Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 317 (2015) 117–131

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Certifying Square Root and Division Elimination

Pierre Neron [1]

T.U. Delft
Delft, Netherlands

Abstract

This paper presents the implementation of a program transformation that removes square roots and divisions from functional programs without recursion, producing code that can be exactly computed. This transformation accepts different subsets of languages as input and it provides a certifying mechanism when the targeted language is Pvs. In this case, we provide a relation between every function definition in the output code and its corresponding one in the input code, that specifies the behavior of the produced function with respect to the input one. This transformation has been implemented in OCaml and has been tested on different algorithms from the NASA ACCoRD project.

Keywords: Program transformation; Real number computation; Certifying transformation; Semantics preservation

1 Introduction

Critical embedded systems, for example in aeronautics, require a very high level of safety. One approach to produce code that may satisfy this required level of safety is to verify its correctness in a proof assistant such as Pvs. The embedded systems do not run the Pvs code but from the proved Pvs specification we can extract a corresponding program in a real language that corresponds to this specification, (see [6] for an example of extraction).

However, these embedded systems may also be cyber-physical systems and therefore have an extended use of mathematical operations over real numbers that can not be computed exactly. In particular, this is a problem if we aim at satisfying the usual requirements of embedded systems, *i.e.,* bounded memory and bounded loops. Indeed, some methods have been developed to compute exactly with real numbers (see [2, 17, 18]) or sufficient precision using lazy evaluation (see [13]) but these techniques usually involve unbounded behaviors. Therefore, for embedded

[1] Email: `p.j.m.neron@tudelft.nl`

http://dx.doi.org/10.1016/j.entcs.2015.10.012

systems, one usually rely on a finite representation and an analysis of rounding errors via abstract interpretation or interval arithmetic [3,5]. Alternatively, one might want to directly prove the correctness properties not on real numbers but on the effective implementation the system uses, *e.g.,* floating point numbers [1], but in this case the proofs become very difficult since any mathematical intuition is lost.

Aeronautics embedded systems, for example, use square root and divisions in conflict detection and resolution algorithms. These operations can not be exactly computed in a finite memory since they may produce infinite sequence of digits. This is not the case for addition and multiplication. These operations only produce finite sequences of digits and therefore they can be exactly computed using some fixed point representation. Determining the required size for this fixed point representation is relatively easy in embedded systems with only bounded loops. This paper presents a program transformation tool that eliminates these square roots and divisions, this transformation allows the extracted code to be exactly computed. The transformation also provides a certifying mechanism [16] to prove the semantics preservation.

This paper focuses on the system description and the implementation aspects. The theoretical aspects of this work are presented in [9–11]. The paper is structured as follows. Section 2 focuses on the main implementation of the transformation in OCaml. Section 3 describes the embedding of a subset of PVS to provide the certifying process. Section 4 introduces some of the technical details and features of the transformation. Section 5 presents an application of this transformation to a conflict detection algorithm from the ACCoRD [2] framework.

2 The OCaml Transformation

2.1 Language

The program transformation is defined in OCaml and operates on a language denoted MiniPVS that is a typed functional language containing numerical ($\mathbb{R}$) and Boolean constants, tests (if then else), pairs, the usual arithmetic operators $+$, $-$, $\times$, $/$, $\sqrt{}$, the comparisons $=$, $\neq$, $>$, $\geq$, $<$, $\leq$, Boolean operators ($\wedge$, $\vee$, $\neg$), variable and function definition and application. Figure 1 presents the OCaml definition of the language as an abstract datatype where uvar is the set of variable identifiers.

The semantics of such a language is quite straightforward. The expression Letin x body scope is interpreted as *let* $x =$ *body in scope* and Letfun f (v,tv) t body scope is the definition of the function taking v as argument of type tv and returning an element of type t, *i.e., let* f $(v : tv) : t =$ *body in scope*; their semantics use call by value. The detailed semantics of this language can be found in [11], Chapter 3 and 4. We denote $[\![\mathsf{p}]\!]_{Env}$ the semantics of a the program p in the environment Env. Function and variable definitions allow for multi-variable definitions (*e.g.,*let f (x,y) $=$ x $+$ y) but partial application is not allowed. This language can represent a subset of many programming language and programs written in such a subset can

[2] http://shemesh.larc.nasa.gov/people/cam/ACCoRD/