Theoretical Computer Science 577 (2015) 98–124

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Observational program calculi and the correctness of translations

Manfred Schmidt-Schauß[a], David Sabel[a,*], Joachim Niehren[b], Jan Schwinghammer

[a] Goethe-University Frankfurt am Main, Germany
[b] INRIA Lille, Links Project, France

A R T I C L E   I N F O

Article history:
Received 23 May 2013
Received in revised form 6 January 2015
Accepted 17 February 2015
Available online 25 February 2015
Communicated by V. Sassone

Keywords:
Contextual equivalence
Correctness
Semantics
Translations

A B S T R A C T

For the issue of translations between programming languages with observational semantics, this paper clarifies the notions, the relevant questions, and the methods; it constructs a general framework, and provides several tools for proving various correctness properties of translations like adequacy and full abstractness, with a special emphasis on observational correctness. We will demonstrate that a wide range of programming languages and programming calculi and their translations can make advantageous use of our framework for focusing the analysis of their correctness.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Motivated by our experience in analyzing and proving properties of translations between programming languages with observational semantics, this paper clarifies the notions, the relevant questions, and the methods, and also constructs a general framework, and provides several tools for proving various correctness properties of translations like adequacy and full abstractness. The presented framework can directly be applied to the observational equivalences derived from the operational semantics of programming calculi (also with concurrency), to the relationship between specification and implementation, to the issue of correctness of embedding a language into another, and also to further issues in translations, and thus has a wide range of applications.

In order to be as general as possible, we use the term *observation* in this paper for *any* predicate on programs, *without* any further restriction like being a constructive one or an effectively computable property.

Motivation Translating programs is an important operation in several fields of computer science. There are four main tasks where translations play an important role:

(1) Translation is the standard task of a compiler, where this is usually a conversion from a high-level language into an intermediate or low-level one, like an abstract machine language or an assembly-like language. Correctness of such a translation ensures correctness of the compiler.

* Corresponding author.
E-mail address: sabel@ki.informatik.uni-frankfurt.de (D. Sabel).

http://dx.doi.org/10.1016/j.tcs.2015.02.027
0304-3975/© 2015 Elsevier B.V. All rights reserved.

(2) Translations are required in programming languages for explaining the meaning of surface language constructs by decomposing them into a number of more primitive operations in the core part of the programming language. The translation paradigm is also useful for reasoning about the implementations of language extensions in terms of a core language (which are often packaged into the language's library). Typical examples are implementations of channels, buffers, or semaphores using a synchronizing primitive in the core part of the language, for example using mutable reference cells and futures in Alice ML [7,39,65], or using MVars in Concurrent Haskell [41]. Correctness of these implementations can be proved by interpreting the implementation as a translation from the extended language into the original language and then showing correctness of the translation.
(3) Optimization of programs by transforming them into programs of the same language, like inlining, partial evaluation and dead code removal are a special case of translations, where source and target language are the same. Again correctness of the program transformations can be seen correctness of translations.
(4) Translations are used to compare the expressiveness (and obtain corresponding expressiveness results) between different languages or programming models. Examples are showing adequacy or full abstractness of denotational models (e.g. [45,31,9,2]), where the translation computes the denotation of the program, proving a language extension being conservative, or even showing non-expressiveness by proving the non-existence of "correct" translations (one such example is the non-encodability of the synchronous π-calculus in its asynchronous variant under mild restrictions [40]). A further example is the question for the expressive power of a sublanguage, viewed as embedded, and whether the language can be seen as a conservative extension of the sublanguage. An example scenario is removing syntactic sugar.

Correctness of these translations is an indispensable prerequisite for their safe use. However, there are various different views of the strength of correctness of a translation. This pluralism appears to be necessary and driven by practical needs, since the desired strength of correctness of a translation may depend on the specific setting.

From a bird's eye view a programming language is a set of programs $\mathcal{P}$ equipped with a notion of equivalence $\sim$ of programs (the semantics). A translation T maps programs from a source language $\mathcal{K}$ into a target language $\mathcal{K}'$. This suffices to define the commonly used notions of *adequacy* and *full abstractness*: The translation T is adequate iff for all (closed) programs $p_1, p_2 \in \mathcal{K}$ the implication $T(p_1) \sim_{\mathcal{K}'} T(p_2) \Longrightarrow p_1 \sim_{\mathcal{K}} p_2$ holds; this can also be seen as the "no confusion" property, since it is equivalent to $p_1 \not\sim_{\mathcal{K}} p_2 \Longrightarrow T(p_1) \not\sim_{\mathcal{K}'} T(p_2)$. If additionally $p_1 \sim_{\mathcal{K}} p_2 \Longrightarrow T(p_1) \sim_{\mathcal{K}'} T(p_2)$ holds then T is fully abstract. Requiring fully abstract translations is often a too hard condition, while adequacy is a necessity. Without adequacy, in the target language equivalent programs may be interchanged correctly (since they are equivalent), but from the source level view, the semantics is changed. From a compilation point of view, full abstractness ensures that optimizations of programs can be performed only in $\mathcal{K}$ or only in $\mathcal{K}'$ without missing opportunities. If a compilation is adequate, but not fully abstract (which is unavoidable in a majority of cases), then optimizing in $\mathcal{K}'$ and $\mathcal{K}$ is permitted, but optimizing only in $\mathcal{K}'$ may miss certain optimizations, since after compilation, some intentional knowledge may be lost.

Depending on the definitions of $\sim_{\mathcal{K}}$ and $\sim_{\mathcal{K}'}$, adequacy and even full abstractness may be too weak as a correctness notion. E.g., none of the properties ensures that programs before and after the translation have the same termination behavior, which is an inevitable requirement to conclude that T together with $\mathcal{K}'$ is a correct evaluator for $\mathcal{K}$. Thus we will be more concrete, and provide a general approach for program calculi with an operational semantics, which beside others covers the termination behavior of programs.

Results and applications We give a short and informal overview of the results and applications. Within the scope are (at least) programming languages where programs can be written down (using a syntax) and evaluated (using an operational semantics) and of several observations like successful evaluation in a program context (may-convergence, must-convergence). Syntactically, typing can be formulated and open and closed programs can be distinguished, and operationally, also concurrent systems, e.g. process calculi are in the scope.

A first useful result (where variants are already in the literature) is Theorem 3.16 which shows that there are easily checkable criteria for adequacy, namely compositionality and convergence equivalence. In this paper there are further extensions and variations of this method.

A general result concerning translations is Theorem 3.21 which exhibits conditions, when the translated programs are more or less equivalent to the original programs (in the image calculus); which is a theorem similar to an algebraic homomorphism theorem. It also provides a first result on conditions when a translation is an embedding.

Section 4 exhibits several conditions and scenarios for extending (resp. embedding) languages. Theorem 4.3 puts a common technique in semantics (approximation by $\perp$) for showing full abstraction into the context of our framework, where the approximations are represented as a family of translations.

Theorem 4.6 is a criterion for full abstractness under a minimal set of conditions. It complements the full abstractness criteria of Theorem 3.21. Corollaries 4.7 and 4.8 show that there are easy-to-check preconditions for applying Theorem 4.6.

Applications of the framework and the methods are in Section 5.2 and in Section 5.3. In particular Corollary 4.8 was applied in [38]. Another published result is the proof of conservativity of embedding the deterministic part of the language in the full language [54].

This paper presents a unifying approach for different variations of observational semantics and translations of programming languages. The usage may be as turnkey solution for proving properties of translations, and if this is not applicable,

Download English Version:

https://daneshyari.com/en/article/6876061

Download Persian Version:

https://daneshyari.com/article/6876061

Daneshyari.com