A Framework For Efficient Modular Heap Analysis

Ravichandhran Madhavan
EPFL, Switzerland
ravi.kandhadai@epfl.ch

G. Ramalingam
Microsoft Research, India
grama@microsoft.com

Kapil Vaswani
Microsoft Research, India
kapilv@microsoft.com

the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

R. Madhavan, G. Ramalingam, and K. Vaswani. *A Framework For Efficient Modular Heap Analysis*. Foundations and Trends® in Programming Languages, vol. 1, no. 4, pp. 269–381, 2014.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-003-3

Foundations and Trends® in Programming Languages
Volume 1, Issue 4, 2014

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2014, Volume 1, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Foundations and Trends® in Programming Languages
Vol. 1, No. 4 (2014) 269–381

DOI: 10.1561/2500000020

the essence of knowledge

A Framework For Efficient Modular Heap Analysis

Ravichandhran Madhavan
EPFL, Switzerland
ravi.kandhadai@epfl.ch

G. Ramalingam
Microsoft Research, India
grama@microsoft.com

Kapil Vaswani
Microsoft Research, India
kapilv@microsoft.com

Contents

Abstract

Modular heap analysis techniques analyze a program by computing summaries for every procedure in the program that describes its effects on an input heap, using pre-computed summaries for the called procedures. In this article, we focus on a family of modular heap analyses that summarize a procedure's heap effects using a context-independent, shape-graph-like summary that is agnostic to the aliasing in the input heap. The analyses proposed by Whaley, Salcianu and Rinard, Buss *et al.*, Lattner *et al.* and Cheng *et al.* belong to this family. These analyses are very efficient. But their complexity and the absence of a theoretical formalization and correctness proofs makes it hard to produce correct extensions and modifications of these algorithms (whether to improve precision or scalability or to compute more information). We present a modular heap analysis framework that generalizes these four analyses. We formalize our framework as an abstract interpretation and establish the correctness and termination guarantees. We formalize the four analyses as instances of the framework. The formalization explains the basic principle behind such modular analyses and simplifies the task of producing extensions and variations of such analyses.

We empirically evaluate our framework using several real-world $C^\sharp$ applications, under six different configurations for the parameters, and using three client analyses. The results show that the framework offers a wide range of analyses having different precision and scalability.

R. Madhavan, G. Ramalingam, and K. Vaswani. *A Framework For Efficient Modular Heap Analysis*. Foundations and Trends® in Programming Languages, vol. 1, no. 4, pp. 269–381, 2014.
DOI: 10.1561/2500000020.

1

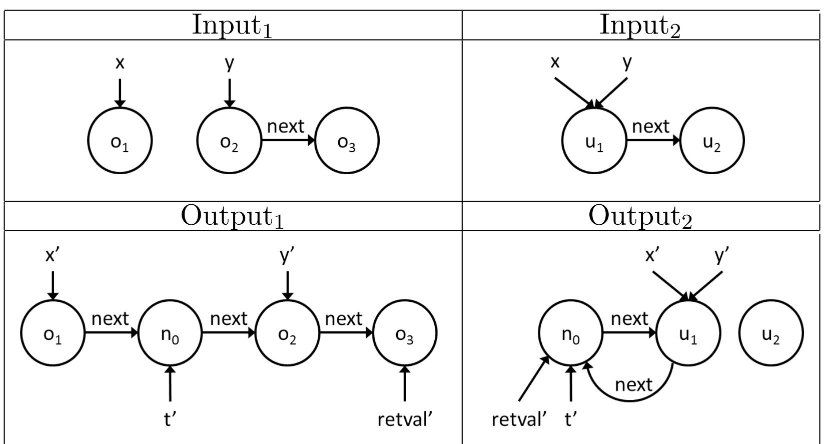
Introduction

Compositional or modular analysis [Cousot and Cousot, 2002] is a key technique for scaling static analysis to large programs. Our interest is in techniques that analyze a procedure in isolation, using pre-computed summaries for called procedures, computing a summary for the analyzed procedure. Such analyses are widely used and have been found to scale well. However, computing such summaries for a heap analysis (or points-to analysis) is challenging because of the aliasing in the input heap. For example, consider the procedure `P` shown in Fig. 1.2(a). Its behaviour on two different input heaps is shown in Fig. 1.2(b) and Fig. 1.2(c). (The heaps are depicted as shape graphs. The input heap is shown at the top and the corresponding output heap at the bottom). It can be seen that the behaviour of `P` varies significantly depending on the aliasing between the variables `x` and `y` in the input heap. A sound summary for `P` should be able to approximate the behaviour of `P` in both these scenarios.

Existing modular heap analyses can be broadly classified into the following categories. (The following classification is not exhaustive. There are modular analyses such as [Nystrom et al., 2004] that cannot be easily classified into any of the categories mentioned. It is also pos-

```
   P (x, y) {
     t = new ();
     x.next = t;
     t.next = y;
     retval = y.next;
   }
```

Figure 1.1: A procedure `P` whose behaviour depends on the aliasing in the input heap.



(a) (b)

Figure 1.2: (a) Output of `P` when `x` and `y` are not aliases in the input heap. (b) Output of `P` when `x` and `y` are aliases in the input heap.

sible to design analyses that belong to more than one of the categories though we aren't aware of any.) (a) Analyses such as [Calcagno et al., 2009] compute *conditional summaries* that are applicable only in the contexts that satisfy certain conditions (e.g., aliasing or non-aliasing conditions). (b) Some analyses such as [Chatterjee et al., 1999], [Dillig et al., 2011], [Jeannet et al., 2010] enumerate all relevant configurations of the input heap belonging to a fixed abstract domain and generate summaries for each configuration. A major challenge with this approach is reducing the number of configurations that are enumerated, which can quickly become intractable, and finding efficient ways of representing them. (c) A few analyses, namely, [Whaley and Rinard, 1999], [Cheng and Hwu, 2000], [Liang and Harrold, 2001], [Lattner et al., 2007], [Buss et al., 2008] compute context-independent summaries that are agnostic to the aliasing in the input heap *without* enumerating the possible configurations of the input heap. To our knowledge, these are the only existing analyses having this property.

The analysis proposed by Whaley and Rinard [Whaley and Rinard, 1999] was later on refined and improved by Salcianu and Rinard [Salcianu and Rinard, 2005]. We will refer to this analysis as the WSR analysis. Adopting the terminology of [Lattner et al., 2007], we will refer to the analysis proposed by Lattner *et al.* as Data Structure Analysis (DSA).

In this article, we consider analyses belonging to the final category. They are interesting for several reasons. (a) They have a number of applications, discussed shortly. (b) The analyses are very efficient. DSA scales to the entire Linux kernel comprising 3 million lines of code in 3 seconds. An optimized version of WSR analysis discussed in [Madhavan et al., 2011] scales to $C^\sharp$ libraries with 250 thousand lines of code. (c) Being modular, they can analyze open programs, libraries, and, in fact, any arbitrary chunk of code without requiring any knowledge of the environment. Moreover, the summaries computed are such that they be refined incrementally when more knowledge about the environment becomes available.

These analyses have been used in a number of applications. Salcianu and Rinard present an application of their analysis to compute the *side-effects* of a procedure, which are the effects of the procedure on the pre-existing state, and use it to classify procedures as *pure* (having no side-effects) or *impure* [Salcianu and Rinard, 2005]. This analysis, referred to as purity analysis, itself has a number of applications.

Whaley and Rinard applied their analysis to identify objects that can be safely allocated in the stack instead of the heap [Whaley and Rinard, 1999]. We use an extension of the WSR analysis to statically verify the correctness of the use of speculative parallelism [Prabhu et al., 2010]. Lattner *et al.* use their analysis to perform *pool allocation* in which different instances of data structures are allocated to distinct memory pools, which enables certain compiler optimizations [Lattner and Adve, 2005b].

However, the complexity of the analyses makes the task of extending and modifying these analyses challenging and time consuming. Questions such as the following often arise while designing new applications based on the analyses and there is no easy way of answering them. Can the scalability of the WSR analysis be improved at the expense of precision? Can DSA be extended to yield more precise results when more time and resources are available? Is it possible to integrate a modular static analysis that requires heap information (such as an *information flow analysis*) with these analyses as typically done in top-down whole program analyses? A sound theoretical formulation of the analyses will greatly aid in answering such questions.

Upon investigating the theoretical basis of these analyses, we realized that, in spite of the apparent dissimilarity between the analyses and the differences in the precision, scalability, and functionality, there are some fundamental ideas common to all of these analyses. This motivated us to develop a parametric framework for designing efficient modular heap analyses. The analyses listed earlier become specific instances of our framework.

We formulate our framework as a parametric abstract interpretation and establish the correctness and termination of the semantics. We present several transformations and optimizations (collectively called as specializations) of our framework and establish their correctness using the standard theory of abstraction interpretation. Our framework with its parametric domains, parametric semantics and several correctness preserving transformations provides a convenient mechanism for obtaining modular heap analyses with different levels of precision and scalability.

We formally establish that the four analyses: [Whaley and Rinard, 1999], [Cheng and Hwu, 2000] [Lattner et al., 2007] (except for the handling of indirect calls), [Buss et al., 2008] are specific instances of our framework. We exclude the analysis proposed in [Liang and Harrold, 2001] (called as *MoPPA*) as it is very similar to [Lattner et al., 2007]. Nevertheless, it can also be expressed as an instance of our framework.

Formulating the analyses as instances of the framework has several advantages. It provides an immediate proof of correctness and termination for the analyses. It also helps understand the abstractions performed by the analyses and identify opportunities for making them more precise or scalable. In fact, we were able to identify several corner cases that were not handled by some of the algorithms and were able to fix them. Since we were unable to find complete formalization of some of the analyses, it is not clear to us if the problems we identified are bugs in the algorithm or gaps in the informal descriptions.

We implemented the framework in our open source heap analysis tool *Seal* (`seal.codeplex.com`). *Seal* is a fairly robust tool which has been used in several program analysis applications. We empirically studied the different configurations of the framework using *Seal*. We present a summary of the results in Chapter 8. The results throw light on the importance of the parameters of the framework by measuring their impact on the precision and scalability of three client analyses.

The framework presented in this article has some limitations. Most importantly, it does not support strong updates on heap locations and path-sensitivity. To our knowledge, all existing modular heap analysis approaches (such as [Dillig et al., 2011], [Jeannet et al., 2010]) that perform strong updates on heap locations enumerate the possible configurations of the input heap. Nevertheless, we believe that both these challenges can be addressed without resorting to enumeration of the input heap configurations. We briefly outline a potential approach in the Future Works section (see Chapter 9).

The following are the main contributions of this article:

- We propose a modular heap analysis framework that is a generalization of a family of existing modular heap analyses. To our knowledge, this is the first attempt to connect and develop a common theory for the different modular heap analyses proposed in the past.
- We formulate our framework as an abstract interpretation and prove the correctness and termination properties.
- We present several correctness preserving transformations that are applicable to all instances of the framework.
- We formalize four existing modular heap analyses as abstractions of instances of our framework, thereby provide a proof of correctness and termination for the analyses. The formalization exposes the relationships between the analyses and provides ways of improving and modifying them.
- We present an empirical evaluation of the framework by analyzing ten open source $C^\sharp$ applications with six different configurations of the framework. We used three client analyses, namely, *Purity and Side Effects Analysis*, *Escape Analysis* and *Call-graph Analysis* to measure the precision and scalability of each of the six configurations.

References

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

Marcio Buss, Daniel Brand, Vugranam C. Sreedhar, and Stephen A. Edwards. Flexible pointer analysis using assign-fetch graphs. In *SAC*, pages 234–239, 2008.

Marcio Buss, Daniel Brand, Vugranam C. Sreedhar, and Stephen A. Edwards. A novel analysis space for pointer analysis and its application for bug finding. *Sci. Comput. Program.*, 75(11):921–942, 2010.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *ECOOP*, pages 665–687, 2012.

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-up shape analysis. In *SAS*, pages 188–204, 2009.

Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 289–298, 2011.

Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32:5:1–5:52, 2010.

Etienne Kneuss, Viktor Kuncak, and Philippe Suter. Effect analysis for programs with callbacks. In *VSTTE*, pages 48–67, 2013.

Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *CC*, pages 125–140, 1992.

Chris Lattner and Vikram Adve. *Macroscopic Data Structure Analysis and Operations*. PhD thesis, University of Illinois at Urbana-Champaign, 2005a.

Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, 2005b.

Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

Ondrej Lhoták. *Program Analysis Using Binary Decision Diagrams*. PhD thesis, 2006.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL*, pages 3–16, 2011.

Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *SAS*, pages 279–298, 2001.

Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *PLDI*, pages 590–601, 2011.

Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. Purity analysis: An abstract interpretation formulation. In *SAS*, pages 7–24, 2011.

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In *SAS*, pages 370–387, 2012.

Ravi Mangal, Mayur Naik, and Hongseok Yang. A correspondence between two approaches to interprocedural analysis in the presence of join. In *ESOP*, pages 513–533, 2014.

Mark Marron, Ondrej Lhoták, and Anindya Banerjee. Programming paradigm driven heap analysis. In *CC*, pages 41–60, 2012.

Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS*, pages 165–180, 2004.

Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *PLDI*, pages 50–61, 2010.

Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.

Alexandru D. Salcianu. Pointer analysis and its applications for java programs. Master's thesis, Massachusetts institute of technology, 2001.

Alexandru D. Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, pages 199–215, 2005.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog*, pages 245–251, 2010.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *PLDI*, page 50, 2014.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

WALA. T. J. Watson libraries for program analysis. URL `https://github.com/wala/WALA`.

John Whaley and Martin C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, page 27, 2014.