UNIVERSITY NAME

DOCTORAL THESIS

Thesis Title

Author:
John SMITH

Supervisor:
Dr. James SMITH

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Research Group Name
Department or School Name

May 8, 2021

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITY NAME

Abstract

Faculty Name
Department or School Name

Doctor of Philosophy

Thesis Title

by John SMITH

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

For/Dedicated to/To my…

Chapter 1

Proving semantic preservation in HILECOP

- Change σ_{injr} and σ_{injf} into σ_i.
- Define the $\texttt{Inject}_{\downarrow}$ and $\texttt{Inject}_{\uparrow}$ relations.
- Keep the $sitpn$ argument in the SITPN full execution relation, but remove it from the SITPN execution, cycle and state transition relations.
- Make a remark on the differentiation of boolean operators and intuitionistic logic operators
- Explain and illustrate the equivalence relation between SITPN and VHDL.
- Talk about the correspondence between combinational signal value and there assignment expression deduced from the code. Explain that this is where the $\mathcal{H}$-VHDL semantics plays it part in the proof; although we are not detailling how assignment expressions are deduced from running the semantics of the $\mathcal{H}$-VHDL code. Give some examples of correspondence between combinational signal value and assignment expressions.

1.1 Semantic preserving transformations in the literature

In this section, we present the review of the work done in the literature pertaining to transformation functions with a focus in the context of formal verification. Here, a transformation function is understood as any kind of mapping from a source representation to a target representation, where the source and target representation possess a behavior of their own (i.e, they are executable). Especially, we are interested in two things:

1. Is there a proper way to build a transformation function? Do standards exist to do this depending on the application domain? How can we build a modular, extensible transformation function? How can we build a transformation function that will ease the proof of semantic preservation?
2. In the context of formal verification, how are expressed the semantic preservation theorems? Are there usual proof strategies?

Here, the goal is to inspire ourselves with the work of the literature, and to see how far the correspondence holds between our specific case of transformation, and other cases of transformations. The results of the literature review are presented in two parts. The two parts have been

prepared based on the same material. The first part will be focusing on the expression of the transformation functions in the literature, and the second part will be focusing on the proof that these transformations are semantic preserving ones.

The material we used for the literature review is divided in three categories. Each category covers a specific case of transformation function, always taken in the context of formal verification. Thus, the three categories are:

- Compilers for generic programming languages
- Compilers for hardware description languages
- Model-to-model and model-to-text transformations

1.1.1 Building transformation functions

As the authors state in [5], "Although theoretically possible, verifying a compiler that is not designed for verification would be a prohibitive amount of work in practice." The question is to know how to design such a compiler? How to anticipate the fact that we will have to prove that the compiler is semantic preserving?

Compilers for generic programming languages

In the context of formally verified compilers for generic programming languages, translation from a source program to a target program mostly straight forward. Not surprisingly, each construct of the source program is mapped to one or many constructs of the target program. Figure 1.1 gives an example of the translation from Java program expressions to Java bytecode expressions, set in the context of a compiler for Java programs written within the Isabelle theorem prover [4]. Here, the mapping between source and target constructs is clearly defined.

```
mkExpr jmb (NewC c) = [New c]
mkExpr jmb (Cast c e) = mkExpr jmb e @ [Checkcast c]
mkExpr jmb (Lit val) = [LitPush val]
mkExpr jmb (BinOp bo e1 e2) = mkExpr jmb e1 @ mkExpr jmb e2 @
   (case bo of
       Eq => [Ifcmpeq 3,LitPush(Bool False),Goto 2,LitPush(Bool True)]
     | Add => [IAdd])
mkExpr jmb (LAcc vn) = [Load (index jmb vn)]
mkExpr jmb (vn::=e) = mkExpr jmb e @ [Dup , Store (index jmb vn)]
mkExpr jmb ( cne..fn ) = mkExpr jmb e @ [Getfield fn cn]
mkExpr jmb (FAss cn e1 fn e2 ) =
   mkExpr jmb e1 @ mkExpr jmb e2 @ [Dup_x1 , Putfield fn cn]
mkExpr jmb (Call cn e1 mn X ps) =
   mkExpr jmb e1 @ mkExprs jmb ps @ [Invoke cn mn X]
mkExprs jmb [] = []
mkExprs jmb (e#es) = mkExpr jmb e @ mkExprs jmb es
```

FIGURE 1.1: Translation from Java expressions to Java bytecode expressions

In the works pertaining to the well-known CompCertproject [3, 1], the many passes building the compiler from C programs to assembly languages are also clearly mapping each construct of source program to target program constructs. This is all the more natural, since the languages like Coq, Isabelle, HOLand other interactive theorem provers permit to perform pattern matching on the abstract syntax tree of the source programs.

The cases of optimizing compilers, like [3] and [6], show that to avoid to write too complex functions when passing from source to target program, that would too difficult to handle during the semantic preservation proof, the compilation is decomposed into many passes. No more than 12 passes for the CakeML compiler, and up to 7 passes for CompCert. This is a way to keep translation functions simple in order to ease reasoning afterwards. Indeed, the more the gap is important between the source representation and the target one, the more the translation function will be complex.

Another point that is noticeable is the expression of translation function is the necessity to keep a binding between source and target representations. For instance, in CompCert, when passing from transformed C programs to an RTL representation (based on registers and control flow graphs), a binding function γ links the variables found in C programs to the registers generated in the RTL representation. The binding is important for the translation (replacing variables by their corresponding registers in the RTL code), and for the proof when values from the source program will be compared to values in the target program. This is a thing that should be anticipated when writing a translation function, i.e what is the correspondence between source and target elements.

In [3], and [2], compilers are written within the Coq proof assistant. Compilers are expressed using the state-and-error monad, thus mimicking the traits of imperative languages into a functional programming language setting.

Compilers for hardware description languages

1.2 Preliminary Definitions

1.3 Behavior Preservation Theorem

1.4 Initial States

1.5 First Rising Edge

1.6 Rising Edge

1.7 Falling Edge

1.8 A detailled proof: equivalence of fired transitions

Appendix A

Reminder on natural semantics

Appendix B

Reminder on induction principles

- Present all the material that will be used in the proof, and that needs clarifying for people who do not come from the field (e.g, automaticians and electronicians)
  - structural induction
  - induction on relations
  - ...

Bibliography

[1] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal Verification of a C Compiler Front-End". In: *FM 2006: Formal Methods*. International Symposium on Formal Methods. Springer, Berlin, Heidelberg, Aug. 21, 2006, pp. 460–475. DOI: 10.1007/11813040_31. URL: https://link.springer.com/chapter/10.1007/11813040_31 (visited on 05/25/2020).

[2] Adam Chlipala. "A Verified Compiler for an Impure Functional Language". In: *ACM SIGPLAN Notices* 45.1 (Jan. 17, 2010), pp. 93–106. ISSN: 0362-1340. DOI: 10.1145/1707801.1706312. URL: https://doi.org/10.1145/1707801.1706312 (visited on 05/22/2020).

[3] Xavier Leroy. "A Formally Verified Compiler Back-End". In: *Journal of Automated Reasoning* 43.4 (Nov. 4, 2009), p. 363. ISSN: 1573-0670. DOI: 10.1007/s10817-009-9155-4. URL: https://doi.org/10.1007/s10817-009-9155-4 (visited on 01/21/2020).

[4] Martin Strecker. "Formal Verification of a Java Compiler in Isabelle". In: *Automated Deduction—CADE-18*. International Conference on Automated Deduction. Springer, Berlin, Heidelberg, July 27, 2002, pp. 63–77. DOI: 10.1007/3-540-45620-1_5. URL: https://link.springer.com/chapter/10.1007/3-540-45620-1_5 (visited on 06/08/2020).

[5] Yong Kiam Tan et al. "A New Verified Compiler Backend for CakeML". In: (Sept. 4, 2016). DOI: 10.17863/CAM.6525.

[6] Yong Kiam Tan et al. "A New Verified Compiler Backend for CakeML". In: (), p. 14.