Themen und weitere Hinweise für das Seminar:

Spezielle Themen zu Softwaresystemen

PD Dr. David Sabel

Sommersemester 2018

Contents

1 Organisatorisches

1.1 Vorbesprechung

Für die Teilnahme am Seminar, ist es u.a. notwendig bei der Vorbesprechung anwesend zu sein. Der Termin für die Vorbesprechung wird noch bekannt gegeben. Vortrags- und Abgabetermine und weitere Treffen werden dort ebenfalls besprochen.

1.2 Webseite

Die Webseite zur Veranstaltung ist über

http://goethe.link/SIW-I-S-2018

zu finden. Sie enthält aktuelle Informationen zum Seminar, wie Terminplan, Raumänderungen, Themenliste usw.

1.3 Modulabschlussprüfung

Die Modulabschlussprüfung ist bestanden, falls sowohl die Ausarbeitung als auch der Vortrag mit mindestens "'ausreichend"' (4.0) bewertet wurden und regelmäßig teilgenommen wurde. Die Note der Prüfung berechnet sich aus dem Durchschnitt der Note für die Ausarbeitung und der Note für den Vortrag.

2 Allgemeines

Die vorgesehenen Themen für das Seminare sind im Abschnitt 3 dieses Dokuments zu finden. Für das erfolgreiche Absolvieren des Seminars ist folgendes Vorgehen empfohlen bzw. ist folgendes zu beachten:

1. Finden der angegebenen Quellen (möglichst sofort damit beginnen):
   - Wo gibt es die angegebene Literatur?
   - Funktionieren die Links ins WWW?
2. Prüfen der angegebenen Quellen:
   - Welche Informationen sind dort zu finden?
   - Welche der gefunden Informationen sind nützlich?
   - Welche Informationen fehlen oder reichen die vorhandenen Quellen?
3. Suchen eigener Quellen

- Literaturrecherche via Bibliotheksdatenbanken (OPAC, Institutsbibliothek)
- Literaturrecherche via Internet (Suchmaschinen wie z.B. Google)

4. Erstellen eines Konzepts

   - Welche Aspekte sollen in die Ausarbeitung?
   - Gliederung der Ausarbeitung
   - Stichpunkte zu den einzelnen Unterpunkten
   - Seitenzahlen abschätzen für die einzelnen Gliederungspunkte.

5. Besprechung des Konzepts. Vor der Besprechung sollte ein Termin per Email ausgemacht werden, im Idealfall sollten unklare Punkte rechtzeitig per Email vor der Besprechung angekündigt werden.

6. Anfertigung der Ausarbeitung

   Die Ausarbeitung soll eine in eigenen Worten verfasste Darstellung des Themas sein. Insbesondere sollte sie nicht aus einer bloßen Übersetzung eines englischen Orginaltexts bestehen. Der Umfang der Ausarbeitung sollte ca. 15 Seiten betragen.

7. Abgabe der Ausarbeitung Eine Vorversion der Ausarbeitung sollte zum Abgabetermin abgegeben werden. 2 Wochen nach dem Vortrag muss die endgültige Ausarbeitung bei angekommen sein.

   Anschließend wird sie begutachtet und es werden eventuell Vorschläge zur Nachbesserung gegeben. Die Nachbesserung sollte normalerweise bis zum Vortrag, in Ausnahmefällen bis zwei Wochen nach dem Vortrag, vorgenommen werden.

8. Präsentation / Vortrag

   Ziel des Vortrages ist zum einen die Vermittlung des Stoffes an die Teilnehmer, zum anderen dient der Vortrag zum Üben einer Präsentation. Hierbei sollte beachtet werden:

   Publikum Berücksichtigen Sie, an wen sich Ihr Vortrag richtet. Dies sind im wesentlichen die übrigen Seminarteilnehmer aber auch der Dozent.

   Zeitbeschränkung Es ist eine Vortragszeit von **60 Minuten** vorgesehen. Diese Begrenzung ist großzügig bemessen, bei wissenschaftlichen Konferenzen haben Sie noch wesentlich weniger Zeit, oft nur 10 - 30 Minuten.

   Die Zeitbeschränkung lässt es oft nicht zu, das gesamte Thema in allen Einzelheiten zu präsentieren, Sie sollten evtl. Teilthemen weglassen, oder unwichtigere Aspekte derart platzieren, so dass Sie sie nur dann vortragen, wenn noch entsprechend Zeit zur Verfügung steht. Bei der Auswahl des Stoffes sollten die folgenden Fragestellungen berücksichtigt werden:

   - Was ist besonders wichtig und muss präsentiert werden?

- Was ist eher uninteressant und kann bei der Präsentation weggelassen werden?
- Was könnte präsentiert werden, falls noch genügend Zeit am Ende Präsentation vorhanden ist?

Zusätzlich zur eigentlichen Vortragszeit sind 5 Minuten für Zwischenfragen und Diskussion eingeplant.

Folien / Bildschirmpräsentation Die Präsentation kann mithilfe einer Bildschirmpräsentation mit Laptop und Beamer erfolgen. Hierbei sollte die Information pro Folie wohldosiert sein: Statt langer Texte sollten die wichtigsten Fakten stichpunktartig auf den Folien festgehalten werden. Die Folien sollten lesbar sein (entsprechend große Schriftgröße wählen).

Die Anzahl der Folien kann je nach Inhalt der Folien variieren, als Maßstab sind ca. 15-20 Folien einzuplanen. Animationen innerhalb der Präsentation sollten Sie weitestgehend vermeiden, und wenn, dann nur sinnvoll einsetzen.

Für die Bildschirmdarstellung von Präsentationen sei PDF angeraten, da diese auf jedem Rechner anzeigbar sind. Beim Einsatz von LaTeX bietet sich hierfür besonders das Paket "Beamer" http://latex-beamer.sourceforge.net/ an.

Zur Vorbereitung auf den Vortrag wird empfohlen, den Vortrag probeweise vor einem Kommilitonen o.ä. zu halten, insbesondere um zu sehen, ob

- der Vortrag zu lang / kurz ist.
- der Vortrag verständlich ist.
- der Vortrag spannend oder eher ermüdend ist.

9. Teilnahme Von den Teilnehmern wird erwartet, dass Sie bei **allen** Vorträgen des Seminars anwesend sind, und aktiv am Seminar teilnehmen, indem Sie sich an der Diskussion beteiligen.

3 Themen

3.1 Algebraic translations, correctness and algebraic compiler construction

Abstract

Algebraic translation methods are argued for in many fields of science. Several examples will be considered: from the field of compiler construction, database updates, concurrent programming languages, logic, natural language translation, and natural language semantics. Special attention will be given to the notion 'correctness of translation'. In all fields this notion can be defined

as the commutativity of some diagram which connects languages, translation and meanings. For algebraically defined compilers, five different definitions are found in the literature. We argue which of these should be considered the 'right' one (it is not the standard choice). We conclude with a first step towards a general algebraic theory of translation.

Literatur

Theo M.V. Janssen: Algebraic translations, correctness and algebraic compiler construction. In Theor. Comput. Sci. , Volume 199, S. 25-56, 1998

aus dem Netz der RBI kostenfrei abrufbar via
http://dx.doi.org/10.1016/S0304-3975(97)00267-3

3.2 Observational program calculi and the correctness of translations

Abstract

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.

Literatur

Manfred Schmidt-Schauß, David Sabel, Joachim Niehren, and Jan Schwinghammer: Observational program calculi and the correctness of translations. In Theor. Comput. Sci. , Volume 577, S. 98-124 2015

aus dem Netz der RBI kostenfrei abrufbar via
http://dx.doi.org/10.1016/j.tcs.2015.02.027

3.3 On Sessions and Infinite Data

Abstract

We investigate some subtle issues that arise when programming distributed computations over infinite data structures. To do this, we formalise a calculus that combines a call-by-name functional core with session-based communication primitives and that allows session operations to be performed "on demand". We develop a typing discipline that guarantees both normalisation of expressions and progress of processes and that uncovers an unexpected interplay between evaluation and communication.

Literatur

Paula Severi, Luca Padovani, Emilio Tuosto and Mariangiola Dezani-Ciancaglini, On Sessions and Infinite Data, In Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, S. 245–261, 2016

Kostenlos verfügbar unter:
http://www.di.unito.it/∼dezani/papers/sptd16.pdf
Langversion mit Beweisdetails verfügbar unter:
https://hal.archives-ouvertes.fr/hal-01297293/document

3.4 Biorthogonality, step-indexing and compiler correctness

Abstract

We define logical relations between the denotational semantics of a simply typed functional language with recursion and the operational behaviour of low-level programs in a variant SECD machine. The relations, which are defined using biorthogonality and stepindexing, capture what it means for a piece of low-level code to implement a mathematical, domain-theoretic function and are used to prove correctness of a simple compiler. The results have been formalized in the Coq proof assistant.

Literatur

Nick Benton and Chung-Kil Hur, Biorthogonality, Step-indexing and Compiler Correctness In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, S. 97-108,

aus dem Netz der RBI kostenfrei abrufbar via
http://doi.acm.org/10.1145/1596550.1596567

3.5 Proving Correctness of a Compiler Using Step-indexed Logical Relations

Abstract

In this paper we prove the correctness of a compiler for a call-by-name language using step-indexed logical relations and biorthogonality. The source language is an extension of the simply typed lambda-calculus with recursion, and the target language is an extension of the Krivine abstract machine. We formalized the proof in the Coq proof assistant.

Literatur

Proving Correctness of a Compiler Using Step-indexed Logical Relations Leonardo Rodríguez , Miguel Pagano , Daniel Fridlender In Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015), Electronic Notes in Theoretical Computer Science Volume 323, 11 July 2016, S. 197 – 214

aus dem Netz der RBI kostenfrei abufbar via
http://dx.doi.org/10.1016/j.entcs.2016.06.013

3.6 Lightweight verification of separate compilation

Abstract

Major compiler verification efforts, such as the CompCert project, have traditionally simplified the verification problem by restricting attention to the correctness of whole-program compilation, leaving open the question of how to verify the correctness of separate compilation. Recently, a number of sophisticated techniques have been proposed for proving more flexible, compositional notions of compiler correctness, but these approaches tend to be quite heavyweight compared to the simple "closed simulations" used in verifying whole-program compilation. Applying such techniques to a compiler like CompCert, as Stewart et al. have done, involves major changes and extensions to its original verification. In this paper, we show that if we aim somewhat lower—to prove correctness of separate compilation, but only for a *single* compiler—we can drastically simplify the proof effort. Toward this end, we develop several lightweight techniques that recast the compositional verification problem in terms of whole-program compilation, thereby enabling us to largely reuse the closed-simulation proofs from existing compiler verifications. We demonstrate the effectiveness of these techniques by applying them to

CompCert 2.4, converting its verification of whole-program compilation into a verification of separate compilation in less than two person-months. This conversion only required a small number of changes to the original proofs, and uncovered two compiler bugs along the way. The result is SepCompCert, the first verification of separate compilation for the full CompCert compiler.

Literatur

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, Viktor Vafeiadis, Viktor: Lightweight Verification of Separate Compilation In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, S. 178 – 190, 2016

aus dem Netz der RBI kostenfrei abufbar via
http://doi.acm.org/10.1145/2837614.2837642

3.7 Fully-abstract compilation by approximate back-translation

Abstract

A compiler is fully-abstract if the compilation from source language programs to target language programs reflects and preserves behavioural equivalence. Such compilers have important security benefits, as they limit the power of an attacker interacting with the program in the target language to that of an attacker interacting with the program in the source language. Proving compiler full-abstraction is, however, rather complicated. A common proof technique is based on the back-translation of target-level program contexts to behaviourally-equivalent source-level contexts. However, constructing such a back-translation is problematic when the source language is not strong enough to embed an encoding of the target language. For instance, when compiling from the simply-typed λ-calculus ($\lambda\tau$) to the untyped λ-calculus (λu), the lack of recursive types in $\lambda\tau$ prevents such a back-translation. We propose a general and elegant solution for this problem. The key insight is that it suffices to construct an approximate back-translation. The approximation is only accurate up to a certain number of steps and conservative beyond that, in the sense that the context generated by the back-translation may diverge when the original would not, but not vice versa. Based on this insight, we describe a general technique for proving compiler full-abstraction and demonstrate it on a compiler from $\lambda\tau$ to λu . The proof uses asymmetric cross-language logical relations and makes innovative use of step-indexing to express the relation between a context and its approximate back-translation. We believe this proof technique can scale to challenging settings and enable simpler, more scalable proofs of compiler full-abstraction.

Literatur

Dominique Devriese, Marco Patrignani, and Frank Piessens: Fully-abstract Compilation by Approximate Back-translation, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16), S. 164-177, 2016.

aus dem Netz der RBI kostenfrei abrufbar via
http://doi.acm.org/10.1145/2837614.2837618

3.8 Refinement reflection: complete verification with SMT

Abstract

We introduce Refinement Reflection, a new framework for building SMT-based deductive verifiers. The key idea is to reflect the code implementing a user-defined function into the function's (output) refinement type. As a consequence, at uses of the function, the function definition is instantiated in the SMT logic in a precise fashion that permits decidable verification. Reflection allows the user to write equational proofs of programs just by writing other programs using pattern-matching and recursion to perform case-splitting and induction. Thus, via the propositions-as-types principle, we show that reflection permits the specification of arbitrary functional correctness properties. Finally, we introduce a proof-search algorithm called Proof by Logical Evaluation that uses techniques from model checking and abstract interpretation, to completely automate equational reasoning. We have implemented reflection in Liquid Haskell and used it to verify that the widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses actually satisfy key algebraic laws required to make the clients safe, and have used reflection to build the first library that actually verifies assumptions about associativity and ordering that are crucial for safe deterministic parallelism.

Literatur

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Phlip Wadler, Ranjit Jhala.2018. Refinement Reflection: Complete Verification with SMT, Proc. ACM Program. Lang., Vol. POPL-2, 53:1–53:31.

Aus dem Netz der RBI kostenlos abrufbar unter:
http://doi.acm.org/10.1145/3158141

3.9 Online detection of effectively callback free objects with applications to smart contracts

Abstract

Callbacks are essential in many programming environments, but drastically complicate program understanding and reasoning because they allow to mutate object's local states by external objects in unexpected fashions, thus breaking modularity. The famous DAO bug in the cryptocurrency framework Ethereum, employed callbacks to steal $150M. We define the notion of Effectively Callback Free (ECF) objects in order to allow callbacks without preventing modular reasoning. An object is ECF in a given execution trace if there exists an equivalent execution trace without callbacks to this object. An object is ECF if it is ECF in every possible execution trace. We study the decidability of dynamically checking ECF in a given execution trace and statically checking if an object is ECF. We also show that dynamically checking ECF in Ethereum is feasible and can be done online. By running the history of all execution traces in Ethereum, we were able to verify that virtually all existing contract executions, excluding these of the DAO or of contracts with similar known vulnerabilities, are ECF. Finally, we show that ECF, whether it is verified dynamically or statically, enables modular reasoning about objects with encapsulated state.

Literatur

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts, Proc. ACM Program. Lang., Vol. POPL-2, 48:1–48:28.

Aus dem Netz der RBI kostenlos abrufbar unter:
http://doi.acm.org/10.1145/3158136

3.10 Linear Haskell: practical linearity in a higher-order polymorphic language

Abstract

Linear type systems have a long and storied history, but not a clear path forward to integrate with existing languages such as OCaml or Haskell. In this paper, we study a linear type system designed with two crucial properties in mind: backwards-compatibility and code reuse across linear and non-linear

users of a library. Only then can the benefits of linear types permeate conventional functional programming. Rather than bifurcate types into linear and non-linear counterparts, we instead attach linearity to function arrows. Linear functions can receive inputs from linearly-bound values, but can also operate over unrestricted, regular values.

To demonstrate the efficacy of our linear type system – both how easy it can be integrated in an existing language implementation and how streamlined it makes it to write programs with linear types – we implemented our type system in ghc, the leading Haskell compiler, and demonstrate two kinds of applications of linear types: mutable data with pure interfaces; and enforcing protocols in I/O-performing functions.

Literatur

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang., Vol. POPL-2, 5:1–5:29.

Aus dem Netz der RBI kostenlos abrufbar unter:
http://doi.acm.org/10.1145/3158093