Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 19–33
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Enhancing Theorem Prover Interfaces with Program Slice Information

Louise A. Dennis[1],[2]

School of Computer Science and Information Technology
University of Nottingham, Nottingham, UK

Abstract

This paper proposes an extension to theorem proving interfaces for use with proof-directed debugging and other disproof-based applications. The extension is based around tracking a user-identified set of rules to create an informative program slice. Information is collected based on the involvement of these rules in both successful and unsuccessful proof branches. This provides a heuristic score for making judgements about the correctness of any rule.
A simple mechanism for syntax highlighting based on such information is proposed and a small case study presented illustrating its operation. No implementation of these ideas yet exists.

Keywords: Proof-Directed Debugging, Program Slicing, Verification

1 Introduction

The use of verification for locating errors in theorems, and more specifically programs, is a relatively neglected area as is the provision of interfaces to assist in this task. This paper considers the proof-directed debugging of functional programs and proposes an extension to current theorem proving interfaces to support this.

The extension is based on the assumption that the debugging process involves locating a program statement or, in the case of functional programs, function case which is incorrect. This incorrect statement will appear in a *program slice* which can be identified during verification. Other program slices leading to correct deductions may also be identified during proof. This information can then be used to create appropriate syntax highlighting of function cases in an interface. A potential highlighting scheme is put forward and a simple case study based around Isabelle/HOL [12] and ProofGeneral [1] is performed to show how this would work.

[1] This research was funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051.
[2] Email:`lad@cs.nott.ac.uk`

doi:10.1016/j.entcs.2006.09.019

No implementation has yet been performed however potential issues are discussed in the context of Isabelle Proof General.

Although the discussion in this paper is based around an application to proof-directed debugging it is likely that similar mechanisms may also be useful in other situations where the cause of a proof failure needs to be identified.

The paper is organised as follows: §2 discusses the concepts of proof-directed debugging and program slicing; §3 present a mechanism for tracking program slices through a proof and §4 presents examples of this mechanism at work via a simple case study; §5 discusses some results using a similar mechanism within an automated system; §6 looks at some related work and §7 discusses implementation issues and other further work.

2 Proof Directed Debugging and Program Slicing

Proof-directed debugging was first suggested by Harper [10] and work is underway to extend this into a framework for locating program errors through the proof process [6]. The idea of using a framework rather than relying on a user's skill at general proof, is based on the example of Algorithmic debugging [15,9,11]. Algorithmic debugging constructs an execution tree of a run of the program on some input and then queries the user each time this tree branches. This identifies branches which are returning false results and so locates sections of code responsible for errors.

Program Slicing was first suggested by Weiser [17]. The key idea was to identify a variable of interest at some point in a program (called the *slicing criterion*) and then extract a fragment of the program (a *program slice*) either containing all those statements upon which the value of the variable at that point depended or that fragment whose values were effected by the value of that variable at that point. Program Slicing techniques for imperative languages have generally followed this work [16] using control flow graphs, data flow graphs or other graph-based representations of programs with statements represented as nodes in the graph and a program slice as a set of nodes from the graph. In functional programs function application takes the place of program statements. The notion of a slicing criterion can also be generalised (e.g. to a projection as in [14]).

The intention behind proof-directed debugging is to use the branching structure of a proof to create program slices and use these to assist in the location of errors. There is clearly a need to provide appropriate tools (i.e. tactics/Isar methods) tailored to this task. This paper does not concentrate on this aspect but considers instead the way a theorem prover's interface could assist a user through the presentation of relevant program slices.

3 Proof Tree Branches as a Slicing Criterion

The verification of functional programs naturally involves splitting a program into a set of equational rules each corresponding to a case in its functional definition. The usage of these rules in the proof can thus be tracked, effectively creating a program