On the Role of Non-functional Properties in Compiler Verification

Jens Knoop[1] and Wolf Zimmermann[2]

[1] TU Wien, Institut für Computersprachen,
A-1040 Wien, Austria
knoop@complang.tuwien.ac.at
[2] Martin-Luther Universität Halle-Wittenberg, Institut für Informatik,
D-06099 Halle/Saale, Germany
zimmer@informatik.uni-halle.de

Abstract. Works on compiler verification rarely consider non-functional properties such as time and memory consumption. This article shows that there are situations where the functional correctness of a compiler depends on non-functional properties; non-functional properties that are imposed by the target architecture, not the application. We demonstrate that this demands for an extended new notion of compilation correctness.

1 Motivation

Functional compilation correctness is informally most commonly considered semantics preservation between source and target program up to deviations due to resource limitations of actual hardware. This notion of correctness has been formalized and made precise in projects on compiler verification such as ProCoS [4] and Verifix [3]. Along these formalizations, compilation correctness essentially boils down towards an appropriate simulation relation between the (binary) target program and the (high-level language) source program up to possible resource limit violations. While intuitive, this notion of compilation correctness does not consider compiler-influenced non-functional properties such as performance and resource utilization, especially time and memory consumption. In fact, the classical notion of compiler correctness is blind wrt non-functional properties.

In this article, we argue that this classical notion of compiler correctness is often too close. We demonstrate this with a practically relevant example: The compilation for *programmable logic controllers* (*PLC*). PLCs follow the control-loop paradigm, i.e., PLCs execute programs iteratively within cycles of a predefined fixed time length. Program portions whose execution has not not been completed at the end of a cycle are skipped. This introduces timing constraints that are not imposed by the application but by the PLC hardware, which makes the straightforward establishing of a simulation relation as required by the classical notion of compilation correctness insufficient. Functional compiler correctness depends in this scenario on adherence to non-functional properties imposed by the target architecture making them first-class citizens for compiler correctness. In this

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 491–495, 2010.

article, we demonstrate this in detail. It shows the need for an enhanced and extended notion of compiler correctness that in addition to functional properties also takes non-functional properties into account.

2 Classical Compiler Correctness

The notion of compiler or translation correctness is often defined as refinement of programming language constructs. In particular each state transition defined by a source language concept (e.g. a conditional statement) must be implemented in exactly the same way by the target machine. Although this is not important for the purpose of this article, it is worth noting that this may forbid some global or interprocedural optimizations. The detailed definition of the notion of compiler correctness took in *Verifix* longer than expected as extensively discussed in [3]. Nowadays, the notion of compiler correctness is based on operational semantics of the programming language and the target languages, respectively.

For this article, we assume that an operational semantics is given by a state transition system. A *state transition system* is a triple $(Q, I, \rightarrow)$ where Q is the (possibly infinite) set of states, $I \subseteq Q$ is the set of initial states and $\rightarrow \subseteq Q \times Q$ is the state transition relation. An *operational semantics* of a programming language L assigns to each program $\pi \in L$ a state transition system $[\![\pi]\!]$. An execution of π is a finite or infinite sequence of states $\langle q_i | i \in \{j | 0 \leq j < k\}\rangle$, $k \in \mathbb{N} \cup \{\infty\}$ such that $q_{i-1} \rightarrow q_i$ for $1 \leq i < k$ and $q_0 \in I$.

From the viewpoint of a compiler user, only the input/output relation of the program is of interest. Each program has such an interaction with an environment which we call *observable behaviour*. The *observable behaviour* of a program consists only of the observable states and state transitions between them induced by the more fine semantics. Compiler users usually only require that the target program preserves the observable behaviour of the source program.

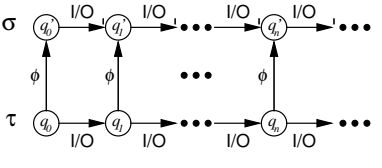

Fig. 1. Preservation of Observable Behaviour

Since usually machine resources are limited while it is easy to write e.g. Java programs that would consume more than 10TByte memory, the target programs may exceptionally stop because of memory overflow. In *Verifix*, we therefore came up with the following notion of correctness: Let τ be a program of the target language with the observable behaviour $(I, Q, \rightarrow)$ and σ be a program of the source language with observable behaviour $(I', Q', \rightarrow')$. *τ preserves the observable behaviour* of *σ up to resource limitations* iff there is a relation $\phi \subseteq Q \times Q'$ such that for any finite or infinite sequence $q_0 \rightarrow q_1 \rightarrow \cdots$ of τ with $q_0 \in I, q_1, q_2, \ldots \in Q$ there is a finite or infinite sequence of states $q_0' \rightarrow q_1' \rightarrow \cdots$ of σ with $q_0' \in I'$ and $q_i \phi q_i'$ for all i except possibly for the last state (if the sequence of observable states of τ is finite). This means that τ halts with violation of resource limitations (cf. Fig. 1). The preservation of observable behaviour up to

resource limitations is transitive and therefore can be applied stepwise for the different phases in a compiler.

Next we apply this classical notion of compiler correctness to PLCs and discuss immediate consequences. Surprisingly, the traditional way to look at embedded systems does not work, i.e., to first ensure functional correctness and to then consider non-functional constraints such as timing conditions. For PLCs it turns out that compiler correctness must include timing constraints from the very beginning as they are imposed by the hardware, not the application.

3 Compiler Correctness for PLCs

Programmable Logic Controllers (abbr. PLCs) are the dominating devices in today's industrial automation systems. The programming of PLCs follows the International Standard IEC 61131-3 that specifies the programming languages. They are rather popular among engineers because of their predictable timing behaviour. Fig. 2 shows the behaviour of a PLC. A PLC-program is iterated infinitely often and implements the control-loop paradigm. At the beginning of each iteration an input phase reads sensor data into the memory of the PLC. Then the PLC-program is executed, and at the end of the iteration an output phase writes some memory cells into actors.

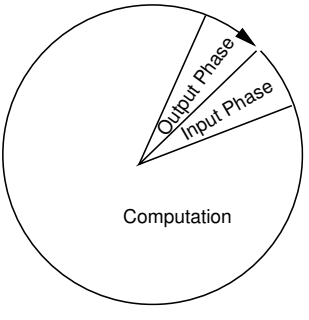

Fig. 2. Execution of programs in PLCs

At first glance, there seems nothing special in proving compiler correctness if the target is a PLC. However, a special feature of PLCs is that the execution of the PLC-program is stopped if a certain time limit (specified by the hardware) has been reached and the next iteration starts again the program.

This behaviour of skipping the execution of program portions on reaching the cycle deadline has implications on the correctness of compilers (as e.g. compilers for *Structured Text* into *Instruction List*).

For compilers for PLCs we thus need an additional requirement for the notion of correctness: Let t be the cycle time, τ be a PLC-program of the target language with the observable behaviour $(I, Q, \rightarrow)$ and σ be a PLC-program of the source language with observable behaviour $(I', Q', \rightarrow')$ such that states contain time and state transitions an increment of time which allows to define the WCET in terms of elapsed time for possible sequences of state transitions. τ *preserves the observable behaviour* of σ *up to resource limitations* iff

i. the worst-case execution time of τ is at most t and

ii. there is a relation $\phi \subseteq Q \times Q'$ such that for any finite or infinite sequence $q_0 \rightarrow q_1 \rightarrow \cdots$ of τ with $q_0 \in I, q_1, q_2, \ldots \in Q$ there is a finite or infinite sequence of states $q'_0 \rightarrow q'_1 \rightarrow \cdots$ of σ with $q'_0 \in I'$ and $q_i \phi q'_i$ for all i except possibly for the last state (if the sequence of observable states of τ is finite).

Hence, the notion of compiler correctness for compilers for PLCs depends on the non-functional property *execution time*. This situation is different to the general situation in embedded systems when general purpose processors are used. The functional correctness of a compiler does not depend on the execution time, only the correctness of the application may depend on execution times. The reason for this difference is that in PLCs the functional behaviour of the target machine depends on the execution time while this is not the case for general purpose processors.

4 Related Work and Conclusions

Correctness of compilers was first considered by McCarthy and Painter [6]. They discussed the compilation of arithmetic expressions. There are a number of works using denotational semantics, e.g. [8,9]. Other works use the approach of refining language constructs, e. g. [4,7,10], or structural operational semantics, e.g. [1]. More recent works [11,5] use a similar notion of correctness as described in this article (although most of them do not consider resource limitations). Glesner et al. considered correct compilers for embedded systems [2]. However, we are not aware of works that consider correct compilation for PLCs as target systems.

In this article we have shown that for PLCs the notion of compiler correctness cannot be based just on a simple simulation relation because the iteration in cycles stops the execution of the program when the cycle time is reached. This shows that non-functional properties can play a surprising and essential role in compiler verification. They are not only needed e.g. to ensure real-time constraints but also for verifying compilers. The classical notion of compiler correctness is insufficient to capture this, and we are not aware of any work that considers non-functional properties quantitatively and integrates them in a verified (optimizing) compiler. In particular, the rapidly developing fields of embedded software demand for such work. Similarities of PLCs and synchronous languages such as Esterel and Lustre need to be further investigated.

References

1. Diehl, S.: Semantics-Directed Generation of Compilers and Abstract Machines. PhD thesis, Universität Saarbrücken (1996)
2. Glesner, S., Geiß, R., Bösler, B.: Verified code generation for embedded systems. In: 1st Workshop on Compiler Optimization meets Compiler Verification COCV 2002. Electronic Notes in Theoretical Computer Science, vol. 65 (2002)
3. Goos, G., Zimmermann, W.: Verification of compilers. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 201–230. Springer, Heidelberg (1999)
4. Hoare, C.A.R., Jifeng, H., Sampaio, A.: Normal Form Approach to Compiler Design. Acta Informatica 30, 701–739 (1993)
5. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7) (2009)

6. McCarthy, J., Painter, J.A.: Correctness of a compiler for arithmetical expressions. In: Proceedings of a Symposium in Applied Mathematics, vol. 19. AMS, Providence (1967)
7. Müller-Olm, M.: Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction. LNCS, vol. 1283. Springer, Heidelberg (1997)
8. Palsberg, J.: An automatically generated and provably correct compiler for a subset of Ada. In: IEEE International Conference on Computer Languages (1992)
9. Polak, W.: Compiler Specification and Verification. LNCS, vol. 124. Springer, Heidelberg (1981)
10. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer, Heidelberg (2001)
11. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. Journal of Universal Computer Science 9(3), 223–247 (2003)