohlhase, D. Nesmith, and J. Richts. Guaranteeing correctness through the communication of checkable proofs. In [5].

[30] X. Huang, M. Kerber, M. Kohlhase, D. Nesmith, J. Richts, and Jörg Siekmann. Ω-MKRP: A proof development environment. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 111–120. Springer Verlag, 1994.

[31] G. Huet and G. Plotkin, editors. *Logical Environments*. Cambridge University Press, 1993.

[32] S. Jagannathan and G. Agha. A reflective model of inheritance. In *6th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer Verlag, 1992.

[33] T.B. Knoblock and R.L. Constable. Formalized metareasoning in type theory. In *Symposium on Logic in Computer Science*, pages 237–248. IEEE Press, 1986.

[34] S. Kromodimoeljo and W. Pase. Proof logging and proof checking in NEVER. In [5].

[35] P. Maes. Introspection in knowledge representation. In *Proceedings of the 7th European Conference on Artificial Intelligence*, pages 256–269, 1986.

[36] P. Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. Elsevier Science Publishers B. V., 1988.

[37] K. Malmkjaer. On some semantic issues in the reflective tower. In *5th International Conference on Mathematical Foundations in Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 229–246. Springer Verlag, 1989.

[38] S. Matthews. *Meta-Level and Reflexive Extension in Mechanical Theorem Proving*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1991.

[39] S. Matthews, A. Smaill, and D. Basin. Experience with FS_0 as a framework theory. In [31].

[40] G. Nadathur and D. Miller. An overview over λProlog. In *5th International Logic Programming Conference*, pages 810–827. MIT Press, 1988.

[41] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[42] L.C. Paulson. The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[43] F. Pfenning. ELF: a language for logic definition and verified metaprogramming. In *4th Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Press, 1989.

[44] W. Reif. Correctness of full first-order specifications. In *4th Conference on Software Engineering and Knowledge Engineering*. Capri, Italy, IEEE press, 1992.

[45] W. Reif. The KIV-system: Systematic construction of verified software. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[46] W. Reif. Verification of large software systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[47] W. Reif and A. Schönegge. Meta-level reasoning: Proving monomorphicity of specifications. In Deduktionstreffen 1994. Technical report. Technische Hochschule Darmstadt, Fachbereich Informatik, October 1994.

[48] A. Schönegge. Verifying tactics in the KIV system: the tautology checker example. Technical report, Universität Karlsruhe, Fakultät für Informatik, 1994. Forthcoming.

[49] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1:407–434, 1985.

[50] B.C. Smith. Reflection and semantics in LISP. In *11th Annual Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1983.

[51] F. van Harmelen. A classification of meta-level architectures. In *Meta-Programming in Logic Programming*, chapter 5, pages 103–122. MIT Press, 1989.

[52] R.W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[53] R.W. Weyhrauch. An example of FOL using metatheory: Formalizing reasoning systems and introducing derived inference rules. In *Proceedings of the 6th International Conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, pages 151–158. Springer Verlag, 1982.

[54] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier Science Publishers B. V., 1990.

[55] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, and M. Broy. On hierarchies of abstract data types. *Acta Informatica*, 20:1–33, 1983.

[56] D. Wolf. Algebraische Spezifikation einer Dynamischen Logik zur Verifikation von Beweistaktiken im KIV–System. Studienarbeit an der Universität Karlsruhe, Fakultät für Informatik, 1994.