ISSN 2186-7437

NII Shonan Meeting Report

No. 2011-5

Automated Techniques for Higher-Order Program Verification

Naoki Kobayashi
Luke Ong
David Van Horn

September 23–26, 2011

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Automated Techniques for Higher-Order Program Verification

Organizers:
Naoki Kobayashi (Tohoku University)
Luke Ong (University of Oxford)
David Van Horn (Northeastern University)

September 23–26, 2011

With the increasing importance of software reliability, program verification has been an important and hot research topic. Recently, we have seen some good progress in automated techniques for verification of higher-order programs. Studies of game semantics have yielded compositional model checkers and automated program equivalence checkers for Algol-like programming languages, and studies of higher-order recursion schemes and pushdown automata have yielded model checkers for higher-order functional programs. Classical control flow analysis has been recently revisited to yield more precise and/or efficient methods than Shivers' k-CFA.

The aim of the workshop was to bring together researchers on automated techniques for higher-order program verification and analyses, and provide them with an opportunity to exchange new research results, and discuss further extensions. The workshop also aimed for cross-fertilization of different techniques for higher-order program verification, such as game semantics, type theories, higher-order grammars and pushdown systems, control flow analyses, and abstract interpretation.

The workshop was organized on an invitation basis, and 25 researchers (including 3 organizers) attended it. The workshop program consisted of tutorials, technical talks, and discussions. We collect abstracts of the talks below.

Overview of Talks

Algorithmic game semantics

Andrzej Murawski (University of Leicester)

I will give a tutorial survey of results on proving program equivalences using game semantics.

Games with names

Nikos Tzevelekos (University of Oxford)

The dynamic creation of new resources is a ubiquitous feature in modern computing and emerges in a wide range of scenarios: from process calculi and the semantics of mobility, to programming languages with objects, references, exceptions, etc. These resources are identified by use of unique, fresh identifiers which we call names. This talk is about formal techniques for reasoning about names which have emerged in the last years.

We will focus on game semantics, a denotational theory of programming languages which models programs as interactions (games) between two players, representing the program and its environment respectively. Nominal games constitute a fresh strand of the theory which is founded on a universe of sets with names called nominal sets. Such games provide models for programming languages with new resources such as fragments of ML. Moreover, nominal game models can be given algorithmic representations by means of newly introduced abstract machines, called Fresh-Register Automata, which operate on infinite alphabets of names.

A tutorial on the small-step approach to CFA

Matthew Might (University of Utah)

The small-step approach to static analysis of higher-order programs unifies and generalizes many concepts in classical CFA theory. This tutorial will introduce: (1) the general framework of small-step analysis; (2) its foundations in abstract machines; (3) its connections to CFAs descending from Shivers's and Jones's formulations of control-flow analysis; and (4) unique advantages of the small-step approach.

Flow-Sensitive Type Recovery in Linear-Log Time

Jan Midtgaard (Aarhus University)

The flexibility of dynamically typed languages such as JavaScript, Python, Ruby, and Scheme comes at the cost of run-time type checks. Some of these checks can be eliminated via control-flow analysis. However, traditional control-flow analysis (CFA) is not ideal for this task as it ignores flow-sensitive information that can be gained from dynamic type predicates, such as JavaScript's instanceof and Scheme's pair?, and from type-restricted operators, such as Scheme's car. Yet, adding flow-sensitivity to a traditional CFA worsens the already significant compile-time cost of traditional CFA. This makes it unsuitable for use in just-in-time compilers.

In response, we have developed a fast, flow-sensitive type-recovery algorithm based on the linear-time, flow-insensitive sub-0CFA. The algorithm has been implemented as an experimental optimization for the commercial Chez Scheme compiler, where it has proven to be effective, justifying the elimination of about 60run-time type checks in a large set of benchmarks. The algorithm processes on average over 100,000 lines of code per second and scales well asymptotically, running in only $O(n \log n)$ time. We achieve this compile-time performance and scalability through a novel combination of data structures and algorithms.

Deriving control-flow analyses with abstract interpretation

Thomas Jensen (INRIA)

Abstract interpretation techniques are used to derive a control-flow analysis for a simple higher-order functional language. The analysis approximates the interprocedural control-flow of both function calls and returns in the presence of first-class functions and tail-call optimization. The analysis is systematically derived by abstract interpretation of the stack-based C_aEK abstract machine of Flanagan et al. using a series of Galois connections. The analysis induces an equivalent constraint-based formulation, thereby providing a rational reconstruction of a constraint-based, higher-order CFA from abstract interpretation principles.

Joint work with Jan Midtgaard.

CFA2: Pushdown Flow Analysis for Higher-Order Languages

Dimitrios Vardoulakis (Northeastern University)

Flow analysis is a valuable tool for creating better programming languages; its applications span optimization, debugging, verification and program understanding. Despite its apparent usefulness, flow analysis of higher-order programs has not been made practical. The reason is that existing analyses do not model function call and return well: they remember only a bounded number of pending calls because they approximate programs with control-flow graphs. Call/return mismatch results in imprecision and increases the analysis time.

In this talk I will describe CFA, a flow analysis that provides unbounded call/return matching in the presence of hard-to-analyze language features, such as first-class functions, tail recursion and first-class control. The key insight is that we can model a higher-order program as a pushdown automaton. By pushing return points on the stack, we eliminate call/return mismatch. Besides the CFA2 semantics, I will talk about some ongoing implementation work I've been doing with Mozilla, on using CFA2 to analyze Firefox add-ons.

Recursion Schemes, Collapsible Pushdown Automata, and the Model Checking of Higher-Order Computation

Luke Ong (University of Oxford)

Recursion schemes are in essence closed base-type terms of the simply-typed lambda calculus with recursion, generated from uninterpreted first-order symbols. Higher-order pushdown automata are finite-state machines with higher-order pushdown stacks (e.g. a second-order stack is a stack of stacks). Old models of computation much studied in the 70's, there has been a revival of interest in recursion schemes and higher-order pushdown automata (and their variants) as generators of rich infinite structures such as infinite trees and graphs. In this tutorial talk, we will present connections between these generator families, survey recent proofs of the decidability of monadic second order theories of these structures, and briefly discuss applications to the verification of higher-order computation.

Algorithms and applications of higher-order model checking

Naoki Kobayashi (Tohoku University)

Model checking of higher-order recursion schemes, or higher-order model checking for short, can be applied to a variety of verification or analysis problems for higher-order programs. As an example of applications, I will first show a reduction of control flow analysis to higher-order model checking. This may sound like a crazy idea, as higher-order model checking is k-EXPTIME complete in general, for order-k recursion schemes. We think, however, that it is worth considering because: (i) the approach yields the exact flow information for closed, simply-typed programs with recursion and finite base types, (ii) the complexity of higher-order model checking is linear-time in the program size if certain parameters are fixed and properties are restricted to safety properties; thus, the resulting analysis can answer each flow query in linear time, and compute all flow information in cubic time under (admittedly strong) assumptions, and (iii) even if the exact analysis is impractical, certain practical restrictions can be considered, which may yield new hierarchies of CFA.

In the second part, I review two practical algorithms for higher-order model checking, presented at PPDP 2009 and FoSSaCS 2011. They are both type-based, but the latter borrows ideas also from game semantics, by which achieving fixed-parameter linear time complexity in the program size. Both algorithms have their own limitations, which may be overcome by further integration of types and game semantics.
(The first part is joint work with Yoshihiro Tobita and Takeshi Tsukada.)

Higher-order Call-by-value Software Verification

Luca Paolini (Università di Torino)

An extension of the call-by-value lambda-calculus with reductions to manage call-by-value effects is presented. The aim of this calculus is to provides a support to the higher-order verification techniques developed by Kobayashi-Ong for the call-by-name lambda-calculus, by avoiding the need for a continuation-passing style transformation.

Pattern Matching Recursion Schemes for the Verification of Functional Programs

Steven Ramsay (University of Oxford)

We introduce pattern-matching recursion schemes (PMRS) as an accurate model of computation for functional programs that manipulate algebraic datatypes. PMRS are a natural extension of higher-order recursion schemes that incorporate pattern-matching in the defining rules. We are concerned with the following verification problem: given a correctness property A, a functional program P (qua PMRS) and a regular set I of inputs to the program, does every term that is reachable from I under rewriting by P satisfy A? To solve the PMRS verification problem, we present a sound semi-algorithm which is based on model-checking and counterexample guided abstraction refinement. Given a no-instance of the verification problem, the method is guaranteed to terminate.

Higher-Order Program Verification with Liquid Types

Ranjit Jhala (University of California)

Traditional software verification algorithms work by using a combination of Floyd-Hoare Logics, Model Checking and Abstract Interpretation, to infer (and check) suitable program invariants. However, these techniques are problematic in the presence of complex (but ubiquitous) constructs like generic data structures, first-class functions.

We show how modern type systems are capable of the kind of analysis needed to analyze the above constructs, and we use this observation to develop Liquid Types, a new static verification technique which combines the complementary strengths of Floyd-Hoare logics, Model Checking, and Types, resulting in a system that can be used to statically verify properties ranging from memory safety to data structure correctness.

(This tutorial is based on joint work with Patrick Rondon and Ming Kawaguchi.)

Using Proofs-from-Tests to Verify Higher-Order Programs

Suresh Jagannathan (Purdue University)

Dependent type systems such as liquid types provide a promising way to verify useful safety properties of higher-order functional programs. However, annotating programs that have complex control- and data-flow properties with meaningful logical qualifiers, and effectively refining types to yield stronger, more precise invariants, can be complicated. In contrast, systems like Dash combine counterexample-guided abstraction refinement with testing to automatically strengthen coarse abstractions and to discover new ones, albeit for first-order imperative programs. In this talk, we consider the integration of liquid type inference with a Dash-like ”proof-from-tests” refinement strategy. We use dependent type summaries to verify higher-order functions modularly, and leverage concrete tests to strengthen these types through iterative refinement. A prototype implementation of our ideas provides evidence to support the feasibility of our approach.

This is joint work with He Zhu

Relatively Complete Refinement Types from Counterexamples

Tachio Terauchi (Nagoya University)

The existing refinement type systems proposed for the automated verification of higher-order functional programs are incomplete, even relative to a hypothetical complete theorem prover for deciding the refinement predicate entailment judgements. This paper presents an extension to the type system for attaining relative completeness in such systems, and a counterexample-driven type inference for the extended refinement type system. We prove that our approach is complete for the class of “closure-bounded” programs, relative to a complete theorem prover. The extension is in the familiar form of universally quantified types, and the type inference utilizes the existing refinement type inference algorithms by iteratively generating program translation instances from the failed type inference attempts.

To demonstrate the effectiveness, we apply the method to an existing refinement type inference system and show that it is able to automatically verify some programs that have been impossible with the previous approaches.

Predicate Abstraction and CEGAR for Higher-Order Model Checking

Hiroshi Unno (Tohoku University)

Higher-order model checking (more precisely, the model checking of higher-order recursion schemes) has been extensively studied recently, which can automatically decide properties of programs written in the simply-typed lambda-calculus with recursion and finite data domains. This talk presents predicate abstraction and counterexample-guided abstraction refinement (CEGAR) for higher-order model checking, enabling automatic verification of programs that use infinite data domains such as integers. A prototype verifier for higher-order functional programs is also demonstrated.

(Joint work with Naoki Kobayashi and Ryosuke Sato)

Abstract Reduction Semantics for Modular Higher-Order Contract Verification

David Van Horn (Northeastern University)

In this talk, I will describe a new approach to the modular verification of higher-order programs that leverages behavioral software contracts as a rich source of symbolic values. Our approach is based on the idea of an abstract reduction semantics that gives meaning to programs with missing or opaque components. Such components are approximated by their contract and our semantics gives an operational interpretation of contracts-as-values. The result is an executable semantics that soundly approximates all possible instantiations of opaque components, including contract failures. It enables automated reasoning tools that can verify the contract correctness of components for all possible contexts. We show that our approach scales to an expressive language of contracts including arbitrary programs embedded as predicates, dependent function contracts, and recursive contracts. We argue that handling such a feature-rich language of specifications leads to powerful symbolic reasoning that utilizes existing program assertions.

(Joint work with Sam Tobin-Hochstadt)

Termination Analysis of the Untyped lambda-Calculus

Neil D. Jones (University of Cophenhagen)

An algorithm is developed that, given an untyped lambda-expression, can certify that its call-by-value evaluation will terminate. It works by an extension of the "size-change principle" earlier applied to first-order programs. The algorithm is sound (and proven so) but not complete: some lambda-expressions may in fact terminate under call-by-value evaluation, but not be recognised as terminating.

The intensional power of size-change termination is reasonably high: It certifies as terminating all primitive recursive programs, and many interesting and useful general recursive algorithms including programs with mutual recursion and parameter exchanges, and Colson's "minimum" algorithm. Further, the approach allows free use of the Y combinator, and so can identify as terminating a substantial subset of PCF.

(joint work with Nina Bohr)

Higher-order rules: termination and confluence

Jean-Pierre Jouannaud (INRIA-LIAMA)

We will review higher-order rewrite rules, how to check their termination under appropriate typing assumptions, and how to check their confluence assuming termination.

Soundness is not sufficient

Fritz Henglein (DIKU)

Wide-spread real-world static program analysis techniques operate on the Machiavellian principle of "A bug is a bug, and whichever way you catch it is fine". More formal approaches require soundness with respect to a well-specified semantics as entry ticket to being called a respected static analysis. High-powered program analyses, which warrant the terminological switch from "analysis" to "verification", may have towering complexity-theoretic lower bounds, but are oftentimes claimed to work in practice. But what does "practice" mean? It is typically represented by a small, finite number (less than, say, a thousand) of a priori known problem instances. Even if practice is, impressively, represented by all known publicly accessible problem instances, how does one know that tomorrow's unknown problem instance is within the scope of today's practice? If the problem instances are samples, then of which space?

In this rambling talk with no new results I'll draw on my subjective experience with type-based program analysis (most of which would now be played on the classical rather than the indie channel, if it were a radio program) and propose that analysis and verification techniques explicitly address the trade-offs embodied in Rice's Theorem; in particular, that they not accept soundness or diffuse practice arguments as sufficient qualification for first-class respectability. Fundamental questions about static program analyses are in my opinion:

- Which semantic properties is a static analysis *invariant* under (linear beta-reduction, integer constant folding, let-expansion, let-reduction, etc.), if any? That is, which semantic equalities are guaranteed not to change the outcome of the static analysis? Ensuring invariance properties supporting local code transformations is important to avoid the "magic optimization" phenomenon ("I did a small change, and now the code suddenly runs 16% faster/slower.") As is well-known, one direction of invariance (reduction) is often sufficient to yield soundness. Less well-known, the other direction (expansion) can be used to prove structural complexity-theoretic lower bounds and inseparability results. (The latter implies for an analysis that there is no analysis that is both more precise and more efficient to compute; i.e. it abides by the principle "You have to pay more if you want more" or, conversely, "You get (no less than) what you pay for".)

- Is the analysis algorithm *adaptive* in the sense that it computes results for certain subclasses of inputs with predictable complexity and competitive with specialized methods for those subclasses? What are the relevant input parameters for defining subclasses that allow expressing the complexity in terms of these parameters, that is using Fixed-Parameter Complexity (FPC)?

- Is the analysis compositional? Is there a way of analyzing fragments and templates of code, not just whole programs, writing down the results for documentation and reuse? And does having a compositional analysis mean that the static analysis algorithm should also be implemented compositionally, that is compute the result by induction on the syntax? (Here I am blatantly advertising type-based analysis and constraint-based algorithmic techniques, respectively: Type-based analyses are inherently compositional, and the answer to the last question is "no".)

View updatablity checking for graph queries

Keisuke Nakano (University of Electro-Communications)

View updating problem is concerned with translating a view update into a corresponding update against the base data source. In our previous work, we solve the view updating problem in which both sources and views are represented by graph-structured data for general purposes. Since the solution is based on a sort of program inversion techniques, it often requires expensive computation to find the translation of view updating. The problem is that the expensive computation may be in vain when the updated view is invalid in the sense that either there is no candidate of corresponding sources or the corresponding source does not conform to user's intention. In this talk, we present a method for checking view updatability in order to know whether the updated view is valid or not before computing the corresponding sources. To achieve a simple computation of view updatability checking, we introduce a new graph schema whose conformance is defined by graph simulation. Although the idea of our schema comes from the simulation-based graph schema proposed by Buneman et al., our schema can describe necessity of out-going edges, which was impossible in their schema. This improvement helps us to give more precise solution for view updatability checking.

Decomposition of Higher-Order Programs to Garbage-Free First-Order Programs

Kazuhiro Inaba (Google)

In this talk, I present a decomposition of a functional tree-processing program into a sequence of composition of first-order functions where intermediate results are always kept small compared to the final output tree. I show how this decomposition can be used to derive complexity upperbound for several verification problems, and discuss further applications.

Static Approximation of Dynamically Generated Web Pages with Context-Free Grammars

Yasuhiko Minamide (University of Tsukuba)

We developed a program analyzer for PHP that approximates the string output of a program with a context-free grammar. By applying the analysis to a server-side program, dynamically generated Web pages can be approximated with a context-free grammar. The approximation obtained by the analysis has many applications in checking the validity and security of a server-side program. It is successfully applied to publicly available PHP programs to check the validity of dynamically generated pages. Wassermann and Su extended the analyzer to detect SQL injection and cross-site scripting vulnerabilities.

Verifying Liveness Properties of ML Programs

Martin Lester (University of Oxford)

Higher-order recursion schemes are a higher-order analogue of Boolean Programs; they form a natural class of abstractions for functional programs. We present a new, efficient algorithm for checking CTL properties of the trees generated by higher-order recursion schemes, which is an extension of Kobayashi's intersection type-based model checking technique. We show that an implementation of this algorithm, THORS, performs well on a number of small examples and we demonstrate how it can be used to verify liveness properties of OCaml programs. Example properties include statements such as "all opened sockets are eventually closed" and "the lock is held until the file is closed".

A syntax-directed approach to model checking higher-order recursion schemes

Robin Neatherway (University of Oxford)

We offer an alternative algorithm for model checking higher-order recursion schemes that aims to unify the 'traversal' approach of Ong in his original proof of decidability of the model checking problem with the type-based approach of Kobayashi. We hope that this new algorithm will offer additional insight into the problem and the symmetries that arise from recursion scheme computation.

Linear Intersection Types and Game Semantics

Takeshi Tsukada (Tohoku University)

Intersection type systems and game semantics are two promising approaches to verification of higher-order programs. To establish their relationship, we propose a linear intersection type system, in which intersections are not idempotent (i.e., $A \wedge A$ is not equal to A), and show the correspondence between derivations in the linear intersection type system and traversal trees of game semantics.

As an application of the correspondence, we give a type-inference algorithm equivalent to the (constructive) definition of the traversal tree. The algorithm can be seen as an alternative formalization of Kobayashi's algorithm.

HORS model checking and logical relation

Akihiko Tozawa (IBM Research, Tokyo)

Let us recall the relation between type/semantic-based HORS model checking algorithm and Plotkin's logical relation, i.e., inherited relations at each type, satisfied by the denotation of any definable simply-typed lambda term. This view may explain some intuitive idea behind the algorithm.

A family of abstract interpretation for static analysis of concurrent higher-order programs

Matthew Might (University of Utah)

We develop a framework for computing two foundational analyses for concurrent higher-order programs: (control-)flow analysis (CFA) and may-happen-in-parallel analysis (MHP). We pay special attention to the unique challenges posed by the unrestricted mixture of first-class continuations and dynamically spawned threads. To set the stage, we formulate a concrete model of concurrent higher-order programs: the P(CEK*)S machine. We find that the systematic abstract interpretation of this machine is capable of computing both flow and MHP analyses. Yet, a closer examination finds that the precision for MHP is poor. As a remedy, we adapt a shape analytic technique – singleton abstraction – to dynamically spawned threads (as opposed to objects in the heap). We then show that if MHP analysis is not of interest, we can substantially accelerate the computation of flow analysis alone by collapsing thread interleavings with a second layer of abstraction.

Functional Reachability

Nikos Tzevelekos (University of Oxford)

What is reachability in higher-order functional programs? We formulate reachability as a decision problem in the setting of the prototypical functional language PCF, and show that even in the recursion-free fragment generated from a finite base type, several versions of the reachability problem are undecidable from order 4 onwards, and several other versions are reducible to each other. We characterise a version of the reachability problem in terms of a new class of tree automata introduced by Stirling at FoSSaCS 2009, called Alternating Dependency Tree Automata (ADTA). As a corollary, we prove that the ADTA non-emptiness problem is undecidable, thus resolving an open problem raised by Stirling. However, by restricting to contexts constructible from a finite set of variable names, we show that the corresponding solution set of a given instance of the reachability problem is regular. Hence the relativised reachability problem is decidable.

The Complexity of kCFA

David Van Horn (Northeastern University)

Flow analysis aims to predict properties of programs before they are run, but how hard is this to do? We answer the question in the case of the kCFA hierarchy, a ubiquitous family of flow analyses for higher-order languages, by deriving tight lower bounds on the computational complexity of the hierarchy.

For 0CFA, and many of its known approximations, a natural decision problem answered by analysis is complete for polynomial time. This theorem relies on the insight that linearity of programs subverts approximation in analysis, rendering analysis equivalent to evaluation.

For any $k > 0$, we prove the decision problem is complete for exponential time. This theorem validates empirical observations that such control flow analysis is intractable. It also provides more general insight into the complexity of abstract interpretation.