Annotation Issues in Isabelle

Sara Kalvala sk@cl.cam.ac.uk
Department of Computer Science, Warwick University
Coventry, CV4 7AL, UK

August 13, 1995

Abstract

Mechanized proof tools have not been eagerly adopted by the potential user community. This lack of penetration is sometimes attributed to the fact that the employment of theorem provers for formal verification is often thought to be a difficult and involved task. As a consequence, many researchers are addressing the issue of usability, and are working towards straightforward and effective ways of interacting with proof tools.

One way I propose to achieve better applicability of provers is to develop a method to incorporate useful domain knowledge into the proof process, facilitating the use of *particular* pieces of ad-hoc information. In this talk I present a general method to exploit such additional, informal information in the Isabelle proof cycle, and discuss some technical issues in the programming involved.

1 Introduction

Theorem-proving tools are set out as reasoning tools for a variety of domains by representing properties to be reasoned about as proof goals, exploiting the intuition that the process of proving these goals allows one to reason about the correctness of the original property. The idea is that representing a domain by way of symbols and manipulating the symbols without reference to their meaning can provide an impartial, rigorous method of describing and ascertaining properties of the domain.

While this approach underlies the many advances in the use of formal methods, it has left a large methodological gap between theorem proving (an exercise in manipulation of form, of syntax) and the domain of application itself (the *meaning* of properties being modelled). This separation of form and meaning does not take into account the fact that the use of provers involves more information than what can be properly described in the (often thought of as constricted) logical notation, such as suppositions and explanations of a more informal, ad-hoc nature. While such information is not integral to the proof on a purely logical level, it is instrumental in mapping formulae and proofs to models in the domain of discourse, which in turn can help proving a goal and understanding the proof.

Missing from the current technology for mechanized proofs is a better facility for incorporating this kind of ad-hoc information. In most cases, proof systems in use provide the logical kernel and leave it up to the user to structure the proof. Other systems incorporate domain heuristics available *a priori* and integrate them into proof search without much direction as to characteristics of the *particular* problem being tackled.

Even when the formal proof is complete, another aspect of the proof process is to convince others of the veracity of this proof. For a proof to be most insightful, it must have some intuitive validity. Not only must a theorem be obtained, but the proof process must be informative. The proof itself must be amenable to examination. One must understand *why* a conjecture is correct, and not only be told that it is. Without such an explanation, the formality imposed in automated reasoning systems can actually become a burden, by cluttering a proof or a specification with many (to a human reviewer) irrelevant details. Ultimately, the mechanical aspects of the proof may not be as important as the intuition behind the proof structure when the goal is to understand the correctness of the result.

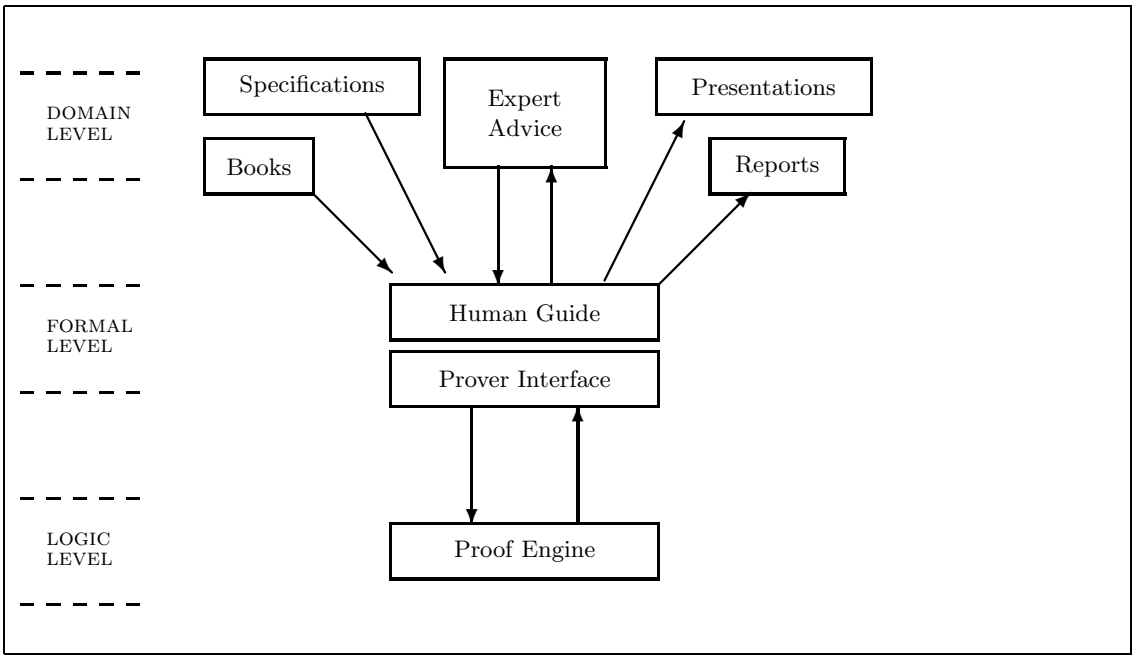

Figure 1: Layered view of proofs

The informal semantics of the domain provides the glue that links the processes of specification, proof, and journal description of the proof. In current methodologies, it is up to the user to provide this glue, with very limited mechanical support. The consequence of this lack of support is that aspects of symbolic manipulation and the necessity to confront minute details can result in cumbersome proof processes where the actual meaning of what one wants to define and prove often gets lost.

I refer to the many aspects of problem definitions that cannot be easily expressed in the logical notation as *extra-formal* information. Extra-formal information can be thought of as partway between vague, unstructured informal knowledge and that kind of information present in formal specifications. This information is usually transmitted via documentation or through consultation, and is not traditionally supported mechanically.

The role of extra-formal information can be understood by regarding any complex mechanized proof as composed of layers, as illustrated in Figure 1. In the established approach, the mechanized proof as such is performed in the formal level through the interaction between the proof system and the human guiding the system. In an even lower level, this proof is built upon the underlying implementation of the logic, details of which are usually hidden from the human guide but necessary for the correct working of the theorem prover. However, there is also the *context* in which the particular proof exists in: explanations as to what are the assumptions, what will be achieved by proving any particular goal, how it can be explained intuitively, and so on. The processes that occur at this level are the ones that involve the extra-formal information. This type of information is usually relegated to commentary which is attached to the specification of the problem or a journal-style description of the proof, where non-essential technical details can also be hidden.

This last layer has not traditionally been identified, and has usually been assumed to be the province of the "experts" who apply the proof system in a certain domain. These experts must, in fact, be conversant in both the domain being reasoned about as well as the intricacies of the proof system used. The use of formal proof systems usually involves concurrent activity at two levels: one manipulates proof goals using logical steps, but one is often interested not in the formal proof *per se* but in what it means, what insights it provides. A related concept applies to anyone attempting to understand someone else's proof: while the fact that the proof has been performed

in some respectable theorem prover provides some assurance of correctness, they still would like to understand the structure of the proof, in the form of an informal justification.

The method of achieving this triple-layered view of proofs is the theme of this paper. In the method being put forward, the integration between theorem proving and domain reasoning is facilitated by recognising explicitly the role of extra-logical information, and providing mechanisms for manipulating this information through new, semantic rules linked to the proof rules that make up the formal (syntax-based) proof system. The concrete way to achieved this linkage is through *annotations*, integral parts of the data structures underlying theorem provers which store extra-formal information, and which therefore allow the information to be mechanically manipulated. Annotations provide the kind of support necessary to develop formal proofs that confirm the more intuitive notions underlying the verification process.

2 Related work

The usefulness of extra-formal information for both proof development and proof explanation has been put forward in Section 1. These ideas are not radically new: other researchers have also realized the importance of adding informal, ad-hoc information to the proof process.

Wegbreit discusses many of the issues which independently guided the current work, in the context of program verification [Weg76], and motivates a need for the "explicit link between the specifications and executable instructions" needed for correct instantiations and guiding of proofs. Wegbreit supplies this link in the form of *justifications*, intermixed with code and correctness specifications in the program. These justifications provide information on how the correctness proof can be obtained. Justifications document the connection between instructions and specifications, and are used to suggest instantiations and cases that structure the proof. For example, when a final assertion uses a quantified variable e, justifications provide information on how e can be constructed, either from the variables in the initial assertion or from the values obtained in the executable statements.

Another example of the use of *annotations* in program verification is provided by Dershowitz [Der83]. Dershowitz proposes annotating programs with **purpose** and **suggest** statements. The **purpose** statement describes the intent of the code following it; **suggest** statements are weaker than assertions, in the sense that they may or may not be correct but can be fine-tuned into an assertion. The assertions being suggested are *desired* invariants; they are compared with the assertions generated by actually analysing the program.

The original Boyer-Moore theorem prover has been modified to allow better user guidance. Before, the user could control the proof process only in an indirect way—by programming its rule base in a certain order. To enhance user guidance even more, the Boyer-Moore system has been augmented to include an explicit *hint* facility [BM88]. Possible hints include specifying which definitions to expand, whether induction should be used or not, etc. Having alternative induction schemes makes the theorem prover potentially more powerful when it is considering what induction should be used on a given conjecture. The hint facility thus makes it easier to lead the theorem prover towards a proof.

Each of these works (introducing the terms justifications, annotations, and hints) have addressed the same issue, namely the addition of ad-hoc information into formal descriptions and uniform procedures. However, none of them have described a truly general theory for using extra-formal information. The approach described in this paper is intended to subsume these other methods.

3 Generic annotation structures

One common characteristic of the previous research described in the last section is that annotations were being used as a useful trick for solving particular problems, rather than an interesting theoretical concept which could be seen as a foundation for applied theorem proving. The research described in this paper is aimed at developing a common framework within which the more specialised approaches to the use of extra-formal information can be embedded.

First it is necessary to identify a desiderata for such a scheme, and to develop a general methodology, ideally as independent as possible of any particular implementation. So, taking into account the experiments described in the last chapter, one may identify several characteristics of an extension designed for improving interaction:

Generality Different application-specific provers are developed with the claim that they ease the proof process in a particular domain. Annotations can be conjectured to alleviate this necessity. Thus, one solution to having a myriad of theorem provers is to develop annotation mechanisms in general-purpose provers and make them more geared to specific applications in a way similar to application-specific provers. The first step in developing a theory of annotations is to make the idea generic, and examine not how particular annotations can be supported but how arbitrary kinds of annotations can be integrated into proof systems.

Programability As a consequence of the generality, one would want such a system to be adapted to ones own needs in a straightforward way, without the necessity of re-implementing the whole prover, or understanding internal details. For this to happen, one can think of several steps in the use of such a system: for each application domain, one would define once and for all a taxonomy of annotations, and their mechanisms of use. Then, for any particular proof endeavour in this domain, one would already have the infra-structure to write logic formulas and their associated annotations.

Uniformity A clean and tidy extension of theorem provers should identify the points in the proof cycle in which extra-formal information is made available and in which it is used. These should be the parts of the prover that must be affected for any taxonomy of annotations being built. There should be a few 'hooks' in well defined points of the proof methodology, such as the writing of a definition, the choice of next proof step, etc. at which the designer of the proof environment can specify the role of the particular family of annotations being implemented.

Outreach The proof development process has several phases: problem definition, proof development, re-proof, and documentation. The methodology should allow a seamless transfer of extra-formal information throughout these phases. For example, a comment can be gleaned in a CAD tool, used as a hint in the proof of correctness, and then available in the final documentation of the product. Information must flow in several directions in a proof development: from specification to proof effort, from proof back to specification (in case of proof failure), from proof development and specification to documentation, etc.

User-friendliness The support for annotations should be built in such a way that the process of entering annotations and using them should be natural rather than a new complication for the person trying to develop a proof. That would make it possible for slots for annotations to be filled in at the same time as the problem is being described, or in the proof development process. There can be several different front-ends for these phases, and they should be made to support the particular taxonomy of annotations.

Transparency The presence of a taxonomy of annotations should be optional, as should be the use of annotations even when a taxonomy is built for the application. Being a methodology to *extend* a theorem prover, the original prover should still be seen as a complete system on its own. Also, one should be able to switch the use of annotations on-and-off. For example, one may want to look at all the annotations making part of a specification when attempting

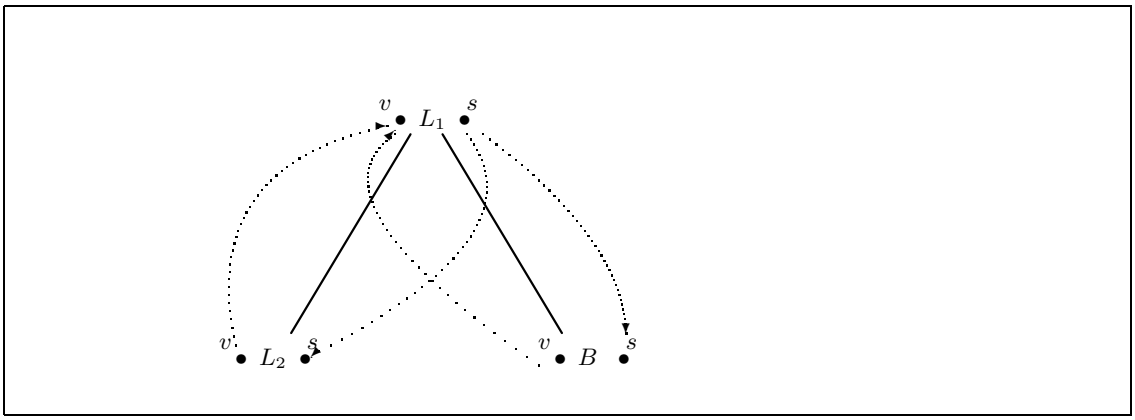

Figure 2: Example production with attribute dependencies

a proof the first time, but not when re-running the proof. Annotations should not clutter up proof scripts when they are not needed.

A well-known paradigm that support these characteristics is Knuth's specification of *attribute trees*, an extension of syntax trees for programming languages [Knu68]. The syntax of a language is represented by a usual context free grammar, to which are added *attributes* which associate values with non-terminals. Syntax rules are then expanded with semantic rules for the computation of attributes. The attributes can be computed in either of the two directions of following a parent-child link, as illustrated in Figure 2. Attributes and rules for computing them provide a method for embedding semantic notions into formal proofs.

My plan is to carry this idea over to proof systems. One often talks about proof trees, so there is a natural counterpart to the world of grammars and trees; the tree-building rules in syntax trees are the productions, while in the world of proofs they are the inference rules. There is an important distinction in the two domains as to how the tree is built: while syntax trees were studied particularly in the context of automatic parsing and compiling, proof trees are developed interactively, and the process of developing the proof often makes use of much expertise. As a consequence, the kind of information which could be added to proof trees differs significantly from the semantics which gets associated with syntax trees. However, there are many insights into proof annotations that can be obtained from Knuth's work. One is that semantic information can be either *synthesised* or *inherited* by a node in the tree. In terms of proof trees, this means that at any point in the proof process one can make use of information pertaining to the subgoal one is examining, but once a proof step is executed one may also provide feedback regarding the overall proof effort and even the original specification of the problem and definitions used.

Another paradigm to be considered is provided by the Curry-Howard isomorphism [Dum77]. Propositions in the logic can be extended with *proof terms*, which represent a procedure to obtain the proof. Proof terms are often described as "annotations", but they represent a very specific, restricted instantiation of the kind of annotations being proposed. Proof terms cannot be entered at random by the human guiding the proof, but they have to be derived formally through the proof process. They are associated with the whole body of the theorem, rather than being local to particular points of the term structure representing theorems. Nevertheless, they provide insight into how annotations can be transmitted through a proof tree created by inference rules.

4 Implementing annotations

In the previous section I discussed what kind of extension to theorem provers I would consider useful as a way of providing support for extra-formal information in a general, clean way. These ideas, to be shown feasible, must be translated into practice. In this section I briefly summarise my results in augmenting the HOL theorem prover with annotations, and then discuss progress in implementing them in the Isabelle prover.

4.1 Annotations in HOL

HOL is a general-purpose proof system in the LCF tradition [GM93]. The mapping of the idea of programmable generic annotations to the HOL system has been direct and not very complicated. The resulting proof environment has several of the desired properties from the desiderata. The system is described in detail elsewhere [Kal94]; here it will be described briefly, and factors which have made the implementation work will be pointed out.

Comments inserted in the specification (definitions and axioms that define the problem) are brought to bear in the proof development phase, and comments typed in during the proof process make part of the display of the final proof. For example, Figure 3 shows how an annotation suggesting a case analysis on a particular variable is mapped into the execution of the appropriate proof tactic when that annotation is active.

The annotations themselves are entered and visualised by extending the parsing and pretty-printing facilities available in HOL. In the example of Figure 3, annotations are printed by being enclosed in `[*` and `*]`.

The behaviour on the case-analysis annotation is one of many easily programmable ones. Figure 4 shows how simple automation through the processing of annotations is programmed, showing the datatype for the taxonomy of annotations and how to process each kind of annotation. Other functions that make part of the taxonomy of annotations are syntax information, book-keeping on annotations, etc. These obligations for customisation of the annotation infra-structure are programmed as two SML modules satisfying the signatures shown in Figure 5. The datatype of annotations itself can be arbitrary, apart from the one restriction that it must support an equality test.

One can conclude that it has been reasonably straightforward to implement annotations in HOL; many factors contributed to this success. The LCF-style proof checking paradigm provides complete extensibility and programmability through Standard ML, therefore supporting a methodology of experimenting with controlling the proof. In particular, the existence of modules in SML was felicitous, at it made it possible to parametrise the complete prover for a taxonomy of annotations to be entered at a later stage. This, and the easy way of declaring datatypes, illustrates the support SML provides for this kind of programming.

Another factor that helped this work is the proof-checking approach behind HOL. Automation is not inbuilt into HOL, which makes it possible to experiment with different degrees of mechanisation of proof procedures using annotations. The interaction with HOL is direct (one guides it by asking it to something, rather than by thwarting its default behaviour), and it is clear to see how annotations can be part of the proof cycle.

Another characteristic of HOL that has been exploited successfully is the distinction between the datatypes of a goal and a theorem. This means that it has been possible to process annotations in one way when creating subgoals from a goal (where the annotations are *inherited* top-down) and in another when theorems are combined to built a new resulting theorem (where annotations are *synthesised* bottom-up). It should be noted in the standard HOL system the explicit tree of subgoals is not kept, for reasons of efficiency. However, it has been easy to extract the tree data structure of proofs. Even without annotations, proof tree visualisations help understand how the proof is organised, and where any relevant lemmas fit into the overall proof.

Slind has incorporated these ideas into the distribution version of HOL, making annotations an integral aspect of user interaction. However, he has annotations only at the proof tree level, rather than in the term level. The full functionality of annotations is not achieved, as some of the characteristics of extra-formal information arises from a close inter-meshing with subterms of formal descriptions.

So what can be learnt about the generic annotation mechanism vis-a-vis this implementation? One criticism made to the fact that it has been easy to implement the ideas in HOL has been, of course, that the way I envision annotations to work is coloured by the experience in implementing them in HOL. Therefore, one way to show the generality of the use of annotations, I plan to use them in several proof systems. For this purpose, in the rest of this paper I discuss the beginnings of my work in implementing annotations in Isabelle.

```
(--‘(?c. b = c * a) /\
    (?c. a = c * b) ==>
    (a = b) [* cases: b = 0 *]‘--)

val it = () :unit
- expand (DISCH_TAC);

 new goal:

(--‘a = b [* cases: b = 0 *]‘--)

     [(--‘?c. b = c * a‘--)]
    [(--‘?c. a = c * b‘--)]

case analysis on b = 0

 new goals:

(--‘a = b‘--)

    [(--‘~ (b = 0) ‘--)]
    [(--‘?c. b = c * a‘--)]
    [(--‘?c. a = c * b‘--)]

(--‘a = b‘--)

    [(--‘b = 0‘--)]
    [(--‘?c. b = c * a‘--)]
    [(--‘?c. a = c * b‘--)]


val it = () :unit
-
```

Figure 3: Partially automated proof

```
     datatype note =  conjecture of string
                    | input_constraint of string
                    | behavioral_factor of string
                    | case_analysis of string
                    | label of string
                    | analogy of string
                    | induct
                    | suggestion of string ;


...
fun process_each_note (case_analysis x) =
        (print ("case analysis on "^x^"\n";
         apply_tac "ASM_CASES_TAC (--‘" ^x^ "‘--)THEN REDUCE_NOTE_TAC")
  | process_each_note (induct) =
        (print "suggested: INDUCT_TAC\n\n" )
  | ...
```

Figure 4: Annotations used for automating proof

```
signature note_sig =
sig
        eqtype note
        val empty_note : note
        val make_note : string list -> note
        val pp_note : note -> unit
        val join_notes : note -> note -> note
        val show_note : note -> unit
end;


signature hook_sig =
sig
        val hook_in : (term_plus list * term_plus) -> string -> unit
end;
```

Figure 5: Signature for annotation instantiation

```
datatype term =
    Const of string * typ * note
  | Free  of string * typ  * note
  | Var   of indexname * typ * note
  | Bound of int * note
  | Abs   of string *typ * term * note
  | op $  of term* term * note;

datatype thm =
 Thm of {sign: Sign.sg, maxidx: int,
         hyps: term list, prop: term,
         anno: note};
```

Figure 6: Term and theorem structure with annotations

4.2 Annotations in Isabelle

Since implementing annotations in HOL, I have started using the Isabelle theorem prover [Pau94]. I am currently adapting the mechanism for using annotations to Isabelle. While Isabelle differs from HOL in several relevant ways, which make this work challenging, both systems share many ideas, both being tactic-based provers under the so-called LCF tradition. The commonalities between them facilitate the transfer the methodology from one to the other.

Extending Isabelle in the same way as was done in HOL is easy. The core of the implementation of Isabelle—the term structure—is incremented with a new field, called `note`, as illustrated in Figure 6. A field with the same type is also added to the datatype for theorems. The total implementation of this idea is more tedious: all the discriminators, destructors and constructors for terms and theorems have to be recoded to account for this field, as well as the parsing and pretty-printing code.

On the other hand, terms in Isabelle exist at two levels: the *object* logic and the *meta* logic. One may think that annotations can be more superficial, in a way similar to the addition of proof terms in first order logic (Isabelle theory FOLP). While in FOLP proof terms exist at the outer level, it is possible to bring them deeper into the term structure of propositions. Thus deciding at which level annotations should exist is an interesting question.

The next phase is to prototype a mechanism for transmission of annotations. It is here that some of the unique characteristics of Isabelle come into play. For one, the tree-structure of proofs is not as obvious as in HOL, and therefore the tree-like ancestry of subgoals cannot be directly used for the inheritance of annotations. Proofs are produced not by refining goals into subgoals (though it appears to be so at the user level), but by refining one meta-theorem into another. That is, when the proof state consists of a 'goal' A and the user applies a tactic to produce the subgoals B_1 and B_2, Isabelle does the refinement of theorem "$A \Rightarrow G$" to the theorem "$[B_1, B_2] \Rightarrow G$", where G is the initial goal.

One particular difficulty in transmitting annotations in an Isabelle proof is that tactics cannot be used directly to transmit them. HOL is characterized by a large number of tactics, each one corresponding to a particular inference in the forward direction. Isabelle, on the other hand, has a small set of tactics, and it lets the resolution and lifting mechanism take care of the book-keeping involved in creating the subgoals. On the other hand, it might be possible to make use of the mechanisms already in place for unification, particularly if the proof-term style is used.

These and other issues have to be examined in further detail before an implementation is made. It is expected that users of the Isabelle system will have some input as to what kind of annotations they would find useful, and how they would envisage interacting with them.

5 Summary

Annotations are enhanced comments that can be added to terms to store information of different types, and can provide a useful method for enhancing the proof environment. I have previously described a methodology for implementing annotations in the HOL system. In this talk I will discuss the issues that arise in their implementation in Isabelle, and show how they can be used to deal with several issues that are of interest to many Isabelle users, such as visualizing the structure of proofs and the use of proof terms.

References

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Der83] Nachum Dershowitz. *The Evolution of Programs*. Birkhauser, 1983.

[Dum77] Michael Dummett. *Elements of Intuitionism*. Oxford University Press, 1977.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press,, 1993.

[Kal94] Sara Kalvala. Annotations in formal specifications and proofs. *Formal Methods in Systems Design*, 5(1/2), July 1994.

[Knu68] Donald Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 1968.

[Pau94] Lawrence Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Weg76] Ben Wegbreit. Constructive methods in program verification. Technical Report CSL-76-2, XEROX Palo Alto Research Center, 1976.