09301 Abstracts Collection
Typing, Analysis and Verification of Heap-Manipulating Programs
— Dagstuhl Seminar —

Peter O'Hearn[1], Arnd Poetzsch-Heffter[2] and Shmuel Mooly Sagiv[3]

[1] Queen Mary College - London, GB
[2] TU Kaiserslautern, D
[3] Tel Aviv University, IL
msagiv@tau.ac.il

Abstract. From July 19 to 24, 2009, the Dagstuhl Seminar 09301 "Typing, Analysis and Verification of Heap-Manipulating Programs" was held in Schloss Dagstuhl – Leibniz Center for Informatics. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar as well as abstracts of seminar results and ideas are put together in this paper. The first section describes the seminar topics and goals in general. Links to extended abstracts or full papers are provided, if available.

Keywords. Ownership types, static analysis, program verification, heap-manipulating programs

09301 Executive Summary – Typing, Analysis, and Verification of Heap-Manipulating Programs

Jonathan Aldrich (Carnegie Mellon University - Pittsburgh, US)

The document contains an executive summary of the Dagstuhl Seminar "Typing, Analysis, and Verification of Heap-Manipulating Programs" that took place July 2009.

Keywords: Typing, Static Analysis, Verification, Heap-Manipulating Programs

Joint work of: Sagiv, Mooly; Poetsch-Heffter, Arnd; O'Hearn, Peter

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2010/2435

Invariant-Carrying Permissions: Modular Dependence on Shared State

Jonathan Aldrich (Carnegie Mellon University - Pittsburgh, US)

A longstanding verification challenge is allowing the invariant for an object to depend on information about another, shared mutable object's state. Existing solutions restrict sharing or mutation (e.g. ownership), or require passing around logical predicates in a way that sometimes violates information hiding and modularity (separation logic).

We have been developing an approach called Invariant-Carrying Permissions, in which access to state is coordinated through fractional permissions that carry an invariant. Each client can rely on the invariant and can mutate the object's state but must respect the state invariant for the benefit of other clients. Our solution allows these permissions to be hidden inside other abstractions such that clients need not be aware of their presence.

We have implemented a prototype of invariant-carrying permissions in our typestate verification system, Plural. Preliminary experience indicates that the flexibility and modularity properties of invariant-carrying permissions are useful in many practical examples. We believe the concept is applicable beyond typestate, to enforce arbitrary predicates.

Full Paper:
http://www.cs.cmu.edu/∼kbierhof/papers/typestate-verification.pdf

Challenge Problem: Modular Verification of Customized Invariants over Shared State

Jonathan Aldrich (Carnegie Mellon University - Pittsburgh, US)

This presentation proposes a challenge problem involving modular verification of customized invariants over shared state. The accompanying code (in Other.zip, the name was chosen by the web software) shows a solution in the Plural typestate verification system; see README.TXT in Other.zip file for how to run the example.

Challenge: remove the Plural specification annotations and insert those from your favorite verification system.

Keywords: Invariant, heap, sharing, modular reasoning, invariant

Higher-Order Separation Logics for Reasoning About First-Class Invariants

Andrew W. Appel (Princeton University, US)

We would like reason using Concurrent Separation Logic about imperative programs that store function-pointers and lock-pointers in the heap, and that contain "assert" statements. We have strong demands on the expressiveness of the logic. Invariants must be able to characterize the binding of memory locations to other invariants, for example, "location l is a pointer to a lock (or function) with resource invariant (or precondition) P". Invariants must have semantics independent of program syntax, so that they survive compilation to lower-level languages. We need polymorphism, that is, impredicative quantification over invariants. Finally, we need it to apply smoothly to real programming languages.

We have constructed a powerful program logic with machine-checked proofs in Coq. The mathematical foundations use (a new formulation of) Separation Algebras combined (a generalization of) the Very Modal Model of mutable references to model a multimodal substructural logic.

We apply this logic to C minor, the input language of Leroy's CompCert compiler, again with machine-checked proofs.

Region Logic

Anindya Banerjee (Madrid Institute for Advanced Studies, ES)

Shared mutable objects pose grave challenges in reasoning, especially for data abstraction and modularity. We present a novel logic for error-avoiding partial correctness of programs featuring shared mutable objects. Using a first order assertion language, the logic provides heap-local reasoning about mutation and separation, via ghost fields and variables of type 'region' (finite sets of object references). A new form of modifies clause specifies write, read, and allocation effects using region expressions; this supports effect masking and a frame rule that allows a command to read state on which the framed predicate depends. Soundness is proved using a standard program semantics. The logic facilitates heap-local reasoning about object invariants. Disciplines such as ownership are expressible without being hard-wired in the logic. Joint work with David A. Naumann and Stan Rosenberg

Keywords: Verification, object-based programs, local reasoning, heap

Joint work of: Banerjee, Anindya; Naumann, David A.; Rosenberg, Stan

Verification Condition Generation for Unstructured Quantifiers

Michael Barnett (Microsoft Research - Redmond, US)

In the Spec# project, we generated BPL from MSIL, the bytecode for the .NET virtual machine. This required a new algorithm for generating verification conditions from code in the form of unstructured programs, namely general control-flow graphs. However, for several reasons, all specifications were preserved in a separate expression language.

One of the main reasons was to support the easy translation of quantifiers. Now, with the Code Contracts project, we cannot rely upon such language/compiler support. Instead, a mechanism is needed to generate verification conditions for specifications from general MSIL.

Compiling Functional Types to Relational Specifications for Low Level Imperative Code

Nick Benton (Microsoft Research UK - Cambridge, GB)

We describe a semantic type soundness result, formalized in the Coq proof assistant, for a compiler from a simple functional language into an idealized assembly language. Types in the high-level language are interpreted as binary relations, built using both second-order quantification and separation, over stores and values in the low-level machine.

Joint work of: Benton, Nick; Tabareau, Nicolas

Meta Predicate Abstraction for Hierarchical Symbolic Heaps

Joshua Berdine (Microsoft Research UK - Cambridge, GB)

Automatic program analyses generally require operations to logically weaken candidate invariants; e.g., in finding loop invariants, procedure preconditions, concurrently accessed resource invariants.

We propose a new technique to define these operations, and apply the technique in the context of a logical domain for expressing complex hierarchically nested data structures, with arbitrarily nested disjunction—to which this technique is particularly well suited.

The basis of this approach is to conceptually blur the distinction between the assertion logic's proof theory and model theory, using the way the syntactic structure of formulae reflects the graph structure of the represented models. This enables a form of predicate abstraction, where predicates expressed in first-order

logic with transitive closure are evaluated globally over syntactic occurrences of terms within formulae. The formulae are then weakened using logical implications that are enabled or disabled based on the predicate evaluations on formulae terms.

Our approach provides a general, declarative means of defining logical-weakening operations, which simulate the abstraction operations in existing separation logic based analyses. Where previously complicated syntactic side-conditions on algorithmic rewrite systems were used, we express operations in terms of predicate evaluations. Our parametric definition scheme enables operation specialization, e.g., to a class of programs, or even to different control points or analysis phases of the same program. These definitions are a key stepping stone to an analysis that automatically refines the degree of logical weakening, perhaps guided by counterexamples—which has been a key advance in automatic verification for arithmetic and control-flow properties.

Keywords: Shape Analysis, Separation Logic

Joint work of: Berdine, Josh; Emmi, Mike

Reduction in End-User Shape Analysis

Bor-Yuh Evan Chang (University of Colorado, US)

Xisa is a shape analyzer parametrized by user-provided data structure definitions that guide the analysis abstraction. These definitions come in the form of data structure validation code, which are interpreted as inductive definitions in separation logic. The user may provide different definitions that correspond to equivalent or related concretizations, which makes the framework quite expressive. However, as a consequence, we must deal with multiple possible abstractions at any point during the program analysis. In this talk, we observe that interestingly, we can derive lemmas about related abstractions by applying and reusing our parametric abstract domain on the user-provided data structure definitions (that will then be its input for the program analysis). Such lemmas are needed by a reduction operator for Xisa that converts between abstractions during the program analysis phase.

Minimal Ownership for Active Objects

David Clarke (K.U. Leuven, BE)

Active objects offer a structured approach to concurrency, encapsulating both unshared state and a thread of control. For efficient data transfer, data should be passed by reference whenever possible, but this introduces aliasing and undermines the validity of the active objects. This paper proposes a minimal variant of ownership types that preserves the required race freedom invariant yet

enables data transfer by reference between active objects (that is, without copying) in many cases, and a cheap clone operation where copying is necessary. Our approach is general and should be adaptable to several existing active object systems.

Keywords: Ownership, concurrency, uniqueness, active objects

Joint work of: Clarke, David; Wrigstad, Tobias; Ostlund, Johan; Johnsen, Einar Broch

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2010/2437

Full Paper: http://www.springerlink.com/content/d502311675h73147/

Observational Purity by Underspecification

David Cok (Eastman Kodak Comp. - Rochester, US)

It is helpful in specification to use methods of a program itself in its own specifications. Specification languages currently require such methods be at least weakly pure. This is too constraining. We describe how a combination of underspecification and the separation of program state into disjoint regions can allow the use of observational purity. This relaxation of purity requires a compensating specification mechanism for describing and checking any recursive modifications of particular regions of program state.

Keywords: JML, specification, observational purity

Compositional Shape Analysis by means of Bi-Abduction

Dino Distefano (University of London, GB)

This talk describes a compositional shape analysis, where each procedure is analyzed independently of its callers.

The analysis uses an abstract domain based on a restricted fragment of separation logic, and assigns a collection of Hoare triples to each procedure; the triples provide an over-approximation of data structure usage.

Compositionality brings its usual benefits –increased potential to scale, ability to deal with unknown calling contexts, graceful way to deal with imprecision – to shape analysis, for the first time.

The analysis rests on a generalized form of abduction (inference of explanatory hypotheses) which we call bi-abduction. Bi-abduction displays abduction as a kind of inverse to the frame problem: it jointly infers anti-frames (missing portions of state) and frames (portions of state not touched by an operation), and is the basis of a new interprocedural analysis algorithm.

We have implemented our analysis algorithm and we report case studies on smaller programs to evaluate the quality of discovered specifications, and larger programs (e.g., an entire Linux distribution) to test scalability and graceful imprecision.

The need for flexible object invariants

Sophia Drossopoulou (Imperial College London, GB)

Specification and verification of object oriented programs usually features in some capacity the concept of an object invariant, used to describe the consistent states of an object. Unavoidably, an object's invariant will be broken at some points in its lifetime, and as a result, invariant protocols have been suggested, which prescribe the times at which object invariants may be broken, and the points at which they have to be re-established.

The fact that currently available invariant protocols do not handle well some known examples, together with the fact that object invariants and invariant protocols can largely be encoded through methods' pre- and post- conditions has recently raised the question of whether they still have a role to play, or should be replaced by more explicit pre- and post- conditions for methods.

In this paper we argue that invariant protocols express programmers' intuitions, lead to better design, allow more succinct specifications and proofs, and allow the expression of properties which involve many objects in a localised manner. In particular, the resulting verification conditions can be made simpler and more modular through the use of invariant-based reasoning.

We also argue that even though encoding invariant protocols through methods' pre- and post-conditions is possible, such an encoding loses important information, and as a result makes specifications less explicit and program evolution (whereby the program evolves after the encoding has taken place) more error-prone. Finally, we show that such encodings often cannot express properties over inaccessible objects, whereas an appropriate invariant protocol can handle them simply.

Proving that non-blocking algorithms don't block

Alexey Gotsman (University of Cambridge, GB)

A concurrent data-structure implementation is considered non-blocking if it meets one of three following liveness criteria: wait-freedom, lock-freedom, or obstruction-freedom. Developers of non-blocking algorithms aim to meet these criteria. However, to date their proofs for non-trivial algorithms have been only manual pencil-and-paper semi-formal proofs. This paper proposes the first fully automatic tool that allows developers to ensure that their algorithms are indeed non-blocking. Our tool uses rely-guarantee reasoning while overcoming the technical challenge of sound reasoning in the presence of interdependent liveness properties.

Keywords: Formal Verification, Concurrent Programming, Liveness, Termination

Automatic Verification of Integer Array Programs

Peter Habermehl (LIAFA University Paris VII, FR)

We provide a verification technique for a class of programs working on integer arrays of finite, but not a priori bounded length. We use the logic of integer arrays SIL to specify pre- and post-conditions of programs and their parts. Effects of non-looping parts of code are computed syntactically on the level of SIL. Loop pre-conditions derived during the computation in SIL are converted into counter automata (CA).

Loops are automatically translated—purely on the syntactical level—to transducers. Pre-condition CA and transducers are composed, and the composition over-approximated by flat automata with difference bound constraints, which are next converted back into SIL formulae, thus inferring post-conditions of the loops. Finally, validity of post-conditions specified by the user in SIL may be checked as entailment is decidable for SIL.

Relational semantics for effect-based program transformations: higher-order store

Martin Hofmann (LMU München, DE)

We give a denotational semantics to an effect system tracking reading and writing to global variables holding values of arbitrary type including effect-triggering functions. Effects are interpreted in terms of the preservation of certain binary relations on the store. The semantics validates a number of effect-dependent program equivalences and can thus serve as a foundation for effect-based compiler transformations. The definition of the semantics requires the solution of a mixed-variance equation which is not accessible to the hitherto known methods. We illustrate the difficulties with a number of small example equations one of which is still not known to have a solution.

Keywords: Denotational semantics, higher-order store, logical relation, effect analysis

Joint work of: Benton, Nick; Beringer, Lennart; Hofmann, Martin; Kennedy, Andrew

See also: to appear in PPDP2009

TVLA for Low-level C

Joerg Kreiker (TU München, DE)

With few exceptions, shape analyses of C programs employ an abstraction, where records are modeled as atomic entities and pointer-valued components of records as binary relations on them.

This is too coarse when programs use pointers to components of records, the address-of operator, pointer arithmetic to compute the head of a record from a pointer to a component, pointer arithmetic to iterate over arrays, or combinations of nested records and arrays. All these features are not uncommon in systems code.

We refined the usual memory model used in shape analyses to be able to deal with such constructs and implemented our analyses in the TVLA framework. We used the implementation to analyze Linux device driver code samples featuring, for instance, arrays of so-called "pprev lists". The results can be used for race detection.

It shows that many of the insights gained in TVLA can be beneficially re-used in the new setting, for example, certain instrumentation predicates and integrity constraints.

Finally, I present a few ideas on abstractions for arrays of dynamically allocated data structures.

Keywords: Shape Analysis, Pointer Arithmetic, System Code, Arrays

Preventing Cross-Type Aliasing: A Problem Description

Gary T. Leavens (University of Central Florida - Orlando, US)

This talk explains the problem of preventing cross-type aliasing and why it is important. The main motivation is to reason about history constraints using static type information, while relaxing the overly strict rule of strong behavioral subtyping, which imposes history constraints on all new methods of all subtypes. To avoid these problems, a mechanism is needed that allows upcasts of objects and borrowing, but prevents cross-type aliasing.

This work was supported in part by NSF grant CNS 08-08913.

Keywords: Types, subtyping, upcast, capability

A comparison of heap models: Ownership, dynamic frames, permissions

K. Rustan M. Leino (Microsoft Research - Redmond, US)

I'd like to provide a comparison of ownership-based techniques (like in Spec#), dynamic frames (like in Dafny), and permission- and abstract-predicate-based techniques (like in separation logic or Chalice). What are pros and cons of each? Can they be combined?

Proving Copyless, Contract-based Message-Passing

Etienne Lozes (ENS - Cachan, FR)

Handling concurrency using locks is tedious and error-prone. One alternative is to use message-passing synchronization instead. I will present a particular use of message-passing for shared memory concurrency where the ownership of the heap region representing the content of a message is lost upon sending, which can lead to efficient implementations. I will define a proof system for a concurrent imperative programming language implementing this idea and inspired by the Singularity operating system research project. The proof system is an extension of separation logic with particular small axioms for message-passing.

A soundness result for this proof system using best local actions, as in abstract separation logic, is feasible, but does not imply that provable programs do not leak memory. I will present a more refined semantics for which I will state a "transfers erasure property" that entails the absence of memory leaks.

Communications are ruled by contracts, similar to session types, and this particular aspect of the programming language plays a very important role in the proof of this last result.

Keywords: Separation Logic, Session Types, Singularity

Joint work of: Lozes, Etienne; Villard, Jules; Calcagno, Cristiano

Using Stereotypes to Reason in Regional Logic

Peter Mueller (ETH Zürich, CH)

I will report on work in progress that simplifies the application of Regional Logic, especially to recursive data structures. Stereotypes are a layer on top of Regional Logic that conveniently hide updates of ghost states and facilitate the application of SMT solvers to the verification of complex heap data structures. We applied Stereotypes to verify a Priority Inheritance Protocol.

Joint work of: Müller, Peter; Rudich, Arsenii

Dynamic encapsulation boundaries: a second order frame rule for region logic

David Naumann (Stevens Institute of Technology, US)

We consider local reasoning in object-based programs via reasoning about explicit footprints of formulas and commands.

Footprints are state-dependent expressions that denote certain location sets.

We give a second order frame rule for soundly hiding a module invariant from clients that respect a dynamic encapsulation boundary.

The boundary is a footprint for the hidden invariant, expressed in client-visible terms.

Keywords: Verification, local reasoning

Joint work of: Naumann, David; Banerjee, Anindya

Abstraction vs Verification : Classes vs Patterns

James Noble (Victoria University of Wellington, NZ)

Verification approaches based on ownership (such as Boogie and JML) are reasonably successful at handling designs that can be represented as individual classes. Unfortunately, object ownership does not work as well where designs involve multiple classes - such as many Design Patterns. In this talk, I'll outline some proposed research on new abstractions that can represent patterns explicitly, involving multiple classes. I hope this may not only make pattern implementations easier to write and reuse, but also support their verification via forms of ownership.

Equivalences of object-oriented program components

Arnd Poetzsch-Heffter (TU Kaiserslautern, DE)

We consider object-oriented program components as units for modular program verification. In particular, we investigate so-called dynamic boxes as components. A dynamic box is a component that can have multiple component instances at runtime.

The implementation of such a box consists of a creation class K and the minimal declaration-closed set of classes and interfaces containing K. At runtime, a box instance C is created by creating a K-object X. C comprises all objects that are directly or indirectly created by X.

The talk investigates the equivalence of boxes in certain classes of program contexts. To abstract from the implementation, we define a behavioral semantics based on the incoming and outgoing messages of a box instance. To prove equivalence, we derive a heap-independent and suitable representation of the component behavior, the so-called behavior denotation. Using program verification techniques, we prove the behavior denotation correct w.r.t. to the operational program semantics. The equivalence proof is based on an adequacy result for the behavioral semantics and a proof that the behavior denotations are equal.

Logical Concurrency Control From Sequential Proofs

Ganesan Ramalingam (Microsoft Research India - Bangalore, IN)

In this talk, we consider the problem of making a sequential library safe for concurrent clients.

Informally, given a sequential library that works satisfactorily when invoked by a sequential client, we wish to synthesize concurrency control code for the library that ensures that it will work satisfactorily even when invoked by a concurrent client (which may lead to overlapping executions of the library's procedures).

Formally, we consider a sequential library annotated with assertions along with a proof that these assertions hold in a sequential execution.

We show how such a proof can be used to derive a concurrency control for the library that guarantees that the library's execution will satisfy the same assertions even when invoked by a concurrent client.

We extend the approach to guarantee linearizability: any concurrent execution of a procedure is not only guaranteed to satisfy its specification, it also appears to take effect instantaneously at some point during its execution.

An interesting open question is how such an approach can be extended to deal with heap allocated data and whether it is possible to derive fine-grained locking schemes in the presence of heap allocated data.

Keywords: Synthesizing concurrency control, atomicity, linearizability

Joint work of: Deshmukh, Jyotirmoy; Ramalingam, G.; Ranganath, Venkatesh Prasad; Vaswani, Kapil

Sequential Verification for Concurrency

Noam Rinetzky (University of London, GB)

Designing, implementing, and verifying concurrent programs is a a difficult and challenging problem.

In this talk, I describe two approaches which allow to lift existing sequential reasoning techniques to the concurrent setting.

Keywords: Sequential reasoning, concurrency, serializability, linearizability, separation logic, TVLA, resilient assertions, hindsight lemma

Joint work of: Rinetzky, Noam; Attiya, H.; O'Hearn, P.W.; Ramalingam, G.; Vechev, M.T.; Yahav, E.; Yorsh, G.

Shape Analysis Applied to C Code

Xavier Rival (ENS - Paris, FR)

Xisa is an abstract interpretation based shape analyzer, which infers invariants for C programs. The analyzer uses data structure definitions provided by the user in the form of inductive definitions.

The analysis process relies on both unfolding (materialization) and folding (widening) of inductive definitions. We will present ongoing work on extending

the subset of the C language handled by Xisa. The C memory model presents many significant challenges. In particular, we formalize the abstraction of pointers, composites (structs and unions), and arrays, and we show how these features affect the transfer functions of the analysis.

Separation logic and the C5 generic collection library

Peter Sestoft (IT University of Copenhagen, DK)

We present fragments of a substantial software library that will be used as case study in a project whose goal is to extend (a version of) separation logic to cover a realistic programming language such as C#.

Keywords: Separation logic, specification, verification, collection library, C#

A glimpse of my Ph.D.: pointer analysis and separation logic

Elodie-Jane Sims (ENS - Paris, FR)

We are interested in modular static analysis to analyse softwares automatically. We focus on programs with data structures, and in particular, programs with pointers. The final goal is to find errors in a program (problems of dereferencing, aliasing, etc) or to prove that a program is correct (regarding those problems) in an automatic way.

Isthiaq, Pym, O'Hearn and Reynolds have recently developed separation logics, which are Hoare logics with assertions and predicates language that allow to prove the correctness of programs that manipulate pointers. The semantics of the logic's triples ($\{P\}C\{P'\}$) is defined by predicate transformers in the style of weakest preconditions.

We expressed and proved the correctness of those weakest preconditions (wlp) and strongest postconditions (sp), in particular in the case of while-loops. The advance from the existing work is that wlp and sp are defined for any formula, while previously existing rules had syntactic restrictions.

We added fixpoints to the logic as well as a postponed substitution which then allow to express recursive formulae. We expressed wlp and sp in the extended logic and proved their correctness. The postponed substitution is directly useful to express recursive formulae. For example,

$$nclist(x) = \mu X_v.(x = nil) \vee \exists x_1, x_2.(isval(x_1) \wedge (x \mapsto x_1, x_2 * X_v[x_2/x]))$$

describes the set of memory where x points to a list of integers.

Next, the goal was to use separation logic with fixpoints as an interface language for pointer analysis. That is, translating the domains of those analyses into formulae of the logic (and conversely) and to prove their correctness. One

might also use the translations to prove the correctness of the pointer analysis itself.

We illustrate this approach with a simple pointers-partitioning analysis. We translate the logic formulae into an abstract language we designed which allows us to describe the type of values registered in the memory (nil, integer, booleans, pointers to pairs of some types, etc.) as well as the aliasing and non-aliasing relations between variables and locations in the memory. The main contribution is the definition of the abstract language and its semantics in a concrete domain which is the same as the one for the semantics of for- mulae. In particular, the semantics of the auxiliary variables, which is usually a question of implementation, is explicit in our language and its semantics. The abstract language is a partially reduced product of several subdomains and can be parametrised with existing numerical domains. We created a subdomain which is a tabular data structure to cope with the imprecision from not having sets of graphs. We expressed and proved the translations of formulae into this abstract language.

Keywords: Separation logic, fixpoints, postponed substitution, pointer analysis, shape analysis, abstract interpretation

Full Paper:
http://www.di.ens.fr/∼sims

Nondeterministic Abstract Regular Tree Model Checking

Tomas Vojnar (Brno University of Technology, CZ)

Abstract regular tree model checking (ARTMC) is a generic technique for automated formal verification of various kinds of infinite-state and parameterised systems, including, e.g., parameterised protocols or programs manipulating dynamic, pointer-linked data structures. ARTMC is based on representing infinite sets of configurations of the examined systems by finite tree automata whose efficient manipulation is thus crucial for an overall efficiency of ARTMC. It turns out that a significant bottleneck in dealing with finite tree automata is the need to determinise them when checking inclusion or within the classical automata minimisation procedure. We present several recent works whose aim is to eliminate the need to determinise the automata being handled in ARTMC. In particular, we present methods for an antichain-based inclusion checking on nondeterministic tree automata and efficient methods for reducing the size of such automata based on various types of upward, downward, and composed (bi)simulations. According to our experimental results, these techniques significantly improve the performance of ARTMC (in particular, when applied for verification of tree-manipulating programs).

The talk is based on joint work with Ahmed Bouajjani, Peter Habermehl, and Tayssir Touili from LIAFA, University Paris Diderot—Paris7/CNRS, Parosh Aziz Abdulla and Lisa Kaati from the Uppsala University, and Lukas Holik from FIT, Brno University of Technology.

Counterexample-Guided Focus

Thomas Wies (EPFL - Lausanne, CH)

The automated inference of quantified invariants for heap-manipulating programs is considered one of the next challenges in software verification.

The question of the right precision-efficiency tradeoff for the corresponding program analyses here boils down to the question of the right treatment of disjunction below and above the universal quantifier. In the closely related setting of shape analysis one uses the focus operator in order to adapt the treatment of disjunction (and thus the efficiency-precision tradeoff) to the individual program statement. One promising research direction is to design parameterized versions of the focus operator which allow the user to fine-tune the focus operator not only to the individual program statements but also to the specific verification task. We carry this research direction one step further. We fine-tune the focus operator to each individual step of the analysis (for a specific verification task). This fine-tuning must be done automatically. Our idea is to use counterexamples for this purpose. We realize this idea in a tool that automatically infers quantified invariants for the verification of a variety of heap-manipulating programs. This is joint work with Andreas Podelski.

Keywords: Quantified invariants of heap programs, shape analysis, abstraction refinement

Phalanx: Parallel Checking of Expressive Heap Assertions

Greta Yorsh (IBM TJ Watson Research Center - Hawthorne, US)

We present Phalanx - a practical framework for dynamically checking expressive heap properties such as ownership, sharing and reachability.

Phalanx uses novel parallel algorithms to efficiently check heap properties utilizing the available cores in the system.

To debug her program, a programmer can annotate it with expressive heap assertions in JML, which use heap primitives provided by Phalanx.

The framework combines a modified version of the JML compiler with a specialized runtime to efficiently evaluate these assertions using parallel algorithms. The Phalanx runtime has been implemented on top of a production virtual machine.

We applied Phalanx to real world applications in various scenarios, and found expressive heap assertions to be extremely valuable in debugging and program understanding. Further, our experimental results indicate that evaluating heap queries using parallel algorithms can lead to significant performance improvements, often resulting in linear speedups as the number of cores increases.

Keywords: Reachability, ownership, runtime assertion checking, parallel algorithms, garbage collector

Joint work of: Yorsh, Greta; Yahav, Eran; Vechev, Martin; Bloom, Bard