SPACE 2001

Preliminary Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2001)

Organizing and editorial committee:

Fritz Henglein, DOI, The IT University of Copenhagen
John Hughes, Chalmers University of Technology
Henning Makholm, DIKU, University of Copenhagen
Henning Niss, DIKU, University of Copenhagen

IT University of Copenhagen (ITU)
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
World-Wide Web: `http://www.it.edu`

Foreword

These pages contain informal proceedings documenting the talks presented at the workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2001), held Januar 15–16, 2001, in London, UK.

Since the inception of programming languages, increasingly powerful runtime support has been devised for memory management, starting with static memory management, via runtime stacks with explicitly managed heaps and, more recently, garbage collection. Correspondingly, (static) reasoning about and harnessing dynamic memory management for predictability, performance, real-time applications, security, mobility and distribution has become increasingly more challenging.

Highlighted by the advent of region inference in the early 90s, compile-time techniques have been proposed for reasoning about and improving dynamic memory management. Such techniques bear the promise of reducing memory footprints, guaranteeing real-time constraints, meeting resource constraints for embedded devices, etc., and may in the end provide a foundation for the next generation of memory management.

This workshop was organized to bring together researchers addressing static control of dynamic memory management issues at different levels of abstraction and with somewhat differing, but related aims. This was motivated by our sense that progress on reasoning about memory management and arriving at innovative implementation technology benefits from an exchange of research — a "value chain" if you will — on semantics and program logics via type systems and efficient program analysis to language design and exploitation of static (compile-time) information in efficient memory management systems. It is also motivated by our — admittedly more diffuse — sense that progress on memory management is likely to be bound to more general progress on reasoning about programs and, vice versa, to aid the latter by providing a fruitful concrete application area.

Early-on we made a decision to emphasize and foster a workshop atmosphere by avoiding a peer review of paper submissions, minimizing administration (for everybody) by eliminating a registration fee, and setting flexible deadlines for registration and for submission of abstracts and papers. There was neither a requirement to submit full papers on the one hand, nor any page limits on the material submitted on the other hand.

The workshop consisted of 4 invited talks and 12 presentations of 25 minutes' duration each, 6 presentations of 10 minutes' duration, and, not least, numerous opportunities for interaction (lunches, breaks, dinner, final plenary discussion). Close to 50 people registered for participation before we, regretfully, had to close registration for space reasons ("space" in lower-case). The invited talks were given by (in the order given): Mads Tofte and Niels Hallenberg (The IT University of Copenhagen); John Reynolds (Carnegie-Mellon University) and Peter O'Hearn (Queen Mary and Westfield College); Colin Runciman (York University); and Greg Morrisett (Cornell University).

It is important to note that these proceedings are informal and are not to be construed as a (peer-reviewed) publication – which they are simply not. We are sure that many of the contributions will find proper publication elsewhere. Only the future, however, can be the judge as to whether the workshop accomplished its broader goals of contributing to progress on static support for memory management and, more generally, reasoning about programs and putting it to effective use.

Fritz Henglein
SPACE 2001 workshop chairman

Acknowledgements

We would like to thank Lotte Møller, Henrik Reif Andersen, Erik van der Meer and Michael Florentin Nielsen at ITU for their practical help. Special thanks go to Chris Hankin and Steffen van Bakel for their invaluable, extensive help with arranging SPACE 2001 in London and adopting it as a colocated event at POPL 2001; to Imperial College for providing room, facilities and services for the workshop; and to DIKU, University of Copenhagen, for hosting the SPACE 2001 web site and supporting Henning Makholm's and Henning Niss' work as workshop organizers.

Last, but not least, we would like to thank the sponsors of SPACE 2001. The Danish Research Council provided for substantial funding for the workshop under grant no. 9502816 ("DART — Design and Reasoning About Tools"), and the Department of Innovation at ITU provided legal sponsorship and administrative support for the workshop. In the same vein, we would like to acknowledge EAPLS and ACM/SIGPLAN for granting SPACE 2001 in-cooperation status and colocation with POPL 2001.

Fritz Henglein, DOI, The IT University of Copenhagen
John Hughes, Chalmers University of Technology
Henning Makholm, DIKU, University of Copenhagen
Henning Niss, DIKU, University of Copenhagen

Contents

Session 3: 9:00–12:00 (chaired by Henning Niss)

Session 4: 14:00–17:00 (chaired by John Hughes)

Program

	Monday	**Tuesday**
9:00	M. Tofte and N. Hallenberg *Invited talk: Region-based memory management in perspective*	C. Runciman *Invited talk: Heap profiling for theoreticians*
9:45	Coffee	
10:05	S. Helsen *Syntactic type soundness for the imperative region calculus*	S. Monnier *Principled scavenging*
10:30	L. Pareto *Sized region types*	A. Garthwaite *Memory management = Partitioning + Alpha-renaming*
10:55	Coffee	
11:15	N. Mazur *Practical structure reuse for Mercury*	P. Keyngnaert *Conflict graph based allocation of static objects to memory banks*
11:40	A. Mycroft *Statically allocated systems*	A. Bakewell *Looking for leaks*
11:50	M. Elsman *A stack machine for region based programs*	D. Sands *Spikes and ballast: The algebra of space*
12:00	Lunch	
14:00	J. Reynolds and P. O'Hearn *Invited talk: Reasoning about shared mutable data structure*	G. Morrisett *Invited talk: Next generation low-level languages*
14:45	Coffee	
14:05	A. Møller *Verification of data type implementations using graph types and monadic second-order logic*	H. Makholm and H. Niss *Towards a more flexible region type system*
14:30	H. Yang *An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm*	D. Gay *A type system for reference-counted regions*
15:55	Coffee	D. Walker *On linear types and regions*
16:15	M. Hofmann *A type system for controlling heap space and its translation to JavaCard*	
16:40	C. Calcagno *Program logics in the presence of garbage collection*	Discussion
16:50	D. Clarke *On deleting aggregate objects*	

Region-based memory management in perspective

Mads Tofte Niels Hallenberg
The IT University of Copenhagen
DENMARK

Abstract

Originally, Region-based Memory Management was conceived as a purely theoretical idea intended to solve a practical problem in the context of Standard ML, namely inventing a dynamic memory management discipline which is less space demanding and more predictable than generic garbage collection techniques, such as generational garbage collection.

Over the subsequent eight years, considerable effort was devoted to refining the idea, proving it correct and finding out whether it worked in practice. The practical experiments revealed weaknesses, which led to new program analyses, new theory and yet more experimentation. In short, we have sought to work on memory management as an experimental science: the work should be scientific in the sense that it should rest on a solid mathematical foundation and it should be experimental in the sense that it should be tested against competing state-of-the-art implementation technology.

The purpose of this talk is to step back and consider the process as a whole. What does it take to develop a new implementation technology and solid foundations to go with it? What might one hope to gain? What were the risks involved? What successes have been achieved? What failures have there been? What is the next logical step?

Embedded in the talk will be an overview of the main idea of region inference and their implementation in the ML Kit with Regions, intended for those not already familiar with region inference.

The ML Kit with Regions web-site: `http://www.it.edu/research/mlkit/`

Syntactic type soundness for the imperative region calculus

Simon Helsen
Institute for Computer Science
University of Freiburg
GERMANY

Abstract

In previous work, we defined a structural operational semantics for the region calculus of Tofte and Talpin. This lead to simple syntactic soundness proofs for region inference based on a general technique of Wright, Felleisen, and Harper. Part of the simplicity of that framework is due to its storeless formulation. In this talk, we fill in some missing links: first, we show that the original big-step semantics of Tofte and Talpin is semantically equivalent to our small-step semantics. Second, we extend our semantics to include an explicit store. And third, we include explicit (ML-style) references in our framework. The proofs become slightly more involved, but pose no conceptual difficulties.

(This is joint work with Cristiano Calcagno and Peter Thiemann.)

Sized region types

Lars Pareto
Computing Science Dept.
Chalmers University of Technology
SWEDEN

Practical structure reuse for Mercury

Nancy Mazur
Dept. of Computer Science
K.U. Leuven
BELGIUM

Abstract

In previous work we developed a module based analysis to obtain structure reuse for the logic programming language Mercury. It also gave preliminary results of a prototype implementation of the analysis. The results being really promising, we are developing a full implementation of the analysis system within the Melbourne Mercury compiler. In this work we present this implementation as a five step analysis and discuss different practical issues that have an effect on the precision of the reuse analysis, on the speed of the analysis, and finally on the obtained speedup and memory usage of the optimised programs.

(Joint work with Peter Ross, Gerda Janssens and Maurice Bruynooghe.)

Statically allocated systems

Alan Mycroft
Cambridge University Computer Laboratory
UK

Abstract

Static allocation of variables, processes and the like is a highly desirable property for languages which compile to hardware. We consider some properties which guarantee static allocatability.

See `http://www.cl.cam.ac.uk/users/am/research/sa` for more details.

A stack machine for region based programs

Martin Elsman
Dept. of Mathematics and Physics
The Royal Veterinary and Agricultural University
DENMARK

Abstract

In this talk we present a stack based abstract machine, called the Kit Abstract Machine, as a target for compiling Standard ML programs with the ML Kit with Regions. Although similar to the abstract machines used as targets for other ML compilers, such as OCaml and Moscow ML, the Kit Abstract Machine is different in that it has no garbage collector. Instead, the Kit Abstract Machine has instructions for allocating regions, deallocating regions, and allocating memory in regions.

Region based programming is promising in the area of embedded systems with real time requirements to the software running and where memory is a limited resource. Region based programming is also promising in situations where programs run shortly and are executed often; in such situations region inference can provide very good cache behavior and thus high efficiency. This latter situation arises often in server based web applications where clients requests the execution of programs on a single server.

To get practical experience with the above scenarios, we have started two projects. The Palm ML project (PaML) focuses on using the ML Kit for developing Palm Pilot applications with extensive event-based computing. The SMLserver project focuses on developing a framework for running web-server applications built with the ML Kit. Both projects make use of the Kit Abstract Machine.

(Joint work with Niels Hallenberg and Ken Friis Larsen.)

The ML Kit with Regions web-site: `http://www.it.edu/research/mlkit/`

Reasoning about shared mutable data structure

John Reynolds
School of Computer Science
Carnegie Mellon University
USA

Peter O'Hearn
Dept. of Computer Science
Queen Mary & Westfield College
UK

Abstract

We describe an extension of Hoare logic to permit reasoning about shared mutable data structure.

The simple imperative language is extended with commands (not expressions) for accessing and modifying shared mutable structures, and with nondetermininstic allocation, explicit deallocation, and unrestricted address arithmetic. Semantically, this extension involves separating the state of the computation into a store that maps variables into integers, and a heap that maps location (a subset of the integers) into integers.

Assertions are extended by introducing an independent conjunction operation that asserts that its subformulas hold for disjoint parts of the heap. Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of data structures with controlled sharing.

Although address arithmetic is unrestricted, the nondeterminacy of storage allocation insures that valid specifications never describe programs that violate record or array boundaries.

Verification of data type implementations using graph types and monadic second-order logic

Anders Møller
BRICS
University of Aarhus
DENMARK

Abstract

We present a new framework for verifying partial specifications of programs in order to catch type and memory errors and check data structure invariants. Our technique can verify a large class of data structures, namely all those that can be expressed as *graph types*. Earlier versions were restricted to simple special cases such as lists or trees. Even so, our current implementation is as fast as the previous specialized tools.

Programs are annotated with partial specifications expressed in Pointer Assertion Logic, a new notation for expressing properties of the program store. We work in the logical tradition by encoding the programs and partial specifications as formulas in monadic second-order logic. Validity of these formulas is checked by the MONA tool, which also can provide explicit counterexamples to invalid formulas.

Other work with similar goals is based on more traditional program analyses, such as shape analysis. That approach requires explicit introduction of an appropriate operational semantics along with a proof of correctness whenever a new data structure is being considered. In comparison, our approach only requires the data structure to be abstractly described in Pointer Assertion Logic.

(Joint work with Michael I. Schwartzbach.)

An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm

Hongseok Yang
Dept. of Computer Science
University of Illinois at Urbana-Champaign
USA

Abstract

Reasoning about programs manipulating pointers has been considered as difficult not because of the lack of formalisms for verifying pointer programs, but because of the significant increase in the complexity of proofs in each formalism over an informal argument. Recently, there has been significant development in handling the complexity by exploiting locality of memory access within a code fragment. Reynolds introduced a pointer logic based on a "spatial" interpretation, where different parts of a formula refer to different area of memory; this holds the promise of managing complexity of aliasing information. O'Hearn proposed a "tight" interpretation of Hoare triples which reflects locality of memory access: in addition to the usual requirement of Hoare triples, a command is required to access only those memory cells "mentioned" in a precondition. These two ideas are combined in the Frame Introduction rule, proposed by Ishtiaq and O'Hearn in BI pointer logic, to exploit locality of memory access in verification of pointer programs. Frame Introduction supports two stage reasoning for a piece of code: first reasoning about a local fact for the piece of code, and then deriving a global fact from the local one by incorporating all other unaccessed cells on the heap. O'Hearn calls this style of reasoning local reasoning. In this talk, we show the promise of local reasoning in BI pointer logic by an example: the Schorr-Waite graph marking algorithm. Our verification gives an evidence that even in a program with no clear separations of data structures, the locality of memory access can still be exploited in a formal proof with Frame Introduction and BI multiplicative connectives and that the resulting verification becomes significantly simpler.

A type system for controlling heap space and its translation to JavaCard

Martin Hofmann
Laboratory for Foundations of Computer Science
University of Edinburgh
UK

Abstract

JavaCard is a subset of Java intended for programming smart cards. The most striking difference to Java proper is the absence of garbage collection (to reduce overhead) and of explicit deallocation (for the sake of safety). Accordingly, every object created remains in memory throughout the lifetime of the program. Given the limited amount of memory on a smart card it therefore appears to be sensible to reuse existing objects whenever possible rather than creating new ones.

To that end we have adapted an existing type system for a linear functional language (M.H. ESOP'00) with inductive datatypes to admit a translation to JavaCard. The main result is that the heap usage of translated programs is statically bounded and that their behaviour agrees with an obvious functional semantics.

The main technical novelties beyond our previous work (M.H. ESOP'00) are the inclusion of general (first-order) recursive datatypes, arrays, string literals, as well as a tentative version of "read-only types" which relax the strict linearity regime in the case of non-modifying access and provide a controlled amount of aliasing for recursive datastructures (eg representation of certain binary trees as dags).

Our approach to read-only types extends and unifies parts of existing work by Wadler-Odersky (observer types), Kobayashi (quasi-linear types), and O'Hearn-Pym (logic of bunched implication).

(Joint work with David Aspinall.)

Program logics in the presence of garbage collection

Cristiano Calcagno
Queen Mary
University of London
UK

Abstract

Garbage collection relieves the programmer of the burden of managing dynamically allocated memory, by providing an automatic way to reclaim unneeded storage. This eliminates or lessens program errors that arise from attempts to access disposed memory, and generally leads to simpler programs. One might therefore expect that reasoning about programs in garbage collected languages would be much easier than in languages where the programmer has more explicit control over memory. But existing program logics are based on a low level view of storage that is sensitive to the presence or absence of unreachable cells, a view that is not invariant under garbage collection, and Reynolds has pointed out that the Hoare triples derivable in these logics are even incompatible with garbage collection. We present a semantics of program logic assertions based on a view of the heap as finite, but extensible; this is for a logical language with primitives for dereferencing pointer expressions. The essential property of the semantics is that all propositions are invariant under operations of adding or removing garbage cells; in short, they are garbage insensitive.

On deleting aggregate objects

David Clarke
School of Computer Science and Engineering
University of New South Wales
AUSTRALIA

Abstract

We describe the intuitions underlying a typed object calculus which makes explicit the nesting between objects. The calculus is based on Abadi and Cardelli's object calculus extended with regions. Regions have properties describing their nesting and the bounds on their access. Regions are used not only in a stack-based manner, but also to store an object's private implementation. This creates opportunities to improve memory management. In particular, the calculus allows the entire private implementation of an aggregate object to be deleted when the interface to the aggregate becomes garbage. The calculus also allows entire aggregate object to be allocated in a stack-based manner.

Heap profiling for theoreticians

Colin Runciman
Dept. of Computer Science
University of York
UK

Abstract

Heap profiling began as an improvised solution to an urgent practical problem: if a program demands too much memory, which program components should be modified to save the most space? A basic heap profiler takes a complete census of live heap memory at intervals throughout a computation, and summarises this data graphically using various classifications of memory – most of them linked to program components. (Fancier two-pass methods involving the examination of garbage are needed for the fullest classification of heap cross-sections.) The programmer gazes alternately at profile and program until inspiration strikes and a modified program runs in the available space.

Such pragmatism seems a long way from abstract theories used to analyse and govern the rules of memory management. What exactly is a heap profile? Does it have a semantics? How are different kinds of profile related? What is the relative power of the methods used to extract them? Is there a theory of heap profiling? Has it any connection with other theories to do with memory space? These are the sorts of questions to be addressed in this talk.

Principled scavenging

Stefan Monnier
Dept. of Computer Science
Yale University
USA

Abstract

Proof-carrying code and typed assembly languages aim to minimize the trusted computing base by directly certifying the actual machine code. Unfortunately, these systems cannot get rid of the dependency on a trusted garbage collector. Indeed, constructing a provably type-safe garbage collector is one of the major open problems in the area of certifying compilation.

Building on an idea by Wang and Appel, we present a series of new techniques for writing type-safe stop-and-copy garbage collectors. We show how to use intensional type analysis to capture the contract between the mutator and the collector, and how the same method can be applied to support forwarding pointers and generations. Unlike Wang and Appel (which requires whole-program analysis), our new framework directly supports higher-order functions and is compatible with separate compilation; our collectors are written in provably type-safe languages with rigorous semantics and fully formalized soundness proofs.

(This is joint work with Zhong Shao and Bratin Saha.)

Memory management = Partitioning + Alpha-renaming

Alex Garthwaite
Sun Microsystems Labs
USA

Abstract

Safe languages like Java and ML are an important advance in the construction of correct, robust, and scalable applications. By providing a strong type system and runtime services like automatic memory management, these languages eliminate whole classes of programming errors. However, programs written in these safe languages must rely on the correctness of their underlying runtime services.

This reliance on the correctness of these underlying services makes formal frameworks for modelling their correctness important. Recent work by Greg Morrisett, for example, on operational semantics that model the interaction of memory management with the supported language has shown the correctness of collection techniques for partitioning the heap based on reachability. While representing an important advance, however, his lambda-gc framework ignores several important aspects of memory management:

- the correctness of allocation policies.
- the fact that collection is more than a partitioning problem; for example, copying collection tecniques also must correctly perform alpha-renaming on the values of the heap as these values are moved during collection.
- the way in which the language and services interact through read and write barriers.

Because of these omissions, the lambda-gc framework is not able to distinguish among different tracing collection techniques, and has a limited ability to model the behavior of generational, incremental, and concurrent collection techniques.

Our work extends the lambda-gc framework to model these aspects. In particular, we extend the operational semantics to include:

- the making of the tracing of references in heap values explicit.
- the introduction of mapping functions that are built up during the course of collection.
- the use of these mapping functions in both the collection's and the supported language's operational semantics.

Using these extensions, we show that specific tracing techniques may be specified including copying, mark-sweep, mark-compact, and Baker-style incremental copying collection. Finally, we extend the set of properties we can prove about these techniques to include the fact that they correctly rename all references for objects moved during collection.

Conflict graph based allocation of static objects to memory banks

Peter Keyngnaert
Dept. of Computer Science
K.U. Leuven
BELGIUM

Abstract

Several architectures, in particular those specifically designed for digital signal processing, have a memory structure that consists of a number of banks with different characteristics (waitstate, size, ...). There may also exist constraints on the accessibility of these banks, as some bank combinations can be accessed in parallel, while others can not. As memory access conflicts lead to pipeline stalls, the assignment of the data objects of a program to the set of memory banks is crucial with respect to a program's execution speed. Programmers usually do the assignment of the static objects manually. We present a method to automate this process at/post link-time, as the linker is the first moment at which both the entire program as well as the target architecture's characteristics are fully known. Based upon statistics drawn from an execution trace of the program, an ordering of conflicts is derived according to the possible execution time penalties they generate. By allocating the objects of those conflicts that have the most negative impact on the program execution time first, a decent allocation can be derived automatically.

(Joint work with Bart Demoen, Bjorn De Sutter, Bruno De Bus and Koen De Bosschere, still in a rather early stage.)

Looking for leaks

Adam Bakewell
Dept. of Computer Science
University of York
UK

Abstract

Implementations of programming languages that assist the programmer by providing automatic memory management can contain *space leaks*. We define a leaky implementation as one with asymptotically worse space usage than the language standard for some program — the leak witness.

This paper is about proving that an implementation — or rather its operational semantics, expressed as a term-graph rewriting system —- has a space leak. We do this by conducting an automated search for candidate leak witnesses. These are programs that do not terminate and keep on allocating more memory without limit.

To make the problem decidable we restrict ourselves to finding witnesses from a class of looping programs which are evaluated by repeatedly applying the same sequence of rules from the operational semantics. A non-standard unification procedure is used to construct a *super-rule* — the repeating rule sequence. A matching procedure tests whether a super-rule can self feed, producing its own redex and so representing a set of non-terminating programs. An approximation is applied to test if the super-rule will also allocate at each iteration, thus selecting candidate witnesses.

This brute force search is slow, so we employ the idea of *proof planning* to reduce the search space. We also use an approximation to the exact super-rule construction procedure to avoid generating multiple solutions.

The search technique is applied to variants of a simple call by name operational semantics for Core Haskell.

Spikes and ballast: The algebra of space

David Sands
Computing Science Dept.
Chalmers University of Technology
SWEDEN

Abstract

The space-usage of lazy functional programs is perhaps the most thorny problem facing programmers using languages such as Haskell. Programmers unable to predict or control the space behaviour of their lazy programs. Even the most advanced programmers, who are able to visualise the space use of their programs, complain that the "state-of-the-art" compilers *introduce* space-leaks into programs that they believe ought to be space-efficient.

Apart from a few high-level operational semantics which claim to model space behaviour, to the best of our knowledge there have been no formal/theoretical/semantics-based approaches to reasoning about space behaviour of programs. Rather than tackling the problem of determining the absolute space behaviour of a program, we study relative space efficiency. We pose the question: when it is space-safe to replace one program fragment by another? To this end we introduce a space-improvement relation on terms, which guarantees that transformations can never lead to asymptotically worse space (heap or stack) behaviour, for a particular model of computation and garbage collection. Space improvement satisfies an interesting and nontrivial collection of algebraic laws which are expressed with the help of syntactic representations of stack and heap-use phenomena: *spikes* and *ballast*.

(Joint work with Jörgen Gustavsson.)

Next generation low-level languages

Greg Morrisett
Dept. of Computer Science
Cornell University
USA

Abstract

High-level languages, such as ML and Java, provide demonstrated safety and software engineering benefits. Nonetheless, today's critical systems are still coded in low-level, unsafe, and error-prone languages such as C. A big reason for this is inertia. But there are also technical reasons. For instance, C provides good programmer control over data representations and memory management, whereas high-level languages do not. This is witnessed by the fact that the run-time system of almost any high-level language implementation is coded in C.

The question is, can we provide a next generation low-level language that provides the programmer control of C and the safety of ML? I claim that, leveraging recent breakthroughs in linear-, region-, and alias-based type systems, we can come pretty close to achieving this goal.

Towards a more flexible region type system

Henning Makholm Henning Niss
Dept. of Computer Science
University of Copenhagen
DENMARK

Abstract

Region-based memory management was proposed by Tofte and Talpin in 1994 as an alternative to garbage collection for call-by-value functional programming languages. It consists of a compile-time analysis that augments a program with explicit deallocation instructions, and a run-time memory manager which implements the ”region” abstraction used by the analysis.

Unfortunately, the original Tofte-Talpin analysis leads to unacceptably large memory use when it is used on programs that use popular programming patterns like tail recursion in the straightforward way. Several proposals for how to deal with this problem have been proposed; most involve doing additional compile-time analysis on the results of the Tofte-Talpin analysis, and all require most actual programs to be rewritten to fit the solution, often in ways that can only be understood by programmers who are intimately familiar with the region model.

The goal of this project is to clean up this situation by replacing the Tofte-Talpin analysis with a stronger analysis that incorporated the earlier proposals. At present we have a theory on paper and thought experiments that indicate that our solution can work in many cases without rewriting the programs at all. An implementation of our system is under construction.

(Joint work: Fritz Henglein, Henning Makholm, and Henning Niss.)

A type system for reference-counted regions

David Gay
Computer Science Division
University of California
Berkeley
USA

Abstract

Region-based memory management systems, where objects are allocated in "regions" and memory is only reclaimed by deleting regions, have traditionally been either

- statically verified through a type system that guarantees that the objects in a region are not accessed after the region is deleted;
- completely unsafe, with no restrictions on where or how regions are deleted.

Our compiler for C with regions, RC, prevents unsafe region deletions by keeping a count of references to each region. There are thus no restrictions on C's type system or where regions are deleted, but there are no static safety guarantees and the reference counting imposes a certain overhead.

To make the structure of a program's region more explicit and to reduce the overhead of reference counting we have extended RC in two ways:

- We have added the concept of a "subregion": if A is a subregion of B then A's lifetime is strictly contained within B's lifetime.
- RC includes type annotations declaring properties of pointers within the program's region hierarchy. Assignments to locations of annotated type are verified either statically or dynamically.

We generalise these annotations in a region type system, similar to the type systems of statically verified region systems, which can represent RC's annotated types. The main novelty of this type system is the use of existentially quantified abstract regions to represent pointers to objects whose region is partially or totally unknown. An analysis of RC programs based on the concepts from this type system allows us to eliminate up to 99% of the checks that would be performed in a purely dynamic system. The extensions and analysis reduce the overhead of reference counting from a maximum of 27% to a maximum of 18% on a collection of eight small to large benchmarks.

On linear types and regions

David Walker
School of Computer Science
Carnegie Mellon University
USA

Abstract

We explore how two different mechanisms for reasoning about state, linear typing and the type, region and effect discipline, complement one another in the design of a strongly typed functional programming language. The basis for our language is a simple lambda calculus containing first-class regions, which are explicitly passed as arguments to functions, returned as results and stored in user-defined data structures. In order to ensure appropriate memory safety properties, we draw upon the literature on linear type systems to help control access to and deallocation of regions. In fact, we use two different interpretations of linear types, one in which multiple-use values are freely copied and discarded and one in which multiple-use values are explicitly reference-counted, and show that both interpretations give rise to interesting invariants for manipulating regions. We also explore new programming paradigms that arise by mixing first-class regions and conventional linear data structures.

(Joint work with Kevin Watkins.)