Electronic Notes in Theoretical Computer Science 40 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume40.html` 14 pages

Proof-theoretic notions for software maintenance

Reinhard Kahle [1,2]

Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen,
Sand 13, D-72076 Tübingen, Germany

Institut für Informatik,
Ludwig-Maximilians-Universität München,
Oettingenstr. 67, D-80538 München, Germany

Abstract

We discuss proof-theoretic notions as a useful tool to deal with software maintenance in a formal setting.

1 Introduction

This paper is concerned with the following question:

> Let a program P and a formal system $\mathcal{F}$ be given such that we can prove a certain property $\varphi(P)$ in $\mathcal{F}$. Now we change P into a new program P'. Is there any possibility to use information of the proof of $\varphi(P)$ for a proof of $\varphi(P')$?

We give an outline how proof-theoretic notions can help to deal with this question.

One crucial notion is the notion of *use* in a proof-theoretic setting. We give suggestions for formal definitions of such a notion depending on the underlying calculus. It allows us to control explicitly the parts of a program which are necessary or sufficient for a certain property. In particular, it provides us with a form of *locality*. This locality is essential for the possibility to reuse proofs, or parts of proofs, when a program is changed.

The definitions are illustrated by some (elementary) examples which show our approach at work. The examples are taken from logic programming and as a formal framework we choose a Hilbert-style calculus. However, in principle

[1] Email: `kahle@informatik.uni-tuebingen.de`
[2] Home-page: `http://www-ls.informatik.uni-tuebingen.de/logik/kahle/`

the approach should work for other programming languages and other calculi in the same manner.

Therefore, we also give a brief overview of formal frameworks for the different programming languages. The paper is finished by a discussion of limitations, applications and related work.

2 Formal analysis of computer programs

For our purpose we consider computer programs as syntactic object, i.e. as a piece of text. This is, of course, something different than the *algorithm* [24] or the (mathematical) *function* which is implemented by a program. *Software maintenance* is analogously understood as the update of a program, i.e. the change of the program text.

In theoretical computer science there is a standard procedure for the formal analysis of programs. A programming language $\mathcal{S}$ is associated with a formal framework $\mathcal{F}$. By use of a translation $\mathcal{T}$ we can interpret programs of $\mathcal{S}$ in the formal framework $\mathcal{F}$.

Usually $\mathcal{F}$ has to contain a fixed part A which describes the computational behavior of $\mathcal{S}$ in general. Then the interpretation $\mathcal{T}(P)$ of a concrete program P of $\mathcal{S}$ is added to A and we prove — or disprove — certain properties, like termination or correctness, in $\mathsf{A} \cup \mathcal{T}(P)$.

Given a (possible infinite) set of axioms $\mathcal{A}$, on one hand we can look at all formula φ which are *derivable* from $\mathcal{A}$, i.e. the deductive closure of $\mathcal{A}$, the set $\mathcal{DC}(\mathcal{A}) := \{\varphi \mid \mathcal{A} \vdash \varphi\}$. On the other hand, we can consider the set of *logical consequences* of $\mathcal{A}$, i.e. the set of formulae which hold in all models of $\mathcal{A}$: $\mathcal{LC}(\mathcal{A}) := \{\varphi \mid \mathcal{A} \models \varphi\} := \{\varphi \mid \mathfrak{M} \models \mathcal{A} \Rightarrow \mathfrak{M} \models \varphi\}$. If we consider first order theories only, the usual completeness result states that both sets are equal: $\mathcal{DC}(\mathcal{A}) = \mathcal{LC}(\mathcal{A})$. (Here, we have sketched the standard picture only. There exists a lot of special accounts to particular programming languages using non-standard derivability notions, like non-monotonic ones, or many valued models.)

For this reason, the semantics of a program P is often identified with the theory $\mathcal{DC}(\mathsf{A} \cup \mathcal{T}(P))$ or $\mathcal{LC}(\mathsf{A} \cup \mathcal{T}(P))$ associated with it. Let us call this view the view of *programs-as-theories.*

But in this view, we give up a lot of structure (or information) which was provided in the calculation of $\mathcal{DC}(\mathsf{A} \cup \mathcal{T}(P))$ or $\mathcal{LC}(\mathsf{A} \cup \mathcal{T}(P))$. The easiest example is the fact that a formula φ can have several proofs in the axiom system $\mathsf{A} \cup \mathcal{T}(P)$, but, obviously, φ is contained only once in the set $\mathcal{DC}(\mathsf{A} \cup \mathcal{T}(P))$. It is our goal to make use of such additional structure when we study software maintenance.

It is quite obvious that there will be changes of a program which do not affect the proven properties. For instance, one can remove "irrelevant parts" or replace a part of the program by an "equivalent" one. We will use the additional structure provided by proofs to make "irrelevancy" and "equivalency" explicit. The main concept therefore is the notion of *use*. If we have

$$\mathsf{A} \cup \mathcal{T}(P) \vdash \varphi$$

we can ask which axioms of the set $\mathsf{A} \cup \mathcal{T}(P)$ have really been used in the proof of φ. As mentioned above, there may exist several proofs of φ. Therefore, we have to take the concrete proofs of φ into our consideration. But for a given proof $\mathcal{B}$ of φ the question which axioms have been used can be defined in a precise way. In the next section we will discuss some possibilities of such definitions.

With a given notion of *use*, we can deal with software maintenance. Let us give a more detailed view on the formal treatment of programs. First, we have programming language $\mathcal{S}$ and assume that there is an adequate framework $\mathcal{F}$ in which the computational behavior of $\mathcal{S}$ is axiomatized by a set of axioms A. Now consider a program P of $\mathcal{S}$. Let $C_1, \ldots, C_n$ be the clauses of P, i.e. the shortest phrases of P which can be handled separately by a formal framework. The notion of *clauses* anticipates the programming language Prolog which we will use in the examples below. However, for other programming languages it is clearly possible to divide the program text in parts which can be handled separately.

For any translation $\mathcal{T}$ which translates the program clauses C_i into formulae $\mathcal{T}(C_i)$ of $\mathcal{F}$ we have $\mathsf{A} \cup \mathcal{T}(P) = \mathsf{A} \cup \mathcal{T}(C_1) \cup \ldots \cup \mathcal{T}(C_n)$ as associated axiom system. (In a more rigorous treatment we would have to deal with *multisets* since there could be different clauses C_i and C_j which result in the same axiom $\mathcal{T}(C_i) = \mathcal{T}(C_j)$. To keep the presentation simple we do not do this. However, there are well-known formal frameworks dealing with multisets, for instance *substructural logics* [27] or *linear logic* [10]. The concepts defined in this paper can be easily worked out for these frameworks, too.)

Let $\varphi(P)$ be a property which is provable in $\mathsf{A} \cup \mathcal{T}(P)$. When we change P into P' by replacing the clause C_i by the clause C_i', we can ask whether $\varphi(P')$ still holds in $\mathsf{A} \cup \mathcal{T}(P')$. But, if $\mathcal{T}(C_i)$ was not used in a certain proof of $\varphi(P)$, it follows that we can prove $\varphi(P')$ by the very same proof in $\mathsf{A} \cup \mathcal{T}(P')$. The underlying notions for this argument will be defined precisely in the following section.

We will finish this section by addressing an interesting point in the comparison of the proof-theoretic and model-theoretic view. Assuming completeness, the proof-theoretic derivability and the model-theoretic validity are equivalent: $\mathcal{A} \models \varphi \Leftrightarrow \mathcal{A} \vdash \varphi$. However, behind this equivalence there is an important *duality*: On the model-theoretic side we prove a *universal statement*: "For all models $\mathfrak{M}$ it holds ..." while we have an *existential statement* on the proof-theoretic side: "There is a proof $\mathcal{B}$ of ...". On the other hand, for the rejection of a property we have an existential statement in the model-theoretic framework: "There is a (counter-) model $\mathfrak{M}$ such that ..." while we have on the proof-theoretic side a negated existential statement which is equivalent to a an universal one: "There is no proof $\mathcal{B}$ of ...".

In general, it is often easier to deal with a single object than with a class of objects. Here, that means, for a (positive) property φ it is easier to deal with a witness proof $\mathcal{B}$ of φ than with a class of models. If you look at the example above, the proof which does not use $\mathcal{T}(C_i)$ is an object which can be immediately transfered in the context of the program P'. However, the relation between the models of $\mathsf{A} \cup \mathcal{T}(P)$ and those of $\mathsf{A} \cup \mathcal{T}(P')$ could be arbitrarily complicated. (Of course, this does not mean that it has to be easier to *find* a proof than to determine a class of models. Also the proof, as an object, could be much more complex than the description of the models. But with a *given* proof we can often deal more easily, in particular, with respect to the question of *used* formulae.)

In contrast, if we want to disprove a property, it is, in general, easier to deal with counter-models than to prove an unprovability statement. Of course, if we have syntactical completeness, i.e. $\mathsf{A} \not\vdash \varphi$ implies $\mathsf{A} \vdash \neg\varphi$, the proof-theoretic account has again some advantages. However, in general, we cannot expect syntactical completeness. Moreover, if we use it, it corrupts our notion of *use* of an axiom. This problem is addressed below in the section about limitations.

3 Proof-theoretic notions

In the general situation we have a given axiom system A containing a particular axiom α and we know that φ is provable from A: $\mathsf{A} \vdash \varphi$. Now we change A to A' by replacing α by α'. The question is whether φ is derivable from A', too and, if so, whether we can use some information from the proof in A or whether we have to prove it from scratch. For the second part we can ask the following three more detailed questions:

(i) Was α *used* in a given proof $\mathcal{B}$ of φ in A?

(ii) Was α *necessary* to prove φ in A?

(iii) Is α provable in A'?

If the answer to the third question is positive, we can obviously transform the proof of φ in A into a proof of φ in A' by replacing the axiom α — if it occurs in the proof — by its proof in A.

For the first question we have to give a formal explanation of notion of *use*. This will be discussed in the following. However, assuming that we have a notion of *use* we can already give a precise notion of *necessary*:

Definition 3.1 Let an axiom system A be given. We call an axiom α of A *necessary* for φ, if

(i) There is a proof of φ in A: $\mathsf{A} \vdash \varphi$.

(ii) Every proof of φ in A *uses* α.

The first condition is needed to avoid pathological cases. In fact, (here) we are not interested in necessity for unprovable formulae. But the second condition should capture our informal intuition of necessity in the case of provable formulae.

For the definition of a notion of *use* we give three suggestions depending on the underlying calculus.

Definition 3.2 Let $\mathcal{B}$ be a proof in a *Hilbert-style* calculus. Then we say that

$$\alpha \text{ is used in the proof } \mathcal{B}$$

if there is a single line

$$\vdash \alpha$$

in $\mathcal{B}$.

Definition 3.3 Let $\mathcal{B}$ be a proof in a *natural deduction* calculus. Then we say that

$$\alpha \text{ is used in the proof } \mathcal{B}$$

if α is an open leaf of $\mathcal{B}$.

We could also discuss the more liberal notion where α could be a closed leaf, too. Since we will restrict ourselves to axioms α in the following, the given definition is sufficient for our purpose.

Definition 3.4 Let $\mathcal{B}$ be a proof in a *sequent* calculus. Then we say that

$$\alpha \text{ is used in the proof } \mathcal{B}$$

if α is a main formula of a rule applied in $\mathcal{B}$.

It is easy to observe that these three notions are essentially equivalent, if we restrict ourselves to *axioms* α in a Hilbert calculus. This fact would be quite complicated to state as a formal theorem. However, a given proof which uses the axiom α, can be transformed in a proof of the “same” end-formula in one of the other calculi which uses α, too.

In the following we will restrict ourselves to the case of Hilbert calculi, cf. e.g. [29].

Here, we do not give a (philosphical) discussion of the adequacy of these definitions but we appeal to the intuitiveness. In the following section we give some examples how these notions can applied to answer the questions (1)–(3).

4 Examples

Our approach is very general and should be applicable for nearly all programming languages. All we need is for a given programming language $\mathcal{S}$ a formal framework $\mathcal{F}$ and a translation $\mathcal{T}$ which allows one to translate programs of $\mathcal{S}$ into axioms of $\mathcal{F}$.

Such frameworks exist for essentially all higher computer languages. There are even different ones for a particular programming language which compete which each other with respect to complexity, expressivity and also practice handling. They can even differ in their intention, focusing on the *denotational* or *operational* semantics. But these aspects do not affect our approach. It works for theories axiomatizing the denotational semantics in the same way as for the operational semantics. However, often the operational semantics is more closely related to a proof-theoretic view while the denotational one is related to a model-theoretic view, cf. e.g. [25]. At the end of this section we give a brief discussion of formal frameworks given in the literature.

For the concrete examples, a programming language with a logical background is easier to handle. For this reason, we work with PROLOG. Moreover, since we would like to give an illustration of our proof-theoretic notions only, we restrict ourselves to the (almost trivial) case of *propositional* PROLOG *programs*. But this case is sufficient to give a picture of the defined notion and to show the essential features without need of a complex background theory.

The propositional PROLOG programs are built in the well-known way. We have formal symbols $\mathtt{a}, \mathtt{b}, \ldots,$ for *propositional variables*. If we use $a, b, \ldots$ as metavariables for propositional variables, a propositional PROLOG program consists of a list of clauses

$$a \;\; \mathtt{:-} \;\; b_1, \ldots, b_n.$$

where $n \in \mathbb{N}$. In the case $n = 0$ we say that a is a *fact*, otherwise the clauses are called *rules*.

As formal framework $\mathcal{F}$ we choose a standard Hilbert calculus for propositional logic, in particular, we have a set A of axioms which allows to derive all tautologies.

Let us assume that we have an enumeration of the propositional variables in $\mathcal{F}$ such that each formal symbol a of our programming language is associated uniquely with one propositional variable. Therefore, we can identify both kinds of variables. Now, $\mathcal{T}$ is a function which translates a rule

$$a \;\; \mathtt{:-} \;\; b_1, \ldots, b_n$$

into the axiom

$$b_1 \wedge \ldots \wedge b_n \rightarrow a.$$

A fact a is interpreted by the axiom a.

Example 4.1 Let P_1 be the program consisting of the following three clauses:

```
b :- a.
a.
c.
```

The set of logical consequences of P_1 is the deductive closure starting from `a`, `b`, `c`: $\mathcal{LC}(P_1) = \mathcal{DC}(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$.

In Prolog we could ask for the goal `b`:

```
?- b.
```

We get the expected answer `Yes`, since $\mathtt{b} \in \mathcal{LC}(P_1)$. On the proof-theoretic side we have $\mathcal{T}(P_1) = \{\mathtt{a} \to \mathtt{b}, \mathtt{a}, \mathtt{c}\}$ and we get the following proof of `b`:

$$\frac{\vdash \mathtt{a} \qquad \vdash \mathtt{a} \to \mathtt{b}}{\vdash \mathtt{b}}$$

If we choose definition 3.2 for the notion of *use*, it follows obviously that `a` was used in this proof, but not `c`. It is even trivial to realize that the given proof is essentially the only one of `b`. (Of course, in a Hilbert-style calculus we get infinitely many other proofs by weakening this proof by adding additional lines containing derivable formulae and their derivations. However, there is no proof which does not contain — *use* — the two given lines). Thus, `a` is even *necessary* for `b` in P_1.

This information will be used when we consider changes of P_1.

Example 4.2 Let P_2 be the program resulting from P_1 by retracting `c`:

```
b :- a.
a.
```

Obviously `b` is an element of $\mathcal{LC}(P_2)$. If we look back to the proof of $\mathtt{b} \in \mathcal{LC}(P_1)$ one realizes that `c` was *not used* in this proof. Therefore, it can be transferred *literally* to the case of P_2. The objective of this example is the fact that, when P_2 arises from P_1 by retracting `c`, we need no (new) calculation of $\mathcal{LC}(P_2)$ to conclude $\mathtt{b} \in \mathcal{LC}(P_2)$.

The retraction of `a`, however, will change the derivability of `b`:

Example 4.3 Let P_3 be the program resulting from P_1 by retracting `a`:

```
b :- a.
c.
```

`b` is no longer in $\mathcal{LC}(P_3)$, but this follows already from the fact that `a` was *necessary* for `b`. Of course, this kind of argument works only, as long as we retract something. When we add new clauses, there could be a new possibility to derive `b`.

Now let us consider the following program:

Example 4.4 Let P_4 be the program consisting of the following four clauses:

```
b :- a.
a.
```

```
c.
b :- c.
```

If we compare this program with P_1 it turns out that the set of logic consequences is the same: $\mathcal{LC}(P_4) = \mathcal{LC}(P_1) = \mathcal{DC}(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. But the associated axiom system is different: $\mathcal{T}(P_4) = \{\mathtt{a} \rightarrow \mathtt{b}, \mathtt{a}, \mathtt{c}, \mathtt{c} \rightarrow \mathtt{b}\}$. It is exactly this difference which is crucial for the analysis of software maintenance. As for P_1 we can ask whether `b` follows from P_4, which is obviously the case. But on the proof-theoretic side this time we have two (essentially different) proofs:

$$\frac{\begin{array}{l}\vdash \mathtt{a}\\[1ex] \vdash \mathtt{a} \rightarrow \mathtt{b}\end{array}}{\vdash \mathtt{b}} \qquad\qquad \frac{\begin{array}{l}\vdash \mathtt{c}\\[1ex] \vdash \mathtt{c} \rightarrow \mathtt{b}\end{array}}{\vdash \mathtt{b}}$$

Again we can look at the consequence of the retraction of `a`:

Example 4.5 Let P_5 be the program resulting from P_4 by retracting `a`:

```
b :- a.
c.
b :- c.
```

Now, `b` still follows. But this fact can, by no means, be deduced from $\mathcal{LC}(P_4)$ alone, since this set is equal to $\mathcal{LC}(P_1)$. And for P_1 the retraction of `a` affects the derivability of `b`. But looking at $\mathcal{T}(P_4)$ and, in particular, to the proofs of `b` we get that `b` is derivable. The derivability already follows from the fact that `a` was not necessary for `b`, since there is a proof of `b` which does not use `a`.

In a last example let us change P_1 by replacing `a` by `a :- c`:

Example 4.6 Let P_6 be the program resulting from P_1 by the replacement of `a` by `a :- c`:

```
b :- a.
a :- c.
c.
```

In this case, the knowledge that `a` was necessary for `b` cannot be used directly. In particular, not in the way that the retraction of `a` disables the derivation of `b`. In fact, the addition of `a :- c` saves the derivability of `b`. To see this, we do not need to calculate $\mathcal{LC}(P_6)$ as a whole. It is enough to show that the necessary axiom `a` which was retracted can be derived in the new context. This follows from the derivation:

$$\frac{\begin{array}{l}\vdash \mathtt{c}\\[1ex] \vdash \mathtt{c} \rightarrow \mathtt{a}\end{array}}{\vdash \mathtt{a}}$$

Thus, example 4.6 serves as an example for a positive answer to a question of the third kind which we have asked at the beginning of section 3.

5 Formal frameworks

We will discuss briefly formal frameworks for the different programming languages. In all these frameworks we can directly work with our notion of *use* and *necessity.*

As a general reference we recommend the second volume of the *Handbook of Theoretical Computer Science* [36]. As is generally known, the pioneer formal approach to programming language was given by Hoare [15], cf. Cousot [6] which contains an impressive list of more than 400 references.

For imperative languages frameworks of *dynamic logic* became popular, because it allows us to express the change of variables in a more natural way [13,20]. Of course, the *dynamic* of this logic deals with the program flow not with changes of a program. But, the main problem with respect to the questions discussed here is the lack of *locality.* The change of a program will in general result in a complete new proof. At least, the study of the consequences of the change for a given proof in dynamic logic would require a much more involved analysis of the proof objects.

From a logical point of view, *declarative* programming languages are of special interest. There exist several special logical formalisms to deal with such programming languages. For functional programming languages, like Scheme, LISP, or ML which are based on the λ-calculus [3], we refer as an example to the framework of *explicit mathematics* introduced by Feferman [8,9,14,32,33]. Studer has even used this framework to deal with the functional core of the programming language Java [35].

In *logic programming* which is based on *resolution*, cf. Apt [2], there are interesting proof-theoretic approaches by Hallnäs and Schroeder-Heister [12], Jäger and Stärk [17], or Elbl [7].

The *functional* core of arbitrary programming languages is discussed from a *type theoretic* point of view by Mitchell [23].

At the moment, the programming language Java is extremely popular. The development of formal systems to deal with it is still ongoing. As a first reference we suggest the collection edited by Alves-Foss [1]. A recent approach worked out in all details can be found in the book of Stärk, Schmid, and Börger [34].

6 Limitations

Our main task was to show how proof-theoretic notions can help to deal with questions arising from software maintenance. In this section we discuss some limitations of our approach. The main one is the requirement of *locality.* If the derivability of a formula depends on the system as a whole, our approach does not really help.

This is the case, if we think of *non monotonic* systems. In such a system the consequences of a change of a program is much harder to control. In logic programming we face this problem if we work with *closed world assumption* or *negation as failure.*

More generally, every form of *metareasoning* will affect our approach: the use of a formula is not only definable on the basis of a given proof, but it could be "used" in a *meta argument.* A (trivial) consequence of this observation is that we are not allowed to deal with *derived rules* in the derivations considered. We would have to store all formulae which are *used* in the derivation of the derived rule.

Moreover, as mentioned above, the use of *syntactical completeness* to derive a negated property from the underivability of the positive one is also a very problematic argument, since it remains unclear which formulae are *used* in the (meta-)proof of the underivability.

7 Applications

The defined notions are very general and they should be applicable in arbitrary contexts as long as we have an appropriate formal framework. However, for many computer programs the calculation and the bookkeeping of used formulae would probably be too space and time consuming. Nevertheless, beside the conceptual clarification given by our approach, there are several areas where it should be applicable directly.

First, we have to mention *databases* and *database update* [19]. In database theory, proof-theoretic accounts are well established. In particular, *deductive databases* could be seen as an implementation of the proof-theoretic view of databases. To control the consequences of an update, our defined notion of *use* is obviously relevant.

Another area where our notions are useful is *object-oriented programming.* In its pure form it is based on the idea that an object is a black-box for the programmer who is using it. That means, changes of the implementations should not affect the bigger program which is using the object. In fact, an object should be determined by its *specification* only. In practice, a programmer has no real chance to check whether and how the specification is fulfilled. In particular, he cannot check whether changes in the implementation of the object will really not affect the bigger program. Again, our approach can help to control such changes.

As a last, but maybe most important topic we mention *proof carrying code.* This very new field arises from problems caused by internet programming. If a browser is allowed to download programs from an other server, it has to ensure that this program can not do nasty things on the local computer. For instance, the use of memory has to be restricted to a defined area which the program is not allowed to leave. For example, the so-called *byte code verifier* should do this for Java applets. It is well-known that, in general, *proof search* is much more “expensive” than *proof checking.* Therefore, the idea is to send the proof of the correctness of a Java program together with the program through the net. However, the whole proof could be already too big. So it is a question of balance which parts of the proof should be packed in the program in order to get an optimal relation between the size of the transferred code and the time for the local verification. To study these kinds of questions the analysis of *used* and *necessary* parts of proofs is clearly highly relevant.

8 Related work

There is a lot of work related to our approach, both from the conceptual as well as from the practical point of view.

The splitting of the axioms describing a program in a *fixed* part for the programming language and a so-to-say *variable* part for a concrete program can be model-theoretically handled by use of *modal logic.* There, the fixed axioms would be modeled by *necessary axioms.* But with exception of database theory, cf. [21,22], we are not aware of a modal approach to programming languages which uses this framework for software maintenance or the other possible applications mentioned above.

The view of *programs-as-deductive systems* introduced by Hallnäs and Schroeder-Heister [12,26], and also adopted by Jäger and Stärk [16,17,30,31], for the analysis of logic programming starts with a proof-theoretic perspective, too. Mainly, it emphasizes the usefulness and importance of *rules* in the modeling of extensions of logic programming.

As a somehow complementary approach we can consider the approach of *proofs as programs.* Here, we extract programs from proofs of the desired specification. Thus, the verification of an extracted program comes for free. As an example for an implementation of this approach we refer to Schwichtenberg's system Minlog [28,4,5]. There we have a strong correspondence between the *used* proof strategies and the resulting programs. In particular, a change of the proof can result in a different program and the extraction procedure gives some kind of control. One key example for this is the use of an induction on the proof side which results in a recursion on the algorithmic side.

Within this framework the idea of *pruning* realizes some aspects of our aims [11]. Let us assume we have extracted a program from a given proof which uses case distinctions. New information could result in a reduction of the possible cases. By using this information systematically, one can *prune* the distinctions and end up with a better, i.e. more efficient, program.

Finally, there is already a discussion of the proof-theoretic notions, introduced here, in a logical and in a linguistic context [18].

References

[1] Alves-Foss, J., editor, “Formal Syntax and Semanatics of Java,” Lecture Notes in Computer Science **1523**, Springer, 1999.

[2] Apt, K., *Logic programming*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 493–574.

[3] Barendregt, H., *Functional programming and lambda calculus*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 323–363.

[4] Benl, H., U. Berger, H. Schwichtenberg, M. Seisenberger and W. Zuber, *Proof theory at work: Program development in the Minlog system*, in: W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume II*, Kluwer, 1998 pp. 41–71.

[5] Berger, U., H. Schwichtenberg and M. Seisenberger, *The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction*, Journal of Automated Reasoning **26** (2001), pp. 205–221.

[6] Cousot, P., *Methods and logics for proving programs*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 841–993.

[7] Elbl, B., *A declarative semantics for depth-first logic programs*, Journal of Logic Programming **41** (1999), pp. 27–66.

[8] Feferman, S., *Logics for termination and correctness of functional programs*, in: Y. Moschovakis, editor, *Logic from Computer Sciences* (1991), pp. 95–127.

[9] Feferman, S., *Logics for termination and correctness of functional programs II: Logics of strength PRA*, in: P. Aczel, H. Simmons and S. S. Wainer, editors, *Proof Theory*, Cambridge University Press, 1992 pp. 195–225.

[10] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[11] Goad, C., *Computational uses of the manipulation of formal proofs*, Technical report, Stanford Department of Computer Science (1980), report No. STAN-CS-80-819.

[12] Hallnäs, L. and P. Schroeder-Heister, *A proof-theoretic approach to logic programming. I. Clauses as rules*, Journal of Logic and Computation **1** (1990), pp. 261–283.

[13] Harel, D., *Dynamic logic*, in: D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic. Volume II*, Kluwer, 1984 pp. 497–604.

[14] Hayashi, S. and H. Nakano, “PX — A computational logic,” MIT Press, Cambridge Mass., 1988.

[15] Hoare, C., *An axiomatic basis for computer programming*, Communications ACM **12** (1969), pp. 576–583.

[16] Jäger, G., *A deductive approach to logic programming*, in: H. Schwichtenberg, editor, *Proof and Computation*, Springer, 1994 pp. 133–172.

[17] Jäger, G. and R. Stärk, *A proof-theoretic framework for logic programming*, in: S. Buss, editor, *Handbook of Proof Theory*, Elsevier, 1998 pp. 639–682.

[18] Kahle, R., *A proof-theoretic view of intensionality*, in: P. Dekker, editor, *Proceedings of the 12th Amsterdam Colloquium*, Amsterdam University, 1999 pp. 163–168.

[19] Kanellakis, P., *Elements of relational database theory*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 1073–1156.

[20] Kozen, D. and J. Tiuryn, *Logics of programs*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 789–840.

[21] Lipski, W., *On semantic issues connected with incomplete information databases*, ACM Transactions on Database Systems **4** (1979), pp. 262–296.

[22] Lipski, W., *On databases with incomplete information*, J. ACM **28** (1981), pp. 41–70.

[23] Mitchell, J., *Type systems for programming languages*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 365–458.

[24] Moschovakis, Y., *What is an algorithm?*, in: B. Engquist and W. Schmid, editors, *Mathematics unlimited — 2001 and beyond*, Springer, 2001 pp. 929–936.

[25] Mosses, P., *Denotational semantics*, in: J. van Leeuwen, editor, *Handbook of TCS. Volume B*, Elsevier and MIT Press, 1990 pp. 575–631.

[26] Schroeder-Heister, P., *Hypothetical reasoning and definitional reflection in logic programming*, in: P. Schroeder-Heister, editor, *Extensions of Logic Programming*, Lecture Notes in Artifical Intelligence **475**, Springer, 1991 pp. 327–339.

[27] Schroeder-Heister, P. and K. Došen, editors, "Substructural Logics," Oxford, 1993.

[28] Schwichtenberg, H., *Proofs as programs*, in: P. Aczel, H. Simmons and S. Wainer, editors, *Proof Theory*, Cambridge University Press, 1992 pp. 79–113.

[29] Shoenfield, J., "Mathematical Logic," Addison-Wesley, Reading, MA, 1967.

[30] Stärk, R., *A complete axiomatization of the three-valued completion of logic programming*, Journal of Logic and Computation **1** (1991), pp. 811–834.

[31] Stärk, R., *Cut-property and negation as failure*, International Journal of Foundations of Computer Science **5** (1994), pp. 129–164.

[32] Stärk, R., *Call-by-value, call-by-name and the logic of values*, in: D. van Dalen and M. Bezem, editors, *Computer Science Logic '96*, Lecture Notes in Computer Science **1258**, Springer, 1997 pp. 431–445.

[33] Stärk, R., *Why the constant ‘undefined’? Logics of partial terms for strict and non-strict functional programming languages*, Journal of Functional Programming **8** (1998), pp. 97–129.

[34] Stärk, R., J. Schmid and E. Börger, “Java and the Java Virtual Machine — Definition, Verification, Validation,” Springer, 2001, URL: `http://www.inf.ethz.ch/~jbook/`.

[35] Studer, T., *Constructive foundations for featherweight Java*, in: R. Kahle, R. Stärk and P. Schroeder-Heister, editors, *Proof Theory in Computer Science*, Lecture Notes in Computer Science **2183**, Springer, 2001 pp. 202–283.

[36] van Leeuwen, J., editor, “Handbook of TCS. Volume B: Formal Models and Semantics,” Elsevier and MIT Press, 1990.