University of California
Los Angeles

Expressive Type Systems for Object-Oriented Languages

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Christian Grothoff

2006

The dissertation of Christian Grothoff is approved.

Rupak Majumdar

Todd Millstein

Alan Laub

Jens Palsberg, Committee Chair

University of California, Los Angeles

2006

To Krista

Table of Contents

1 Introduction **1**

1.1 Standard Object-Oriented Type Systems 2

1.2 Types vs. Flow Analysis 4

1.2.1 Plugin Architectures 7

1.2.2 Experimental Results 8

1.2.3 Scaling 0-CFA 12

1.3 Contributions of Our Work 14

2 Encapsulating Objects with Confined Types **16**

2.1 Introduction 16

2.2 A Practical Example of Confinement 19

2.3 Confined Types 21

2.3.1 Anonymity Rules 22

2.3.2 Confinement Rules 24

2.3.3 Discussion and Special Cases 24

2.4 Constraint-Based Analysis 26

2.4.1 Control Flow Analysis 27

2.5 Implementing Confinement Inference: Kacheck/J 29

2.5.1 Example 31

2.5.2 Simplifying Assumptions 32

2.5.3 Constraint Generation 32

2.6 Analysis Results 34

2.6.1 The Purdue Benchmark Suite 34

2.6.2 Confined Types 38

2.6.3 Confinement Violations 40

2.6.4 Confinement with Generics 43

2.6.5 Inferring Access Modifiers 44

2.6.6 Hierarchical Packages 47

2.6.7 Evolution of Confinement 48

2.6.8 Runtime Performance 50

2.7 Containers and Language Extensions 50
2.7.1 Coding for Confinement 50
2.7.2 Improved Language Support 53
2.8 Refactoring for Confinement 55
2.8.1 The Example Program 55
2.8.2 Refactoring: Remove Simple Confinement Violations . . . 56
2.8.3 Refactoring: Remove Widening Violations 60
2.8.4 Refactoring: Summary 60
2.9 Related Work 64

3 Memory Safety through Region Types 69
3.1 Background: X10 72
3.1.1 Overview 72
3.1.2 Places and Activities 73
3.1.3 Arrays, Regions and Distributions 75
3.1.4 Foreach and Ateach 75
3.1.5 Examples 76
3.2 Dependent Types for Regions 77
3.3 Example Programs 79
3.4 The Core Language 82
3.4.1 Syntax 83
3.4.2 Semantics 85
3.4.3 Heap Types 88
3.4.4 Type Equivalence 88
3.4.5 Satisfiability and Entailment 88
3.5 Proof of Type Soundness 93
3.6 Extended Region Algebra 109
3.6.1 Points 109
3.6.2 Regions 110
3.6.3 Places 110
3.6.4 Distributions 111
3.6.5 Additional Point Operations 112
3.6.6 Additional Region Operations 112

3.6.7 Additional Distribution Operations 113
3.7 Implemented Design Features 114
3.7.1 Rank 114
3.7.2 Overloading 115
3.7.3 Defaults 116
3.7.4 Type declarations 117
3.7.5 Decision Procedure for Subtyping 119
3.8 Experiments 120
3.8.1 The ArrayBench Benchmark Suite 120
3.8.2 Region and Place Casts in the Benchmarks 122
3.8.3 Measurements and Assessment 124
3.9 Related Work 127

4 Conclusion 131

A The XTC Framework 135
A.1 The Runabout 135
A.1.1 Using the Runabout 137
A.1.2 Semantics 140
A.2 Representation of Java Code 143
A.2.1 Static Analysis 144
A.2.2 Performance 146
A.2.3 Code Manipulation 147
A.3 Representation of X10 Code 147
A.3.1 Types 148
A.3.2 IR Elements 150
A.3.3 Graph invariants 169
A.3.4 Compiling `X10` to `X10` IR 169
A.3.5 `Array`, `region`, `distribution` and `point` 170
A.3.6 Runtime and runtime library interactions 170
A.3.7 Region types 171
A.4 The Abstract Interpretation Framework 174
A.4.1 Abstract Interpretation 175

A.4.2 Abstract Domain 176
A.4.3 Abstract Semantics 178
A.4.4 Termination and Fixpoint Theory 179
A.4.5 API 181
A.4.6 Implementation 182
A.4.7 Obtaining the Entailment Graph 185
A.4.8 Future Work 186
A.5 License for XTC-X10 186

B ArrayBench Benchmarks 194
B.1 Example Programs 194
B.1.1 init 194
B.1.2 partialinit 194
B.1.3 copy 195
B.1.4 expand 195
B.1.5 shiftleft 195
B.2 Crypt 195
B.3 Crypt-2 200
B.4 DSOR 204

Index 206

References 209

List of Figures

1.1 Illustration of Java's type rules 3
1.2 Illustration of a problem with array subtyping 4
1.3 Example highlighting differences between 0-CFA and CHA 6
1.4 Example code for loading a plugin 9

2.1 Illustration of confinement 17
2.2 Overview of which code must be analyzed for what 19
2.3 Example of hidden widening 23
2.4 Example of static widenings that can be ignored 26
2.5 Example program used to illustrate constraint generation. 28
2.6 Simple example of a confinement violation. 31
2.7 Benchmark characteristics: program sizes. 35
2.8 Benchmark characteristics: member encapsulation. 36
2.9 Percentage of confined types for various benchmarks. 38
2.10 Impact of package-scoping on confinement. 40
2.11 Example of event handler violating anonymity. 41
2.12 Percentage of generic-confined types for various benchmarks. . . . 44
2.13 Percentages of confinable types for various benchmarks. 45
2.14 Confinable types and package-scoping. 47
2.15 Confinement with hierarchical packages. 48
2.16 Changes in confinement between different versions of TomCat . . 49
2.17 Timing data for inferring confined types 50
2.18 Example Hashtable interface. 52
2.19 Java annotations for @Anonymous and @Confined 54
2.20 Doubly-linked list interface. 56
2.21 Doubly-linked list implementation. 57
2.22 Doubly-linked list client code. 58
2.23 Code for another doubly-linked list client. 59
2.24 Refactored doubly-linked list implementation 61
2.25 Refactored client code 62
2.26 Final doubly-linked list implementation 63

2.27 Confinement violation $\mathcal{C}1$. 65
2.28 Example why anonymity does not need preservation in subtypes . 65
2.29 Example for allowed widening between confined types 66

3.1 Overview of X10 Places and Activities, simplified from [CDE05]. . 73
3.2 Example programs: init. 80
3.3 Example programs: partialinit. 80
3.4 Example programs: copy. 81
3.5 Example programs: expand. 81
3.6 Example programs: shiftleft. 82
3.7 Example programs: shift. 82
3.8 Syntax of the core language. 83
3.9 Semantics of the core language (part 1). 86
3.10 Semantics of the core language (part 2). 87
3.11 Semantics of the core language (part 3). 88
3.12 Equivalence rules (1). 89
3.13 Equivalence rules (2). 90
3.14 Type rules (part 1). 91
3.15 Type rules (part 2). 92
3.16 Example for use of rank information 115
3.17 Use of overloading with region types. 115
3.18 Local variable type declarations causing trouble with overloading. 117
3.19 Local variable type declarations can be tedious. 118
3.20 Local variable type declarations – not always necessary. 118
3.21 Overlap computation for Knuth-Morris-Pratt. 123

A.1 The visitee classes with a dedicated method (`dedicated`). 138
A.2 Visitors require `accept` methods in the visitees. 139
A.3 Example for multi-dispatch specification in MultiJava 140
A.4 Runabouts extend the Runabout class to inherit `visitAppropriate`. 141
A.5 Using the Runabout on any kind of visitee. 142
A.6 Example of a `lookup` method. 143
A.7 Example visit method 145

A.8 Interaction between flyweight instructions and buffer 145
A.9 Performance evaluation for Kacheck/J 146
A.10 The TYPE instruction 153
A.11 The CONST instruction 153
A.12 The IS_INSTANCEOF instruction 154
A.13 The IS_DUMMY instruction 154
A.14 The IS_NOT instruction 155
A.15 The IS_AND instruction 155
A.16 The IS_OR instruction 155
A.17 The REGION instruction 156
A.18 The INVOKEVM instruction 157
A.19 The CAST instruction 157
A.20 The IF instruction 158
A.21 The ENTER instruction 159
A.22 The RETURN instruction 159
A.23 The GET instruction 160
A.24 The PUT instruction 160
A.25 The ANCHOR instruction 161
A.26 The CALL instruction 162
A.27 The FCALL instruction 162
A.28 The FINISH instruction 163
A.29 The ASYNC instruction 164
A.30 The END instruction 165
A.31 The FOR instruction 166
A.32 The ATOMIC instruction 167
A.33 The CONDITIONAL-BLOCK instruction 168
A.34 The CEND and END instructions 168
A.35 The `Engine` API (simplified). 181
A.36 The `TransferFunction` interface (slightly simplified). 182
A.37 The `AbstractValue` interface (slightly simplified). 182
A.38 Entailment graph with potential exponential execution time . . . 184
A.39 Ordering that achieves linear execution time 185

List of Tables

1.1 Experimental results for devirtualization effectiveness 11
1.2 Time and space consumption for the flow analyses 14

2.1 Anonymity rules. 22
2.2 Confinement rules. 24
2.3 The constraints generated from the example in Figure 2.5. 29
2.4 The Purdue Benchmark Suite (PBS v1.0). 37
2.5 Statistics for the benchmarks. 39
2.6 Number of confined and confinable classes. 46
2.7 Time required for the analysis. 51
2.8 Alternative confinement rules 53

3.1 Benchmark characteristics 121
3.2 Numbers of dynamic checks required for the benchmarks. 124
3.3 Features of HPC languages. 127
3.4 HPC language terminology overview. 128

Acknowledgments

The work on confined types was supported by grants from DAAD, Lockheed-Martin, CERIAS, and NSF (CCR–9734265). I want to thank Ben Titzer and Theodore Witkamp for developing the modifier inference package, and Mario Südholt as well as the members of IFIP WG2.4 for helpful comments. Some of the programs from the Purdue Benchmark Suite originate from the Ashes suite; thanks to the Sable research group at McGill for making these available.

Thanks to Christopher Donawa and Rajkishore Barik for providing me with performance data on the cost of bounds-checking in Java and X10. I thank Neal Glew for finding a bug in a previous version of the type rules. Vivek Sarkar wrote the original X10 version of the SOR benchmark. X10 is being developed in the context of the IBM PERCS (Productive Easy-to-use Reliable Computing Systems) project, which is supported in part by DARPA under contract No. NBCH30390004.

Chapter 2 is an extended version of [GPV01] that was joint work with Jens Palsberg and Jan Vitek. Chapter 3 is the result of a collaboration with Jens Palsberg and Vijay Saraswat. Some of the material in the introduction is closely related to work done with Neal Glew and Jens Palsberg [GPG05]. Finally, the description of parts of the X10 language in Section 3.1 is a condensed version of joint work done with Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Allan Kielstra, Vivek Sarkar and Christoph Von Praun [CDE05]. I am grateful for the collaboration with all of these co-authors and thank them for their contributions to this thesis.

I thank my committee for discussions about and constructive comments on this work. I am particularly grateful to my wife Krista for editing and feedback. Finally, I thank my advisor Jens Palsberg for his advice and friendship.

Vita

1977	Born, Wuppertal, Germany.
2000	Diplom II (Mathematics), BUGH Wuppertal.
2001	1. Staatsexamen (Chemistry), BUGH Wuppertal.
2003	M.S. (Computer Science), Purdue University.
2000-2005	Research Assistant, Computer Science Department, Purdue University.
2005-2006	Research Assistant, Computer Science Department, UCLA.

Publications

Krista Bennett, Christian Grothoff, Tzvetan Horozov and Ioana Patrascu. "Efficient Sharing of Encrypted Data". In *Proceedings of the 7th Australasian Conference on Information Security and Privacy (ACISP 2002).*, pages 107–120. Springer-Verlag (LNCS 2384), 2002.

Krista Bennett and Christian Grothoff. "gap – Practical Anonymous Networking. In *Designing Privacy Enhancing Technologies (PET 2003).*, pages 141–160. Springer-Verlag (LNCS 2760), 2003.

Ronaldo A. Ferreira, Christian Grothoff and Paul Ruth. "A Transport Layer Abstraction for Peer-to-Peer Networks". In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (GRID 2003)*, pages 398–403, IEEE Computer Society, 2003.

Christian Grothoff. "An Excess-Based Economic Model for Resource Allocation in Peer-to-Peer Networks". In *Wirtschaftsinformatik*, 3-2003, pages 285–292, June 2003.

Christian Grothoff. "Walkabout revisited: The Runabout". In *Proceedings of the European Conference on Object-oriented Programming (ECOOP 2003)*, pages 103-125, Springer-Verlag (LNCS 2743), 2003.

Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi and Jan Vitek. “Engineering a Customizable Intermediate Representation”. In *Science of Computer Programming*, Volume 57 Issue 3, pages 357–378, Elsevier 2005 (supercedes “Engineering a Customizable Intermediate Representation” in *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 2003)*, 2003).

Christian Grothoff, Krista Grothoff, Ludmila Alkhutova, Ryan Stutsman and Mikhail Atallah. “Translation-based steganography”. In *Proceedings of the Information Hiding Workshop (IH 2005)*, pages 219–233, Springer-Verlag, 2005.

Neil Glew, Jens Palsberg and Christian Grothoff. “Type-Safe Optimization of Plugin Architectures”. In *Static Analysis, 12th International Symposium (SAS 2005)*, pages 135–154, Springer Verlag (LNCS 3672), 2005.

Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vijay Saraswat, Vivek Sarkar and Christoph Von Praun. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2005)*, pages 519–538, 2005.

Mangala Gowri, Christian Grothoff and Satish Chandra. “Deriving object typestates in the presence of inter-object references”. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2005)*, pages 77–96, 2005.

Christian Grothoff. “Reading File Metadata with extract and libextractor”. In *LinuxJournal 6’2005*, pages 86-88, SCC Publishing, 2005.

Christian Grothoff, Jens Palsberg, and Jan Vitek. “Encapsulating Objects with Confined Types”. In *ACM Transactions on Programming Languages and Systems (TOPLAS 2006)*, pages XX–YY, 2006 (supercedes “Encapsulating Objects with Confined Types” in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2001)*, pages 241–253, 2001).

Ryan Stutsman, Mikhail Atallah, Christian Grothoff and Krista Grothoff. “Lost in Just the Translation. In *Proceedings of the ACM Symposium On Applied Computing (SAC 2006)*, ACM, 2006.

Abstract of the Dissertation

Expressive Type Systems for Object-Oriented Languages

by

Christian Grothoff
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2006
Professor Jens Palsberg, Chair

Expressive type-systems for object-oriented languages can help improve programmer productivity and system performance. This thesis describes type-based solutions for problems that are commonly addressed using flow-analysis. Specifically, this thesis introduces two extensions to standard object-oriented type systems that enable the programmer to concisely specify additional common and useful invariants. A concise specification is facilitated in both cases by using problem-specific constraint systems in the type rules.

The first extension enables programmers to control aliasing of objects and thereby strengthens the encapsulation paradigm in object-oriented languages. Existing object-oriented languages provide little support for encapsulating objects. Reference semantics allow objects to escape their defining scope, and the pervasive aliasing that ensues remains a major source of software defects. Chapter 2 introduces a type system that can guarantee *confinement* – the property that all instances of a given type are encapsulated in their defining package Chapter 2 presents `Kacheck/J`, a tool for inferring confined types for large Java programs. Our goal is to develop practical tools to assist software engineers, thus we focus on simple and scalable techniques. Confinement can be used to identify accidental leaks of sensitive objects, as well as for compiler optimizations. We report on the analysis of a large body of code and discuss language support and refactoring

for confinement.

The second extension focuses on memory safety, specifically for accesses to distributed arrays. A fundamental concept for programming with these arrays is the use of regions. A region is a set of points, and enables the programmer to give high-level specifications for the shape of arrays and for iterations over sets of indices. Arrays over regions were introduced in ZPL in the late 1990s and later adopted in Titanium and X10 as a means of simplifying the programming of high-performance software. While convenient, regions do not eliminate the risk of memory safety violations in the form of accesses outside of the bounds of arrays. Until now, language implementations have resorted to checking array accesses dynamically or to warning the programmer that bounds violations lead to undefined behavior. Chapter 3 shows that a type system for a language with arrays over regions can guarantee that array bounds violations cannot occur. Our type system uses dependent types and enables safety without dynamic bounds checks. We have developed a core language and a type system, proven type soundness and settled the complexity of the key decision problems.

A prototype implementation which embodies the ideas of our core language has been developed for X10, a new language for high performance computing. We have implemented a type checker for our variant of X10 and experimented with a variety of benchmark programs. Some of the internals of the implementation of both type system extensions are covered in Appendix A.

CHAPTER 1

Introduction

Correctness and performance are fundamental goals in software engineering. This thesis will demonstrate that sophisticated type systems can be an important tool in helping programmers achieve these goals. Type systems combine the benefits of program verification and static analysis techniques. Specifically, they give the programmer certain correctness guarantees and at the same time enable the compiler to generate faster code. Successful type systems allow programmers to concisely declare important properties that are interesting for the correctness of the program and at the same time also convey new information for program optimization. Capturing such properties with type systems is desirable for multiple reasons. First of all, type checking is modular and is, thus, scalable. Types enforce a certain programming discipline which can in turn give certain guarantees, such as memory safety. Type systems usually rely on type rules that state constraints for particular operations. In order to facilitate programming, these rules are generally formulated in ways that allow modular checking. Type system designs attempt to use only a limited amount of context; type-checking usually allows the type checker to independently verify small program units. Furthermore, types often provide useful documentation about the program and in ways that are concise and standardized. The fact that the compiler checks the type annotations also ensures that the documentation provided by the types is always up-to-date.

However, types, as they are used in mainstream languages, have limitations. They can rarely offer the complete correctness guarantees desired in program verification and are at best a complementary tool for program optimization. Program verification tools and optimizing compilers often rely instead on flow analysis as an alternative to types. The information obtained from flow analysis determines if data or control is guaranteed to (not) reach certain locations. Flow analysis can generally give more precise answers about program correctness and enable more aggressive compiler optimizations than types can. However, the results of a flow analysis are harder to predict for the programmer, as the source code usually does not describe the expected results from the flow analysis and changes in one part of the code can unexpectedly impact the flow analysis results of other parts of the application. Flow analysis is also usually harder to scale than equivalently powerful analyses based on types.

This thesis describes two extensions to standard object-oriented type systems that enhance the expressiveness of such type systems. As a result, the type checker can verify the absence of a broader set of programming errors, thus improving programmer productivity. Furthermore, compilers can use the additional type information to generate better code. This use of types improves on the state-of-the-art use of flow analysis for program verification and optimization by improving scalability, usability and predictability.

The capabilities of standard object-oriented type systems will be illustrated later in the introduction, followed by a comparison of the uses of flow analysis and types and their respective advantages and disadvantages. This illustration focuses on a standard compiler optimization, devirtualization and inlining. The approach based on types is called class-hierarchy analysis (CHA), and the flow analysis approach is known as control-flow analysis (0-CFA).

While type systems and flow analysis are often competing techniques, it is sometimes beneficial to combine them. Type systems can use flow analysis in their type rules, and similarly, flow analysis can exploit type information. An example of a type system using flow analysis results is described in Section 2.4.1. Section 1.2.3 is an example of the use of type information in flow analysis.

The remainder of the introduction is structured as follows. Section 1.1 describes features common to most statically typed object-oriented languages; such languages form the basis for the extensions presented in the thesis. Section 1.2 will contrast type-based and flow analysis-based techniques for a well-known compiler optimization, devirtualization and inlining in object-oriented languages. Finally, Section 1.3 highlights the contributions of this thesis.

1.1 Standard Object-Oriented Type Systems

Commonly-used ahead-of-time-compiled object-oriented languages such as C++, C# and Java use type systems to ensure the absence of errors like "message not understood" and "no such field".[1] In these statically typed object-oriented languages, the types in the type system closely mirror the class hierarchy. Figure 1.1 gives some simple examples in Java.

Subclasses understand the same messages and contain the same fields as their parent classes (because they inherit these members). Thus, subtyping and sub-

[1]Note that runtime errors in Java, such as `NoSuchMethodException` can be caught by the type checker and are only thrown at runtime due to reflection or incompatible class changes (version mismatches) that occur after processing of the code by `javac`.

```
class A {
    int f;
}
class B {
}
class C extends A {
    void m() {
        A a = this; // legal: C subtype of A
        a.f++; // type system ensures field f exists
        B b = this; // illegal: C not subtype of B
        b.m(); // illegal: B has no method m
    }
}
```

Figure 1.1: This example illustrates some of Java's type rules. `C` is a subclass of `A` and thus it is legal to assign instances of `C` (here `this`) to variables of type `A`. `C` is not a subclass of `B` so the assignment to `b` is illegal. The method `m()` is not defined in `B` and hence the call to `b.m()` is also illegal. Note that in an untyped language the above code would run despite the presence of type errors.

classing relationships become homomorphisms in these type systems. This simple duality breaks down, however, with the addition of generics or templates to these languages, resulting in a large design space [IPW01, RS06, MBL97, CS98, OAC05, Tho97]. Consequently, some of the largest differences between the type systems of C++, C# and Java are in terms of generics. We will now briefly describe the problem that lies beneath these differences in order to illustrate some of the inherent limitations of type systems.

The fundamental problem with generics was first formalized in [Mad95, MMM90] and is easily illustrated using arrays, which can be thought of as a generic type `Array<T>`. Suppose `S` is a subtype of `T`. The question is, should `Array<S>` be a subtype of `Array<T>`? Surprisingly, the answer is that it depends: if the array is read-only, `Array<S>` can be a subtype of `Array<T>` (covariance). If the array is write-only, `Array<T>` can be a subtype of `Array<S>` (contravariance). And if the array is both read and written to, the two types should be incomparable (invariance). Figure 1.2 illustrates the problem why `Array<S>` cannot be a subtype of `Array<T>` in the case of writes. The design space is further complicated by the possibility of resorting to runtime checks for assignments (as in Java).

Ignoring complications arising from generics, simple object-oriented type systems are often considered to be easy to understand (in part probably because the

```
class T {
}
class S extends T {
    int f;
    void m() {
        S[] a = new S[1];
        T[] b = a; // array subtyping
        b[0] = new T(); // (*)
        S s = a[0]; // magic: T becomes S
        s.f++; // oops
    }
}
```

Figure 1.2: This example illustrates a fundamental problem with generics using the example of array subtyping. Note that while Java allows the assignment `b = a` in this case, the VM performs a costly dynamic check for the assignment in the line marked with `(*)`. Since these costly dynamic checks are required for any `aastore` in Java, Java's array subtyping rule (covariance) is generally considered a design mistake.

confusion between subclassing and subtyping blurs the vision). However, they are inherently limited, in that they cannot prevent a large class of runtime errors. Examples include the absence of operations dereferencing `null`, safe stack allocation of objects, guaranteeing liveness requirements and various forms of controlling aliasing and side-effects. More powerful type systems are constantly being developed. A good general overview of type systems can be found in a recent book by Pierce [Pie02].

1.2 Types vs. Flow Analysis

From the point of view of the compiler, *flow analysis* is the primary alternative to using type information. With types, the type verification stage checks that all statements obey the respective type rules; as a result, it constructs a proof that the invariants described by the types are satisfied. On the other hand, with flow analysis, the compiler analyzes the data and control flow in the program to establish the desired invariants.

In general, flow analysis is theoretically capable of deriving stronger invariants than type systems; however, type systems can be constructed that are equal to a large class of flow analyses [PO95, Pal98, PP01]. Consequently, the real distinction between type systems and flow analysis arises from the kind of usage that is being considered. Type systems are used to reject unsafe code and thus types are commonly used for program verification, whereas a flow analysis must accept all programs, making flow analysis suitable for optimization. Furthermore, type systems are usually defined inductively. As a result, programmers can easily reason about the kinds of programs that the type system will accept. A common way to define a flow analysis, on the other hand, is coinductively, making the results harder to predict for the application programmer. The coinductive definition is also one reason that flow analyses are often more expensive than type-based approaches.

This point is easily illustrated with a technique that is frequently used in optimizing compilers for Java-like languages: inlining of devirtualized function calls. In order to inline a virtual call site, the optimizing compiler first needs to devirtualize the call site – that is establish the existence of a unique receiver. A common type-based technique for devirtualization is called *class hierarchy analysis* (CHA) [DC94, DGC95]. Here the compiler looks at the type hierarchy. The call can be devirtualized if a single class in the applicable part of the hierarchy implements the respective method. An alternative approach based on flow analysis is 0-CFA – *control flow analysis* with zero context. By "context", we refer to the call stack at the site of the virtual function call. In 0-CFA, the analysis is *context insensitive*.

Figure 1.3 gives a simple example that illustrates some of the differences between CHA and 0-CFA. Since CHA only looks at the class hierarchy using the receiver type, it can devirtualize fewer calls. A program verifier based on 0-CFA could determine that the example would always print `01`; an analysis using CHA, on the other hand, would have to assume that it might also output `11`.[2]

The rest of this section will contrast and quantify the differences between type-based and flow analysis-based approaches to inlining in a particular setting, the optimization of software that has a plug-in architecture and that is written in a type-safe object-oriented language. Section 1.2.1 first motivates and illustrates the complications that arise when analyzing code that uses plugins. Section 1.2.2

[2]The example also illustrates the problem with 0-CFA which is addressed with TSMI, a 0-CFA variant for which we also give experimental results. Suppose the `main` method only contained the second print statement. Then an inliner based on 0-CFA would determine that `B.o` can be inlined in `A.n`. However, a type checker would reject the resulting method `n` with the statement `return this.f` because `A` does not have a field `f`. This does not make 0-CFA unsound (because `this` in `A.n` would, during execution, always be of type `B` for which the field `f` exists). However, it makes it impossible to use 0-CFA in a compiler that wants to preserve typeability in its intermediate representation. The TSMI variant of 0-CFA adds additional constraints to 0-CFA which ensure that code will type check after inlining.

```
class A {
    static void main() {
        System.out.print(new A().m());
        System.out.print(new B().m());
    }
    int m() {
        return this.n();
    }
    int n() {
        return this.o();
    }
    int o() {
        return 0;
    }
}
class B extends A {
    int f = 1;
    int m() {
        return this.o();
    }
    int n() {
        return 2;
    }
    int o() {
        return this.f;
    }
}
```

Figure 1.3: The calls to `m` above can be devirtualized by both CHA and 0-CFA. The call to `n` can also be devirtualized by 0-CFA, because `this` in `m` always has type `A`. However, CHA must assume that `this` could also be of type `B` and would thus not devirtualize the call to `n`. The call to `o` in `A` cannot be devirtualized by CHA for the same reason, but the call to `o` in `B` can be devirtualized using both CHA or 0-CFA.

gives experimental results comparing CHA and 0-CFA. Finally Section 1.2.3 discusses some of the limitations of using control flow analysis in this context. More details about this particular analysis can be found in [GPG05].

1.2.1 Plugin Architectures

In a rapidly changing world, software has a better chance of success when it is extensible. Rather than having a fixed set of features, extensible software allows new features to be added on the fly. For example, modern browsers such as Firefox, Konqueror, Mozilla, and Viola [Wei94] allow downloading of plug-ins that enable the browser to display new types of content. Using plugins can also help keep the core of the software smaller and make large projects more manageable thanks to the resulting modularization. Plugin architectures have become a common approach to achieving extensibility and include well-known software such as Eclipse [Fou04] and Jedit [Pes04].

While good news for users, plug-in architectures are challenging for optimizing compilers. Consider the following typical snippet of Java code for loading and running a plugin.

```
String className = ...;
Class c = Class.forName(className);
Object o = c.newInstance();
Runnable p = (Runnable) o;
p.run();
```

The first line gets the name of a plugin class. The list of plugins is typically supplied in the system configuration and loaded using I/O, preventing the compiler from doing a data-flow analysis to determine all possible plugins. The second line loads a plugin class with the given name. The third line creates an instance of the plugin class, which is subsequently cast to an interface and used.

In the presence of this dynamic loading, a compiler has two choices: either treat dynamic-loading points very conservatively or make speculative optimizations based only on currently loaded classes. The former can pollute the analysis of much of the program, potentially leading to little optimization. The latter can often lead to more optimization, but dynamically-loaded code might *invalidate* earlier optimization decisions, thus requiring the compiler to undo such optimizations. When a method inlining is invalidated by class loading, the runtime must *revirtualize* the call; that is, it must replace the inlined code with a virtual call. Naturally, rolling back optimizations, in particular optimizations such as inlining that often enable additional code transformations, can be costly.

The next section gives experimental results for analyses based on CHA (types) and 0-CFA (flow analysis) that quantify the potential invalidations and suggest how to significantly decrease the number of invalidations. We count which sites are likely candidates for future invalidation, which sites are unlikely to require invalidation, and which sites are guaranteed to stay inlined forever. These numbers suggest that speculative optimization is beneficial and that invalidation can be kept manageable. An optimizing compiler can use these hints to decide whether or not to inline, for which call sites data for fast deoptimization should be preserved, and what subset of sites will have to be actively considered for revirtualization when the application loads plugins.

1.2.2 Experimental Results

Using the plugin architectures Eclipse and Jedit as our benchmarks, we have conducted an experiment that addresses the following questions:

- How many call sites can be inlined?
- How many inlinings remain valid and how many can be invalidated?
- Is there a measurable and significant gain in preanalyzing the plugins that are statically available?

Preanalyzing plugins can be beneficial. Consider the code in Figure 1.4. The devirtualization analysis can see that the plugin calls method `m` in `Main` and passes it an `Main.B2`; since `main` also calls `m` with a `Main.B1`, it is probably not a good idea to inline the `a.n()` call in `m` as it will be invalidated by loading the plugin. The analysis can also see which methods are overridden by the plugin, in case only `run` of `Runnable` is. The analysis must still be conservative in some places, as, for example, with the three statements of the `for` loop, since these could load any plugin. However, it can gather much more information about the program and make decisions based on likely invalidations by dynamically loading the known plugins.

Being able to apply the inlining optimization in the first place still depends on the flow analysis being powerful enough to establish the unique target. Thus, the answer to each of the three questions raised earlier depends on which static analysis is used to determine what call sites have a unique target. We have experimented with four different interprocedural flow analyses, all implemented for Java bytecode (here listed in order of increasing precision):

- Class Hierarchy Analysis (CHA, [DC94, DGC95])
- Rapid Type Analysis (RTA, [BS96, Bac97])
- subset-based, context-insensitive, flow-insensitive flow analysis for type-safe method inlining (TSMI, [GP04]) and

- subset-based, context-insensitive, flow-insensitive flow analysis, commonly known as 0-CFA ([Shi91, PS91]).

```
class Main {
    static Main main;
    public static void main(String[] args) throws Exception {
        main = new Main();
        for (String arg : args) {
            Class c = Class.forName(arg);
            Runnable p = (Runnable) c.newInstance();
            p.run(); // virtual if plugins define multiple run methods
        }
        main.m(new B1()); // can stay optimized for given Plugin
    }
    void m(A a) { a.n(); } // needs to be virtual for given Plugin
    static abstract class A {
        abstract void n();
    }
    static class B1 extends A {
        void n() { }
    }
    static class B2 extends Main.A {
        void n() { }
    }
}
class Plugin implements Runnable {
    public void run() { new Main().m(new Main.B2()); }
}
```

Figure 1.4: Example code loading a known plugin. The Plugin does not modify `Main.main`, which ensures that the call to `main.m()` can remain inlined. If only `Plugin` is loaded, `p.run()` can also be inlined. Pre-analyzing `Plugin` reveals that `a.n()` should be virtual, even if the flow analysis of the code without `Plugin` may say otherwise. Note that for the example, CHA would not be able to inline `a.n()`, while CHA would inline `main.m()`, which could be wrong if a plugin is loaded that subclasses `Main` and updates `Main.main`.

In order to show that deoptimization is a necessity for optimizing compilers for plugin architectures, we also give the results for a simple intraprocedural flow analysis ("local") which corresponds to the number of inlinings that will never have to be deoptimized, even if arbitrary new code is added to the system. The "local" analysis essentially makes conservative assumptions about all arguments, including the possibility of being passed new types that are not known to the analysis. A runtime system that cannot perform deoptimization is limited to the optimizations found by "local" if loading arbitrary plugins is to be allowed.

We use two benchmarks in our experiments:

Jedit 4.2pre13 A free programmer's text editor which can be extended with plugins from `http://jedit.org/`, 865 classes; analyzed with GNU classpath 0.09, from `http://www.classpath.org`, 2706 classes.

Eclipse 3.0.1 An open extensible Integrated Development Environment from `http://www.eclipse.org/`, 22858 classes from the platform and the CDT, JDT, PDE and SDK components; analyzed with Sun JDK 1.4.2 for Linux[3], 10277 classes.

While these are "only" two benchmarks, note that the combined size of SPECjvm98 and SPECjbb2000 is merely 11% of the size of Eclipse. Furthermore, these are the only freely available large Java systems with plugin architectures that we are aware of. Analyzing benchmarks that do not have plugins, such as the SPEC benchmarks, is pointless. We are also not aware of any previously-published results on 0-CFA for benchmarks of this size.

We will use *app* to denote the core application together with the plugins that are available to ahead-of-time analysis. Automatically drawing a clear line between plugins and the main application is difficult considering that parts of the core may only be reachable from certain plugins.

Flow analyses are usually implemented with a form of reachability analysis built in, and more powerful powerful analyses are better at reachability. To further ensure a fair comparison of the analyses, reachability is first done once in the same way for all analyses. Then each of the analyses is run with reachability disabled. This ensures that all analyses work with the same total number of virtual call sites and the same code overall. The initial reachability analysis is based on RTA and assumes that all of *app* is live; in particular, all local plugins are treated as roots for reachability. The analysis determines the part of the library (classpath, JDK) which is live, denoted *lib*, and then we remove the remainder of the library.

The combination *app + lib* is the "closed world" that is available to the ahead-of-time compiler, in contrast to all of the code that could theoretically be dynamically loaded from the "open world". We use the abbreviations:

[3]using the JARs dnsns, rt, sunrsasign, jsse, jce, charsets, sunjce_provider, ldapsec and localedata

Jedit	Can be inlined						Cannot be inlined	
	Remain valid		Can be invalidated					
			By DLCW		By DLOW not DLCW			
	app	lib	app	lib	app	lib	app	lib
Local	682	297	0	0	0	0	20252	7808
CHA	682	297	69	7	18720	6178	1463	1623
RTA	682	297	97	51	18723	6178	1432	1579
TSMI	682	297	99	59	19449	7091	704	658
0-CFA	682	297	103	83	19592	7191	557	534

Eclipse	Can be inlined						Cannot be inlined	
	Remain valid		Can be invalidated					
			By DLCW		By DLOW not DLCW			
	app	lib	app	lib	app	lib	app	lib
Local	15497	472	0	0	0	0	481939	26512
CHA	15497	472	4105	61	366114	20796	111720	5655
RTA	15497	472	9024	169	366169	20797	106746	5546
TSMI	15497	472	11479	439	420029	23097	50431	2976
0-CFA	15497	472	9921	46	428944	23971	43074	2495

Table 1.1: Experimental results; each number is a count of virtual call sites. The total number of virtual call sites for Jedit are 20934 (app) and 8105 (lib). The total number of virtual call sites for Eclipse are 497436 (app) and 26984 (lib).

DLCW = Dynamic Loading from Closed World
DLOW = Dynamic Loading from Open World.

In other words, DLCW means loading a local plugin, whereas DLOW means loading a plugin from, say, the Internet. In effect, DLCW means loading one of the locally available plugins that the compiler was able to include in the original whole-program flow analysis, while DLOW means loading a plugin from, say, the Internet. The assumption is that loading locally available plugins is a frequent event, whereas downloading new code, while allowed, should happen rarely and may thus be more costly.

Table 1.1 shows the static number of virtual call sites that can be inlined under the respective circumstances. The numbers show that loading from the local set of plugins results in an extremely small number of possible invalidations (DLCW). The numbers also show that preanalyzing plugins is about 50% more effective for 0-CFA than for CHA: the number of additional devirtualizations is respectively 57% and 49% higher for 0-CFA after compensating for the higher number of devirtualizations of 0-CFA. When loading arbitrary code from the open world (DLOW), the compiler has to consider almost all devirtualized call sites for invalidation. Only a tiny fraction of all virtual calls can be guaranteed to never require revirtualization in a setting with dynamic loading—a compiler that cannot revirtualize calls can only perform a fraction of the possible inlining optimizations.

The data also shows that TSMI and 0-CFA are quite close in terms of precision, which is good news since this means it is possible to use the type-safe variant without losing a large number of opportunities for optimization. As expected, using 0-CFA or TSMI instead of CHA or RTA cuts the number of virtual calls left in the code after optimization in half. Notice that for Eclipse, in the column for call sites that can be inlined and invalidated by DLCW, 0-CFA has a *smaller* number than TSMI. This is not an anomaly; on the contrary, it shows that 0-CFA is so good that it both identifies 7,357 more call sites in *app* for inlining than TSMI *and* determines that many call sites cannot be invalidated by DLCW. Obviously the flow analyses (TSMI and 0-CFA) perform significantly better than the type-based analyses (CHA and RTA). However, this improved precision comes at a dramatic cost in performance and scalability.

1.2.3 Scaling 0-CFA

As expected, the 0-CFA [Shi91, PS91] style analyses have more precision than CHA on inlining in the presence of dynamic class loading. The numbers also show that if a compiler analyses all plugins that are locally available, then dynamically loading from these plugins will lead to a miniscule number of invalidations. In contrast, when dynamically loading an unanalyzed plugin, the runtime will have to consider a significantly larger number of invalidations. However, running an analysis like 0-CFA on large benchmarks is an extraordinary challenge. While CHA is trivial to scale, 0-CFA generally takes $O(n^3)$ time and $O(n^2)$ in space.

Running benchmarks like Eclipse with an algorithm that uses $O(n^2)$ space is not feasible, in particular since all of our experiments were run with at most 1.8 GB of memory.[4] In order to be able to run Eclipse, our implementation trades space for speed. In essence, instead of representing flow sets directly (with $O(n)$ flow-sets of size $O(n)$ each) the analysis represents flow sets implicitly using dependency graphs using only $O(n)$ space. However, as a result the analysis has to traverse the flow graph more often; and the execution time is increased to $O(n^4)$.

This huge cost in computation time can be offset with help from CHA. Since CHA already devirtualizes a large number of call sites, all of those sites can be ignored by the 0-CFA implementation. Furthermore, even if a call site cannot be devirtualized, CHA can also give an upper bound on the set of interesting types of receivers. Thus, it is possible to exploit type information in the control flow analysis to reduce the computation time to be cubic in practice. Table 1.2 gives space and time measurements for the various analyses. The time given is system execution time, including parsing, abstract interpretation and constraint solving. Space is the maximum number of megabytes memory utilization observed using top for the JVM process.

In conclusion, a type-based approach (represented in this section by CHA and RTA) is often trivial to scale for large applications. Using types can give the compiler many opportunities for code optimization, and these opportunities are generally easy to predict for the programmer. Type annotations also increase code readability; both CHA and standard practices necessitate declaration of types of arguments, fields and local variables (the reader should imagine erasing these type declarations from Java or C++ applications to see what the programs would look like without these standard types). The experiments presented in this section also show that information obtained from flow analyses (represented in this section by TSMI and 0-CFA) is generally more precise compared with information obtained from types. However, flow analyses can also be significantly more expensive. The results are more difficult to predict for the programmer and easily invalidated by changes to the application; for the devirtualization and inlining analysis described in this section, this is demonstrated by the fact that if the code changes due to

[4]1.8 GB is the maximum total process memory for the Hotspot Java Virtual Machine running on OS X as reported by `top` and also the memory limit specified at the command line using the `-Xmx` option.

	Time (s)	Space (MB)
Local, Jedit	14	84
CHA, Jedit	14	83
RTA, Jedit	14	80
TSMI, Jedit	85	93
0-CFA, Jedit	144	122
Local, Eclipse	6749	1800
CHA, Eclipse	13710	1800
RTA, Eclipse	4719	1800
TSMI, Eclipse	209371	1800
0-CFA, Eclipse	183171	1800

Table 1.2: Time and space consumption for the various analyses. Note that implementations of the five analyses share as much code as possible; our goal was to create the fairest comparison possible, not to optimize the analysis time of a particular analysis.

dynamic loading (DLCW), more than twice as many devirtualizations might have to be undone for 0-CFA compared to CHA.

1.3 Contributions of Our Work

This thesis explores new type systems in areas that have traditionally been dominated by flow analysis. Specifically, the two main chapters of the thesis will present two extensions of object-oriented type systems for Java-like languages that allow the programmer to use concise annotations to address common programming problems. By extending the type systems we are able to reap the general benefits of types over flow analysis – scalability, predictability and modularity – for these problems.

Confined types (Chapter 2) can help programmers reason about aliasing, achieving encapsulation of objects. The crucial invariant established by confined types is that all instances of a confined type are encapsulated in their defining package. Encapsulated, in this context, means that no direct references exist to instances of confined type from outside of its defining package. The rules for confinement are simple to understand and allow for fast static verification and inference of the confinement property. Experimental results show that confinement occurs frequently.

Region types (Chapter 3) give programmers memory safety guarantees when manipulating complex distributed and (possibly) multi-dimensional arrays. While generally applicable to other languages, region types have been developed specifically for X10, a new type-safe programming language for distributed high-performance computing. The Chapter 3 describes an extension to the basic object-oriented Java-like type system of X10. Region types exploit a high-level algebra of operations over sets of indices to reason about essential safety properties of accesses to X10's sparse and distributed arrays. The type system allows the programmer to express why accesses to possibly sparse and distributed X10 arrays are both in-bounds and place-local. As a result, the type system enables the programmer to use concise type annotations to provide useful documentation that allows the compiler to eliminate safety checks resulting in faster, statically checked code.

Both extensions simplify the use of the type system for programmers by using constraint solvers in their type checking algorithms. The constraint solvers reduce the amount of annotations that are necessary for the type checker to verify that the desired invariants hold. The remaining annotations have the desirable property that they simply express the interface of the code; in previous work [VB01], the programmer had to declare additional invariants that did not directly relate to the interface and instead were only there to help the type checker verify the type constraints.

The language extensions for the two type system extensions have been implemented in a new compiler framework called XTC (pronounced *ecstasy*, short for *ext*ensible *a*rchtecture for *c*ompilation). Aspects of the framework are discussed in Appendix A.

CHAPTER 2

Encapsulating Objects with Confined Types

2.1 Introduction

Object-oriented languages rely on reference semantics to allow sharing of objects. Sharing occurs when an object is accessible to different clients, while aliasing occurs when an object is accessible from the same client through different access paths. Sharing and aliasing are both powerful tools as well as sources of subtle program defects. One potential consequence of aliasing is that methods invoked on an object may depend on each other in a manner not anticipated by designers of those objects, and updates in one sub-system can affect apparently unrelated sub-systems, thus undermining the reliability of the program.

While object-oriented languages provide linguistic support for protecting access to fields, methods, and entire classes, they fail to provide any systematic way of protecting objects. A class may well declare some field private and yet expose the contents of that field by returning it from a public method. In other words, object-oriented languages protect the state of individual objects, but cannot guarantee the integrity of systems of interacting objects. They lack a notion of an *encapsulation boundary* that would ensure that references to 'protected' objects do not escape their scope.

The goal of this chapter is to describe a pragmatic notion of encapsulation that enables us to provide software engineers with tools to guide them in the design of robust systems. To this end, we focus on simple models of encapsulation that can easily be understood. We deliberately ignore more powerful escape analyses [Bla99, Bla03, BH99, Deu95] which are sensitive to small code changes and may be difficult to interpret, as well as more powerful notions of ownership [AKC02, BN02, BDF04, BLR02, BSB03, Boy01, Cla01, CW03b, DLN96, LM04, MP99]. Of course, the tradeoff is that we will sometimes deem an object as 'escaping' when a more precise analysis would discover that this is not the case. In particular, we have chosen to investigate the concept of *confined types* introduced by Bokowski and Vitek in [VB01]. Confined types give rise to a form of encapsulation that is relatively simple to understand. Furthermore, as we demonstrate in this chapter, confined types are useful in practice and can be checked or inferred with little cost. The basic idea underlying confined types is

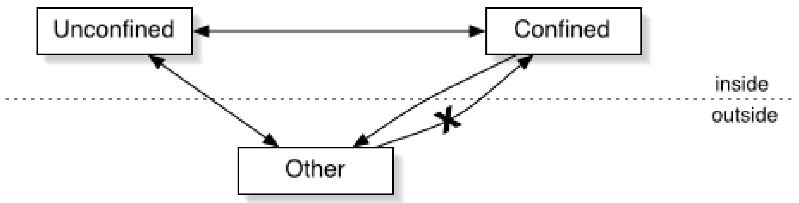

Figure 2.1: Confinement is a property of object references. The diagram illustrates confinement in a tiny program with two packages called `inside` and `outside` and three classes `inside.Confined`, `inside.Unconfined` and `outside.Other`. Arrows denote allowed reference patterns. If the class `Confined` obeys the confinement rules, then objects defined in package `outside` cannot hold references to instances of the class `Confined`, the class is said to be encapsulated in the `inside` package.

the following:

> *Objects of a confined type are encapsulated in their defining, sealed package.*

The notion of *sealed* ensures that all of the code in the package is known. In Java, sealing a package forces the Java Virtual Machine to load all classes from that package from the same jar file. Thus, if a class is confined, instances of that class (and all of its subclasses) cannot be manipulated by code belonging to other packages. An instance of a confined type cannot flow to an object outside the package of the confined type. In terms of aliasing, confinement allows aliases within a package but prevents them from spreading to other packages as illustrated in Figure 2.1.

The original definition of confinement [VB01] required explicit annotations and thus presupposes that software is designed with confinement in mind. In some sense, the underlying assumption was that confinement is an unusual property that may require substantial changes to the original program. In this work we take a different point of view. We claim that confinement is a natural property of well designed software systems. We validate our hypothesis empirically with a tool that *infers* confinement in Java programs. We gathered a suite of forty-six thousand Java classes and analyzed them for confinement. Our results show that, without any change to the source, 24% of the package-scoped classes (exactly 3,804 classes or 8% of all classes) are confined. Furthermore, we found that by using generic container types, the number of confined types could be increased by close to one thousand additional classes. Finally, with appropriate tool support to tighten access modifiers, the number of confined classes can be well over 14,500 (or over 29% of all classes) for that same benchmark suite. While a more powerful program analysis may yield higher numbers of confined classes, especially if a whole-program approach is taken, our current numbers are already high and can be obtained efficiently as the average time to analyze a class file is less than eight milliseconds.

In a related effort, Zhao, Palsberg and Vitek [ZPV06] have shown that the confinement rules are sound for a simple object calculus inspired by Featherweight Java [IPW01] in which sharing is impossible. This was achieved by recasting the confinement rules into a type system. They also showed the soundness of an extension of the confinement rules to generic types; we will discuss that extension later in this chapter.

Since their introduction, confined types have been applied in several different contexts. Clarke, Richmond, and Noble have shown that minor changes to the confinement rules presented here can be used to check the architectural integrity constraints that must be satisfied by Enterprise Java Beans applications [CRN03]. Zhao, Noble and Vitek have applied the same ideas to Real-time Specification for Java to ensure static safety of scoped memory usage [ZNV04]. Herrmann introduced confinement as a software engineering mechanism for a new programming language [Her03]. Skalka and Smith have studied a somewhat different notion of confinement within the context of programming language security [SS05]. In their work the main goal is to control not the flow of references to objects but rather which methods are invoked on those objects.

This chapter makes the following contributions and improvements on previous work on confinement:

- We simplify and generalize the confinement rules presented in the original paper on confined types [BV99].
- We present an efficient constraint-based confinement inference algorithm.
- We give an overview of the implementation of `Kacheck/J`, our confinement inference tool.
- We give results of the confinement analysis of a large corpus of programs.
- We discuss refactorings aimed at improving confinement as well as better language support.

Chapter overview The chapter is organized as follows. Section 2.2 presents a practical an example of confinement using class taken from the Java standard library. Section 2.3 introduces confined types and the associated confinement

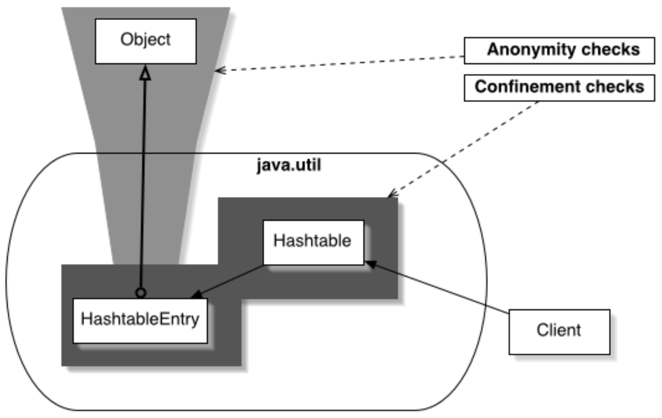

Figure 2.2: Analysis overview. All classes in the enclosing package, `java.util` in this case, are checked for confinement. Parent classes of confined classes (e.g. `Object`) are checked for anonymity. Client code need not be checked.

rules. Section 2.4 presents our constraint-based analysis algorithm. Section 2.5 discusses the implementation of the inference tool, with the complete constraint generation rules given in Section 2.5.3. Section 2.6 gives result of the analysis of the benchmark suite. Section 2.7 discusses refactoring and language support. In Section 2.8, we present an example from the Freenet benchmark [CSW00, CMH02], where `Kacheck/J` is used to detect that a class is not confined. The code is then refactored such that the class becomes confined. Section 2.9 gives an overview of related work.

2.2 A Practical Example of Confinement

In statically typed object-oriented programming languages such as Java, confinement can be achieved by a disciplined use of built-in access control mechanisms and some simple coding idioms. We will give a simple motivating example and use it to illustrate our analysis. Consider the class `HashtableEntry` used within the implementation of `Hashtable` in the `java.util` package. The access modifier for this class is set to default access, which, in Java, means that the class is scoped by its defining package. `HashtableEntry` instances are used to implement linked lists of elements which have the same hash code modulo table size. They are a prime example of an internal data structure which is only relevant to one particular implementation of a hashtable and that should not escape the context of that table and definitely not of its defining package. Yet how can we be sure that code outside of the package cannot get access to an entry object?

Since `HashtableEntry` is a package-scoped class we need not worry that outside code will create instances of the class. However, the implementation of the hash table class itself could cast an entry object to some public superclass, and then expose a reference to the object. Alternatively, in the case where a public method were to return an entry object or a public field held a reference to such an object, outside code might obtain a reference to it (possibly causing an unexpected memory leak – for example in a weak hash map). The outside code could also use that reference as an argument (which might have security implications if the object were representing ownership of a permission), or cast it to some public parent class and invoke methods on it (which may be problematic, particularly if the methods are overridden in the subclass and were not intended to be reachable from outside of the package).

It is likely that a programmer would consider these scenarios to be the result of a programming error, and a good programmer would take care to prevent such confinement breaches. One can view this issue as an escape problem: can references to instances of a package-scoped class escape their enclosing package? If not, then the objects of such a class are said to be *encapsulated* in the package. In the example at hand, `HashtableEntry` is indeed encapsulated, as programmers have carefully avoided exposing them to code outside of the `java.util` package.

Confinement can be checked by a simple program analysis which relies on access modifiers of classes, fields and methods and performs a context- and flow-insensitive analysis of the code of the confining package. We have implemented a tool called `Kacheck/J` which uses this analysis, discovering potential confinement violations and returning a list of confined types for each package analyzed. For instance, in the above example, the expected result of the analysis would be that `HashtableEntry` is confined to the package `java.util`, while `Hashtable` is not, since `Hashtable` has been declared public. In order to determine this, the tool must analyze the body of all classes declared in the package `java.util` package, as well as all parent classes of confined classes. Figure 2.2 illustrates the checks performed by the tool. The analysis is modular since only one package (and the parent classes of confined types) needs to be considered at a time; this turns out to be a key feature for scalability. Furthermore, since client code is not required when checking confinement, it is possible to use our tool on library code.

It turns out that contrary to our expectation, our analysis infers that the class `HashtableEntry` is not confined because the method `clone()` is invoked on one of its instances. The problem is that `clone()` returns a copy of an entry which is typed as `Object`. Manual inspection of the code reveals that each invocation of this methods is immediately followed by a cast to `HashtableEntry`. Thus, instances of the class do not actually escape; this is a typical pattern in a language without adequate support for genericity. Our analysis is not precise enough to discover the idiom—this is part of the price we pay for simplicity. One could consider extending the analysis to catch such idioms, and we leave that for future work.

2.3 Confined Types

The goal of confinement is to satisfy the following soundness property: *An object of confined type is encapsulated in the scope of that type.* Notice that *scope* is a static notion whereas *confinement* controls runtime flow of objects. The idea of confined types is to make the static scope define a bound on where an object can flow. In this work we have set the granularity of confinement to be the package (other granularities have been studied in [CRN03, ZNV04]; only minimal changes to the rules are required). Thus, no instance of a confined type may escape the package in which that type is defined. In order to ensure that the analysis is modular and sound in the presence of dynamic loading we must ensure that new code does not show up inside of the encapsulation boundary after the analysis. In Java, this can easily be achieved by requiring that the encapsulating package is sealed [Mic06, ZFA00]. Henceforth, when we say that instances of a confined class are encapsulated in their defining package, we require that the package is sealed.

Confinement is enforced by two sets of constraints. The first set of constraints, *confinement rules*, apply to the enclosing package (the package in which the confined class is defined). These rules track values of confined types and ensure that they are neither exposed in public members nor *widened* to non-confined types. We use the term widening to denote both:

- *static widening* from C to B: an expression or a statement that requires a check that C is a subtype of B, and
- *hidden widening* to B: an expression or a statement which requires that the type of the distinguished variable `this` is a subtype of B.

A typical example of static widening is an assignment `x=y`, where `x` is of type B and `y` is of type C; Java requires that C is a subtype of B. A typical example of hidden widening is an assignment `x=this`, where `x` is of type B; the dynamic type of the `this`-object cannot be determined locally, so we say that the assignment results in a hidden widening from that dynamic type of `this` to B.

The second set of constraints, so-called *anonymity rules*, applies to methods inherited by the confined classes (potentially including library code) and ensures that these methods do not leak a reference to `this`, which may refer to an object of confined type.

In this section, we adapt the rules of Bokowski and Vitek to infer confinement. The new rules are both simpler and less restrictive (*i.e.*, more classes can be shown confined), while remaining sound. As in the original paper, the rules presented here do not require a closed-world assumption. Confinement inference is performed at the package level. The rules assume that all classes in a package are known and, for confined classes, that their superclasses are available.

Enforcing confinement relies on tracking the spread of encapsulated objects within a package and preventing them from crossing package boundaries. We have chosen to track encapsulated objects via their types. Thus, a confinement breach will occur as soon as a value of a confined type can escape its package. Since we track types, widening a value from a confined type to a non-confined type is a violation of the confinement property.

2.3.1 Anonymity Rules

Anonymity rules apply to inherited methods which may (but do not have to) reside in classes outside of the enclosing package. The goal of this set of rules is to prevent a method from leaking a reference aliasing the distinguished `this` pointer. The motivation for these rules is that if `this` refers to an encapsulated object, returning or storing it amounts to hidden widening.

We say that a method is *anonymous* if it satisfies the three rules in Table 2.1. The first rule prevents an inherited method from storing or returning `this` unless the static type of `this` also happens to be confined. The second rule ensures that `native` methods are never anonymous. While rules $\mathcal{A}1$ and $\mathcal{A}2$ are direct anonymity violations, the rule $\mathcal{A}3$ tracks transitive violations. The call mentioned in rule $\mathcal{A}3$ depends on the dynamic type of `this` (the target of the call). Thus, anonymity of a method is determined in relation to a specific type. One can use a conservative flow analysis to determine a set of possible target methods, or one can rely on the static type to determine possible targets.

$\mathcal{A}1$	An anonymous method cannot widen `this` to a non-confined type.
$\mathcal{A}2$	An anonymous method cannot be `native`.
$\mathcal{A}3$	An anonymous method cannot invoke non-anonymous methods on `this`.

Table 2.1: Anonymity rules.

Figure 2.3 gives an example of a problematic piece of code where a non-

```
public abstract class C {
    public abstract int getSecret();
    public C escape() {
        return this;
    }
}
class Internal extends C {
    public int getSecret() {
        return 42;
    }
    public C notAnonymous() {
        return escape();
    }
}
public class Container {
    Internal i = new Internal();
    public C exposeAccidentally() {
        return i.notAnonymous();
    }
}
```

Figure 2.3: Example of hidden widening in a non-anonymous method resulting in a confinement breach. A client outside of the package could execute `container.exposeAccidentally().getSecret();` to obtain the secret, despite the fact that `Internal` is package-scoped and there is no static widening of `Internal` to `C`.

anonymous method allows presumably encapsulated objects to escape their container, possibly leaking private information. The interesting thing to note here is that for all assignments in the code the static types match exactly. In particular, the widening of the type of the presumably encapsulated object happens in the `escape` method when the static type of `this` is `C`. Detecting such hidden widenings is the purpose of the anonymity rules. Explicit (static) widenings, that is assignments where the static types of the variables involved are different, are covered by rules described in the next section.

An alternative approach would be to simplify the rules (as taken in [CRN03]) and to disallow confined types from extending types other than `Object`. Anonymity rules are then no longer needed: the only place where hidden widening can occur with that limitation is `Object`. The only violation in `Object` is `clone()`, which is then handled with a specific rule. However, while this approach significantly simplifies the rules for confinement, it also severely restricts the set of classes that can be confined. In this chapter, we focus on the approach using anonymous methods.

2.3.2 Confinement Rules

Confinement rules are applied to all classes of a package. A class is *confined* if it satisfies the five rules of Table 2.2. Rule $\mathcal{C}1$ ensures that no inherited method invoked on a confined type will leak the `this` pointer. Together with the anonymity rules this rule prevents hidden widening. Note that the rule does not preclude a confined type from *inheriting* (or even declaring) non-anonymous methods, as long as they are never called. Rule $\mathcal{C}2$ prevents public classes from being confined. This is necessary since code outside of the package must not be able to instantiate a confined type. Rule $\mathcal{C}3$ ensures that no exposed member (public or protected) is of a confined type. This applies to all non-confined types in the package. Rule $\mathcal{C}4$ prevents non-confined classes (or interfaces) from extending confined types. This rule is primarily a design choice from the point of view that if a confined type encapsulates internal information, that information should also not be leaked as part of a subtype. In [ZPV06] it was shown that leaking references to confined types from a package can be prevented without this rule. Finally, rule $\mathcal{C}5$ prevents static widening of references of confined type to non-confined types.

$\mathcal{C}1$	All methods invoked on a confined type must be anonymous.
$\mathcal{C}2$	A confined type cannot be public.
$\mathcal{C}3$	A confined type cannot appear in the type of a public (or protected) field or the return type of a public (or protected) method of a non-confined type.
$\mathcal{C}4$	Subtypes of a confined type must be confined.
$\mathcal{C}5$	A confined type cannot be widened to a non-confined type.

Table 2.2: Confinement rules.

2.3.3 Discussion and Special Cases

Exceptions are a case of widening which is not explicitly listed in our rules. Instead, we consider that `throw` widens its argument to the class `Throwable`, which is declared public and thus violates rule $\mathcal{C}5$.

Our confinement rules do not forbid packages from having native code, but rule $\mathcal{A}2$ explicitly states that native methods are not anonymous. The motivation for this design choice is that while the developer of a package may be expected to manually inspect native code in the current package, it would be difficult to check native code of parent classes belonging to standard libraries. Furthermore, uses of `this` that violate $\mathcal{A}1$ are usually not perceived as bad behavior for native code. Essentially, we assume that native code within the enclosing package is, to some extent, trusted. In other words, with respect to anonymity, we make the safe choice that a native method cannot be anonymous; it can do whatever it wants. With respect to confinement, we trust the native methods to not violate the confinement rules. The reason for this design decision is that native code that does not conflict with the Java type system may still violate the anonymity rules. However, confinement violations can happen anywhere in native code, thus if we do not want to analyze or rule out all native code, we must trust that native code does not violate confinement. We have manually inspected some of the native code in GNU Classpath, and we found that anonymity violations do happen. We did not, however, find any confinement violations in the native code that we inspected.

In Java, `System.arraycopy` is often used to copy elements from one array to another. While the signature of this special native method takes arguments of type `Object` and thus calls to this method would constitute a widening to a non-confined type, this method is used frequently enough to warrant a special treatment in `Kacheck/J`. The tool treats calls to `System.arraycopy` as a widening from the inferred source-array type to the inferred destination array type. This is safe if the language implements `System.arraycopy` correctly.

Another optimization in `Kacheck/J` is the treatment of static widenings of `this`. Static widenings of `this` are covered by both rules for anonymity ($\mathcal{A}1$) and for confinement ($\mathcal{C}5$). But while rule $\mathcal{A}1$ will only have an impact on confinement if the anonymous method is actually invoked ($\mathcal{C}1$), rule $\mathcal{C}5$ would always make the statically widened type non-confined. While this makes no difference in many cases, this does have an impact on some types if the code in which the widening takes place is dead. In some sense, $\mathcal{A}1$ implicitly contains a limited flow-sensitive dead code analysis, while $\mathcal{C}5$ does not. The `Kacheck/J` tool can be made to relax the rule $\mathcal{C}5$ to not include static widenings of `this` (since those would be caught by rule $\mathcal{A}1$ if the code is not dead). An example for this is shown in Figure 2.4. If the optimization is enabled, the liveness of the `dead()` method determines whether `Conf` is confined. This illustrates how relaxing $\mathcal{C}5$ makes the analysis more fragile in the sense that small changes in the code are more likely to change the set of confined classes.

In a related effort, Zhao, Palsberg and Vitek [ZPV06] have shown that the

```
class Conf {
    public Object dead() {
        return this;
    }
}
```

Figure 2.4: Static widenings of `this` can be ignored. This can eliminate confinement violations in dead code.

confinement rules are sound for a simple object calculus inspired by Featherweight Java [IPW01] in which sharing is impossible. In that paper, the three anonymity rules are consolidated into just one rule, namely "*the* `this` *reference is only used to select fields and as receiver in invocation of other anonymous methods.*" That can be done because the calculus does not have native methods or assignment statements.

Potanin et al. [PNC04] have presented an alternative means to check package confinement, using reduction to Java generics.

Clarke et al. have shown that minor changes to the confinement rules presented here can be used to check the architectural integrity constraints that must be satisfied by Enterprise Java Beans applications [CRN03]. One main difference between their rules and ours is that their uses don't employ a notion of anonymous methods. Roughly speaking, we can understand their rules as the result of removing $\mathcal{A}1$–$\mathcal{A}3$ from our rules and changing $\mathcal{C}1$ to "All methods invoked on a confined type must be defined in a confined type." Clarke et al [CRN03] make a few further restrictions on the confinement rules which are appropriate in the domain of Enterprise Java Beans. In contrast to our analysis, their analysis enables different classes to appear as both confined and as unconfined in different parts of the application (i.e., in different beans). The overall result is an analysis which works at a different level of granularity than ours and offers confinement per bean, rather than per package. The experimental results of Clarke, Richmond, and Noble [CRN03] demonstrate that their analysis works well in the domain of Java Beans.

2.4 Constraint-Based Analysis

We use a constraint-based program analysis to infer method anonymity and confinement. Constraint-based analyses have previously been used for a wide variety of purposes, including type inference and flow analysis. Constraint-based analysis proceeds, as usual, in two steps: (**1**) Generate a system of constraints from program text. (**2**) Solve the constraint system. The solution to the constraint system is the desired information. In our case, constraints are of the following forms:

$$\begin{aligned} A &::= \textsf{not-anon}(\text{methodId}) \\ T &::= \textsf{not-conf}(\text{classId}) \\ C &::= A \mid T \mid T \Rightarrow A \mid A \Rightarrow A \mid A \Rightarrow T \mid T \Rightarrow T \end{aligned}$$

A constraint not-anon(methodId) asserts that the method methodId is *not* anonymous; similarly, not-conf(classId) asserts that the class classId is *not* confined. The remaining four forms of constraints denote logical implications. For example, not-anon(`A.m()`) $\Rightarrow$ not-conf(`C`) is read "if method `m` in class `A` is not anonymous then class `C` will not be confined."

We generate constraints from the program text in a straightforward manner. The example of Figure 2.5 illustrates the generation of constraints. For each syntactic construct, we have indicated in comments the associated rule from Section 2.3. Table 2.3 details the constraints that are generated for that example. A complete description of the constraints generated from Java bytecode is given in Appendix A. All our constraints are ground Horn clauses. Our solution procedure computes the set of clauses not-conf(`classId`) that are either immediate facts or derivable via logical implication. This computation can be done in linear time [DG84] in the number of constraints, which, in turn, is linear in the size of the program.

2.4.1 Control Flow Analysis

The rule $\mathcal{C}1$ poses a control flow problem as it mandates that only methods that are actually invoked on a confined type need to be anonymous. Any conservative control flow analysis can be used to yield a set of candidate methods. We have chosen to perform a simple flow insensitive analysis that is practical and precise enough for our purposes.

Methods of confined classes cannot be invoked from outside of their defining package since confined types are by definition not public ($\mathcal{C}2$) and cannot be widened to non-confined types ($\mathcal{C}5$). So, for anonymity violations that are relevant to a given type, the analysis only needs to consider invocations of methods on instances of that type and its subtypes. Subtypes must be included since confined types may be widened to other confined types.

Our analysis performs a fixed-point iteration starting with the assumption that all non-public classes could potentially be confined. The analysis then

```
public class A {
    A a;
    public A m() {
        a = this;                      (𝒜1)
        new B().t(this);               (𝒜1)
        return this;                   (𝒜1)
    }
    native void o();                   (𝒜2)
}

class B extends A {
    void t(A a) {}
    A p() {
        return this.m();               (A3)
    }
    public A getD() {
        return new D().p();            (𝒞1)
    }
}
public class C {                       (𝒞2)
    public D getD() {                  (𝒞3)
        return new D();
    }
    public D d = new D();              (𝒞3)
}

class D extends B {                    (𝒞4)
    A getA() {
        this.t(this);                  (𝒞5)
        a = new D();                   (𝒞5)
        return new D();                (𝒞5)
    }
}
```

Figure 2.5: Example program used to illustrate constraint generation.

Case	Constraint	Explanation
($\mathcal{A}1$)	not-conf(`A`) $\Rightarrow$ not-anon(`A.m()`)	`this` widened to `A`
($\mathcal{A}2$)	not-anon(`A.o()`)	`o` is `native`
($\mathcal{A}3$)	not-anon(`A.m()`) $\Rightarrow$ not-anon(`B.p()`)	`B.p()` calls `m()` with `this` as receiver
($\mathcal{C}1$)	not-anon(`D.p()`) $\Rightarrow$ not-conf(`D`)	`p()` invoked on a `D`-object
($\mathcal{C}2$)	not-conf(`C`)	class `C` declared to be public
($\mathcal{C}3$)	not-conf(`C`) $\Rightarrow$ not-conf(`D`)	public method `C.getD()` has return type `D`; public field `C.d` has type `D`
($\mathcal{C}4$)	not-conf(`D`) $\Rightarrow$ not-conf(`B`)	`D` extends `B`
($\mathcal{C}5$)	not-conf(`A`) $\Rightarrow$ not-conf(`D`)	`D` widened to `A`

Table 2.3: The constraints generated from the example in Figure 2.5.

records invocations of the type `x.m()`, where the type of `x` is in the current candidate set for confinement. These invocations form the root set for the control flow analysis. Calls of the form `this.m()` that are reachable from this root set are recorded in accordance with anonymity rule $\mathcal{A}3$. The set of types of `this` that are used for resolving virtual method calls is the static type of `x`, as inferred during bytecode verification, and all subtypes of that type that are ever found to be widened to it. Naturally, such widenings (rules $\mathcal{A}1$ and $\mathcal{C}5$) may be detected at any time during the flow analysis, which is the reason that a fixed-point computation is necessary. When the fixed-point computation terminates and all invocation chains for all applicable confinement candidates have been traversed, the remaining types for which no anonymity violations were found are declared confined.

The analysis does not attempt to perform dead-code detection, so while the method that includes an invocation such as `x.m()` may be dead, we will nevertheless add `m` to the root set. This simplifies the analysis but costs some precision. Doing dead code detection would lead to analysis results that are much more sensitive to changes in the source program.

2.5 Implementing Confinement Inference: Kacheck/J

Although the confinement and anonymity rules have been described as source level constraints, we have chosen to implement `Kacheck/J` as a bytecode analyzer.

The main advantage of working at the bytecode level is that there are a large number of class files freely available to apply our tool to. The implementation of `Kacheck/J` leverages the XTC static analysis framework which was developed as part of the Ovm JVM [OVM04] and which is presented in Appendix A. In XTC, bytecode verification is implemented using the Flyweight pattern [GHJ94]. For each of the 200 bytecode instructions defined in the Java Virtual Machine Specification, the XTC verifier creates an `Instruction` object that is responsible for computing the effect this instruction will have on an abstract state. Verification is a simple fixed-point iteration. The verification starts with an initial state which includes the instruction pointer, operand stack, and variables. The verifier follows all possible flows of control within the method.

By instrumenting the transfer functions of only 9 of the 200 `Instruction` objects, we can use XTC's abstract interpretation engine to generate constraints. The instrumentation performs some simple checks and records basic facts about the program execution. For instance, the code for the `areturn` instruction checks to see if `this` is used as return value, and if so, it reports that `this` is widened to the return type of the method. The `invoke` instructions record dependencies such as the use of `this` as an argument or method invocations on `this`. Overall, the following changes are applied to the verifier:

- In non-static methods, local variable 0 (`this`) is tracked, and uses of `this` are recorded.
- All static widenings are recorded; thrown exceptions are considered widened to `Throwable`.

Widenings are captured by intercepting subtype checks done by the verifier. Anonymity checks only require slight modifications to the transfer functions that correspond to the nine instructions: a check is added to record operations on `this`. See Appendix A for details. The flow analysis computes the implication chains for each potentially confined type T, such that

$$(T' \Rightarrow A_1) \wedge (A_1 \Rightarrow A_2) \wedge \ldots\ (A_{n-1} \Rightarrow A_n) \wedge (A_n \Rightarrow T)$$

is collapsed to

$$T' \Rightarrow T.$$

The code specific to confined types (including verbose reporting of violations) is about 2,200 lines. The code reused from XTC (including reading and writing of Java 5.0 class files) is about 30,000 lines of code.

```
public class P {
    public Object nonAnon() {
        return this;                          (1)
    }
}

class B extends P {
    public Object nonAnonInd() {
        return this.nonAnon();                (2)
    }
}

class C extends B {                           (3)
}

public class X {
    public Object invocation() {
        return new C().nonAnonInd();          (4)
    }
}
```

Figure 2.6: Simple example of a confinement violation.

2.5.1 Example

Figure 2.6 gives an example of a chain of constraints that results in classes being not confined. Although the tool reorders parts of the solving process, we will in the following explain only the final chain of constraints. Notice first that `Object` is a non-confined class, so a constraint of the type C is generated by rule $\mathcal{C}2$:

$$\textsf{not-conf}(\texttt{Object})$$

The method `P.nonAnon()` widens `this` to `Object`. This will generate a constraint of type $C \Rightarrow A$ by rule $\mathcal{A}1$:

$$\textsf{not-conf}(\texttt{Object}) \Rightarrow \textsf{not-anon}(\texttt{P.nonAnon()})$$

The invocation of `nonAnon` in `nonAnonInd` with `this` as the receiver generates a constraint of the type $A \Rightarrow A$ by rule $\mathcal{A}3$:

$$\textsf{not-anon}(\texttt{P.nonAnon()}) \Rightarrow \quad \textsf{not-anon}(\texttt{B.nonAnonInd()})$$

The method `nonAnonInd()` is invoked on `C`. By rule $\mathcal{C}1$ a constraint of the type $A \Rightarrow C$ is generated:

$$\textsf{not-anon}(\texttt{B.nonAnonInd()}) \Rightarrow \textsf{not-conf}(\texttt{C})$$

As `C extends B`, a constraint of the type $C \Rightarrow C$ is generated by rule $\mathcal{C}4$:

$$\textsf{not-conf}(\texttt{C}) \Rightarrow \textsf{not-conf}(\texttt{B})$$

Solving this constraint system will result in `B` and `C` being non-confined (and `P` and `X` cannot be confined either because they are `public`).

2.5.2 Simplifying Assumptions

`Kacheck/J` operates under some simplifying assumptions which we detail here.

Reflection The analysis assumes that reflection is not used to circumvent language access control. In other words, it assumes that the semantics of private, protected and default access modifiers are respected by the reflection mechanisms. This assumption can be violated by changing the settings of the Java Security Manager. This may result in additional confinement breaches.

Native code Native methods are not checked by `Kacheck/J` and may breach confinement. The results obtained from `Kacheck/J` are only valid if native methods do not violate any of the confinement rules. Furthermore, we assume that native code does not violate the semantics of the language by ignoring access control declarations. Manual inspection of a number of native methods indicates that these assumptions are reasonable. We do not assume that native methods satisfy the anonymity rules. Note, however, that during manual inspection of various native methods, we did not find any that violate the rules for anonymous methods.

2.5.3 Constraint Generation

This section summarizes the constraints that need to be generated by the various Java opcodes. Inference or checking of confined types essentially requires generation of constraints according to these rules followed by a simple constraint solving phase.

InvokeStatic

- If `this` occurs in the argument list, record widening of `this` to the type T of the matching argument in the current method m. This generates the constraint: $C \Rightarrow A$ where C is $\mathsf{not\text{-}conf}(T)$ and A is $\mathsf{not\text{-}anon}(m)$.

- For each argument a of inferred type T that is an object, record the corresponding declared type T' of the parameter. This generates constraints $C' \Rightarrow C$ where C' is $\mathsf{not\text{-}conf}(T')$ and C is $\mathsf{not\text{-}conf}(T)$.

Areturn, Putfield, Putstatic, Aastore

- If the variable that is returned or stored is `this`, record widening of `this` to the declared type T' (the return type, type of the field or the component type of the array). This generates a constraint $C \Rightarrow A$ where C is $\mathsf{not\text{-}conf}(T')$ and A is $\mathsf{not\text{-}anon}(m)$ with m being the current method.

- If the variable that is used is an object but not `this` and has inferred type T, record widening to the corresponding declared type T'. This generates constraints $C' \Rightarrow C$, where C' is $\mathsf{not\text{-}conf}(T')$ and C is $\mathsf{not\text{-}conf}(T)$.

InvokeInterface, InvokeVirtual, InvokeSpecial

- If `this` occurs in the argument list, record widening of `this` to the type T of the matching argument in the current method m. This generates the constraint: $C \Rightarrow A$, where C is $\mathsf{not\text{-}conf}(T)$ and A is $\mathsf{not\text{-}anon}(m)$.

- If a call in method m is of the form `this.n()`, calling a method n from method m on `this`, record method invocation (distinguishing between invokevirtual, invokeinterface and invokespecial). This generates the constraint $A \Rightarrow A'$, where A is $\mathsf{not\text{-}anon}(n)$ and A' is $\mathsf{not\text{-}anon}(m)$.

- If the call is not on `this`, but of the form $a.n()$, record an invocation on type T where T is the inferred type of a. This generates the constraint $A \Rightarrow C$, where A is $\mathsf{not\text{-}anon}(n)$ and C is $\mathsf{not\text{-}conf}(T)$.

- For each argument a of inferred type T that is an object, record the corresponding declared type T' of the parameter. This generates constraints $C' \Rightarrow C$, where C' is $\mathsf{not\text{-}conf}(T')$ and C is $\mathsf{not\text{-}conf}(T)$.

Athrow

- If the variable that is thrown is `this`, record widening of `this` to `Throwable`. This generates a constraint $C \Rightarrow A$, where C is not-conf(`Throwable`) and A is not-anon(m) with m being the current method. Because the condition not-conf(`Throwable`) is always true, a primitive constraint A can be used as well.

- If the thrown variable is an object, but not `this`, and has inferred type T', record widening to `Throwable`. This generates a constraint $C \Rightarrow C'$, where C is again always true (not-conf(`Throwable`)) and C' is not-conf(T').

Call Propagation A call to method m on a type T must generate additional constraints for all subtypes S_i of T that are widened to T.

2.6 Analysis Results

`Kacheck/J` has been evaluated on a large data set. This section gives an overview of the benchmark programs and presents the results of the analysis. The first goal of the evaluation is to show that confinement is a common property in actual code (Section 2.6.2). The second goal is to identify common reasons why certain types are not confined and thereby gauge the limitations of our technique (Section 2.6.3). Studying reasons for nonconfinement also points out possible areas where slight modifications to the analysis would dramatically increase opportunities for confinement. We present three such modifications in sections 2.6.4, 2.6.5 and 2.6.6. Having the programmer declare types as confined using annotations in the source code would not be practical if confinement was a fragile property and annotations would need to be changed frequently. In order to give evidence that confinement is a stable property which a programmer might want to declare in the program text, Section 2.6.7 studies how confinement properties of types change during the lifetime of a particular application. Finally, in order for confinement to be useful in practice we will demonstrate that checking or inferring confinement scales and can be done quickly. Section 2.6.8 shows that `Kacheck/J` can rapidly analyze huge benchmarks.

2.6.1 The Purdue Benchmark Suite

The Purdue Benchmark Suite (Table 2.4) consists of 33 Java programs and libraries of varying size, purpose and origin. The entire suite contains 46,165 classes (or 115 MB of bytecode) and 1,771 packages. To the best of our knowledge the PBS is the largest such collection of Java programs. Most of the benchmarks are freely available.

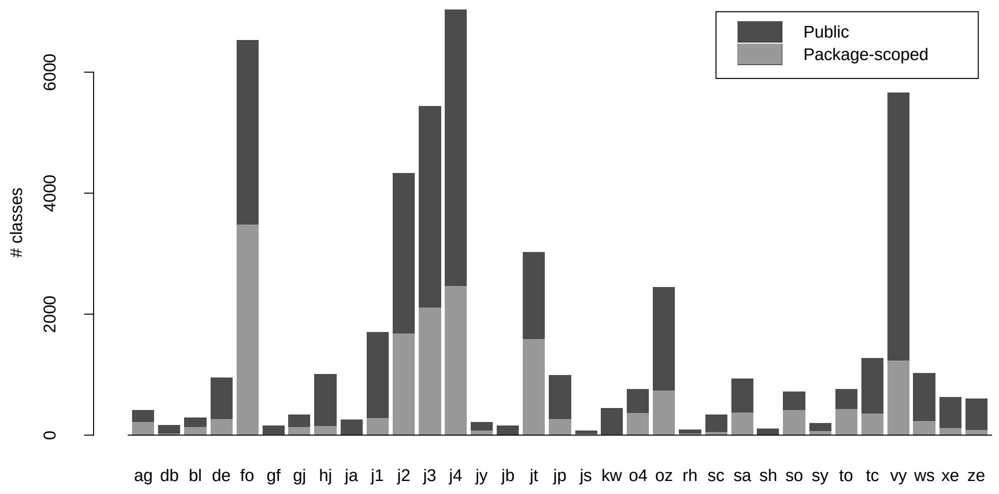

Figure 2.7: Benchmark characteristics: program sizes.

Figure 2.7 gives an overview of the sizes, in number of classes, for each program or library that is part of the PBS. Appendix B provides additional data about the benchmarks. Our largest benchmarks, over 2,000 classes each, are Forte, JDK 1.2.2, JDK 1.3, Ozone, Voyager and JTOpen. Ozone and Forte are applications, while the others are libraries. The number of package-scoped classes is indicated in light gray for each application. This number is an upper bound for the number of confined classes; public classes cannot be confined.

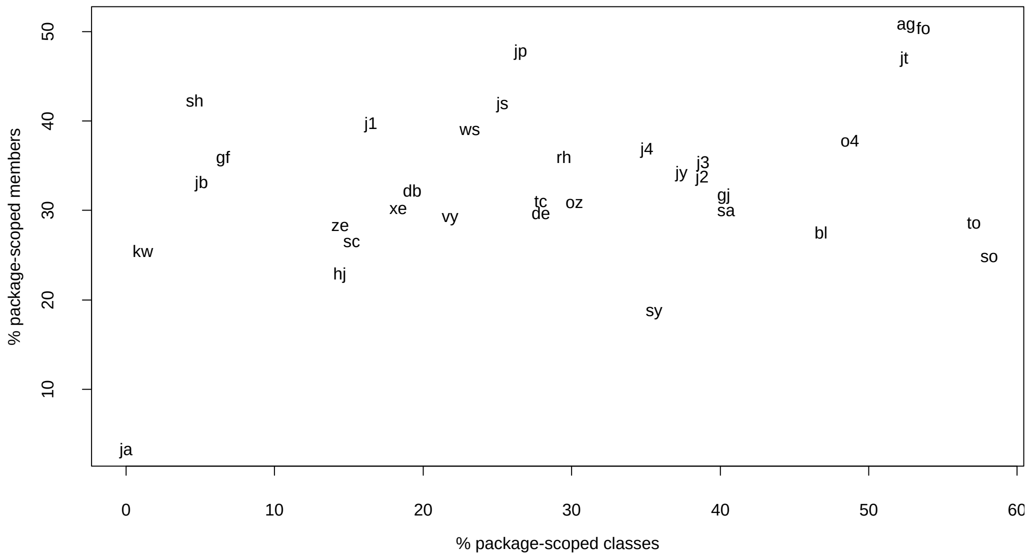

Figure 2.8: Benchmark characteristics: member encapsulation.

Figure 2.8 relates the proportion of package-scoped members to package-scoped classes. Package-scoped members are fields and methods that are declared to have either private or default access. Most coding disciplines encourage the use of package-scoped methods and package-scoped classes. Not surprisingly, programs that were designed with reuse in mind, such as libraries and frameworks, are better-written than one-shot applications. For instance, the Aglet workbench and JTOpen, both libraries, exhibit high degrees of encapsulation. Forte is noteworthy because, although it is an application, it has over 50% package-scoped classes and members. Compilers and optimizers written in an object-oriented style, such as Bloat, Toba and Soot, have high numbers of package-scoped classes because of the many classes used to represent syntactic elements or individual bytecode instructions. At the other extreme, we have applications such as Jax and Kawa which have almost no package-scoped classes. It is also worth noting the increase in encapsulation between different versions of the JDK. From JDK1.1.8 to JDK1.3.1, the absolute number of classes tripled, yet the percentage of package-scoped classes doubled. The reason for this is largely that most of

Name	Description	
Aglets	Mobile agent toolkit	ag
AlgebraDB	Relational database	db
Bloat	Purdue bytecode optimizer	bl
Denim	Design tool	de
Forte	Integrated dev. environment	fo
GFC	Graphic foundation classes	gf
GJ	Java compiler	gj
HyperJ	IBM composition framework	hj
JAX	Packaging tool	ja
JDK 1.1.8	Library code (Sun)	j1
JDK 1.2.2	Library code (Sun)	j2
JDK 1.3.0	Library code (IBM)	j3
JDK 1.3.1	Library code (Sun)	j4
JavaSeal	Mobile agent system	js
Jalapeno 1.1	Java JIT compiler	jp
JPython	Python implementation	jy
JTB	Purdue Java tree builder	jb
JTOpen	IBM toolbox for Java	jt
Kawa	Scheme compiler	kw
OVM	Java virtual machine	o4
Ozone	ODBMS	oz
Rhino	Javascript interpreter	rh
SableCC	Java to HTML translator	sc
Satin	Toolkit from Berkeley	sa
Schroeder	Audio editor	sh
Soot	Bytecode optimizer framework	so
Symjpack	Symbolic math package	sy
Tomcat	Java servlet reference impl.	tc
Toba	Bytecode-to-C translator	to
Voyager	Distributed object system	vy
Web Server	Java Web Server	ws
Xerces	XML parser	xe
Zeus	Java/XML data binding	ze

Table 2.4: The Purdue Benchmark Suite (PBS v1.0).

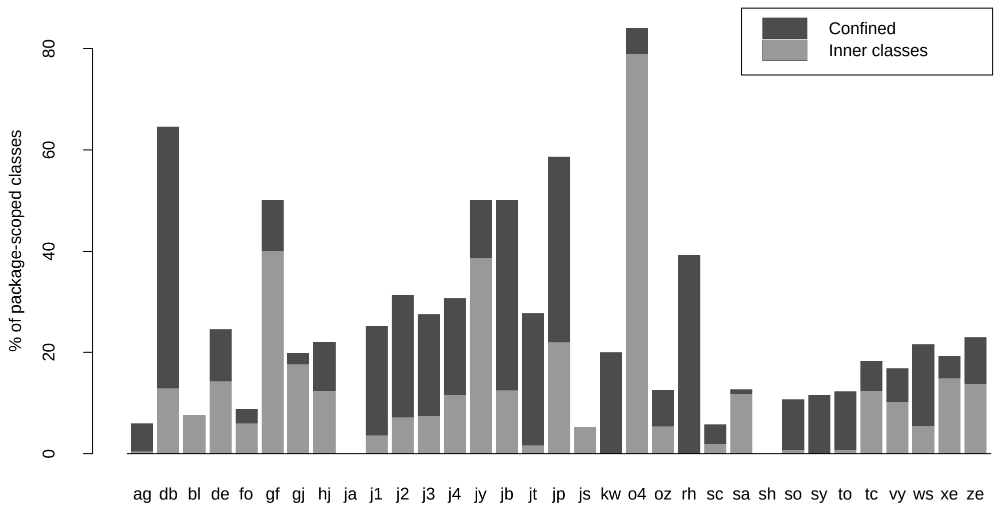

Figure 2.9: Percentage of confined types for various benchmarks.

the JDK1.1.8 code implements the simple, public core classes of the Java runtime (java.*), whereas JDK1.3.1 has substantial amounts of code that the main application does not interface with directly.

Coding style has an impact on confinement. While the relation between package-scoped classes and confined types is obvious, there is a more subtle connection between package-scoped members and confined types: public and protected methods can return potentially confined types. So it is reasonable to expect that programs with low proportions of package-scoped members will also have comparatively fewer confined types.

2.6.2 Confined Types

Running `Kacheck/J` over the PBS yields 3,804 confined classes – 24% of the package-scoped classes and 8% of all classes are confined. Figure 2.9 shows confined classes in terms of percentage of all classes. The numbers are broken down per program, with confined inner classes in light gray. Raw numbers are given in Table 2.6.

There are 6 programs where more than 40% of the package-scoped types are confined (db, gf, jy, jb, jp, o4). It is interesting to note that these programs have very little in common: they are a mix of libraries (gf), frameworks (o4) and applications (db, jy, jb, jp). Their ratios of package-scoped classes and their sizes vary widely. Indeed, manual inspection of the programs indicates that programming style is essential to confinement. For example, in early versions

Benchmark	Classes			Pkgs	Opcodes
	All	Public	Inner		
Aglets	410	193	133	31	107846
AlgebraDB	161	130	9	9	51218
Bloat	282	150	127	20	84212
Denim	949	684	271	84	288140
Forte	6535	3053	3769	231	1123362
GFC	153	143	8	22	58003
GJ	338	202	189	14	105323
HyperJ	1007	862	70	29	211269
JAX	255	255	0	21	97932
JDK 1.1.8	1704	1423	29	90	917132
JDK 1.2.2	4338	2655	1365	133	958619
JDK 1.3.0	5438	3326	1780	177	1180406
JDK 1.3.1	7037	4569	2043	213	2010305
JPython	214	134	35	11	103094
JTB	158	150	1	8	48900
JTOpen	3022	1439	557	75	1048704
Jalapeno 1.1	994	730	132	29	255436
JavaSeal	75	56	19	18	34933
Kawa	443	438	100	13	68733
OVM	835	416	590	41	111161
Ozone	2442	1705	490	122	447984
Rhino	95	67	1	8	51752
SableCC	342	290	47	10	45621
Satin	938	559	455	70	194985
Schroeder	108	103	7	13	41422
Soot	721	302	79	9	65137
Symjpack	194	125	0	14	73465
Toba	762	327	79	14	98993
Tomcat	1271	916	221	105	286368
Voyager	5667	4430	1305	312	996077
Web Server	1024	787	52	76	370664
Xerces	622	508	125	45	233919
Zeus	604	517	74	42	180437
Total	49259	31741	14163	2120	11987191

Table 2.5: Statistics for the benchmarks.

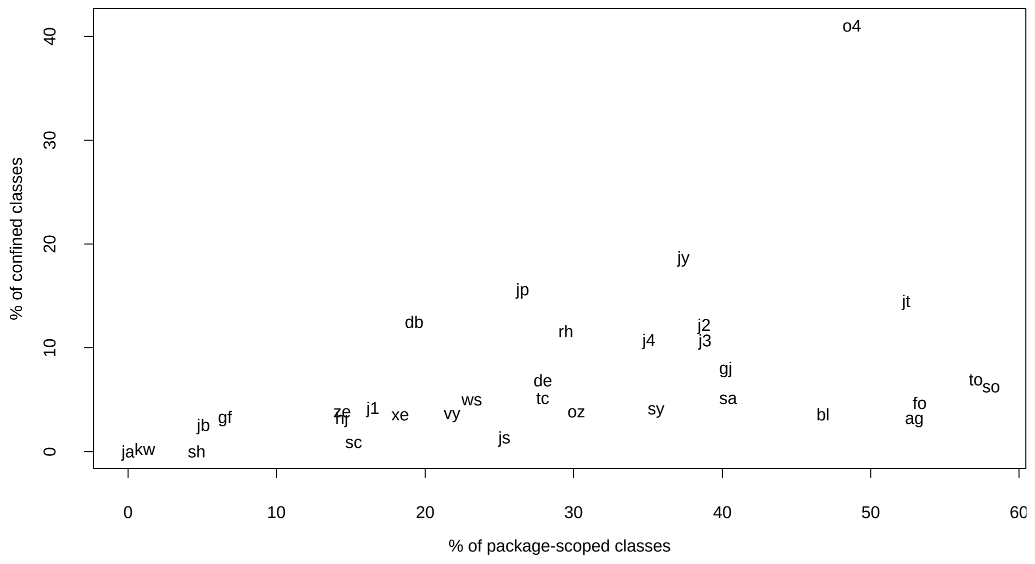

Figure 2.10: Impact of package-scoping on confinement.

of Ovm and `Kacheck/J`, unit tests were systematically stored in sub-packages of the current package. Some methods and classes were declared public only to allow testing of the code. This, in turn, prevented many classes from being confined. The large number of confined inner classes in Ovm (o4) comes from the objects representing bytecode instructions nested in an instruction set class. For Jalapeno, the high confinement ratio of 16% (155 classes out of 994) is partially the result of the single package structure of the program.

Predictably, programs with very few package-scoped classes (e.g. ja, kw, sh, gf) end up with few confined classes. Figure 2.10 shows the relationship between package-scoped classes and confined classes. Notice that the fraction of package-scoped classes varies considerably from benchmark to benchmark. For instance, libraries like Aglets (ag) which have very high ratios of package-scoped members and classes still perform quite poorly with only 13 classes (3%) being confined out of 410. Why does this happen? To answer that question, we start with a discussion of confinement violations.

2.6.3 Confinement Violations

It is difficult to quantify confinement violations in terms of categories based on the constraints. This is mainly because many of the constraints work in concert. For example, widening one class to another ($\mathcal{C}5$) may violate confinement because the second class is not confined due to the fact that a non-anonymous method ($\mathcal{A}1$) is invoked ($\mathcal{C}1$) on it. And the reason that the aforementioned method is not anonymous could be that a third class is public ($\mathcal{C}2$). Notice that the confinement violation for the original class involves four different constraints. Rather than trying to quantify confinement violations, this section attempts to describe the causes for non-confinement based on a few characteristic examples.

Most confinement breaches are caused by a small number of widely used programming idioms. For any violation `Kacheck/J` returns a textual representation of the implication chain that caused the violation. We give examples of the main causes for classes not being confined.

2.6.3.1 Anonymity Violations

The three primary anonymity violations in the entire JDK come from methods in the AWT library which register the current object for notification. The method `addImpl` from Figure 2.11 is representative.

```
protected void addImpl(Component cp, Object cn, int i) {
    synchronized ( getTreeLock() ) { ...
        e = new ContainerEvent(this, COMPONENT_ADDED, comp);
        ...
    }
}
```

Figure 2.11: Example of event handler violating anonymity. The `addImpl` method passes `this` to the constructor of the event that it is creating. Since the event expects a first argument of type `Object` (the source of the event), this widens `this` to `Object`, which is an anonymity violation.

2.6.3.2 Widening to superclass

Widening to a superclass is among the most frequent kind of confinement breach. For instance, `Kacheck/J` signals the following widening in the Aglet benchmark:

```
com/ibm/aglets/tahiti/SecurityPermissionEditor:
    - illegal widening to:
        - com/ibm/aglets/tahiti/PermissionEditor
```

The `PermissionEditor` class is an abstract superclass of the non-public `Security-PermissionEditor`. `PermissionEditor` is the part of the interface that is exported outside the package.

2.6.3.3 Widening in Containers

A large number of violations comes from the use of container classes in Java. `Kacheck/J` works with bytecode, not source code. At the bytecode level of Java, generic types are absent. As a result, data structures such as vectors and hashtables always take arguments of type `Object`; thus, any use of a container will entail widening to the most generic super-type. One might implement a `Kacheck/J`-like tool at the source level which handles generic types in a non-trivial way. We leave that for future work; inspiration might come from the paper by Zhao, Palsberg, and Vitek [ZPV06], which presented rules for handling confinement of generic types. For instance, `Kacheck/J` reports that `NativeLibrary`, an inner class of `ClassLoader`, is not confined.

```
java/lang/ClassLoader$NativeLibrary:
    Illegal Widening to java/lang/Object
```

The error occurs because an instance of `NativeLibrary` is stored in a vector:

```
systemNativeLibraries.addElement(lib);
```

As such, this violation may indicate a security problem. The internals of class loaders should really be encapsulated. Inspection of the code reveals that the `Vector` in which the object is stored is private.

```
private static Vector systemNativeLibraries = new Vector();
```

After further inspection, it is obvious that the vector does not escape from its defining class. But this requires inspection of the source code and only remains true only until the next patch is applied to the class. This example shows the usefulness of tools such as `Kacheck/J` as they can direct the attention of software engineers towards potential security breaches or software defects.

2.6.3.4 Anonymous Inner Classes

This violation occurs frequently when inner classes are used to implement callbacks. For example, in Aglets, the `MouseListener` class is public. Thus, the following code violates confinement of the anonymous inner class.

```
mlistener = new MouseAdapter() {
    public void mouseEntered(MouseEvent e) { ... } };
```

Similar situations occur with package-scoped classes that implement public interfaces. They are package-scoped to protect their members, but are exported outside of the package.

2.6.3.5 Summary

Even though confinement violations are often the result of chains of events, there are two rules which by themselves eliminate most opportunities for confinement and thus deserve further consideration. The confinement rules whose violations are the cause of the largest number of non-confined types overall are $\mathcal{C}2$ (class is public) and $\mathcal{C}5$ (instance widened to non-confined type). The dramatic effect of these rules is shown in the following sections, where small modifications are made which limit the scope of these rules and result in a significant increase in the number of confined types. In Section 2.6.4, widening of confined types is discounted whenever it occurs in conjunction with containers (eliminating many common applications of rule $\mathcal{C}5$). In Section 2.6.5, the access modifiers are inferred, making many classes and methods package-scoped which were previously public. Both variations result in a sharp increase in the number of confined types.

2.6.4 Confinement with Generics

In Java, vectors, hashtables, and other containers are pervasive. Every time an object is stored in a container, its type is widened to `Object`, leading to a widening violation for the object's class. If Java supported proper parametric polymorphism (on the bytecode level), the large majority of container-related violations would disappear (existing applications rarely use containers with heterogeneous contents).

In order to attempt to assess the impact of generics without rewriting all of the programs in the PBS, we modified `Kacheck/J` to ignore widening violations linked to containers. This was done by ignoring all widenings to `Object` that

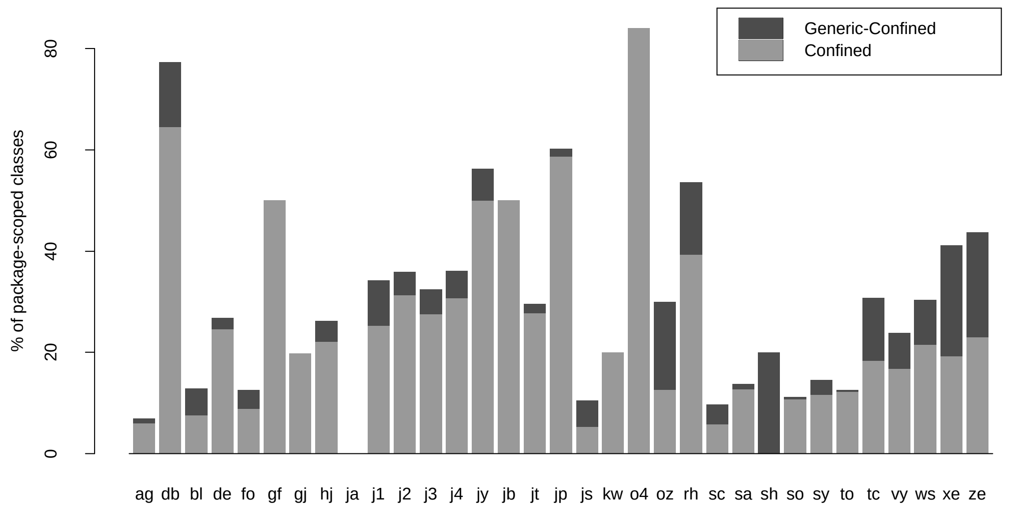

Figure 2.12: Percentage of generic-confined types for various benchmarks.

occur in calls to methods of `java.util` classes. Figure 2.12 gives the resulting percentages of confined classes without generic violations; we call these classes Generic-Confined (GC). The light gray bars show the original number of confined classes. The dark grey bars show the effect of adding genericity. The number of confined types increases from 3804 (8%) to 4862 (10%) over all programs in the PBS. These results should be interpreted with caution, as they might represent an overestimate of potential gains; this is due to the fact that we do not guarantee that the container instances are package-scoped.

2.6.5 Inferring Access Modifiers

Some of the benchmarks have an extraordinarily low number of confined classes compared to the others. Inspection of the access modifiers of classes in these benchmarks makes the reason for this lack of confined classes immediately clear. For example, in Kawa, out of 443 classes, only 5 (1%) are package-scoped. Similarly, many benchmarks contain methods and/or fields that are declared as public and thus prevent certain types from being confined. This raises the question of whether or not the access modes are the tightest possible, or whether they are more permissive than necessary. To address this question we infer the tightest access modes and then use the inferred modes for confinement checking. This analysis is performed by the Java Access Modifier Inference Tool (JAMIT), which is also available on our webpage.

JAMIT infers the tightest legal access modes by looking at all accesses to

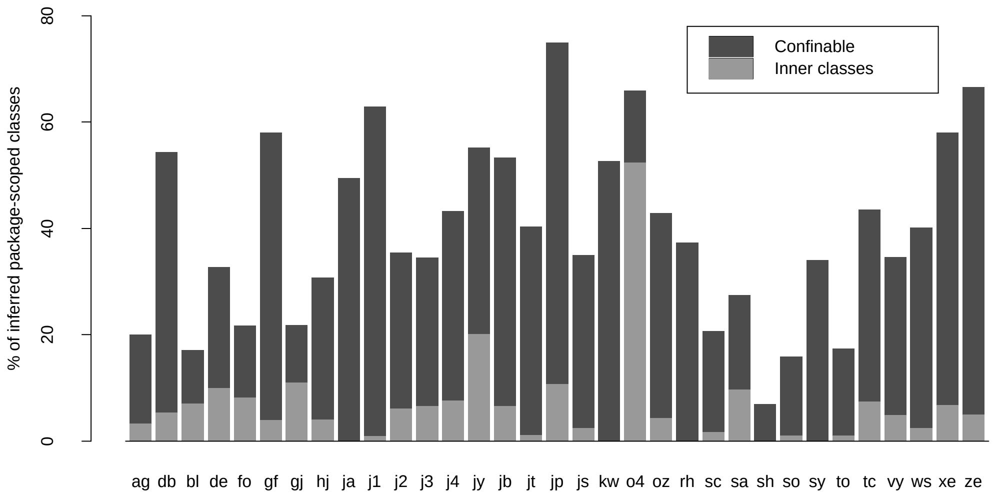

Figure 2.13: Percentages of confinable types for various benchmarks.

a given member or type. It then determines what the most restrictive access modifier would be that will permit all accesses according to Java's visibility rules. The analysis takes subtyping into account; subtypes can view protected members, and overriding methods can only relax access modifiers. More importantly, in order to preserve overriding the access modifier in the parent may need to be relaxed from private to package-scoped (if all overriding subtypes reside in the same package) or protected.

Figure 2.13 shows the result of running `Kacheck/J` on code for which access modifiers were strengthened using JAMIT. Classes that become confined with modifier inference are called *confinable* (CA). With mode inference, the number of confinable classes jumps from 3804 (8%) to 12,880 (26.1%) for the entire PBS. Furthermore, if we combine confinable and generics, we obtain 14,591 (29.6%) generic-confinable classes.

Figure 2.14 relates the results of this new analysis to the original number of package-scoped classes. It is quite telling to see that Jax and Kawa, which were applications with the lowest numbers of confined classes, suddenly consist of about 40% confinable classes. Of course, using this option on library code may yield an overestimate of the potential gains, as some classes that are never used from within the library can be made package-scoped, even though client code requires access to these classes. Nevertheless, the results give a good indication of the potential gains.

Benchmark	Conf	Confinable	GenConf	GenConfinable
Aglets	13	60	15	66
AlgebraDB	20	81	24	97
Bloat	10	29	17	39
Denim	65	187	71	211
Forte	306	1149	437	1346
GFC	5	58	5	58
GJ	27	51	27	52
HyperJ	32	193	38	212
JAX	0	99	0	104
JDK 1.1.8	71	712	96	744
JDK 1.2.2	527	1062	603	1173
JDK 1.3.0	581	1297	685	1476
JDK 1.3.1	756	2126	891	2344
JPython	40	90	45	107
JTB	4	8	4	8
JTOpen	438	1049	467	1113
Jalapeno 1.1	155	543	159	549
JavaSeal	1	14	2	17
Kawa	1	177	1	177
OVM	119	243	302	428
Ozone	93	754	221	920
Rhino	11	28	15	33
SableCC	3	24	5	28
Satin	48	206	52	218
Schroeder	0	6	1	7
Soot	45	90	47	92
Symjpack	8	53	10	89
Toba	53	102	55	104
Tomcat	65	377	109	448
Voyager	208	1268	295	1442
Web Server	51	255	72	301
Xerces	22	221	47	279
Zeus	20	237	38	278
Total	3804	12880	4862	14591

Table 2.6: Number of confined and confinable classes.

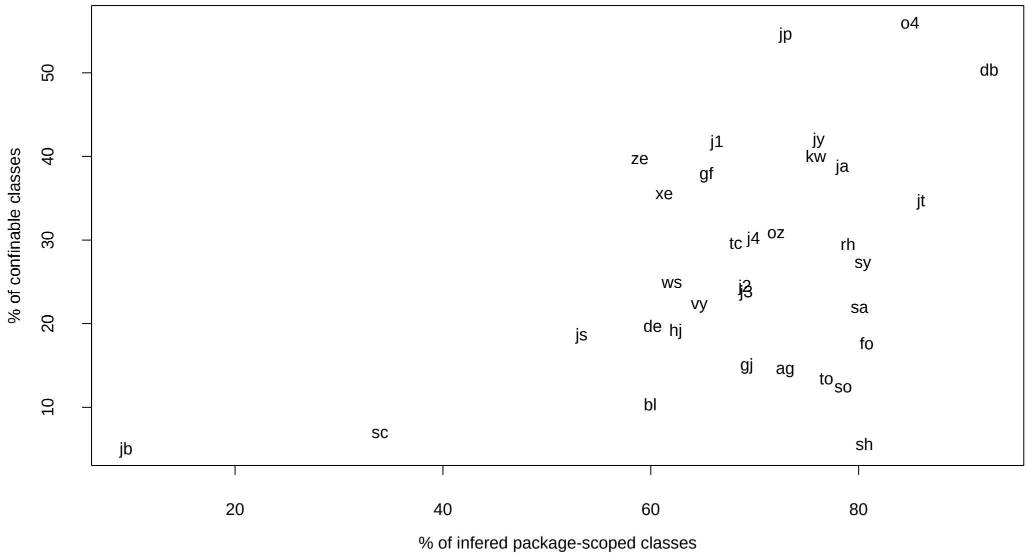

Figure 2.14: Confinable types and package-scoping.

2.6.6 Hierarchical Packages

Our last experiment involves changing the semantics of the Java package mechanism. Currently, Java has a flat package namespace; that is, even though package names can be nested, there is no semantics in this nesting. This creates a dilemma between data abstraction and modularity. Good design practice suggests that applications be split into packages according to functional characteristics of the code. On the other hand, creating packages forces certain classes to become public even if those classes should not be used by clients of the program. From a confinement perspective, we could say more packages result in fewer confined classes. One extreme is Jalapeno, which is structured as a single package. This diminishes the usefulness of the confinement property.

To evaluate the impact of the package structure on confinement, we modified `Kacheck/J` to use a hierarchical package model. The general idea is to extend package-access to neighbor packages. We introduce a definition of scope that we call *n-package-scoped*. n-package-scoped limits access to classes in packages that are less than n nodes in the tree of package names away from the defining package. For example, the class `java.util.HashtableEntry` would be visible for `java.lang.System` for $n = 2$. The unnamed package is defined to have distance ∞ from all other packages, making a n-package-scoped class `a.A` invisible for `b.B` regardless of the choice of n.

Figure 2.15 shows the cumulative improvements yielded by increasing the proximity threshold n. With $n = 9$ most programs are treated as a single package,

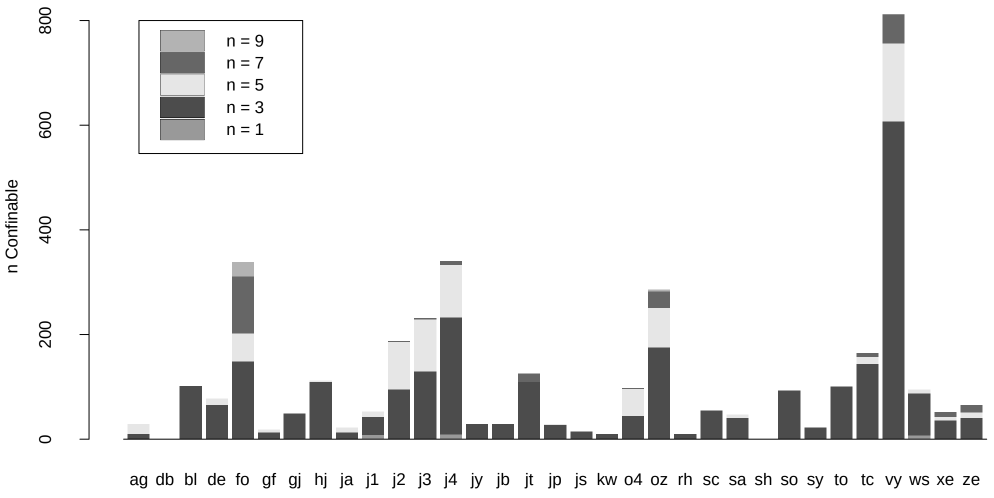

Figure 2.15: Confinement with hierarchical packages.

increasing the number of confined types from 3,804 to 7,495. The largest increase in confined classes comes from the Voyager benchmark, where the number of confined classes increases from 208 to 1021.

2.6.7 Evolution of Confinement

For the working software engineer, it may be of interest to know whether confinement is preserved when software evolves. If a class is confined in one software version, it would be helpful to know whether the class will also likely be confined in the next version. If the answer is yes, then confinement can be viewed as a meaningful, fundamental property of a type, not just a coincidence of the arrangement of the code. To shed light on this issue, we present a study of the confined types in 14 versions of TomCat, ranging from version 3.0 to an early snapshot of 5.0. The results are unambiguous. Even with dramatic changes to the code base that involve adding and removing hundreds or thousands of classes, only a few existing classes suddenly become confined or stop being confined. Almost all confined classes stay confined (or are removed from the code base), and almost all non-confined classes stay non-confined (or are removed).

Figure 2.16 shows the differences in the numbers of confined types between versions. The upward arrows indicate the number of types that are new in a particular version of the code. The top of the upward arrows is anchored at the number of confined types for the specific version. The dashed arrows that go down diagonally from that point indicate the number of types that used to be confined and that have been removed from the codebase. The fact that in almost

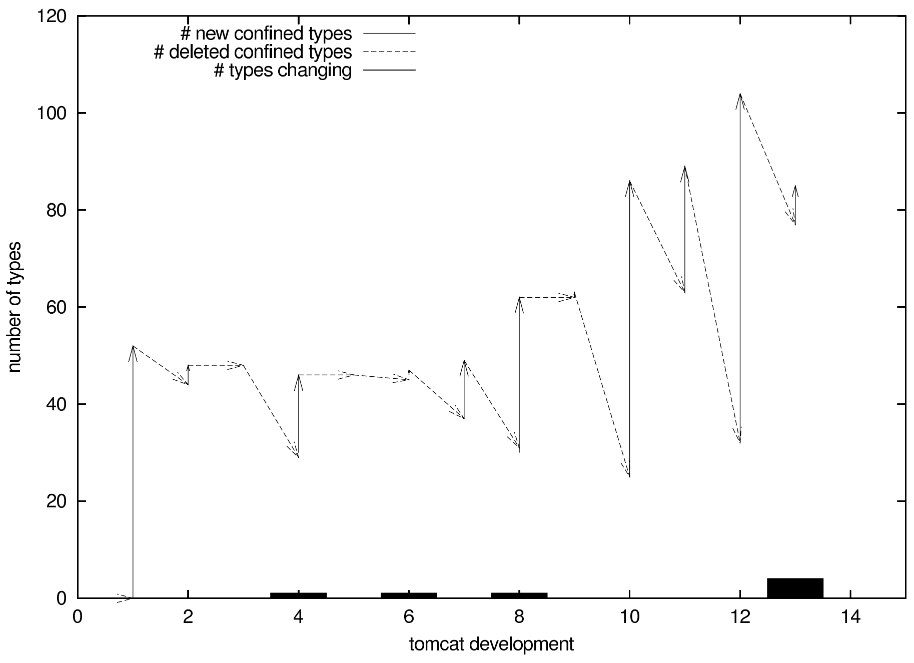

Figure 2.16: Number of confined types in different versions of the TomCat benchmark. The top of the solid arrows marks the number of confined types in each version. The dashed arrows refer to the number of confined types that were already present and confined in the previous version of the code. The bars at the bottom represent the number of types that change confinement (become confined or are no longer confined) and exist in both the current and the previous version of the code.

all places both arrows meet in exactly the same point shows that it is rare that confined types become non-confined and vice versa. The height of the bars at the bottom also illustrates this; the height of the bar is the number of types that are live, were live in the previous version and changed from confined to non-confined or vice versa. The graph shows that while the overall changes to the code are quite significant, the number of types that change their confinement property is marginal (with a total of 6 changes from version 3.0 to 5.0; the total number of confined types in the different versions in between ranges from 46 to 104 with an average of 68). This stability of the confinement property over time supports the thesis that confinement is be a reasonable annotation for a type.

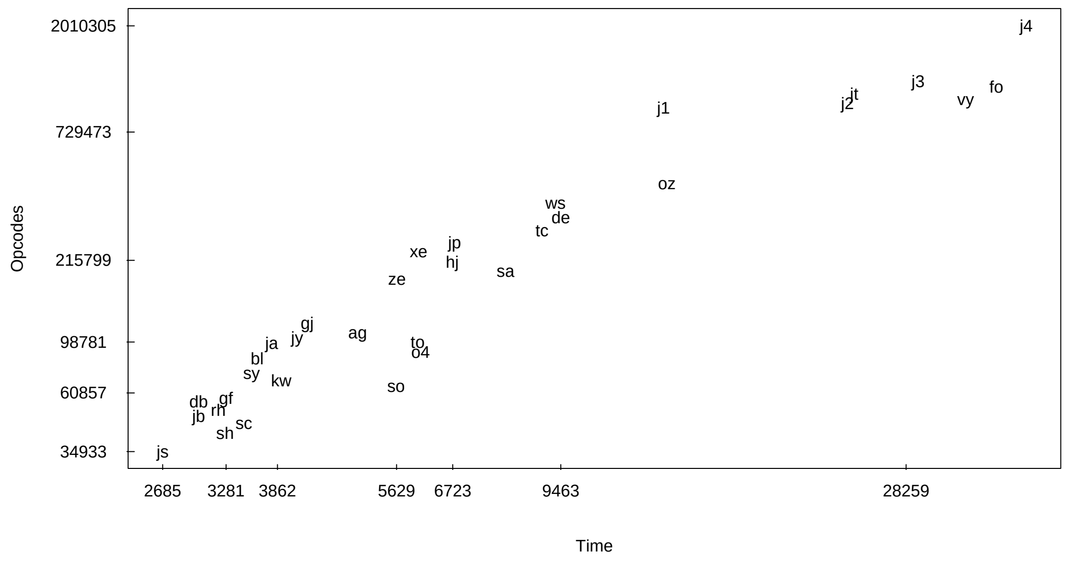

Figure 2.17: Running times for the analysis in ms (log-log scale).

2.6.8 Runtime Performance

All benchmarks were performed on a Pentium III 800 with 256 MB of RAM running Linux 2.2.19 with IBM JDK 1.3. Except for the JDK tests (j1, j2, j3, j4), all running times include loading and analyzing required parts of the Sun JDK 1.3.1 libraries. The longest running time is that of JDK 1.3.1, which consists of 7,037 classes and is analyzed in 41 seconds. On average, `Kacheck/J` needs 7.5 milliseconds per class. Figure 2.17 summarizes the cost of confinement checking; detailed timings are in Table 2.7.

2.7 Containers and Language Extensions

2.7.1 Coding for Confinement

Our results clearly point to containers as one source of confinement violations. We considered using generic extensions of Java to increase confinement; unfortunately, the homogeneous translation strategies adopted by Java implies that at the bytecode level, code written with generics is translated back to code that uses `Object` and casts. One might be able to uncover patterns of bytecode compiled from generics and use those to improve the analysis; however, `Kacheck/J` makes no attempt to do this and thus cannot verify that classes stored in generic containers remain confined. Heterogeneous translation strategies would have the drawback of causing code duplication. Fortunately, it is possible to achieve the de-

Benchmark	Time (ms)		
	real	user	sys
Aglets	4979	4540	160
AlgebraDB	3009	2860	70
Bloat	3623	3530	90
Denim	9463	8020	300
Forte	37565	29870	1380
GFC	3284	3070	90
GJ	4245	3960	60
HyperJ	6711	6160	220
JAX	3790	3580	100
JDK 1.1.8	13103	11750	450
JDK 1.2.2	23463	19270	750
JDK 1.3.0	29336	25760	760
JDK 1.3.1	41304	39730	850
JPython	4107	3890	90
JTB	3009	2810	80
JTOpen	23950	21720	800
Jalapeno 1.1	6770	6270	230
JavaSeal	2685	2490	50
Kawa	3910	3440	180
OVM	6072	5270	200
Ozone	13245	11190	480
Rhino	3201	2920	70
SableCC	3470	3130	110
Satin	7955	6310	270
Schroeder	3270	2730	90
Soot	5622	5190	250
Symjpack	3559	3270	100
Toba	6020	5550	270
Tomcat	8918	7790	330
Voyager	34082	25960	1090
Web Server	9308	8060	250
Xerces	6038	5560	220
Zeus	5640	4960	150
Total	347567	303330	10670

Table 2.7: Time required for the analysis.

```
public interface Entry {
    boolean equal( Entry e);
    int hashCode();
}

public class Hashtable {
    public void put( Entry e) { ... }
    public Entry get( Entry e) { ... }
}

class MyEntry implements Entry {
    ConfinedKey key;
    ConfinedValue val;
    public boolean equal( Entry e) { ... }
    public int hashCode() { ... }
}
```

Figure 2.18: Example Hashtable interface.

sired result using some coding techniques; the basic concept is to use the adapter pattern to wrap an unconfined object around each confined object that must be stored in a container.

A confined implementation of a hashtable could provide an interface `Entry` with two methods `equal(Entry e)` and `hashCode()` (Figure 2.18). In the package that contains the confined class `C`, the programmer would define an implementation `MyEntry` of `Entry` with a package-scoped constructor that takes the key and value (where, for example, the value has the type of the confined class) as well as package-scoped accessor methods. The `Hashtable` itself would only be able to access the `public` methods defined in `Entry`.

The cost of this change would be the creation of the extra `Entry` object that might not be required by other implementations of `Hashtable`. On the other hand, to access a key-value pair, this implementation only requires one cast (`Entry` to the `MyEntry` to access key and value), where the default implementation requires a cast on key and value. For other containers, the tradeoffs may be more significant.

Zhao, Palsberg and Vitek [ZPV06] suggested an alternative that involves extending confinement to generic types and annotating bytecode with confinement assertions. In addition to the existing rules presented so far, they require the rules given in Table 2.8. The combined rules $\mathcal{C}5$ and $\mathcal{C}6$ correspond to the sub-

$\mathcal{C}6$	A generic type or type variable cannot be widened to a type containing a different set of type variables.
$\mathcal{C}7$	A confined type cannot replace a public type variable in the instantiation of a generic type.
$\mathcal{C}8$	Overriding must preserve anonymity of methods.

Table 2.8: Alternative confinement rules

typing partial order that prevents reference widening for Generic ConfinedFJ. $\mathcal{C}7$ corresponds to the extra requirement in the definition of well-formed generic types. Unlike in the base system, $\mathcal{C}8$ is necessary as we are not certain which method may be called before a generic class is instantiated.

2.7.2 Improved Language Support

Java can be extended to support confined types in several ways. Such extensions can be more or less intrusive on the syntax and semantics. We will consider two approaches:

1. explicit annotations for confined classes and anonymous methods, and
2. explicit annotations for confined classes but not for anonymous methods.

Using the metadata facilities of Java 5, it is easy to add such annotations to Java code. Figure 2.19 shows how to specify the `Confined` and `Anonymous` annotations. In order to allow for running a static checker on the bytecode, the confinement property is preserved for the class files. The annotations are not needed at runtime. The rule that subclasses of confined types should also be confined is made explicit by the `inherited` annotation.

2.7.2.1 Explicit annotations of classes and methods

In Bokowski and Vitek's original proposal for confined types [VB01], both confined classes and anonymous methods had explicit modifiers, in the following style:

```
@Confined class C extends B {
    @Anonymous int m() {
        return this.n();
```

```
@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.CLASS)
@java.lang.annotation.Inherited
public @interface Anonymous {}
@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.CLASS)
@java.lang.annotation.Inherited
public @interface Confined {}
```

Figure 2.19: Defining annotations for confinement and anonymous methods in Java 5. The presented code defines two annotations (`@Anonymous` and `@Confined`) which according to the given *retention policy*, will be compiled into the `.class` files, but which will not be available for introspection at runtime. The *inherited* declaration ensures that the annotations are automatically applied to all subtypes.

```
        }
    }
```

The constraints of Bokowski and Vitek are stricter than the constraints presented in this chapter. In particular, Bokowski and Vitek require that the anonymity of a method is preserved in all subclasses, and that the static receiver of a virtual call must be anonymous. In contrast, the constraints checked by `Kacheck/J` only require the unique dynamic targets to be anonymous. Thus, the more modular checking of Bokowski and Vitek results in fewer confinement opportunities when compared to `Kacheck/J`.

The explicit annotation `@Anonymous` for anonymous methods simplifies checking the confinement constraint $\mathcal{C}1$. That rule can be checked by (1) ensuring that every method invoked on a confined type is declared as `@Anonymous` and (2) by checking the constraints given by Bokowski and Vitek [VB01] for anonymous methods. With `@Anonymous` annotations determining confined types no longer requires a complex data flow analysis to determine whether or not a method is anonymous. As a result, programmers only have to look at one statement at a time to reason about confinement. While this simplifies reasoning for the programmer, it also requires additional work in the form of `@Anonymous` annotations on many methods.

2.7.2.2 Explicit annotations of classes, but not of methods

There are many more anonymous methods than confined classes. Thus, the burden on the programmer to annotate code can be lightened considerably by only requiring explicit annotation of classes. Moreover, the resulting inference of anonymous methods can be done according to the rules presented in this chapter. This inference is scalable and more precise than annotations for anonymous methods. Annotating existing code with a `@Confined` modifier can be done automatically with the results from `Kacheck/J`.

The potential issue that without `@Anonymous` annotations reasoning about confinement requires constraint solving can be addressed with appropriate tool support. `Kacheck/J` can be used to report confinement violations with the chains of method calls that resulted in the violation. Consequently, the fact that avoiding `@Anonymous` annotations requires additional constraint solving is hardly an issue for developers, especially since analysis performance is also not an issue (see Table 2.7). Hence having only `@Confined` annotations is clearly the best choice, resulting in less work for the programmer as well as giving better precision.

The latest version of `Kacheck/J` for Java 5 allows both automatically annotating bytecode with `@Confined` metadata attributes (for use by other analyses that need confinement information) as well as checking that all types that are annotated to be `@Confined` in the source are actually confined (for verification of confinement assertions provided by the programmer).

2.8 Refactoring for Confinement

In this section, we detail how `Kacheck/J` can aid the process of first discovering that a class is not confined, and then refactoring the program such that the class can become confined.

2.8.1 The Example Program

We will use an example which stems from the Freenet application [CSW00, CMH02]. The example was found by inspecting the Freenet source code and discovering that class `DoublyLinkedListImpl` has an inner class which probably should be confined. For clarity, we will work with a severely condensed version of class `DoublyLinkedListImpl` and two of its clients. We condensed the code mainly by removing methods and code sections irrelevant to our quest to make a class confined.

Our example program is shown in Figures 2.20, 2.21, 2.22 and 2.23. The ex-

```
package freenet.support;

public interface DoublyLinkedList {
    public interface Item {
        Item getNext();
        Item setNext(Item i);
        Item getPrev();
        Item setPrev(Item i);
    }
    int size();
    java.util.Enumeration elements();
    void push(Item i);
}
```

Figure 2.20: Doubly-linked list interface.

ample program contains a rudimentary interface `DoublyLinkedList` (Figure 2.20) and an implementation `DoublyLinkedListImpl` of doubly-linked lists (Figure 2.21). The example program also consists of two pieces of client code, called `Intervalled-Sum` (Figure 2.22) and `LoadStats` (Figure 2.23) which use doubly-linked lists. Notice that class `DoublyLinkedListImpl` has an inner class `ItemImpl` (which was called `Item` in the Freenet source code). Class `ItemImpl` is used to represent the state of objects of class `DoublyLinkedListImpl`. If we want to encapsulate the state of objects of class `DoublyLinkedListImpl`, then class `ItemImpl` should be confined.

The Freenet code was written by many different authors. The multiple authorship may explain the two inconsistent uses of class `DoublyLinkedListImpl`: one client re-implements the `Item` interface from scratch, whereas another client extends the `ItemImpl` code. The Freenet code contains more than just these two uses of `DoublyLinkedList`; the two clients in Figure 2.22 and 2.23 are simple yet representative samples.

2.8.2 Refactoring: Remove Simple Confinement Violations

In Figure 2.21, the class `ItemImpl` is public and therefore it is not confined by definition. However, confining `ItemImpl` is probably a good idea since its state is the internal representation of the `DoublyLinkedList`. Having clients outside of the package manipulate `ItemImpl` objects might easily break invariants of the

```
public class DoublyLinkedListImpl implements DoublyLinkedList {
    protected int size;
    protected Item head, tail;
    public int size() { return size; }
    public java.util.Enumeration elements() {
        return new java.util.Enumeration() {
            protected Item next = head;
            public boolean hasMoreElements() {
                return next != null;
            }
            public Object nextElement() {
                if (next == null) throw new java.util.NoSuchElementException();
                Item result = next;
                next = next.getNext();
                return result;
            }
        };
    }
    public void push(Item j) {
        // ...
        ++size;
    }
    public static class ItemImpl implements Item {
        private Item next, prev;
        public Item getNext() { return next; }
        public Item setNext(Item i) {
            Item old = next; next = i; return old;
        }
        public Item getPrev() { return prev; }
        public Item setPrev(Item i) {
            Item old = prev; prev = i; return old;
        }
    }
}
```

Figure 2.21: Doubly-linked list implementation.

```
package freenet.support;

import freenet.support.DoublyLinkedList.Item;

public class IntervalledSum {
    private final DoublyLinkedList l = new DoublyLinkedListImpl();
    public void report(double d) {
        l.push(new Report(d));
    }
    static class Report implements Item {
        double value;
        private Item prev, next;
        Report(double value) { this.value = value; }
        public Item getNext() { return next; }
        public Item setNext(Item i) {
            Item r = next; next = i; return r;
        }
        public Item getPrev() { return prev; }
        public Item setPrev(Item i) {
            Item r = prev; prev = i; return i;
        }
    }
}
```

Figure 2.22: Doubly-linked list client code.

```
package freenet.node;

import freenet.support.*;

public class LoadStats {
    private final DoublyLinkedListImpl lru = new DoublyLinkedListImpl();
    private final java.util.Map table = new java.util.HashMap();
    public void storeTraffic(byte[] n, long r) {
        LoadEntry le = new LoadEntry(n, r);
        table.put(le.fn, le);
        lru.push(le);
    }
    class LoadEntry extends DoublyLinkedListImpl.ItemImpl {
        private final Object fn;
        private final long qph;
        private LoadEntry(byte[] b, long qph) {
            this.fn = b;
            this.qph = qph;
        }
    }
}
```

Figure 2.23: Code for another doubly-linked list client.

DoublyLinkedList implementation (such as the size of the list or the head and tail fields).

In order to confine ItemImpl, we must remove all violations of the confinement rules. Let us first consider rule $\mathcal{C}4$, which requires that subtypes of a confined type be confined. This clearly conflicts with the subclassing of ItemImpl by LoadEntry. This problem can be solved using the "Replace Inheritance with Delegation" refactoring pattern [FBB99]. Instead of extending Item, a field value is added to the Item class. We use generics in order to give the field an appropriate type. Using this design also removes the code duplication in Report, which no longer needs to implement Item. Since ItemImpl is now going to be the only implementation of the Item interface, the split between implementation and interfaces is quite useless, so in order to simplify the code, we remove the interfaces and eliminate the Impl from the names of the classes of the implementation. Finally, the access modifier of Item (formerly ItemImpl) is changed from public to default (in order to satisfy confinement rule $\mathcal{C}2$). The result of the first refactoring is the program in Figures 2.24 and 2.25.

2.8.3 Refactoring: Remove Widening Violations

If we run Kacheck/J on the program in Figure 2.24, we will get the result that class Item is still not confined. The problem is that the method nextElement widens Item to Object (upon return). We can refactor the program in Figure 2.24 to remove the violation.

The result of the second refactoring is the program in Figure 2.26. As a result of the refactoring, Item is confined and clients can no longer easily break invariants of the DoublyLinkedList container.

2.8.4 Refactoring: Summary

In our experience, the biggest hurdle in refactoring code for confinement is to find candidates where such refactoring would truly improve the code. The primary obstacle in Java is Java's containers; this obstacle could theoretically be addressed by checking confinement at the source level. Nevertheless, in practice, many classes can easily be confined by flattening the hierarchy and possibly wrapping references to instances in other classes. However, while it is often easy to achieve confinement, refactoring code blindly simply to maximize confinement may result in unnatural data structures with too many layers of abstraction.

```
package freenet.support;

public class DoublyLinkedList<T> {
    private int size;
    private Item<T> head, tail;
    public int size() { return size; }
    public java.util.Enumeration elements() {
        return new java.util.Enumeration() {
            protected Item next = head;
            public boolean hasMoreElements() {
                return next != null;
            }
            public Object nextElement() {
                if (next == null)
                    throw new java.util.NoSuchElementException();
                Item result = next;
                next = next.next;
                return result;
            }
        };
    }
    public void push(T j) {
        // ...
        ++size;
    }
    static class Item<T> {
        Item next, prev;
        public final T value;
        Item(T val) { this.value = val; }
    }
}
```

Figure 2.24: The implementation after the first refactoring.

```
package freenet.support;

import freenet.support.DoublyLinkedList;

public class IntervalledSum {
    private final DoublyLinkedList<Report> l
        = new DoublyLinkedList<Report>();
    public void report(double d) {
        l.push(new Report(d));
    }
    static class Report {
        double value;
        Report(double value) { this.value = value; }
    }
}


package freenet.node;

import freenet.support.*;

public class LoadStats {
    private final DoublyLinkedList<LoadEntry> lru
        = new DoublyLinkedList<LoadEntry>();
    private final java.util.Map table
        = new java.util.HashMap();
    public void storeTraffic(byte[] nr, long rph) {
        LoadEntry le = new LoadEntry(nr, rph);
        table.put(le.fn, le);
        lru.push(le);
    }
    class LoadEntry {
        private final Object fn;
        private final long qph;
        private LoadEntry(byte[] b, long qph) {
            this.fn = b;
            this.qph = qph;
        }
    }
}
```

Figure 2.25: The client code after the first refactoring.

```
package freenet.support;

public class DoublyLinkedList<T> {
    private int size;
    private Item<T> head, tail;
    public int size() { return size; }
    public java.util.Enumeration<T> elements() {
        return new java.util.Enumeration<T>() {
            protected Item next = head;
            public boolean hasMoreElements() {
                return next != null;
            }
            public T nextElement() {
                if (next == null)
                    throw new java.util.NoSuchElementException();
                Item<T> result = next;
                next = next.next;
                return result.value;
            }
        };
    }
    public void push(T j) {
        // ...
        ++size;
    }
    static class Item<T> {
        Item next, prev;
        public final T value;
        Item(T val) { this.value = val; }
    }
}
```

Figure 2.26: The implementation code after the second refactoring.

2.9 Related Work

The the pervasiveness of aliasing in object-oriented programming languages results in a diverse set of programming problems that relate to aliasing. Consequently, there can hardly be only one design for controlling aliasing that solves all problems. Depending on the problem, solutions to controlling aliasing range from disallowing aliases [Wad90], restricting aliases to unique external aliases (but allowing an arbitrary amount of aliasing inside of the component) [CW03b, CW03a, Hog91, Alm97] to completely disallowing external aliases while allowing arbitrary aliasing inside of the component, as presented in this chapter. Other designs do not restrict aliasing in general, but try to control the way aliases can be used [NPV98, KM95, BNR01, BDF04, Mul01]. This chapter has described a particular way to control aliasing using confined types. Many alternative designs for controlling aliasing have been explored in numerous papers in recent years [Alm97, Alm99, CPN98, DLN96, GTZ98, Hog91, HLW92, KM95, NPV98]. These approaches differ widely in which operations with references are allowed and which aliasing patterns are permitted – as well as the complexity of the constraints that are imposed on the programmer. This section discusses the most relevant related work in more detail.

Bokowski and Vitek [BV99] introduced the notion of confined types. In their paper, confined types are explicitly declared. Their paper discussed an implementation of a source-level confinement checker based on Bokowski's CoffeeStrainer [Bok99]. The main difference between that work and the present chapter lies in the definition of anonymity. In both cases, anonymity rules are used to detect the confinement breaches from hidden widening of confined types to public types that can occur with inherited methods (rule $\mathcal{C}1$). However, the rules given by Bokowski and Vitek are much stronger than strictly necessary.

Consider the example of Figure 2.27. Notice that class `Parent` is public, and therefore cannot be confined. Intra-procedural analysis would not reveal that the expression `new NotConf().violation()` will widen `NotConfined` to `Parent`. Thus, Bokowski and Vitek chose to rely on explicit anonymity declarations and added an additional anonymity constraint:

$\mathcal{A}4$	Anonymity declarations must be preserved when overriding methods.

So, once a method is declared anonymous, all overriding definitions of that method have to abide by the constraints. When inferring anonymity, the rule $\mathcal{A}4$ is not necessary. The goal of $\mathcal{A}4$ was to ensure that anonymity of a method is independent from the result of method lookup. If anonymity of methods is

```
public class Parent {
     protected Parent nonAnonymousMethod() {
        return this; // violation of A1
    }
}

class NotConf extends Parent {
    Parent violation() {
        return nonAnonymousMethod(); // hidden widening
    }
}
```

Figure 2.27: Confinement violation $\mathcal{C}1$.

```
public class A { // A is not confined
    Object m() {
        // m() is anonymous in relation to C but not in relation to B
        return null;
    }
    public Object n() {
        return new C().m();
    }
}
class B extends A { // B is not confined
    Object m() { // m() is not anonymous
        return this;
    }
}
class C extends A { } // C is confined
```

Figure 2.28: Anonymity need not be preserved in all subtypes.

```
class Parent {
     protected Parent anonymousMethod() {
        return this; // not a violation of A1
    }
}

class Confined extends Parent {
    Parent noViolation() {
        return anonymousMethod(); // widening, but no escape
    }
}
```

Figure 2.29: Two confined classes.

inferred, dynamic binding can be taken into account. Figure 2.28 shows a confined class `C` that extends a class `A`. The method `A.m()` meets all anonymity criteria except for rule $\mathcal{A}4$. The violation of that rule occurs in class `B`, because `B` extends `A` and redefines `m()` with an implementation that returns `this`. The key point to notice here is that the anonymity violation cannot occur if the dynamic type of `this` is `A`. We say the method `A.m()` is anonymous *in relation* to `C`, but not in relation to `B`.

Another difference between the old and the new anonymity rules is that we allow widening of the `this` reference to other confined types. The old rules forbid returning `this` or using `this` as an argument completely. The new rules allow such cases if the type of the return value or the argument is again a confined type. An example is shown in Figure 2.29, which is a minimal variation of Figure 2.27 (`Parent` is no longer public). In this case, the new rules would allow both classes to be confined.

Noble, Vitek, and Potter [NPV98] presented flexible alias protection as a means to control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing-mode declarations specify constraints on the sharing of references. The mode `rep` protects *representation objects* from exposure. In essence, `rep` objects belong to a single owner object, and the model guarantees that all paths that lead to a representation object go through that object's owner. The mode `arg` marks argument objects which do not belong to the current owner, and therefore may be aliased from the outside. Argument objects can have different *roles*, and the model guarantees that an owner cannot introduce aliasing between roles.

Hogg's Islands [Hog91] and Almeida's Balloons [Alm97, Alm99] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference from [NPV98] is that both proposals strive for full encapsulation; that is, all objects reachable from an owner are protected from aliasing. This is equivalent to declaring everything inside an Island or Balloon as `rep`. This is restrictive, since it prevents many common programming styles; it is not possible to mix protected and unprotected objects as is done with flexible alias protection and confined types. Hogg's proposal extends Smalltalk-80 with sharing annotations, but it has neither been implemented nor formally validated. Almeida presented an abstract interpretation algorithm to decide if a class meets his balloon invariants, but it has also so far not been implemented. Balloon types are similar to confined types in that they only require an analysis of the code of the balloon type and not of the whole program.

Boyland, Noble and Retert [BNR01] introduced capabilities as a uniform system to describe restrictions imposed on references. Their system can model many of the different modifiers used to address the aliasing problem such as immutable, unique, readonly or borrowed. They also model a notion of anonymous references, which is different from the one used in this chapter. Their system of access rights cannot be used to model confined types, mainly because it lacks support for modeling package-scoped access.

Kent and Maung [KM95] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at runtime. Barnett et al. [BDF04] used a simple notion of ownership as the basis for an approach to specifying and checking properties stated as pre- and post-conditions for methods and object invariants; in their system the checking of ownership is itself a proof obligation. Additionally, Müller [Mul01] used ownership in support of verification, but in this case, checked by a type system.

In the field of static program analysis, a number of techniques have been developed. Static escape analyses such as the ones proposed by Blanchet [Bla99, Bla03] and others [BH99, Deu95] provide much more precise results than our technique, but come at a higher analysis cost. They often require whole program analyses, and are sensitive to small changes in the source code.

Clarke, Potter, and Noble [Cla01, CPN98] formalized representation containment by means of ownership types. Their seminal paper has sparked much interest and many papers have explored ownership types since then. Ownership types enforce the rule that all paths from the root of an object system must pass through an object's owner. The paper of Clarke, Potter, and Noble [CPN98] allowed just three annotations, `rep`, `norep`, and `owner` for specifying ownership, while later papers have introduced additional or alternative annotations [AKC02, CW03b, LM04, MP99]. Ownership types are inherently more flexible than confined types, while experiments with inferring ownership types (for example using the approach of Aldrich, Kostadinov, and Chambers [AKC02]) indicate that confined types lead to more scalable inference. Ownership types have been used as the basis for specifying a variety of properties via types, such as the absence of data races and deadlocks [BLR02, BSB03].

Most of the approaches mentioned above use operational semantics to reason about alias protection and ownership. Banerjee and Naumann [BN02] used denotational semantics to prove a representation-independence theorem – that is, a result about whether a class can safely be replaced by another class, independently of the program in which the class occurs. They use a syntactic notion of confinement (as we do) in which the protection domain is an instance rather than a package. Their notion of confinement is more restrictive than ours and it leads to a powerful theorem about classes.

CHAPTER 3

Memory Safety through Region Types

Proper handling of memory has been a challenge for programmers and programming languages since the earliest days of computing. In the last few decades, language systems featuring automatic memory management have become mainstream. Using a combination of techniques including garbage collection, type systems and dynamic checks, modern languages prevent programmers from accessing unallocated or deallocated parts of memory, a guarantee which is commonly called *memory safety*. While memory safety can significantly improve programmer productivity by eliminating a large class of programming errors, memory safety comes at a price in terms of performance. Unlike types, garbage collection and dynamic checks cost significant amounts of execution time and memory.

Ensuring memory safety will become an even bigger challenge for the next generation of hardware. Given that processor clock rates are unlikely to increase significantly in the future, performance improvements from larger numbers of transistors will have to be obtained by increasing parallelism. In addition to instruction-level parallelism, existing systems make use of multi-core and multi-processor designs. The extreme case involves clusters for high performance computing (HPC) consisting of tens of thousands of processors. The cost of accessing data in these parallel and distributed systems which feature a complex memory hierarchy is non-uniform. Accesses to the same location in the global memory may vary by as much as five orders of magnitude depending on the execution core performing the access. Consequently, the notion of a single shared memory with uniform access is inappropriate for these systems. Language systems that are to perform well on this new hardware and guarantee memory safety will not only have to perform parallel, distributed garbage collection, but will also have to ensure data locality for memory accesses.

Current object-oriented language facilities for concurrent and distributed programming fail to address these requirements of modern and next-generation parallel machines (SMPs and clusters). Given the likelihood that the majority of desktop systems in the future will be multi-core SMP designs, there is a need for new language systems that simplify the development and deployment of computations spanning multiple nodes. In order to be able to give a sufficiently simple and adequate cost model to programmers, a new language should require that basic read and write operations are local with respect to the place of execution. If all basic read and write operations are guaranteed to be local by the language, programmers will continue to be able to reason about the performance of their algorithms with respect to memory accesses.

This chapter describes a type system for such new languages that will help the compiler statically reason about the locality of memory accesses. Specifically, we will use types to eliminate the additional dynamic checks that are commonly used to ensure memory safety (including access locality) of array accesses. The type system specifically includes the possibility that the elements of the arrays can be distributed among the nodes of the system. Ensuring locality of access to objects is a simple extension of the presented formalism.

Dynamic checks to ensure memory safety of array accesses are of particular interest because they not only reduce performance for common operations but also represent programming errors that are only partially prevented by the language. Instead of giving the programmer compile-time guarantees that array accesses are safe, these checks may still fail at run-time.

Our work is heavily influenced by X10 [ESS04, ESS05, CDE05]. X10 is an experimental new memory-safe language for high performance computing (HPC) currently under development at IBM in collaboration with academic partners. The primary design goals for X10 are programmer productivity and performance. One reason that X10 requires applications to be memory safe – an unusual property for HPC languages – is the desire to ensure programmer productivity by eliminating the entire category of difficult-to-find errors due to memory corruption. Performance is facilitated in X10 with a design that ensures data locality. X10 uses the model of a partitioned global address space. The language features the abstract notion of a *place* to denote the location at which computations are executed. Each partition of the global address space is associated with a particular place, and X10 mandates that all accesses to mutable data must take place at the current node, or, in X10 terminology, be place-local. Furthermore, X10 features language constructs for explicit parallel and distributed computations. These language constructs reduce the complexity of developing HPC applications by replacing complex libraries, such as MPI [SLG99], that are traditionally used to manage parallelism and data distributions. Using language constructs instead of libraries also theoretically enables a tighter integration between the language system and data distribution mechanisms. Naturally, supporting parallel distributed computations requires support for distributed data structures – for arrays, in particular. In X10, arrays can be scattered over multiple places. X10's requirement that all accesses to mutable data must be place-local includes distributed arrays. Consequently, accesses to X10 arrays must not only be in-bounds (X10 is memory-safe), but must also be executed locally (X10 is place-safe) at the respective place associated with the index by the distribution of the array.

X10 is different from earlier HPC languages in that it requires the programmer to write code that ensures data locality and enables programmers to ensure data locality by providing constructs to programatically change the place of execution (this is called place-shifting in X10 terminology). Having a general place-shifting mechanism is good news for application programmers, since it affords them a great deal of flexibility. However, it does make it more difficult for the language system to generate fast, safe code for these general distributed parallel computations.

A central problem in this context is to statically check that memory accesses are local in the presence of distributed arrays and place-shifting computations. This chapter shows how to solve this problem using a new type system which employs dependent types over a particular vocabulary of constraints. There is a variety of constraint systems that can be chosen for this purpose; for the core language presented in this chapter, we chose a small operational algebra for illustration purposes which is sufficient to cover several fundamental examples. Changes to the constraint system would impact the complexity of the type checker, but do not change the core ideas necessary for the type soundness proof.

If, instead of using a type system, X10's requirements for memory safety and place safety were addressed using the traditional approach of generating dynamic safety checks, performance would be impacted significantly. Because array access operations are among the most frequent operations in scientific applications, their performance is critical; this makes elimination of the safety checks for array accesses important. Experiments show that simple bounds checks in languages like Java can cause performance hits of up to a factor of two. Dynamic checks would be even more costly in X10, since arrays are allowed to be sparse and distributed. The type system approach presented in this chapter proves programs to be memory-safe and place-safe in the absence of such checks. Compared with using traditional static analyses to eliminate some of these checks, this approach improves programmer productivity by moving detection of safety violations from runtime to compile time and ensuring that performance is consistently good.

The results of this chapter are the establishment of an applied dependently-typed lambda calculus [AH05] that can be used to establish locality of access for distributed arrays in a computation with place-shifting operations. Our results substantially generalize the pioneering work of Pfenning and Xi [XP98, XP99] on dependently typed programming languages (by applying them to a clustered setting and extending the constraint domain to include regions) and the work of Liblit and Aiken [LA00] (by covering place-shifting operations). The type system integrates dependent types with a new class of constraints over points, regions of points, and places. For this system, we have proven type soundness and settled the complexity of the key decision problems. The key operation during type checking is constraint entailment; type checking is co-NP-complete. The chapter illustrates this type system with various examples.

We have implemented the type system in an experimental compiler for X10 called XTC-X10. The implementation extends the standard object-oriented type system of X10 with the dependent types of the core language presented in this chapter. We have measured how many dynamic checks can be eliminated with this extension as a result of the static proof for several programs. The chapter presents encouraging experimental results that show that the type system can be used to effectively eliminate dynamic checks and statically ensure memory safety and locality of access, resulting in a substantial reduction in dynamic checks. This shows that use of the type system is a practical approach to eliminating checks for place safety and memory safety.

Chapter overview This chapter is structured as follows. Section 3.1 will give a brief introduction to X10, covering the features relevant to the type system extension presented in particular. Section 3.2 will then give an overview of the basic ideas behind the type system. Section 3.3 presents various examples to illustrate the core of the type system. A core language that is useful for formalizing the type rules is presented in Section 3.4, followed by a type soundness proof in Section 3.5. Extensions of the basic region algebra used in the core language are presented in Section 3.6. Implementation details are described in Section 3.7. Section 3.8 gives experimental results for the prototype implementation. Related work is discussed in Section 3.9

3.1 Background: X10

This section gives a condensed version of the description of the X10 language as it relates to the type system presented in this chapter. The formulations in this section are simplified from the original description of X10 in [CDE05]. The simplifications were made to focus the discussion on the relevant features of the language with respect to the type system.

3.1.1 Overview

X10 starts with a state-of-the-art object-oriented programming model and type system. To address non-uniformities in memory access, X10 introduces a notion of *places*. A place is a repository for data and the activities that operate on

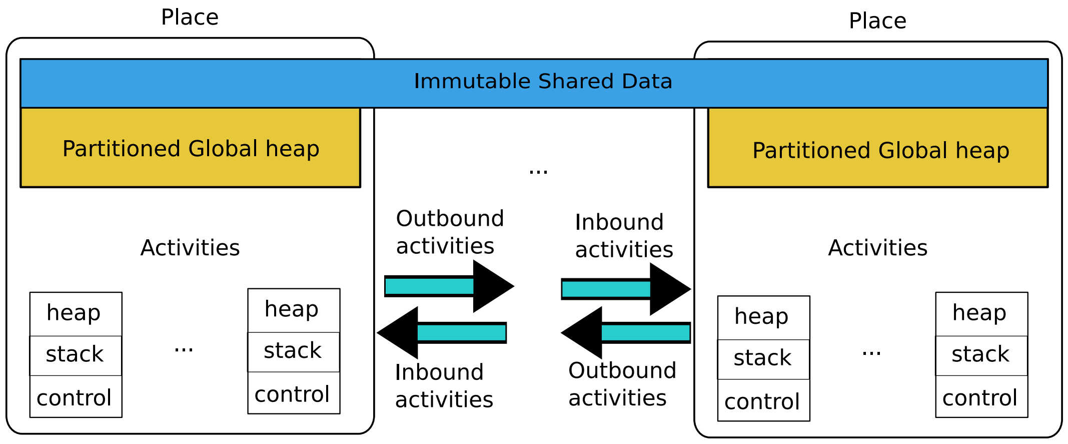

Figure 3.1: Overview of X10 Places and Activities, simplified from [CDE05].

the data. Parallel computation is enabled using asynchronous operations called *activities*. A simple high-level notion of shared-memory interaction is provided in the form of *conditional atomic sections* which are inspired by the conditional critical regions of Hoare [Hoa74] and Brinch Hansen [Han72]. Activities can be coordinated across multiple places with X10's *clocks*, which are generalization of SIMD barriers. X10 permits multi-dimensional arrays to be distributed over multiple places. The shape of arrays is specified using regions, a concept from the language ZPL [CCD04].

The next sections will detail X10 constructs which are important for the type system presented in this chapter.

3.1.2 Places and Activities

Figure 3.1 contains a schematic overview of places and activities in the X10 programming model. An X10 computation acts on *data objects* through the execution of asynchronous lightweight threads called *activities*. A central new concept in X10 is that of a *place*. A place can be thought of as a collection of resident (non-migrating) activities and mutable data objects. X10 has a *partitioned global address space* (PGAS) that spans all the places in the program. An object in the PGAS is allocated at a specified place, but may be referenced by activities at other places. X10 supports a globally asynchronous and locally synchronous model for data access, which enables an activity to synchronously read and write data items in the (local) place where the activity is running but requires that all accesses to remote data be performed asynchronously. Though an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in both the current and remote places. A remote location can be written into only by asynchronously spawning an activity to run at that location and perform the write operation. Asynchronous activities are not limited to a single read or write operation – any X10 program is initiated by spawning of and then waiting for termination of an asynchronous activity.

3.1.2.1 Activity spawning

An X10 computation is initiated as a single activity from the command line. This activity is the *root activity* for the entire computation.

An asynchronous activity is created by a statement `async (P) S` where `P` is a place expression and `S` is a statement. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`. Statement `S` may reference variables in lexically enclosing scopes. An activity A executes the statement `async (P) S` by launching a new activity B at the designated place. Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the pool of activities at the target place and will be executed in sequence or in parallel based on the local scheduler's decisions. If the programmer wishes to sequence their execution, he must use X10 constructs such as clocks and `finish` to obtain the desired effect.

For example, the X10 statement,

```
async (A[99]) { A[99] = k }
```

creates a new activity (assignment of k to $A[99]$) at the place containing element `A[99]` of a global distributed array `A`. The values of local variables such as `k` are passed as implicit parameters to this activity.

3.1.2.2 Objects

In X10, objects are of two kinds: scalar and aggregate. A *scalar* object has a statically fixed set of fields, each of which has a distinct name. Such an object is located at a single place and stays at that place throughout its lifetime. An *aggregate* object has many fields (the number may only be known after the object has been created), is uniformly accessed through an index (e.g. a point), and may be distributed across many places. The distribution of an aggregate object remains unchanged throughout the program's execution. X10 assumes an underlying garbage collector will dispose of (scalar and aggregate) objects and reclaim the memory associated with them once it can be determined that these objects are no longer accessible from the current state of the computation. There are no operations in the language to allow a programmer to explicitly release memory.

3.1.3 Arrays, Regions and Distributions

A *point* is an n-tuple of integers. n, in this context, is called the *rank* of the point. Points are used in array index expressions to select a particular array element. When used as an argument for accessing an array, a point is also often called an *index*.

A *region* is a set of points. For instance, the region `[0,1:200,100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`. In X10, operations are provided to construct regions from other regions and to iterate over regions. Standard set operations, such as union, intersection and set difference are available for regions. An X10 array is a function from a *region* to a base type (which may itself be an array type).

A *distribution* is a map from a region to a subset of places. For example, a distribution that maps all of the points from region `r` to place `p` can be specified as `d = r * p;`. The domain of a distribution is a region. That region can be accessed using the `reg` field of the distribution. In X10, regions and distributions are values. A primitive set of distributions is provided, together with operations on distributions. For example, programmers will use the `blocked` function to obtain a distribution that partitions the points of a region evenly among all of the places available:

```
region r = ...;
distribution d = distribution.blocked(r);
```

The distribution of an array specifies at which place it is legal to access a particular element of the array. A distribution must be provided when creating an X10 array. X10 allows array constructors to iterate over the underlying distribution and specify a value at each item in the underlying region. Such a constructor may spawn activities at multiple places.

3.1.4 Foreach and Ateach

X10 features k-dimensional versions of iteration operations, `for` and `foreach`.

```
for (p : r) { S }
foreach (p : r) { S }
```

The expression `r` is of type `region` and `p` is a fresh variable of type `point` that is available in the body `S` of the iteration. An activity executes a `for` statement by enumerating the points in the region in canonical order. The activity executes the body of the loop with the formal parameter bound to the given point. If the body terminates successfully, the activity continues with the next iteration, terminating successfully when all points have been visited. An activity executes a `foreach` statement in a similar fashion except that separate `async` activities are launched in parallel in the local place for each point in the region. The statement terminates when all the activities have been spawned.

In an `ateach` statement, the expression is intended to be of type `distribution`. This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, `ateach(p : d) S` may be thought of as standing for:

```
foreach (p : d)
    async (d[p]) { S }
```

3.1.5 Examples

The `init` function given below initializes all of the elements of an `Array` of `int`egers to `1`. The `finish` keyword is used to ensure that the `async`hronous communication is terminated before the next iteration of the `for` loop. The access to `a.dist[p]` returns the place at which the element of the array `a` corresponding to point `p` is stored. This code is memory safe because its construction ensures that all accesses are in-bounds and place-local.

```
void init(Array<int> a) {
    for (p : a.dist.reg) {
        finish async(a.dist[p]) {
            a[p] = 1;
        }
    }
}
```

The type system presented in this chapter can be used to statically check that code is memory safe. The `init` function type checks in our extended type system without modifications.

An example of a function that is generally not memory safe is the following `copy` function (here given in `X10` without region type annotations):

```
void copy(Array<int> a, Array<int> b) {
    finish ateach (h : distribution.unique()) {
        for (p : a.dist % here) {
            a[p] = b[p];
        }
    }
}
```

The `copy` function takes two arrays `a` and `b` and copies the elements of *b* into the array `a`. The function uses the built-in `distribution.unique()` function to obtain a bijective distribution that contains exactly one one-dimensional point per place in the `X10` runtime which is then mapped to the respective place. Using the `ateach` statement, the main computation is then executed in parallel at each of these places. The `% here` operation on the distribution of the array ensures that each of the distributed and parallel executions of the sequential `for` loop only iterates over those points of the distribution that are mapped to the current place of execution.

The `copy` function assumes that the two arrays `a` and `b` have the same dimensionality, the same underlying region and the same distribution. Otherwise, the accesses to `b[p]` might be non-local, out-of-bounds or even using a point of the wrong rank. These assumptions are not explicit in the source code. The extended type system presented in this chapter, in contrast, requires the programmer to use types to declare these implicit assumptions. These type annotations could be used to document the limitations of the method and allow the language system to statically guarantee memory safety. Traditional language implementations would instead have to generate dynamic checks to ensure memory safety. Such dynamic checks result in runtime errors and a loss of performance, both of which are eliminated by the type system presented in this chapter.

Additional examples in `X10` syntax using the type system extension presented in this chapter can be found in Section B.1.

3.2 Dependent Types for Regions

As the description of `X10` illustrates, high performance computing (HPC) languages contain a rich language for arrays which are a dominant data-structure in the HPC space. Languages like ZPL [CCL00, Cha01, CS01, DCS02], Titanium [UCB05], and X10 provide the programmer with a rich algebra of region operators to manipulate arrays. A programmer can use regions to specify computations on dense or sparse multidimensional and hierarchical arrays. While convenient, regions do not eliminate the risk of array bounds violations. Until now, language implementations have resorted to checking array accesses dynamically or to warning the programmer that bounds violations lead to undefined behavior. In contrast, for performance and productivity, we prefer that array computations be statically checked to be safe.

The type system presented in this chapter enables the programmer to use concise type annotations to provide useful documentation that allows the compiler to eliminate all safety checks, resulting in faster, statically-checked code. The type system uses the operations of the region algebra as a high-level abstraction which it can exploit for its reasoning. Code type-checks if accesses are performed in a context in which the index can be statically established to be in the region over which the array is defined. X10's region-based iterators such as `for` and `foreach` often provide such a context. For instance in the statement `for (x : r) s` it is the case that within `s` one may assume that `x` lies in the region `r`.

Operations on region values are mapped to corresponding operations on region types. The mapping is defined such that subset relations for region values correspond to subtyping of their respective types. Establishing that an index is in-bounds for a particular array is equivalent to establishing that the region over which the index may range is a subset of the region over which the array is defined. A subtyping relationship between the respective region types implies the desired subset relationship and can thus be used to statically prove the safety of the access.

Furthermore, because our type system needs to take data locality into account, the type system supports the notion of *distributions*: functions from array indices to places. Distributions are used to determine how data stored in arrays is distributed among the places of the computation. The core language features various operations on distributions that are used to ensure locality of access.

Using the new type system requires programmers to write code in such a way that regions and region operations are written explicitly instead of the traditional integer arithmetic. However, this is a small price to pay; in our experience, refactoring programs to use regions often results in code that is easier to understand and more generic.

The next section describes a core language – with formal syntax, semantics and type rules – that is sufficient to illustrate the main issues, including the required kinds of constraints, interesting type rules and essential operations.[1] The semantics of our core language is such that a locality or bounds violation results in the semantics getting stuck. The main contribution of this chapter is a type system which guarantees statically that such violations cannot occur. Thus, our type system enables us to statically verify that code will be safe and fast (in the sense of only accessing local data).

3.3 Example Programs

We will present our core language and type system via six example programs. This section describes the example programs in the syntax of the core language, Section B.1 in Appendix B gives the same example programs using X10 syntax. The first five example programs all type check, while the sixth program (`shift`) does not. We use functions of the form $\lambda^\bullet$x.e which run at the place where they are defined. Our core language also has functions $\lambda x.e$ which run at the place where they are called. Similarly, we use dependent expressions lam$^\bullet\alpha.e$ for which the body will be evaluated at the place where the dependent expression was defined. Additionally, our core language has dependent expressions `lam` $x.e$ which are evaluated at the place they are called.

The function `init` shown in Figure 3.2 initializes all points in an array to 1. The function `init` takes two arguments, namely a region α and an array over region α. The use of the dependent type α makes `init` polymorphic: `init` can initialize any array without the need for any bounds checking. The expression `a.reg` has type `reg` α and `p` which ranges over `a.reg` has type `pt` (σ, α), where α is the important part and σ can be ignored here. At the time of the assignment to `a[p]`, we have that the type of `p` matches the type of the region of `a`. For a point p in a region r we use $r[@p]$ to denote the place at which the element corresponding to index p is stored for arrays over region r. This occurs in `init` in the loop body, which uses `at(a.reg[@p])` { `a[p]=1` } to do the computation of `a[p]=1` at the place of `a[p]`. This is a common idiom in the benchmarks we have studied.

The function `partialinit` (shown in Figure 3.3) allows partial initialization of an array. It takes two extra arguments, namely a place type variable γ and a place value `h` of the singleton place type `pl` γ. The body of `partialinit` initializes those points in the argument array which can be found at the place `h`. In our core language, every region comes with a predefined mapping, called

[1]While the core language is rather large for such a formalization, it is still simplified in that it does not include objects, subtyping and many additional region operators, all of which are handled by our implementation.

```
let init = lam•α : region.λ•a:int[α].
     for (p in a.reg) {
          at(a.reg[@p]) { a[p]=1 } }
in init<reg 0:9>(new int[0:9])
init:  Πα : region. int[α] → int
```

Figure 3.2: Example programs: init.

distribution, of points to places. The expression a.reg % h denotes those points in a.reg which are mapped to the place h by the distribution of a.reg. The for loop iterates only over points in a.reg % h, and since the for loop is wrapped in **at**(h) { ...}, each access a[p] will happen at the place of a[p]. The type of a.reg is reg α and the type of h is pl γ. As a result, the type of a.reg % h is $\alpha \%_t \gamma$, which illustrates that we use a type operator to mirror the expression operator. The variable p then gets the type pt $(\beta, \alpha \%_t \gamma)$, where β is a fresh variable that denotes the type-level identity of p. When we type check the access a[p], the region check determines that the region of p, namely α, is a subset of the region of a, which is also α. For a[p], the place check determines that the current place of execution is the same as the place of a[p]. The place of execution is given by the enclosing **at**(h) expression, and we have that h has type pl γ. The type of the place of a[p] is given by the type expression $\alpha[@_t(\beta, \alpha \%_t \gamma)]$, which says that the place is that of a point in α which has its data located at place γ. So, we can use the type equivalence $\alpha[@_t(\beta, \alpha \%_t \gamma)] \equiv \gamma$ to conclude that the place of execution is indeed the same as the place of the data a[p]. Note the use of $<$ and $>$ to indicate dependent application in contrast to the use of parenthesis for normal function applications.

```
let partialinit = lam•γ:place.λ•h:pl γ.lam•α:region.λ•a:int[α].
     at(h) { for (p in a.reg % h) { a[p]=1 } }
in partialinit <P>(P)<reg 0:9>(new int[0:9])
partialinit:  Πγ:place.pl γ
     → (Πα:region.int[α] → int)
```

Figure 3.3: Example programs: partialinit.

The function copy (Figure 3.4) takes two arrays a and b, both with region α. The body of copy copies elements from b to a. The body of copy uses the construct **forallplaces** h { ...} which iterates over all places available to the program. For each place, the code copies elements that reside at that place. Notice that since a and b have the same region, they also have the same distribution, so for a given point p in that region, both a[p] and b[p] will be at the same place.

```
let copy = lam•α:region.λ•a:int[α].λ•b:int[α].
     forallplaces h { at(h) {
          for (p in (a.reg  %  h)) { a[p] = b[p] } } }
in copy<reg 0:7>(new int[0:7])(new int[0:7])
copy:  Πα:region.int[α] → (int[α] → int)
```

Figure 3.4: Example programs: copy.

The function expand (Figure 3.5 takes an array a and region x, where x must be a superset of the region of a, and creates and returns a new array b over the region x. The function expand partially initializes the new array b with values from a at overlapping points. expand is interesting in that it highlights the importance of keeping upper and lower bounds for the region of arrays during type checking. Notice that argument β comes with the constraint $\alpha \subseteq_t \beta$, which means that the region α must be a subset of the region β. The subscript t is used to indicate that this is a relationship defined on the *t*ype level. The call `partialinit <P>(P)<reg 0:9>(new int[0:9])<reg 1:8>(reg 1:8)` is a good example of the kind of reasoning that the programmer has to do when programming directly in the core language; the call satisfies the constraint $\beta \subseteq_t \alpha$ because $[1:8] \subseteq [0:9]$.

```
let expand = lam•α:region.λ•a:int[α].
              lam•β:region (α ⊆_t β).λ•x:reg β.
     let b = new int[x]
     in { forallplaces h { at(h) {
          for (p in a.reg ∩_s (b.reg % h)) {
               b[p] = at (a.reg[@p]) { a[p] } } } } ; b }
in expand<reg 3:7>(new int[3:7])
          <reg 0:10>(int[0:10])
expand:  Πα:region.int[α]
     → (Πβ:region(α ⊆_t β).reg β → int[β])
```

Figure 3.5: Example programs: expand.

The function shiftleft (Figure 3.6 takes an argument a with region α and shifts all elements one position to the left, while leaving the rightmost element unchanged. In more detail, shiftleft first creates an inner region of α shifting all elements of α by one to the right $(\alpha + 1)$ and then intersecting the result with α. If α is simply an interval, this effectively removes the first element from α. Then shiftleft proceeds with doing `a[p-1] = a[p]` for each point p in the inner region. The inner region has type reg $((\alpha + 1) \cap_t \alpha)$. The expression p-1 is always within the region of a because p-1 has type pt $(((\alpha + 1) \cap_t \alpha) - 1)$ and therefore also, via subtyping, the type pt α (because $+1$ and -1 cancel each other out). Similarly, the expression p is always within the region of a because p has type pt $(\alpha + 1) \cap_t \alpha)$ and therefore also, again via subtyping, the type pt α.

```
let shiftleft = lam$^\bullet\alpha$:region.$\lambda^\bullet$a:int[$\alpha$].
     let inner = (a.reg + 1) $\cap_s$ a.reg
     in { for (p in inner) { at(a.reg[@p-1]) {
          a[p-1] = at(a.reg[@p]) { a[p] } } } } }
in shiftleft<reg 3:7>(new int[3:7])
shiftleft:  $\Pi\alpha$:region.int[$\alpha$] $\rightarrow$ int
```

Figure 3.6: Example programs: shiftleft.

The program shift (Figure 3.7) is a small variation of shiftleft that contains a bug which would result in an array bounds violation – and that consequently does not type check. The problem with shift is that the array access a[p+1] will be out of bounds when p reaches the end of the array.

```
let shift = lam$^\bullet\alpha$:region.$\lambda^\bullet$a:int[$\alpha$].
     let inner = (a.reg + 1) $\cap_s$ a.reg
     in { for (p in inner) {
          at(a.reg[@p+1]) {
               a[p+1] = at(a.reg[@p]) { a[p] } } } }
in ...
```

Figure 3.7: Example programs: shift.

3.4 The Core Language

We now present the syntax, semantics, and type system of our core language. In Section 3.5, we prove type soundness using the standard technique of Nielson [Nie89] and others that was popularized by Wright and Felleisen [WF94].

The core language models the features of X10 that are most relevant for our type system extension. Specifically, the semantics capture the flat, distributed memory model of X10. The language includes X10's distributed arrays and the most interesting operations from X10's region algebra. In order to be able to give simple, deterministic semantics to the core language, X10's asynchronous remote computations are modeled as synchronous remote computations. Objects and synchronization constructs are not included in the core language in order to keep the definition concise. Other languages that feature regions are likely to contain

$$
\begin{array}{lcll}
\texttt{(Kind)} & k & \texttt{::=} & \texttt{point}\ \varphi \mid \texttt{region}\ \varphi \mid \texttt{place} \\
\texttt{(Type)} & t & \texttt{::=} & \texttt{int} \mid \texttt{pt}\ (\sigma, r) \mid \texttt{reg}\ r \mid t[r] \\
 & & \mid & \texttt{pl}\ \pi \mid t \to t \mid \Pi\alpha : k.t \\
\texttt{(Region)} & r & \texttt{::=} & \alpha \mid R \mid r \cup_t r \mid r \cap_t r \\
 & & \mid & r +_t c \mid r \,\%_t\, \pi \\
\texttt{(Point)} & \sigma & \texttt{::=} & \alpha \mid p \mid \sigma \mathbin{++}_t c \\
\texttt{(Place)} & \pi & \texttt{::=} & \alpha \mid P \mid r[@_t(\sigma, r)] \mid \mathit{unknown} \\
\texttt{(Constraint)} & \varphi & \texttt{::=} & r \subseteq_t r \mid \sigma \in_t r \mid \varphi \wedge \varphi \\
 & & & \\
\texttt{(Value)} & v & \texttt{::=} & c \mid p \mid R \mid l \mid P \mid \lambda x : t.e \\
 & & \mid & \texttt{lam}\ \alpha : k.e \\
\texttt{(ValOrVar)} & y & \texttt{::=} & v \mid x \\
\texttt{(Expression)} & e & \texttt{::=} & y \mid e_1\ e_2 \mid e_1\texttt{<}e_2\texttt{>} \\
 & & \mid & \lambda^{\bullet} x : t.e \mid \texttt{lam}^{\bullet}\alpha : k.e \\
 & & \mid & \textbf{new}\ t[e] \mid y_1[y_2] \mid y_1[y_2] = e \\
 & & \mid & e\texttt{.reg} \mid y_1[@_s y_2] \\
 & & \mid & e_1 \cup_s e_2 \mid e_1 \cap_s e_2 \mid e +_s c \\
 & & \mid & e \mathbin{++}_s c \mid y_1 \,\%\, y_2 \\
 & & \mid & \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\} \\
 & & \mid & \textbf{forallplaces}\ x\{e\} \\
 & & \mid & e_1; e_2 \mid \textbf{at}(y)\{e\} \\
\texttt{(Dep Val)} & w & \texttt{::=} & p \mid R \mid P
\end{array}
$$

Figure 3.8: Syntax of the core language.

a similar core language – possibly without the locality constraints for memory accesses in the operational semantics. Relaxing the type rules to match such a simplification of the operational semantics is straightforward.

3.4.1 Syntax

Figure 3.8 gives the syntax for the core language. We use c to range over integer constants, p to range over point constants, R to range over region constants (such as [1:4], which denotes $\{1, 2, 3, 4\}$), l to range over array labels drawn from a set `Label`, P to range over place constants, x to range over variable names, and α to range over type-variable names. In our core language, points are integers, and we will occasionally write a point constant as c. For shifting a region by a constant we use the notation $\{c_1, \ldots, c_n\} + c = \{c_1 + c, \ldots, c_n + c\}$.

The language has seven data types, namely integers, points, regions, arrays, places, functions, and dependently-typed functions. We have deliberately avoided having distributions as values, in an effort to keep the size of the language manageable. We assume a function *distribute* which maps a region and a point in that region to a place. When we create an array over a region R, the array will be distributed according to the function *distribute*. We make no assumptions about *distribute*.

The types are defined in terms of three forms of expressions which, given an interpretation of the variables, evaluate to sets of points (regions), points, and places, respectively. Specifically, if ρ is a mapping from region variables to regions, point variables to points, and place variables to places, then the meaning of the expressions is given as follows:

$$\begin{aligned}
\alpha\rho &= \rho(\alpha) \\
R\rho &= R \\
(r_1 \cup_t r_2)\rho &= r_1\rho \cup r_2\rho \\
(r_1 \cap_t r_2)\rho &= r_1\rho \cap r_2\rho \\
(r +_t c)\rho &= r\rho + c \\
(r \ \%_t\ \pi)\rho &= \{\ p \in r\rho \mid \text{distribute}(r\rho, p) = \pi\rho\ \} \\
p\rho &= p \\
(r \mathbin{++}_t c)\rho &= r\rho + c \\
P\rho &= P \\
(r_1[@_t(\sigma, r_2)])\rho &= \text{distribute}(r_1\rho, \sigma\rho).
\end{aligned}$$

The expression $r \ \%_t\ \pi$ evaluates to a subset of r which contains those points which are mapped to π by *distribute*. The expression $r[@_t(\sigma, r)]$ evaluates to the place of the point σ according the distribution given by *distribute*.

The type of a point is a pair (σ, r) where σ is a type-level identity of the point and r is a region that contains the point. The type of a region is a singleton type consisting of that region itself. A dependently-typed function `lam` $\alpha : k.e$ has its argument constrained by the kind k; its type is $\Pi\alpha : k.t$.

The expression language contains syntax for creating and calling functions, for creating, accessing, and updating arrays, for computing with regions, for iterating over regions, for iterating over all places, and for shifting the place of execution. The expression e`.reg` returns the region of an array. The expression $e \mathbin{++}_s c$ adds a constant c to the point to which e evaluates. The expression $e +_s c$ adds a constant to each of the points in the region to which e evaluates.

We need the set operators to work both on types, expressions, and actual sets. In order to avoid confusion, we give each operator on types the subscript t, on expressions the subscript s, and on sets no subscript at all.

In the example programs earlier in the chapter, we used the syntactic sugar `let` x `=` e `in` $\{\ e'\ \}$ in order to represent $(\lambda^{\bullet}x.e')e$. We also used a few other constructs such as `p+1` which are not part of the core language but which could be added easily. We will use `true` to denote the tautology $\emptyset \subseteq_t \emptyset$.

3.4.2 Semantics

We specify the semantics of the core language using small-step operational semantics (see Figures 3.9, 3.10 and 3.11). We use H to range over heaps:

$$H \in \texttt{Label} \rightarrow \texttt{Point} \rightarrow (\texttt{Value} \times \texttt{Place})$$

A heap maps labels to array representations. An array representation maps each point in the region of the array to its value and its place. Both uses of $\rightarrow$ above denote a space of partial functions. We will use the notation (v, P) for elements of $(\texttt{Value} \times \texttt{Place})$, and we will use the operators $\_.1$ and $\_.2$ to extract the first and second element of a pair, respectively. We use $\mathcal{D}(H)$ to denote the domain of a partial function H.

A state in the semantics is a pair (H, e). We say that (H, e) can *take a step* at place P if we have H', e' such that $P \vdash (H, e) \leadsto (H', e')$ using the rules below. We say that (H, e) is *stuck* at place P if e is not a value and (H, e) cannot take a step at place P. We say that (H, e) *can go wrong* at place P if we have H', e' such that $P \vdash (H, e) \leadsto^* (H', e')$ and (H', e') is stuck at place P.

We assume that the programmer (externally to the program text) provides a function `default` which maps a closed type t to a value for each type t used as an element type of an array in the program. The function `default` must have the property that $\Psi; \varphi; \Gamma \vdash \texttt{default}(t) :\ t$ for a Ψ that contains suitable definitions of the labels used in $\texttt{default}(t)$, and for any φ and Γ. The idea is that we will use $\texttt{default}(t)$ as the initial value at all points in an array with element type t. While we can easily define examples of such a function `default`, we will not show a specific one, simply because all we need to know about it is the property $\Psi; \varphi; \Gamma \vdash \texttt{default}(t) :\ t$.

We also assume a list *places* of the places available during the execution of the program. The only thing a program can do with *places* is to iterate over the places using the `forallplaces` construct.

In order to specify the execution order for the for loop construct, Rule (3.29) uses a function $\text{order}(\{c_1, \ldots, c_n\}) = \langle c_1, \ldots, c_n \rangle$, where $c_1 < \ldots < c_n$.

The following rules define a call-by-value semantics and are mostly standard. The key rules (3.11) and (3.13) both have the side condition that $l \in \mathcal{D}(H)$ and $p \in \mathcal{D}(H(l))$ and $P = H(l)(p).2$. The condition $p \in \mathcal{D}(H(l))$ is the array-bounds

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1\ e_2) \rightsquigarrow (H', e_1'\ e_2)} \tag{3.1}$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e_2')}{P \vdash (H, v\ e_2) \rightsquigarrow (H', v\ e_2')} \tag{3.2}$$

$$P \vdash (H, (\lambda x : t.e)v) \rightsquigarrow (H, e[x := v]) \tag{3.3}$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1\texttt{<}e_2\texttt{>}) \rightsquigarrow (H', e_1'\texttt{<}e_2\texttt{>})} \tag{3.4}$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e_2')}{P \vdash (H, v\texttt{<}e_2\texttt{>}) \rightsquigarrow (H', v\texttt{<}e_2'\texttt{>})} \tag{3.5}$$

$$P \vdash (H, (\texttt{lam}\ \alpha : k.e)\texttt{<}w\texttt{>}) \rightsquigarrow (H, e[\alpha := w]) \tag{3.6}$$

$$P \vdash (H, \lambda^\bullet x : t.e) \rightsquigarrow (H, \lambda x : t.\textbf{at}\texttt{(}P\texttt{)}\{e\}) \tag{3.7}$$

$$P \vdash (H, \text{lam}^\bullet \alpha : k.e) \rightsquigarrow (H, \texttt{lam}\ \alpha : k.\textbf{at}\texttt{(}P\texttt{)}\{e\}) \tag{3.8}$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, \textbf{new}\ t[e]) \rightsquigarrow (H', \textbf{new}\ t[e'])} \tag{3.9}$$

$$P \vdash (H, \textbf{new}\ t[R]) \rightsquigarrow (H[l \mapsto \lambda p \in R.(\texttt{default}(t), \textit{distribute}(R, p))], l) \quad \text{where } l \text{ is fresh} \tag{3.10}$$

$$P \vdash (H, l[p]) \rightsquigarrow (H, H(l)(p).1) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2 \tag{3.11}$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, v_1[v_2] = e) \rightsquigarrow (H', v_1[v_2] = e')} \tag{3.12}$$

$$P \vdash (H, l[p] = v) \rightsquigarrow (H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]], v) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2 \tag{3.13}$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, e.\texttt{reg}) \rightsquigarrow (H', e'.\texttt{reg})} \tag{3.14}$$

$$P \vdash (H, l.\texttt{reg}) \rightsquigarrow (H, \mathcal{D}(H(l))) \qquad \text{if } l \in \mathcal{D}(H) \tag{3.15}$$

$$P \vdash (H, l[@_s p]) \rightsquigarrow (H, H(l)(p).2) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \tag{3.16}$$

Figure 3.9: Semantics of the core language (part 1).

check; p must be in the region of the array. The condition $P = H(l)(p).2$ is the place check; the place of execution must equal the place of the data to be accessed. If the side condition is not met, then the semantics will get stuck.

Notice that in Rule (3.19) we evaluate the syntactic expression $R_1 \cup_s R_2$ to the value $R_1 \cup R_2$.

$$\frac{P \vdash (H, e_1) \leadsto (H', e_1')}{P \vdash (H, e_1 \cup_s e_2) \leadsto (H', e_1' \cup_s e_2)} \tag{3.17}$$

$$\frac{P \vdash (H, e_2) \leadsto (H', e_2')}{P \vdash (H, v \cup_s e_2) \leadsto (H', v \cup_s e_2')} \tag{3.18}$$

$$P \vdash (H, R_1 \cup_s R_2) \leadsto (H, R_1 \cup R_2) \tag{3.19}$$

$$\frac{P \vdash (H, e_1) \leadsto (H', e_1')}{P \vdash (H, e_1 \cap_s e_2) \leadsto (H', e_1' \cap_s e_2)} \tag{3.20}$$

$$\frac{P \vdash (H, e_2) \leadsto (H', e_2')}{P \vdash (H, v \cap_s e_2) \leadsto (H', v \cap_s e_2')} \tag{3.21}$$

$$P \vdash (H, R_1 \cap_s R_2) \leadsto (H, R_1 \cap R_2) \tag{3.22}$$

$$\frac{P \vdash (H, e) \leadsto (H', e')}{P \vdash (H, e +_s c) \leadsto (H', e' +_s c)} \tag{3.23}$$

$$P \vdash (H, d +_s c) \leadsto (H, d + c) \tag{3.24}$$

$$\frac{P \vdash (H, e) \leadsto (H', e')}{P \vdash (H, e +\!\!+_s c) \leadsto (H', e' +\!\!+_s c)} \tag{3.25}$$

$$P \vdash (H, p +\!\!+_s c) \leadsto (H, p + c) \tag{3.26}$$

$$\begin{array}{c} P \vdash (H, R \,\%\, P']) \leadsto (H, R') \\ \text{where } R' = \{\, p \in R \mid \mathit{distribute}(R, p) = P' \,\} \end{array} \tag{3.27}$$

$$\frac{P \vdash (H, e_1) \leadsto (H', e_1')}{P \vdash (H, \textbf{for } (x \textbf{ in } e_1)\{e_2\}) \leadsto (H', \textbf{for } (x \textbf{ in } e_1')\{e_2\})} \tag{3.28}$$

$$\begin{array}{l} P \vdash (H, \textbf{for } (x \textbf{ in } R)\{e\}) \leadsto \\ (H, ((\text{lam}^\bullet \alpha : \texttt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e)\texttt{<}c_1\texttt{>})c_1; \ldots; \\ \quad ((\text{lam}^\bullet \alpha : \texttt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e)\texttt{<}c_n\texttt{>})c_n; 0) \\ \quad \text{where order}(R) = \langle c_1, \ldots, c_n \rangle \end{array} \tag{3.29}$$

Figure 3.10: Semantics of the core language (part 2).

Rule (3.29) unrolls the for loop and replaces the loop variable with an appropriate point in each copy of the body of the loop. Similarly, Rule (3.30) unrolls the loop and replaces the loop variables with an appropriate place in each copy of the body of the loop. The unrolling is specified the way it is to enable the type checker to assign a type variable as first/only part of the type of the loop variable and at the same time achieve that each iteration is executed using the exact value bound to the loop variable.

Rule (3.7) and Rule (3.8) express that the body of $\lambda^\bullet$ or lam$^\bullet$ must execute at the place of the definition. Effectively, each of those rules creates a closure consisting of the function and the current place of execution.

$$\begin{array}{l} P \vdash (H, \textbf{forallplaces}\ x\{e\}) \rightsquigarrow \\ (H, ((\text{lam}^{\bullet}\alpha : \texttt{place}.\lambda^{\bullet}x : \texttt{pl}\ \alpha.e)\texttt{<}P_1\texttt{>})P_1; \ldots; \\ \quad ((\text{lam}^{\bullet}\alpha : \texttt{place}.\lambda^{\bullet}x : \texttt{pl}\ \alpha.e)\texttt{<}P_n\texttt{>})P_n; 0) \\ \quad \text{where } \mathit{places}\ = \langle P_1, \ldots, P_n \rangle \end{array} \tag{3.30}$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1; e_2) \rightsquigarrow (H, e_1'; e_2)} \tag{3.31}$$

$$P \vdash (H, v; e) \rightsquigarrow (H, e) \tag{3.32}$$

$$\frac{P' \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, \textbf{at}\texttt{(}P'\texttt{)}\{e\}) \rightsquigarrow (H, \textbf{at}\texttt{(}P'\texttt{)}\{e'\})} \tag{3.33}$$

$$P \vdash (H, \textbf{at}\texttt{(}P'\texttt{)}\{v\}) \rightsquigarrow (H, v) \tag{3.34}$$

Figure 3.11: Semantics of the core language (part 3).

3.4.3 Heap Types

We use Ψ to range over maps from array labels to types of the form $t[R]$. We use the judgment $\models H : \Psi$ which holds if (1) $\mathcal{D}(H) = \mathcal{D}(\Psi)$ and (2) if for each $l \in \mathcal{D}(H)$ we let $t[R] = \Psi(l)$, then $\mathcal{D}(H(l)) = R$ and for each $p \in \mathcal{D}(H(l))$ we have (i) $\Psi; \varphi; \Gamma; \mathit{here} \vdash H(l)(p).\mathit{1} : t$ and (ii) $\mathit{distribute}(R, p) = H(l)(p).2$. We write $\Psi \lhd \Psi'$ if $\mathcal{D}(\Psi) \subseteq \mathcal{D}(\Psi')$ and Ψ, Ψ' agree on their common domain. Informally, $\models H : \Psi$ says that Ψ maps the label l of each array in the heap H to an array type $t[R]$ and the type of each element of the array is t.

3.4.4 Type Equivalence

We define type equivalence via the judgments $\varphi \vdash t \equiv t'$, $\varphi \vdash r \equiv r'$, $\varphi \vdash \sigma \equiv \sigma'$, and $\varphi \vdash \pi \equiv \pi'$, which hold if they can be derived using the rules in Figures 3.12 and 3.13. The first three rules use a meta-variable q which ranges over t, r, σ, π. The type equivalence rules simply state trivial identities that follow immediately from the operational semantics.

3.4.5 Satisfiability and Entailment

We use ρ to range over mappings from variables to sets. We say that ρ *satisfies* a constraint φ if for all $r_1 \subseteq_t r_2$ in φ we have $r_1\rho \subseteq r_2\rho$ and for all $\sigma \in_t r$ in φ we have $\sigma\rho \in r\rho$. We say that a constraint φ is *satisfiable* if there exists a satisfying assignment for φ.

We say that a constraint is *valid* if all variable assignments satisfy the con-

$$\varphi \vdash q \equiv q \tag{3.35}$$

$$\frac{\varphi \vdash q_1 \equiv q_2}{\varphi \vdash q_2 \equiv q_1} \tag{3.36}$$

$$\frac{\varphi \vdash q_1 \equiv q_2 \quad \varphi \vdash q_2 \equiv q_3}{\varphi \vdash q_1 \equiv q_3} \tag{3.37}$$

$$\frac{\varphi \vdash \sigma \equiv \sigma' \quad \varphi \vdash r \equiv r'}{\varphi \vdash \texttt{pt}\ (\sigma, r) \equiv \texttt{pt}\ (\sigma', r')} \tag{3.38}$$

$$\frac{\varphi \vdash r \equiv r'}{\varphi \vdash \texttt{reg}\ r \equiv \texttt{reg}\ r'} \tag{3.39}$$

$$\frac{\varphi \vdash \pi \equiv \pi'}{\varphi \vdash \texttt{pl}\ \pi \equiv \texttt{pl}\ \pi'} \tag{3.40}$$

$$\frac{\varphi \vdash t_1 \equiv t_1' \quad \varphi \vdash t_2 \equiv t_2'}{\varphi \vdash t_1 \to t_2 \equiv t_1' \to t_2} \tag{3.41}$$

$$\frac{\varphi \vdash t \equiv t'}{\varphi \vdash \Pi\alpha : k.t \equiv \Pi\alpha : k.t'} \tag{3.42}$$

$$\varphi \vdash R_1 \cup_t R_2 \equiv R_1 \cup R_2 \tag{3.43}$$

$$\frac{\varphi \vdash r_1 \equiv r_1' \quad \varphi \vdash r_2 \equiv r_2'}{\varphi \vdash r_1 \cup_t r_2 \equiv r_1' \cup_t r_2'} \tag{3.44}$$

$$\varphi \vdash R_1 \cap_t R_2 \equiv R_1 \cap R_2 \tag{3.45}$$

$$\frac{\varphi \vdash r_1 \equiv r_1' \quad \varphi \vdash r_2 \equiv r_2'}{\varphi \vdash r_1 \cap_t r_2 \equiv r_1' \cap_t r_2'} \tag{3.46}$$

$$\varphi \vdash R +_t c \equiv R + c \tag{3.47}$$

$$\frac{\varphi \vdash r \equiv r'}{\varphi \vdash r +_t c \equiv r' +_t c} \tag{3.48}$$

$$\varphi \vdash R \,\%_t\, P \equiv \{\ p \in R \mid \mathit{distribute}(R, p) = P\ \} \tag{3.49}$$

$$\frac{\varphi \vdash r \equiv r' \quad \varphi \vdash \pi \equiv \pi'}{\varphi \vdash r \,\%_t\, \pi \equiv r' \,\%_t\, \pi'} \tag{3.50}$$

$$\varphi \vdash p \mathbin{++_t} c \equiv p + c \tag{3.51}$$

$$\frac{\varphi \vdash \sigma \equiv \sigma'}{\varphi \vdash \sigma \mathbin{++_t} c \equiv \sigma' \mathbin{++_t} c} \tag{3.52}$$

Figure 3.12: Equivalence rules (1).

$$\frac{\varphi \models p \in_t r \quad \varphi \models r \subseteq_t R}{\varphi \vdash R[@_t(p, r)] \equiv \mathit{distribute}(R, p)} \tag{3.53}$$

$$\frac{\varphi \vdash r_1 \equiv r'_1 \quad \varphi \vdash \sigma \equiv \sigma' \quad \varphi \vdash r_2 \equiv r'_2}{\varphi \vdash r_1[@_t(\sigma, r_2)] \equiv r'_1[@_t(\sigma', r'_2)]} \tag{3.54}$$

$$\frac{\varphi \models \sigma \in_t r \,\%_t\, \pi}{\varphi \vdash r[@_t(\sigma, r \,\%_t\, \pi)] = \pi} \tag{3.55}$$

Figure 3.13: Equivalence rules (2).

straint. We say that φ *entails* φ' if the implication $\varphi \Rightarrow \varphi'$ is valid, and write $\varphi \models \varphi'$.

The *satisfiability problem* is this: given a constraint φ, is φ satisfiable? The *entailment problem* is as follows: given two constraints φ, φ', is $\varphi \models \varphi'$ true?

For our notion of constraints, the satisfiability problem is NP-complete. To understand this, first note that already for the fragment of region constraints with just variables, constants, union, and intersection, the satisfiability problem is NP-hard [AKV93]. Second, to show that the satisfiability problem is in NP we must first argue that we only need to consider sets of polynomial size; we can then guess a satisfying assignment and check that assignment in polynomial time. Let us first *flatten* the constraint by, for each subexpression e, replacing e with a variable α and adding an extra conjunct $\alpha = e$. In the flattened constraint, let n be the number of variables in the constraint, let u be the largest integer mentioned in any region constant in the constraint, and let k be the largest c used in any $e +_s +_s$ or $e ++_s ++_s$ expression in the constraint. In any solution, an upper bound on the largest integer is $n \times u \times k$. To understand this, notice that either the constraint system is not satisfiable or else the biggest integer we can construct is by a sequence of +k operations, each involving a different variable. Similarly, we have a lower bound on the smallest integer used in any solution. So, for each region variable can guess a set of polynomial size, for each point variable we can guess a point in a set of polynomial size, and for each place variable we can guess a place in the list *places*. We can then check that assignment in polynomial time.

For our notion of set constraints, the entailment problem is co-NP-complete. To demonstrate, first note that $\varphi \models \varphi'$ if and only if $\varphi \wedge \neg\varphi'$ is unsatisfiable. For the fragment of cases where $\varphi' =$ `false` we have that the entailment problem is the question of given φ, is φ unsatisfiable, which is co-NP-complete. So, the full entailment problem is co-NP-hard. Second, note that the entailment problem is in co-NP; we can easily collect the set of all points mentioned in the constraints, then guess an assignment, and finally check that the assignment is not a satisfying assignment, in polynomial time.

The complexity of deciding type equivalence is dominated by the time to check constraint entailment. Given that all other aspects of type checking for our core language are in polynomial time, we conclude that type checking is co-NP-complete. In a later section, our experimental results show that the problem instances for entailment are small for our benchmarks and thus type checking is fast.

$$\Psi; \varphi; \Gamma; \mathit{here} \vdash c : \ \texttt{int} \tag{3.56}$$

$$\Psi; \varphi; \Gamma; \mathit{here} \vdash p : \ \texttt{pt}\ (p, R) \qquad (\text{where } p \in R) \tag{3.57}$$

$$\Psi; \varphi; \Gamma; \mathit{here} \vdash R : \ \texttt{reg}\ R \tag{3.58}$$

$$\Psi; \varphi; \Gamma; \mathit{here} \vdash l : \ \Psi(l) \tag{3.59}$$

$$\Psi; \varphi; \Gamma; \mathit{here} \vdash P : \ \texttt{pl}\ P \tag{3.60}$$

$$\frac{\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash e : \ t_2}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \lambda x : t_1 . e : \ t_1 \rightarrow t_2} \tag{3.61}$$

$$\frac{\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; \mathit{unknown} \vdash e : \ t}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \texttt{lam}\ \alpha : k.e : \ \Pi\alpha : k.t} \tag{3.62}$$

$$\Psi; \varphi; \Gamma; \mathit{here} \vdash x : \ \Gamma(x) \tag{3.63}$$

$$\frac{\Psi; \varphi; \Gamma; \mathit{here} \vdash e_1 : \ t_1 \rightarrow t_2 \qquad \Psi; \varphi; \Gamma; \mathit{here} \vdash e_2 : \ t_1}{\Psi; \varphi; \Gamma; \mathit{here} \vdash e_1 \ e_2 : \ t_2} \tag{3.64}$$

$$\frac{\begin{array}{ll} \Psi; \varphi; \Gamma; \mathit{here} \vdash e_1 : \ \Pi\alpha : k.t_1 & \Psi; \varphi; \Gamma; \mathit{here} \vdash e_2 : t_2 \\ \vdash t_2 : \ k \rhd \ W & \varphi \models (\mathit{constraint}(k))[\alpha := W] \end{array}}{\Psi; \varphi; \Gamma; \mathit{here} \vdash e_1\texttt{<}e_2\texttt{>} \ : \ t_1[\alpha := \ W]} \tag{3.65}$$

$$\frac{\Psi; \varphi; \Gamma[x : t_1]; \mathit{here} \vdash e : \ t_2 \qquad \mathit{here} \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \lambda^{\bullet} x : t_1 . e : \ t_1 \rightarrow t_2} \tag{3.66}$$

$$\frac{\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; \mathit{here} \vdash e : \ t \qquad \mathit{here} \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \mathrm{lam}^{\bullet} \alpha : k.e : \ \Pi\alpha : k.t} \tag{3.67}$$

$$\frac{\Psi; \varphi; \Gamma; \mathit{here} \vdash e : \ \texttt{reg}\ r}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \textbf{new}\ t[e] : \ t[r]} \tag{3.68}$$

$$\frac{\begin{array}{c} \Psi; \varphi; \Gamma; \mathit{here} \vdash y_1 : \ t[r_1] \qquad \Psi; \varphi; \Gamma; \mathit{here} \vdash y_2 : \ \texttt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash \mathit{here} \equiv r_1[@_t(\sigma, r_2)] \end{array}}{\Psi; \varphi; \Gamma; \mathit{here} \vdash y_1[y_2] : \ t} \tag{3.69}$$

Figure 3.14: Type rules (part 1).

A type judgment is of the form $\Psi; \varphi; \Gamma; \mathit{here} \vdash e : t$, which holds if it is derivable using the following rules. The type *here* is the type of the current place

$$\frac{\begin{array}{ll}\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 : t[r_1] & \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 : \texttt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 & \varphi \models \sigma \in_t r_2 \\ \varphi \vdash \mathit{here} \equiv r_1[@_t(\sigma, r_2)] & \Psi;\varphi;\Gamma;\mathit{here} \vdash e : t\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[y_2] = e : t} \tag{3.70}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e : t[r]}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e.\texttt{reg} : \texttt{reg}\ r} \tag{3.71}$$

$$\frac{\begin{array}{ll}\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 : t[r_1] & \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 : \texttt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 & \varphi \models \sigma \in_t r_2\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[@_s y_2] : \texttt{pl}\ r_1[@_t(\sigma, r_2)]} \tag{3.72}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 : \texttt{reg}\ r_1 \quad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 : \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 \cup_s e_2 : \texttt{reg}\ r_1 \cup_t r_2} \tag{3.73}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 : \texttt{reg}\ r_1 \quad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 : \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 \cap_s e_2 : \texttt{reg}\ r_1 \cap_t r_2} \tag{3.74}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e : \texttt{reg}\ r}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e +_s c : \texttt{reg}\ r +_t c} \tag{3.75}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e : \texttt{pt}\ (\sigma, r)}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e +\!\!+_s c : \texttt{pt}\ (\sigma +\!\!+_t c, r +_t c)} \tag{3.76}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 : \texttt{reg}\ r \quad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 : \texttt{pl}\ \pi}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 \% y_2 : \texttt{reg}\ r \%_t \pi} \tag{3.77}$$

$$\frac{\begin{array}{l}\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 : \texttt{reg}\ r \quad \mathit{here} \neq \mathit{unknown} \\ \Psi;\varphi \wedge (\alpha \in_t r);\Gamma[x : \texttt{pt}\ (\alpha, r)];\mathit{here} \vdash e_2 : t\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\} : \texttt{int}} \quad \text{(where } \alpha \text{ is fresh)} \tag{3.78}$$

$$\frac{\Psi;\varphi;\Gamma[x : \texttt{pl}\ \alpha];\mathit{here} \vdash e : t \quad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \textbf{forallplaces}\ x\{e\} : \texttt{int}} \quad \text{(where } \alpha \text{ is fresh)} \tag{3.79}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 : t_1 \quad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 : t_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1; e_2 : t_2} \tag{3.80}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y : \texttt{pl}\ \pi \quad \Psi;\varphi;\Gamma;\pi \vdash e : t}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \textbf{at}(y)\{e\} : t} \tag{3.81}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e : t \quad \varphi \vdash t \equiv t'}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e : t'} \tag{3.82}$$

Figure 3.15: Type rules (part 2).

of execution. Notice that the use of entailment is a condition in rules such as Rule (3.65). Rule (3.78) is a key type rule which says that to type check a loop `for` $(x$ `in` $e_1)\{e_2\}$, we check that e_1 has a type `reg` r, and then assign x the type `pt` (α, r) while checking e_2, where α is fresh. The type rules for array lookup, Rule (3.69), and array update, Rule (3.70), ensure that (1) the point is in bounds by requiring that the type of the point is a region which is a subset of the region of the array, and (2) the place of execution equals the location of the array data by requiring that the type *here* is equivalent to the type of the place of the data.

Figures 3.14 and 3.15 give the rules for extracting constraints. We use W to range over regions r and variables α of kind `place`. In the type rules, the meaning of *constraint* is the following:

$$\mathit{constraint}(\texttt{point}\ \varphi) = \varphi \tag{3.83}$$
$$\mathit{constraint}(\texttt{region}\ \varphi) = \varphi \tag{3.84}$$
$$\mathit{constraint}(\texttt{place}) = \texttt{true} \tag{3.85}$$

The rules for kind checking are:

$$\vdash \texttt{pt}\ (\sigma, r) : \ \texttt{point}\ \varphi \rhd \sigma \tag{3.86}$$
$$\vdash \texttt{reg}\ r : \ \texttt{region}\ \varphi \rhd \ r \tag{3.87}$$
$$\vdash \texttt{pl}\ \pi : \ \texttt{place}\ \rhd \pi. \tag{3.88}$$

3.5 Proof of Type Soundness

Lemma 1 (Substitution) *If* $\Psi;\varphi;\Gamma[x : t_1]; \mathit{here} \vdash e : \ t_2$ *and* $\Psi;\varphi;\Gamma; \mathit{here} \vdash v : \ t_1$*, then* $\Psi;\varphi;\Gamma;\mathit{here} \vdash e[x := v] : t_2$*.*

Proof. By induction on the structure of the derivation of
$\Psi;\varphi;\Gamma[x : t_1];\mathit{here} \vdash e : \ t_2$. □

Lemma 2 (Dependent Substitution) *If* $\Psi;\varphi;\Gamma;\mathit{here} \vdash e : \ t$*, then*

$$\Psi;\varphi[\alpha := W];\Gamma;\mathit{here}[\alpha := \ W] \vdash e[\alpha := \ W] : \ t[\alpha := \ W].$$

Proof. By induction on the structure of the derivation of
$\Psi;\varphi;\Gamma;\mathit{here} \vdash e : \ t$. □

Lemma 3 (Weakening) *If* $\Psi;\varphi;\Gamma;\mathit{here} \vdash e : \ t$ *and* $\varphi' \models \varphi$*, then* $\Psi;\varphi';\Gamma;\mathit{here} \vdash e : \ t$*.*

Proof. By induction on the structure of the derivation of
$\Psi;\varphi;\Gamma;\mathit{here} \vdash e : \ t$. □

Lemma 4 (Indifference) *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v :\ t$, then $\Psi;\varphi;\Gamma;\mathit{here}' \vdash v :\ t$.*

Proof. Immediate from the seven type rules for values. □

Lemma 5 (Canonical Forms)

- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : \texttt{int}$, then v is of the form c.*
- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : \texttt{pt}\ (\sigma, r)$, then v is of the form p.*
- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : \texttt{reg}\ r$, then v is of the form R.*
- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : t[r]$, then v is of the form l, and $l \in \mathcal{D}(\Psi)$.*
- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : \texttt{pl}\ \alpha$, then v is of the form P.*
- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : t_1 \to t_2$, then v is of the form $\lambda x : t.e$.*
- *If $\Psi;\varphi;\Gamma;\mathit{here} \vdash v : \Pi\alpha : k.t$, then v is of the form $\texttt{lam}\ \alpha : k.e$.*

Proof. From an examination of the type rules we have that each form of type is the type of exactly one form of value, namely the one given in the lemma. □

Theorem 3.5.1 (Type Preservation) *For a place P, let $Q \in \{P, \mathit{unknown}\}$. If $\Psi;\varphi;\Gamma;Q \vdash e :\ t$, $\models H : \Psi$, and $P \vdash (H, e) \rightsquigarrow (H', e')$, then we have Ψ', t' such that $\Psi \lhd \Psi'$, $\Psi';\varphi;\Gamma;Q \vdash e' :\ t'$, $\models H' : \Psi'$, and $\varphi \vdash t \equiv t'$.*

Proof. We proceed by induction on the structure of the derivation of $\Psi;\varphi;\Gamma;Q \vdash e :\ t$. There are now twenty-five subcases depending on which one of the type rules was the last one used in the derivation of $\Psi;\varphi;\Gamma;Q \vdash e :\ t$.

In eight of those cases, e is a either a value or a variable x, and hence (H, e) cannot take a step. We will now consider each of the remaining seventeen cases.

- Rule (3.64): the derivation is of the form:

$$\frac{\Psi;\varphi;\Gamma;Q \vdash e_1 :\ t_1 \to t_2 \qquad \Psi;\varphi;\Gamma;Q \vdash e_2 :\ t_1}{\Psi;\varphi;\Gamma;Q \vdash e_1\ e_2 :\ t_2}$$

  We now have three subcases depending on which rule was used to make $(H, e_1\ e_2)$ take a step.

  If Rule (3.1), that is,

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1\ e_2) \rightsquigarrow (H', e_1'\ e_2)}$$

was used to take a step, then we have from the induction hypothesis that we have Ψ' such that $\Psi \lhd \Psi'$, $\Psi';\varphi;\Gamma;Q \vdash e_1' : \ t_1 \rightarrow t_2$, and $\models H' : \Psi'$. From $\Psi \lhd \Psi'$ and $\Psi;\varphi;\Gamma;Q \vdash e_2 : \ t_1$ we have $\Psi';\varphi;\Gamma;Q \vdash e_2 : \ t_1$. From $\Psi';\varphi;\Gamma;Q \vdash e_1' : \ t_1 \rightarrow t_2$ and $\Psi';\varphi;\Gamma;Q \vdash e_2 : \ t_1$, and Rule (3.64), we conclude $\Psi';\varphi;\Gamma;Q \vdash e_1' \ e_2 : \ t_2$.

If Rule (3.2), that is,

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e_2')}{P \vdash (H, v \ e_2) \rightsquigarrow (H', v \ e_2')}$$

was used to take a step, then we have from the induction hypothesis that we have Ψ' such that $\Psi \lhd \Psi'$, $\Psi';\varphi;\Gamma;Q \vdash e_2' : \ t_1$, and $\models H' : \Psi'$. From $\Psi \lhd \Psi'$ and $\Psi;\varphi;\Gamma;Q \vdash e_1 : \ t_1 \rightarrow t_2$ we have $\Psi';\varphi;\Gamma;Q \vdash e_1 : \ t_1 \rightarrow t_2$. From $\Psi';\varphi;\Gamma;Q \vdash e_1 : \ t_1 \rightarrow t_2$ and $\Psi';\varphi;\Gamma;Q \vdash e_2' : \ t_1$, and Rule (3.64), we conclude $\Psi';\varphi;\Gamma;Q \vdash e_1 \ e_2' : \ t_2$.

If Rule (3.3), that is,

$$P \vdash (H, (\lambda x : t.e)v) \rightsquigarrow (H, e[x := v])$$

was used to take a step, then we have from Rule (3.61) that $\Psi;\varphi;\Gamma[x : t_1];Q \vdash e : \ t_2$, so we pick $\Psi' = \Psi$ and we have from Lemma 1 that $\Psi;\varphi;\Gamma;Q \vdash e[x := v] : t_2$.

- Rule (3.65): the derivation is of the form:

$$\frac{\begin{array}{c}\Psi;\varphi;\Gamma;Q \vdash e_1 : \ \Pi\alpha : k.t_1 \qquad \Psi;\varphi;\Gamma;Q \vdash e_2 : t_2 \\ \vdash t_2 : \ k \rhd \ W \qquad \varphi \models (\mathit{constraint}(k))[\alpha := W]\end{array}}{\Psi;\varphi;\Gamma;Q \vdash e_1\texttt{<}e_2\texttt{>} \ : \ t_1[\alpha := W]}$$

We now have three subcases depending on which rule was used to make $(H, e_1\texttt{<}e_2\texttt{>})$ take a step.

If Rule (3.4) or Rule (3.5) was used to take a step, then the proof is similar to that given above for the case of function application (Rule (3.1)); we omit the details.

If Rule (3.6), that is,

$$P \vdash (H, (\texttt{lam} \ \alpha : k.e)\texttt{<}w\texttt{>}) \rightsquigarrow (H, e[\alpha := w])$$

was used to take a step, then we have from Rule (3.62) that

$$\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; \mathit{unknown} \vdash e : \ t.$$

We pick $\Psi' = \Psi$. We pick α such that α does not occur free in φ. Let $\varphi' = \mathit{constraint}(k)$. From $\Psi; \varphi \wedge \varphi'; \Gamma; Q \vdash e : \ t$ and Lemma 2, we have $\Psi; (\varphi \wedge \varphi')[\alpha := W]; \Gamma; Q \vdash e[\alpha := W] : \ t[\alpha := W]$, which is the same as $\Psi; \varphi \wedge (\varphi'[\alpha := W]); \Gamma; Q \vdash e[\alpha := W] : \ t[\alpha := W]$. From $\Psi; \varphi \wedge (\varphi'[\alpha := W]); \Gamma; Q \vdash e[\alpha := W] : \ t[\alpha := W]$, $\varphi \models \varphi'[\alpha := W]$, and Lemma 3, we have $\Psi; \varphi; \Gamma; Q \vdash e[\alpha := W] : \ t[\alpha := W]$.

- Rule (3.66): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma[x : t_1]; Q \vdash e : \ t_2 \quad Q \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; Q \vdash \lambda^{\bullet} x : t_1.e : \ t_1 \to t_2}$$

If Rule (3.7), that is,

$$P \vdash (H, \lambda^{\bullet} x : t_1.e) \rightsquigarrow (H, \lambda x : t_1.\mathbf{at}\texttt{(}P\texttt{)}\{e\})$$

was used to take a step, then from $Q \in \{P, \mathit{unknown}\}$ and $Q \neq \mathit{unknown}$, we have $Q = P$. From Rule (3.60) we have $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash P : \ \texttt{pl}\ P$. From $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash P : \ \texttt{pl}\ P$ and $\Psi; \varphi; \Gamma[x : t_1]; P \vdash e : \ t_2$ and Rule (3.81), we have $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash \mathbf{at}\texttt{(}P\texttt{)}\{e\} : \ t_2$. From $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash \mathbf{at}\texttt{(}P\texttt{)}\{e\} : \ t_2$ and Rule (3.61) we have $\Psi; \varphi; \Gamma; Q \vdash \lambda x : t_1.\mathbf{at}\texttt{(}P\texttt{)}\{e\} : \ t_1 \to t_2$.

- Rule (3.67): the derivation is of the form:

$$\frac{\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; \mathit{here} \vdash e : \ t \quad \mathit{here} \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \mathrm{lam}^{\bullet} \alpha : k.e : \ \mathit{\Pi} \alpha : k.t}$$

If Rule (3.8), that is,

$$P \vdash (H, \mathrm{lam}^{\bullet} \alpha : k.e) \rightsquigarrow (H, \texttt{lam}\ \alpha : k.\mathbf{at}\texttt{(}P\texttt{)}\{e\})$$

was used to take a step, then we can prove that $\Psi; \varphi; \Gamma; \mathit{here} \vdash \texttt{lam}\ \alpha : k.\mathbf{at}\texttt{(}P\texttt{)}\{e\} : \ \mathit{\Pi} \alpha : k.t$ in a manner similar to the previous case of Rule (3.66); we omit the details.

- Rule (3.68): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e : \ \texttt{reg}\ r}{\Psi; \varphi; \Gamma; Q \vdash \mathbf{new}\ t[e] : \ t[r]}$$

We now have two subcases depending on which rule was used to make $(H, \mathbf{new}\ t[e])$ take a step.

If Rule (3.9) was used to take a step, then the proof is similar to that given above for the case of function application (Rule (3.1)); we omit the details. If Rule (3.10), that is,

$$\begin{array}{l} P \vdash (H, \mathtt{new}\ t[R]) \rightsquigarrow \\ \qquad (H[l \mapsto \lambda p \in R.(\mathtt{default}(t), \mathit{distribute}(R,p))], l) \\ \qquad\qquad \text{where } l \text{ is fresh} \end{array}$$

was used to take a step, then we have $e = R$, so from Rule (3.58) we have $r = R$. We define Ψ' to be an extension of $\Psi[l \mapsto t[R]]$ such that $\Psi \lhd \Psi'$ and Ψ' contains suitable definitions for the labels used in $\mathtt{default}(t)$; we omit the details. Let H' be an extension of

$$H[l \mapsto \lambda p \in R.(\mathtt{default}(t), \mathit{distribute}(R,p))]$$

such that H' contains suitable definitions for the labels used in $\mathtt{default}(t)$; we omit the details. From Rule (3.59) we have $\Psi'; \varphi; \Gamma; Q \vdash l : \Psi'(l)$. We finally need to show $\models H' : \Psi'$. From the construction of Ψ' and H' we have that they extend the domains of Ψ and H, respectively, with the same labels. From $\models H : \Psi$ we have $\mathcal{D}(H) = \mathcal{D}(\Psi)$, so we conclude $\mathcal{D}(H') = \mathcal{D}(\Psi')$. Moreover, we have $R = \mathcal{D}(\lambda p \in R.(\mathtt{default}(t), \mathit{distribute}(R,p)))$ and we have $\Psi'; \varphi; \Gamma; Q \vdash \mathtt{default}(t) : t$. Finally, for each $p \in R$ we have $H(l)(p).2 = \mathit{distribute}(R,p)$.

- Rule (3.69): the derivation is of the form:

$$\frac{\begin{array}{c} \Psi; \varphi; \Gamma; Q \vdash y_1 :\ t[r_1] \qquad \Psi; \varphi; \Gamma; Q \vdash y_2 :\ \mathtt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 \qquad \varphi \models \sigma \in_t r_2 \qquad \varphi \vdash Q \equiv r_1[@_t(\sigma, r_2)] \end{array}}{\Psi; \varphi; \Gamma; Q \vdash y_1[y_2] :\ t}$$

  If Rule (3.11), that is,

$$\begin{array}{l} P \vdash (H, l[p]) \rightsquigarrow (H, H(l)(p).1) \\ \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2 \end{array}$$

  was used to take a step, then we have $y_1 = l$ and $y_2 = p$. From $\Psi; \varphi; \Gamma; Q \vdash l :\ t[r_1]$ and Rule (3.59) we have that $r_1 = R$ and $\Psi(l) = t[R]$. We pick $\Psi' = \Psi$ and from $\models H : \Psi$ we have $\Psi; \varphi; \Gamma; Q \vdash H(l)(p).1 : t$.

- Rule (3.70): the derivation is of the form:

$$\frac{\begin{array}{ll}\Psi;\varphi;\Gamma;Q \vdash y_1 : \ t[r_1] & \Psi;\varphi;\Gamma;Q \vdash y_2 : \ \texttt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 & \varphi \models \sigma \in_t r_2 \\ \varphi \vdash Q \equiv r_1[@_t(\sigma, r_2)] & \Psi;\varphi;\Gamma;Q \vdash e : \ t\end{array}}{\Psi;\varphi;\Gamma;Q \vdash y_1[y_2] = e : \ t}$$

  We now have two subcases depending on which rule was used to make $(H, y_1[y_2] = e_3)$ take a step.

  If Rule (3.12) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

  If Rule (3.13), that is,

$$\begin{array}{l}P \vdash (H, l[p] = v) \rightsquigarrow (H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]], v) \\ \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2\end{array}$$

  was used to take a step, then we have $y_1 = l$, $y_2 = p$, $e = v$. From $\Psi;\varphi;\Gamma;Q \vdash l : \ t[r_1]$ and Rule (3.59) we have that $r_1 = R$ and $\Psi(l) = t[R]$. We have $\Psi;\varphi;\Gamma;Q \vdash v : \ t$ so we need to prove $\models H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]] : \Psi$. From $l \in \mathcal{D}(H)$ we have $\mathcal{D}(H[l \mapsto (H(l))[p \mapsto v]]) = \mathcal{D}(H)$. Notice that $H(l)(p).2 = H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]](l)(p).2$. The remaining thing to prove is

$$\Psi;\varphi;\Gamma;Q \vdash (H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]])(l)(p).1 : t.$$

  We have $(H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]])(l)(p).1 = v$ and we have $\Psi;\varphi;\Gamma;Q \vdash v : \ t$.

- Rule (3.71): the derivation is of the form:

$$\frac{\Psi;\varphi;\Gamma;Q \vdash e : \ t[r]}{\Psi;\varphi;\Gamma;Q \vdash e.\texttt{reg} \ : \ \texttt{reg}\ r}$$

  We now have two subcases depending on which rule was used to make $(H, e.\texttt{reg})$ take a step.

  If Rule (3.14) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

  If Rule (3.15), that is,

$$P \vdash (H, l.\texttt{reg}) \rightsquigarrow (H, \mathcal{D}(H(l))) \quad \text{if } l \in \mathcal{D}(H)$$

was used to take a step, then we have from $\Psi;\varphi;\Gamma;Q \vdash l : t[r]$ and Rule (3.59) that $\Psi(l) = t[r]$. Moreover we have that r is of the form R. From $\models H : \Psi$ and $\Psi(l) = t[R]$, we have $\mathcal{D}(H(l)) = R$. We pick $\Psi' = \Psi$ and from Rule (3.58) we conclude $\Psi;\varphi;\Gamma;Q \vdash \mathcal{D}(H(l)) : \texttt{reg}\ R$.

- Rule (3.72): the derivation is of the form:

$$\frac{\begin{array}{ll}\Psi;\varphi;\Gamma;Q \vdash y_1 : t[r_1] & \Psi;\varphi;\Gamma;Q \vdash y_2 : \texttt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 & \varphi \models \sigma \in_t r_2\end{array}}{\Psi;\varphi;\Gamma;Q \vdash y_1[@_s y_2] : \texttt{pl}\ r_1[@_t(\sigma, r_2)]}$$

If Rule (3.16), that is,

$$P \vdash (H, l[@_s p]) \rightsquigarrow (H, H(l)(p).2) \qquad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l))$$

was used to take a step, then we have $y_1 = l$ and $y_2 = p$. From $\Psi;\varphi;\Gamma;Q \vdash l : t[r_1]$ and Rule (3.59) we have that $r_1 = R$ and $\Psi(l) = t[R]$. From $\Psi;\varphi;\Gamma;Q \vdash p : \texttt{pt}\ (\sigma, r_2)$ and Rule (3.57) we have that $\sigma = p$. We have $H' = H$ and we pick $\Psi' = \Psi$. From $\models H : \Psi$ we have $H(l)(p).2 = \mathit{distribute}(R, p)$ and $\mathcal{D}(H(l)) = R$. From Rule (3.60) we have that we must show $H(l)(p).2 \equiv r_1[@_t(\sigma, r_2)]$. We have $r_1[@_t(\sigma, r_2)] = R[@_t(p, r_2)]$. We have $H(l)(p).2 = \mathit{distribute}(R, p)$. We also have $\varphi \models r_2 \subseteq_t R$ and $\varphi \vdash p \in_t r_2$ so from Rule (3.53) we have $\varphi \vdash R[@_t(p, r_2)] \equiv \mathit{distribute}(R, p)$. We conclude $H(l)(p).2 = \mathit{distribute}(R, p) \equiv R[@_t(p, r_2)] = r_1[@_t(\sigma, r_2)]$, as desired.

- Rule (3.73): the derivation is of the form:

$$\frac{\Psi;\varphi;\Gamma;Q \vdash e_1 : \texttt{reg}\ r_1 \quad \Psi;\varphi;\Gamma;Q \vdash e_2 : \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;Q \vdash e_1 \cup_s e_2 : \texttt{reg}\ r_1 \cup_t r_2}$$

We now have three subcases depending on which rule was used to make $(H, e_1 \cup_s e_2)$ take a step.

If Rule (3.17) or Rule (3.18) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

If Rule (3.19), that is,

$$P \vdash (H, R_1 \cup_s R_2) \rightsquigarrow (H, R_1 \cup R_2)$$

was used to take a step, then we have from Rule (3.58) that we must show $\varphi \vdash R_1 \cup_t R_2 \equiv R_1 \cup R_2$, which is Rule (3.43).

- Rule (3.74): the derivation is of the form:

  $$\frac{\Psi;\varphi;\Gamma;Q \vdash e_1 : \ \texttt{reg}\ r_1 \qquad \Psi;\varphi;\Gamma;Q \vdash e_2 : \ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;Q \vdash e_1 \cap_s e_2 : \ \texttt{reg}\ r_1 \cap_t r_2}$$

  We now have three subcases depending on which rule was used to make $(H, e_1 \cap_s e_2)$ take a step.

  If Rule (3.20) or Rule (3.21) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

  If Rule (3.22), that is,

  $$P \vdash (H, R_1 \cap_s R_2) \rightsquigarrow (H, R_1 \cap R_2)$$

  was used to take a step, then we have from Rule (3.58) that we must show $\varphi \vdash R_1 \cap_t R_2 \equiv R_1 \cap R_2$, which is Rule (3.45).

- Rule (3.75): the derivation is of the form:

  $$\frac{\Psi;\varphi;\Gamma;Q \vdash e : \ \texttt{reg}\ r}{\Psi;\varphi;\Gamma;Q \vdash e +_s c : \ \texttt{reg}\ r +_t c}$$

  We now have two subcases depending on which rule was used to make $(H, e +_s c)$ take a step.

  If Rule (3.23) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

  If Rule (3.24), that is,

  $$P \vdash (H, R +_s c) \rightsquigarrow (H, R + c)$$

  was used to take a step, then we have from Rule (3.58) that we must show $\varphi \vdash R +_t c \equiv R + c$, which is Rule (3.47).

- Rule (3.76): the derivation is of the form:

  $$\frac{\Psi;\varphi;\Gamma;Q \vdash e : \ \texttt{pt}\ (\sigma, r)}{\Psi;\varphi;\Gamma;Q \vdash e \mathbin{++_s} c : \ \texttt{pt}\ (\sigma \mathbin{++_t} c, r +_t c)}$$

  We now have two subcases depending on which rule was used to make $(H, e \mathbin{++_s} c)$ take a step.

  If Rule (3.25) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

If Rule (3.26), that is,

$$P \vdash (H, p +\!\!+_s c) \rightsquigarrow (H, p + c)$$

was used to take a step, then we have from Rule (3.57) that we must show $\varphi \vdash p +\!\!+_t c \equiv p + c$, which is Rule (3.51).

- Rule (3.77): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y_1 : \ \texttt{reg}\ r \qquad \Psi; \varphi; \Gamma; Q \vdash y_2 : \ \texttt{pl}\ \pi}{\Psi; \varphi; \Gamma; Q \vdash y_1 \ \%\ y_2 : \ \texttt{reg}\ r\ \%_t\ \pi}$$

  If Rule (3.27), that is,

$$\begin{array}{c} P \vdash (H, R\ \%\ P']) \rightsquigarrow (H, R') \\ \text{where } R' = \{\ p \in R \mid \mathit{distribute}(R, p) = P'\ \} \end{array}$$

  was used to take a step, then we have from Rule (3.58) that we must show $\varphi \vdash r\ \%\ \pi \equiv R'$, which is Rule (3.49).

- Rule (3.78): the derivation is of the form:

$$\frac{\begin{array}{c}\Psi; \varphi; \Gamma; Q \vdash e_1 : \ \texttt{reg}\ r \qquad Q \neq \mathit{unknown} \\ \Psi; \varphi \wedge (\alpha \in_t r); \Gamma[x : \ \texttt{pt}\ (\alpha, r)]; Q \vdash e_2 : \ t\end{array}}{\Psi; \varphi; \Gamma; Q \vdash \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\} : \ \texttt{int}} \quad \text{(where } \alpha \text{ is fresh)}$$

  We now have two subcases depending on which rule was used to make $(H, \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\})$ take a step.

  If Rule (3.28) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

  If Rule (3.29), that is,

$$\begin{array}{l} P \vdash (H, \textbf{for}\ (x\ \textbf{in}\ R)\{e_2\}) \rightsquigarrow \\ (H, ((\text{lam}^\bullet \alpha : \texttt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_1\texttt{>})c_1; \ldots; \\ \quad ((\text{lam}^\bullet \alpha : \texttt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_n\texttt{>})c_n; 0) \\ \quad\ \text{where order}(R) = \langle c_1, \ldots, c_n \rangle \end{array}$$

  was used to take a step, then we have $r = R$. From $\Psi; \varphi; \Gamma[x : \ \texttt{pt}\ (\alpha, R)]; Q \vdash e_2 : \ t$, $Q \neq \mathit{unknown}$ and Rule (3.66) we have $\Psi; \varphi; \Gamma; Q \vdash \lambda^\bullet x : (\alpha, R).e_2 : \texttt{pt}\ (\alpha, R) \rightarrow t$.

From $\Psi;\varphi;\Gamma;Q \vdash \lambda^\bullet x : (\alpha, R).e_2 : \texttt{ pt } (\alpha, R) \to t$ and Rule (3.67) we have

$$\Psi;\varphi;\Gamma;Q \vdash \mathrm{lam}^\bullet\alpha.\lambda^\bullet x : (\alpha, R).e_2 : \Pi\alpha : \texttt{point}(\alpha \in_t R).\texttt{pt } (\alpha, R) \to t. \tag{3.89}$$

From Rule (3.57) and the definition of order(R) we have $\Psi;\varphi;\Gamma;Q \vdash c_i : \texttt{pt } (c_i, R)$. From $\Psi;\varphi;\Gamma;Q \vdash c_i : \texttt{pt } (c_i, R)$ and (3.89) and $\vdash (c_i, R) : \texttt{point}(\alpha \in_t R) \rhd c_i$ and $\varphi \models \mathit{constraint}(\texttt{point})[\alpha := c_i]$ and Rule (3.65) we have

$$\Psi;\varphi;\Gamma;Q \vdash (\mathrm{lam}^\bullet\alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_i\texttt{>} : \texttt{ pt } (c_i, R) \to t[\alpha := c_i]. \tag{3.90}$$

From (3.90) and $\Psi;\varphi;\Gamma;Q \vdash c_i : \texttt{pt } (c_i, R)$ and Rule (3.64) we have $\Psi;\varphi;\Gamma;Q \vdash ((\mathrm{lam}^\bullet\alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_i\texttt{>})c_i : t[\alpha := c_i]$. From Rule (3.80) and Rule (3.56) we conclude

$$\begin{aligned}\Psi;\varphi;\Gamma;Q \vdash {} & ((\mathrm{lam}^\bullet\alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_1\texttt{>})c_1; \ldots; \\ & ((\mathrm{lam}^\bullet\alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_n\texttt{>})c_n; 0 : \texttt{ int}.\end{aligned}$$

- Rule (3.79): the derivation is of the form:

$$\frac{\Psi;\varphi;\Gamma[x : \texttt{pl } \alpha];Q \vdash e : t \quad Q \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;Q \vdash \textbf{forallplaces } x\{e\} : \texttt{ int}} \quad \text{(where } \alpha \text{ is fresh)}$$

  If Rule (3.30), that is,

$$\begin{aligned} & P \vdash (H, \textbf{forallplaces } x\{e\}) \rightsquigarrow \\ & (H, ((\mathrm{lam}^\bullet\alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl } \alpha.e)\texttt{<}P_1\texttt{>})P_1; \ldots; \\ & ((\mathrm{lam}^\bullet\alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl } \alpha.e)\texttt{<}P_n\texttt{>})P_n; 0) \\ & \text{where } \mathit{places} = \langle P_1, \ldots, P_n \rangle \end{aligned}$$

  was used to take a step, then we can prove that $\Psi;\varphi;\Gamma;\mathit{here} \vdash ((\mathrm{lam}^\bullet\alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl } \alpha.e)\texttt{<}P_1\texttt{>})P_1; \ldots; ((\mathrm{lam}^\bullet\alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl } \alpha.e)\texttt{<}P_n\texttt{>})P_n; 0) : \texttt{int}$ in a fashion similar to the case for **for**-loops and Rule (3.78) and Rule(3.29); we omit the details.

- Rule (3.80): the derivation is of the form:

$$\frac{\Psi;\varphi;\Gamma;Q \vdash e_1 : t_1 \quad \Psi;\varphi;\Gamma;Q \vdash e_2 : t_2}{\Psi;\varphi;\Gamma;Q \vdash e_1; e_2 : t_2}$$

  We now have two subcases depending on which rule was used to make $(H, e_1; e_2)$ take a step.

If Rule (3.31) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (3.1); we omit the details.

If Rule (3.32), that is,

$$P \vdash (H, v; e) \leadsto (H, e)$$

was used to take a step, then we pick $\Psi' = \Psi$ and we have $\Psi; \varphi; \Gamma; Q \vdash e : t_2$.

- Rule (3.81): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y : \texttt{pl}\ \pi \quad \Psi; \varphi; \Gamma; \pi \vdash e : t}{\Psi; \varphi; \Gamma; Q \vdash \mathbf{at}(y)\{e\} : t}$$

We now have two subcases depending on which rule was used to make $(H, \mathbf{at}(x)\{e\})$ take a step.

If Rule (3.33), that is,

$$\frac{P' \vdash (H, e) \leadsto (H', e')}{P \vdash (H, \mathbf{at}(P')\{e\}) \leadsto (H, \mathbf{at}(P')\{e'\})}$$

was used to take a step, then we have that $y = P'$. From Rule (3.60) we have that $\pi = P'$. So, we can apply the induction hypothesis to $\Psi; \varphi; \Gamma; \pi \vdash e : t$ and get that $\Psi; \varphi; \Gamma; \pi \vdash e' : t$. From Rule (3.81) we conclude that $\Psi; \varphi; \Gamma; Q \vdash \mathbf{at}(y)\{e'\} : t$.

If Rule (3.34), that is,

$$P \vdash (H, \mathbf{at}(P')\{v\}) \leadsto (H, v)$$

was used to take a step, then we have $H' = H$ and we pick $\Psi' = \Psi$. We also have $e = v$. From $\Psi; \varphi; \Gamma; \pi \vdash v : t$ and Lemma 4, we have $\Psi; \varphi; \Gamma; \mathit{here} \vdash v : t$.

- Rule (3.82): the derivation is of the form

$$\frac{\Psi; \varphi; \Gamma; \mathit{here} \vdash e : t \quad \varphi \vdash t \equiv t'}{\Psi; \varphi; \Gamma; \mathit{here} \vdash e : t'}$$

From the induction hypothesis we have Ψ', t'' such that $\Psi \lhd \Psi'$, $\Psi'; \varphi; \Gamma; Q \vdash e' : t''$, $\models H' : \Psi'$, and $\varphi \vdash t \equiv t''$. From $\varphi \vdash t \equiv t'$ and $\varphi \vdash t \equiv t''$ and Rule (3.37) and Rule (3.36), we have $\varphi \vdash t' \equiv t''$. From Rule (3.82) we conclude that $\Psi; \varphi; \Gamma; \mathit{here} \vdash e' : t'$.

□

Theorem 3.5.2 (Progress) *For a place P, let $Q \in \{P, unknown\}$. If $\Psi; true; \emptyset; Q \vdash e : \ t$ and $\models H : \Psi$, then (H, e) is not stuck at place P.*

Proof. We proceed by induction on the structure of the derivation of

$$\Psi; \text{true}; \emptyset; Q \vdash e : \ t. \tag{3.91}$$

There are now twenty-five subcases depending on which one of the type rules was the last one used in the derivation of $\Psi; \text{true}; \emptyset; Q \vdash e : \ t$.

In seven of those cases, the derivation is of the form: $\Psi; \text{true}; \emptyset; Q \vdash v : \ t$. where v is a value, hence (H, v) is not stuck at place P. The derivation cannot be of the form: $\Psi; \text{true}; \emptyset; Q \vdash x : \ t$ because Rule (3.63) cannot apply. We will now consider each of the remaining seventeen cases.

- Rule (3.64): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \ t_1 \rightarrow t_2 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : \ t_1}{\Psi; \text{true}; \emptyset; Q \vdash e_1 \ e_2 : \ t_2}$$

  From the induction hypothesis, we have that $(H, e_1), (H, e_2)$ are not stuck at place P. If (H, e_1) can take a step at place P, then $(H, e_1 \ e_2)$ can take also a step at place P using Rule (3.1). If e_1 is a value and (H, e_2) can take a step at place P, then also $(H, e_1 \ e_2)$ can take a step at place P using Rule (3.2). If e_1, e_2 are both values, then we have from Lemma 5 that e_1 is of the form $\lambda x : t.e$, so $(H, e_1 \ e_2)$ can take a step at place P using Rule (3.3).

- Rule (3.65): the derivation is of the form:

$$\frac{\begin{array}{ll}\Psi; \text{true}; \emptyset; Q \vdash e_1 : \ \Pi\alpha : k.t_1 & \Psi; \text{true}; \emptyset; Q \vdash e_2 : t_2 \\ \vdash t_2 : \ k \rhd \ W & \text{true} \models (\mathit{constraint}(k))[\alpha := W]\end{array}}{\Psi; \text{true}; \emptyset; Q \vdash e_1\texttt{<}e_2\texttt{>} \ : \ t_1[\alpha := W]}$$

  From the induction hypothesis, we have that (H, e_1) is not stuck at place P. If (H, e_1) can take a step at place P, then $(H, e_1\texttt{<}e_2\texttt{>})$ can also take a step at place P using Rule (3.4). If e_1 is a value and (H, e_2) can take a step at place P, then $(H, e_1\texttt{<}e_2\texttt{>})$ can take also a step at place P using Rule (3.5). If e_1, e_2 are both values, then we have from Lemma 5 that e_1 is of the form $\texttt{lam}\ \alpha : k.e$, and we have from $\vdash t_2 : \ k \rhd \ W$ and Lemma 5 that e_2 is of the form w, so $(H, e_1\texttt{<}e_2\texttt{>})$ can take a step using Rule (3.6).

- Rule (3.66): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset[x : t_1]; Q \vdash e : \ t_2 \quad Q \neq \mathit{unknown}}{\Psi; \text{true}; \emptyset; Q \vdash \lambda^\bullet x : t_1.e : \ t_1 \rightarrow t_2}$$

From Rule (3.7) we have that $\lambda^\bullet x : t_1.e$ can take a step.

- Rule (3.67): the derivation is of the form:

$$\frac{\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; Q \vdash e :\ t \quad Q \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; Q \vdash \mathrm{lam}^\bullet \alpha : k.e :\ \Pi\alpha : k.t}$$

From Rule (3.8) we have that $\mathrm{lam}^\bullet \alpha : k.e$ can take a step.

- Rule (3.68): the derivation is of the form:

$$\frac{\Psi; \mathrm{true}; \emptyset; Q \vdash e :\ \texttt{reg}\ r}{\Psi; \mathrm{true}; \emptyset; Q \vdash \textbf{new}\ t[e] :\ t[r]}$$

From the induction hypothesis we have that (H, e) is not stuck at place P. If (H, e) can take a step at place P, then $(H, \textbf{new}\ t[e])$ can also take a step at place P using Rule (3.9). If e is a value, then we have from Lemma 5 that e is of the form R, so $(H, \textbf{new}\ t[e])$ can take a step using Rule (3.10).

- Rule (3.69): the derivation is of the form:

$$\frac{\begin{array}{c}\Psi; \mathrm{true}; \emptyset; Q \vdash y_1 :\ t[r_1] \quad \Psi; \mathrm{true}; \emptyset; Q \vdash y_2 :\ \texttt{pt}\ (\sigma, r_2) \\ \mathrm{true} \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash Q \equiv r_1[@_t(\sigma, r_2)]\end{array}}{\Psi; \mathrm{true}; \emptyset; Q \vdash y_1[y_2] :\ t}$$

We have that y_1, y_2 must be values and we have from Lemma 5 that y_1 is of the form l, $l \in \mathcal{D}(\Psi)$ and y_2 is of the form p. Further we have that $Q = P$ since *unknown* is not equivalent to anything other than itself. Let $t[R]$ denote $\Psi(l)$. From $l \in \mathcal{D}(\Psi)$ and $\models H : \Psi$, we have that $l \in \mathcal{D}(H)$ and $R = \mathcal{D}(H(l))$. We have $r_1 = R$. From the type rule for point constants, we have that r_2 is of the form R' and that $p \in R'$. We have true $\models R' \subseteq_t R$. From *true* $\models R' \subseteq_t R$, we have $R' \subseteq R$, hence $p \in R' \subseteq R$. From $H \models \Psi$ we have $\mathit{distribute}(R, p) = H(l)(p).2$. From Rule (3.57) we have $\sigma = p$. We conclude $P = r_1[@_t(\sigma, r_2)] = \mathit{distribute}(R, p) = H(l)(p).2$. So, $(H, e_1[e_2])$ can take a step using Rule (3.11).

- Rule (3.70): the derivation is of the form:

$$\frac{\begin{array}{ll}\Psi; \mathrm{true}; \emptyset; Q \vdash y_1 :\ t[r_1] & \Psi; \mathrm{true}; \emptyset; Q \vdash y_2 :\ \texttt{pt}\ (\sigma, r_2) \\ \mathrm{true} \models r_2 \subseteq_t r_1 & \mathrm{true} \models \sigma \in_t r_2 \\ \mathrm{true} \vdash Q \equiv r_1[@_t(\sigma, r_2)] & \Psi; \mathrm{true}; \emptyset; Q \vdash e :\ t\end{array}}{\Psi; \mathrm{true}; \emptyset; Q \vdash y_1[y_2] = e :\ t}$$

We have that y_1, y_2 must be values and we have from Lemma 5 that y_1 is of the form l, $l \in \mathcal{D}(\Psi)$ and y_2 is of the form p. Further we have that $Q = P$, since *unknown* is not equivalent to anything other than itself. From the induction hypothesis we have that (H, e) is not stuck at place P. If (H, e) can take a step at place P, then $(H, y_1[y_2] = e)$ can also take a step at place P using Rule (3.12). Suppose now that e is a value. The proof that $y_1[y_2] = e$ can take a step at place P using Rule (3.13) is similar to that given above for the case of array lookup (Rule (3.69)), because Rule (3.11) has the same side condition; we omit the details.

- Rule (3.71): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : \ t[r]}{\Psi; \text{true}; \emptyset; Q \vdash e.\texttt{reg} : \ \texttt{reg}\ r}$$

From the induction hypothesis we have that (H, e) is not stuck. If (H, e) can take a step at place P, then $(H, e.\texttt{reg})$ can also take a step using Rule (3.14). If e is a value, then $(H, e.\texttt{reg})$ can take a step using Rule (3.15).

- Rule (3.72): the derivation is of the form:

$$\frac{\begin{array}{ll}\Psi; \text{true}; \emptyset; Q \vdash y_1 : \ t[r_1] & \Psi; \text{true}; \emptyset; Q \vdash y_2 : \ \texttt{pt}\ (\sigma, r_2) \\ \text{true} \models r_2 \subseteq_t r_1 & \text{true} \models \sigma \in_t r_2\end{array}}{\Psi; \text{true}; \emptyset; Q \vdash y_1[@_s y_2] \ : \ \texttt{pl}\ r_1[@_t(\sigma, r_2)]}$$

We have that y_1, y_2 must be values and we have from Lemma 5 that y_1 is of the form l, $l \in \mathcal{D}(\Psi)$ and y_2 is of the form p. The proof that $y_1[@_s y_2]$ can take a step at place P using Rule (3.16) is similar to that given above for the case of array lookup and update (Rules (3.69) and (3.70)), because Rules (3.11)and (3.13) have merely a stronger side condition; we omit the details.

- Rule (3.73): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \ \texttt{reg}\ r_1 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : \ \texttt{reg}\ r_2}{\Psi; \text{true}; \emptyset; Q \vdash e_1 \cup_s e_2 : \ \texttt{reg}\ r_1 \cup_t r_2}$$

From the induction hypothesis we have that $(H, e_1), (H, e_2)$ are not stuck at place P. If (H, e_1) can take a step at place P, then $(H, e_1 \cup_s e_2)$ can also take a step at place P using Rule (3.17). If e_1 is a value and (H, e_2) can take a step at place P, then $(H, e_1 \cup_s e_2)$ can also take a step at place P using Rule (3.18). If e_1, e_2 are both values, then we have from Lemma 5 that e_1 is of the form R_1 and that e_2 is of the form R_2, so $(H, e_1 \cup_s e_2)$ can take a step at place P using Rule (3.19).

- Rule (3.74): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \texttt{reg}\, r_1 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : \texttt{reg}\, r_2}{\Psi; \text{true}; \emptyset; Q \vdash e_1 \cap_s e_2 : \texttt{reg}\, r_1 \cap_t r_2}$$

  From the induction hypothesis we have that $(H, e_1), (H, e_2)$ are not stuck at place P. If (H, e_1) can take a step at place P, then $(H, e_1 \cap_s e_2)$ can also take a step at place P using Rule (3.20). If e_1 is a value and (H, e_2) can take a step at place P, then $(H, e_1 \cap_s e_2)$ can also take a step at place P using Rule (3.21). If e_1, e_2 are both values, then we have from Lemma 5 that e_1 is of the form R_1 and that e_2 is of the form R_2, so $(H, e_1 \cap_s e_2)$ can take a step at place P using Rule (3.22).

- Rule (3.75): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : \texttt{reg}\, r}{\Psi; \text{true}; \emptyset; Q \vdash e +_s c : \texttt{reg}\, r +_t c}$$

  From the induction hypothesis we have that (H, e) is not stuck at place P. If (H, e) can take a step at place P, then $(H, e +_s c)$ can also take a step at place P using Rule (3.23). If e is a value, then we have from Lemma 5 that e is of the form R, so $(H, e +_s c)$ can take a step at place P using Rule (3.24).

- Rule (3.76): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : \texttt{pt}\ (\sigma, r)}{\Psi; \text{true}; \emptyset; Q \vdash e +\!\!+_s c : \texttt{pt}\ (\sigma +\!\!+_t c, r +_t c)}$$

  From the induction hypothesis we have that (H, e) is not stuck at place P. If (H, e) can take a step at place P, then $(H, e +\!\!+_s c)$ can also take a step at place P using Rule (3.25). If e is a value, then we have from Lemma 5 that e is of the form p, so $(H, e +\!\!+_s c)$ can take a step at place P using Rule (3.26).

- Rule (3.77): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash y_1 : \texttt{reg}\, r \quad \Psi; \text{true}; \emptyset; Q \vdash y_2 : \texttt{pl}\ \pi}{\Psi; \text{true}; \emptyset; Q \vdash y_1 \,\%\, y_2 : \texttt{reg}\, r \,\%_t\, \pi}$$

  We have that y_1, y_2 must be values and we have from Lemma 5 that y_1 is of the form R and y_2 is of the form P'. So, $(H, y_1 \,\%\, y_2)$ can take a step at place P using Rule (3.27).

- Rule (3.78): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \ \texttt{reg}\ r \quad Q \neq \textit{unknown} \qquad \Psi; (\alpha \in_t r); \emptyset[x : \ \texttt{pt}\ (\alpha, r)]; Q \vdash e_2 : \ t}{\Psi; \text{true}; \emptyset; Q \vdash \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\} : \ \texttt{int}} \quad \text{(where } \alpha \text{ is fresh)}$$

  From the induction hypothesis we have that (H, e_1) is not stuck at place P. If (H, e_1) can take a step at place P, then $(H, \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\})$ can also take a step at place P using Rule (3.28). If e_1 is a value, then we have from Lemma 5 that e_1 is of the form R, so $(H, \textbf{for}\ (x\ \textbf{in}\ e_1)\{e_2\})$ can take a step at place P using Rule (3.29).

- Rule (3.79): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma[x : \texttt{pl}\ \alpha]; Q \vdash e : \ t \quad Q \neq \textit{unknown}}{\Psi; \varphi; \Gamma; Q \vdash \textbf{forallplaces}\ x\{e\} : \ \texttt{int}} \quad \text{(where } \alpha \text{ is fresh)}$$

  We have that **forallplaces** $x\{e\}$ can take a step using Rule (3.30).

- Rule (3.80): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \ t_1 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : \ t_2}{\Psi; \text{true}; \emptyset; Q \vdash e_1; e_2 : \ t_2}$$

  From the induction hypothesis we have that (H, e_1) is not stuck at place P. If (H, e_1) can take a step at place P, then $(H, e_1; e_2)$ can also take a step at place P using Rule (3.31). If e_1 is a value, then $(H, e_1; e_2)$ can take a step at place P using Rule (3.32).

- Rule (3.81): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash y : \ \texttt{pl}\ \pi \quad \Psi; \text{true}; \emptyset; \pi \vdash e : \ t}{\Psi; \text{true}; \emptyset; Q \vdash \textbf{at}\texttt{(}y\texttt{)}\{e\} : \ t}$$

  We have that y must be a value and we have from Lemma 5 that y must be of the form P'. From the induction hypothesis we have that (H, e) is not stuck at place P'. If (H, e) can take a step at place P', then $(H, \textbf{at}\texttt{(}y\texttt{)}\{e\})$ can take a step at place P using Rule (3.33). If e is a value, then $(H, \textbf{at}\texttt{(}y\texttt{)}\{e\})$ can take a step at place P using Rule (3.34).

- Rule (3.82): the derivation is of the form

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : \ t \quad \text{true} \vdash t \equiv t'}{\Psi; \text{true}; \emptyset; Q \vdash e : \ t'}$$

From the induction hypothesis we have that e is not stuck at place P.

□

Corollary 3.5.3 (Type Soundness) *For a place P, let $Q \in \{P, unknown\}$. If $\Psi; true; \emptyset; Q \vdash e : \ t$ and $\models H : \Psi$, then (H, e) cannot go wrong at place P.*

Proof. Suppose (H, e) can go wrong at place P, that is, we have H', e' such that $P \vdash (H, e) \leadsto^n (H', e')$ and (H', e') is stuck at place P. From Theorem 3.5.1, Rule (3.37), and induction on n, we have Ψ', t' such that $\Psi'; \text{true}; \emptyset; Q \vdash e' : \ t'$, $\models H' : \Psi'$, and true $\models t \equiv t'$. From Theorem 3.5.2 we have that (H', e') is not stuck at place P, a contradiction. □

3.6 Extended Region Algebra

The discussions in Section 3.4 focused on a simple core algebra of region operations based on the fundamental set operations union ($\cup$), intersection ($\cap$) and shifting. However, additional set operators are required in practice for many benchmarks (see Section 3.8). This section discusses various extensions of the region algebra. The type-theoretic proofs given in previous sections continue to work for any region algebra. Specifically, subtyping will be decidable if constraint entailment for the respective region algebra is decidable.

This section gives a formal definition of the various region operators that are used in the implementation. Many of the region operators were inspired by ZPL; however, the notation and aspects such as distributions and places are different. Some ZPL operators, such as bitmasks, are not listed – our implementation does not support them and the existing benchmarks do not require them.

3.6.1 Points

Definition 1 (Point and Rank) *A point p is an $n+1$-tuple of integers $p = i_0, \ldots, i_n$. We say that $n+1$ is the* rank *of p. Points can never have rank zero.*

Definition 2 (Point Order) *Suppose $p_1 = i_0, \ldots, i_n$ and $p_2 = j_0, \ldots, j_n$. Then $\leq$ is defined lexicographically:*

$$p_1 \leq p_2 := i_0 \leq j_0 \lor ((i_0 = j_0) \land (i_1 \leq j_1 \lor \ldots \land i_n = j_n)) \tag{3.92}$$

Definition 3 (Point Addition and Subtraction) *Suppose $p_1 = i_0, \ldots, i_n$ and $p_2 = j_0, \ldots, j_n$. Point addition and subtraction are defined pairwise (just like vector addition and subtraction):*

$$p_1 \pm p_2 := i_1 \pm j_1, \ldots, i_n \pm j_n. \tag{3.93}$$

3.6.2 Regions

Definition 4 (Region) *A region is a finite set of points $r = \{p_1, \ldots, p_n\}$ where all points p_i must have the same rank. The rank of a region is the rank of all of its points. We distinguish between empty regions by rank. In other words, the empty region of rank 1 is different from the empty region of rank 2. We use $r_1 \subseteq r_2$ to indicate that r_1 is a subset of r_2.*[2]

Definition 5 (Region Construction) *The region constructor $[\_, \_] : point \times point \rightarrow region$ such that for all points p_1, p_2 where the ranks of p_1 and p_2 must be identical:*

$$[p_1 : p_2] := \{p : p_1 \leq p \wedge p \leq p_2\} \tag{3.94}$$

Definition 6 (Region Union, Intersection and Difference) *Union, intersection and difference on regions are defined to be the usual union and intersection on sets. These operations are only defined if the ranks of the regions are identical. Since we cannot use $\cup$, $\cap$ and $\setminus$ in the X10 source language we use different symbols to represent these canonical operations:*

$$r_1 | r_2 : = \{p : p \in r_1 \vee p \in r_2\} \tag{3.95}$$

$$r_1 \& r_2 : = \{p : p \in r_1 \wedge p \in r_2\} \tag{3.96}$$

$$r_1 - r_2 : = \{p : p \in r_1 \wedge p \notin r_2\} \tag{3.97}$$

Definition 7 (Region Move) *The region move operation $\_+\_ : region \times point \rightarrow region$ is defined by*

$$r + p_1 := \{p : p - p_1 \in r\} \tag{3.98}$$

where the rank of p_1 and r must be identical.

3.6.3 Places

A place represents an execution unit in the hardware. An X10 virtual machine (XVM) provides a fixed, finite set of places $\mathcal{E}$; however, this number is not known

[2]In X10 source, $\subseteq$ is expressed using `<=`. The canonical meaning by extension is given to the `<`, `>`, `>=`, `==` and `!=` operations.

for type checking. The type equality rules identify type expressions that are guaranteed to represent the same place. Note that there can be more place expressions than the actual number of places in the XVM. However, given that the source code is finite, there can still only be a finite number of place expressions.

3.6.4 Distributions

A distribution d is a mapping from points of identical rank to places, $d : p \to e$. The rank of a distribution is defined to be the rank of the underlying regions. Distributions are used to represent distributed arrays.

Definition 8 (Distribution Construction) *The distribution constructor* $\_*\_ :$ *region* $\times$ *place* $\to$ *distribution is an operation such that:*

$$\begin{aligned} r * e := d :& r \to \mathcal{E} \\ & p \mapsto e. \end{aligned}$$

Definition 9 (Distribution Restriction) *The restriction operation on distributions* $\_\%\_ :$ *distribution* $\times$ *place* $\to$ *region is defined as*

$$d\%e := \{p : d(p) \equiv e\}.$$

Definition 10 (Distribution Union and Intersection) *For* $d_1 : r_1 \to \mathcal{E}$ *and* $d_2 : r_2 \to \mathcal{E}$ *where the ranks of* d_1 *and* d_2 *must be identical and*

$$\bigwedge_{\substack{e_1 \in \mathcal{E} \\ e_2 \neq e_1}} d_1\%e_1 \qquad \& \qquad d_2\%e_2 \subseteq \emptyset, \tag{3.99}$$

define

$$d_1|d_2 := d : r_1|r_2 \to \mathcal{E} \tag{3.100}$$

$$p \mapsto \begin{cases} d_1(p) & \text{, if } p \in r_1 \\ d_2(p) & \text{, if } p \in r_2 - r_1 \end{cases} \tag{3.101}$$

$$d_1\&d_2 := d : r_1\&r_2 \to \mathcal{E} \tag{3.102}$$

$$p \mapsto d_1(p). \tag{3.103}$$

Lemma 6 (Commutativity)

$$d_1|d_2 = d_2|d_1 \tag{3.104}$$

$$d_1\&d_2 = d_2\&d_1 \tag{3.105}$$

$$\tag{3.106}$$

Proof. Implied by (3.99). □

Note that in (3.99) it would be sufficient to require $e_2 \not\equiv e_1$, however this would make it impossible to statically decide that the given union or intersection operation is sound.

Definition 11 (Subset Relation for Distributions) *Let $d_1 : r_1 \to \mathcal{E}$ and $d_2 : r_2 \to \mathcal{E}$. We say that $d_1 \subseteq d_2$ if $r_1 \subseteq r_2$ and equation* (3.99) *holds.*

Definition 12 (Distribution Access) *In X10, the source-level array element access operation $\_[\_] : distribution \times point \to place$ corresponds to evaluating the distribution function; that is*

$$d[p] := d(p). \tag{3.107}$$

Definition 13 (Region of Distribution) *Let $d : r \to \mathcal{E}$. The region-of operation on distributions $\_.reg : distribution \to region$ is then defined as*

$$d.reg := r.$$

3.6.5 Additional Point Operations

Definition 14 (Point Multiplication) *Suppose $p = i_0, \ldots, i_n$ and $c \in \mathbb{Z}$. Then*

$$p * c := i_1 * c, \ldots, i_n * c. \tag{3.108}$$

Definition 15 (Point Projection) *Suppose $p = i_0, \ldots, i_n$. Then for $k \in [0 : n]$*

$$p[k] := i_k. \tag{3.109}$$

For $k_1, \ldots, k_m$ with $k_l \in [0 : n]$, we define

$$p[k_1, \ldots, k_m] := i_{k_1}, \ldots, i_{k_m}. \tag{3.110}$$

Note that while $m \geq 1$ is required, $m > n$ is allowed.

3.6.6 Additional Region Operations

Definition 16 (Region Multiplication) $\_ * \_ : region \times region \to region$ *is defined as*

$$r_1 * r_2 := \{i_1, \ldots, i_n, i_{n+1}, \ldots, i_{n+m} : i_1, \ldots, i_n \in r_1 \wedge i_{n+1}, \ldots, i_{n+m} \in r_2\} \tag{3.111}$$

Definition 17 (Region Projection) *$\_[\_,\ldots,\_] : region \times \mathbb{N}_0^m \rightarrow region$ is defined as*

$$r[k_1,\ldots,k_m] := \{p[k_1,\ldots,k_m] : p \in r\}. \tag{3.112}$$

3.6.7 Additional Distribution Operations

Definition 18 (Distribution Move) *The distribution move operation $\_ + \_ : distribution \times point \rightarrow distribution$ is defined by*

$$\begin{aligned} d_0 + p_0 := d : d.reg + p_0 &\rightarrow \mathcal{E} \\ p &\mapsto d_0(p - p_0). \end{aligned}$$

where the ranks of d_0 and p_0 must be identical.

Definition 19 (Distribution Multiplication) *$\_ * \_ : region \times distribution \rightarrow distribution$ is defined as*

$$\begin{aligned} r * d_0 := d : (r * d_0.reg) &\rightarrow \mathcal{E} \\ (p_r, p_{d_0}) &\mapsto d_0(p_{d_0}) \ \texttt{where}\ p_r \in r \wedge p_{d_0} \in d_0.reg \end{aligned}$$

$\_ * \_ : distribution \times region \rightarrow distribution$ is defined as

$$\begin{aligned} d_0 * r := d : (d_0.reg * r) &\rightarrow \mathcal{E} \\ (p_{d_0}, p_r) &\mapsto d_0(p_{d_0}) \ \texttt{where}\ p_{d_0} \in d_0.reg \wedge p_r \in r \end{aligned}$$

In other words, the multiplication of a region and a distribution results in a distribution over the points in the products of the respective regions. The resulting distribution maps a point in the product region to the same place to which part of the point was mapped to by the original distribution used in the product.

Distribution multiplication as defined here is associative, but not commutative. Distributions can only be multiplied with regions, not with other distributions.

Definition 20 (Built-in Distributions) *X10 provides various built-in distribution constructors. As far as the type system is concerned, the semantics of these constructors are all identical. For example, the semantics of the* `block` *constructor is given by:*

$$\texttt{block}(r) := d : r \rightarrow \mathcal{E}. \tag{3.113}$$

3.7 Implemented Design Features

We have implemented the presented type system in XTC-X10, a prototype implementation of an X10 variant that is publicly available on my webpage[3]. The implementation is able to type check all of the example programs from Section 3.3. Our prototype has some syntactic differences in comparison with IBM's reference implementation of X10, mainly because we add additional features such as region types, operator overloading, first-order functions and parametric types (generics). This section discusses various implementation and language design choices that relate to the type system. Additional details about the implementation can be found in Section A.3 in the Appendix.

3.7.1 Rank

The core language presented in Section 3.4 does not consider array dimensions. However, it is in general important for the type system to capture the rank of a point, region, distribution or array. The region algebra defined in Section 3.6 specifically requires all points in a region to have the same rank. Capturing the rank as part of the dependent type information is important for various reasons, as detailed below.

Knowing the rank enables the type system to statically verify the safety of the projection operations and accesses to the various point dimensions as integers using X10's exploded syntax (see [CDE05]). The code generator can exploit knowledge about the rank when reserving memory for points and generating `for` loops.

Rank expressions are limited to constant (positive) integers. The various region operations described in Section 3.6 require only addition and multiplication with a constant. This ensures that deciding rank and rank equality is simple and decidable (the relevant constraint expressions stay within Pressburger Arithmetic). Note that adding the rank of a region to the facts tracked by the region type system does not prevent code from being parametric over arrays of different dimensionality. Figure 3.16 shows an example of code that safely operates on arrays of ranks n and $2 \cdot n$. A simple change of the type of the second parameter to `Array<int#1>` could be used to specialize the code for the case that $n = 1$.

[3]http://grothoff.org/christian/xtc/x10/

```
    void compact(Array<int;small.reg*small.reg> big, Array<int> small) {
        for (p : small.reg)
            for (q : small.reg)
                small[p] += big[p * q];
    }
```

Figure 3.16: The `big` array here has rank $2n$ where n is the rank of `small`. The $p{*}q$ operation produces an $n{+}n$ dimensional point by concatenation of the points p and q.

```
class Array<T> {

    nonexcepting T get(point<:this.dist % here> p);

    @warn(slow)
    nonexcepting T get(point<:this.dist.reg> p);

    @warn(very slow)
    T get(point p);

}
```

Figure 3.17: Use of overloading with region types.

3.7.2 Overloading

Dependently typed function arguments result in additional possibilities for overloading of methods. A particularly nice way of using overloading with dependent types is shown in Figure 3.17.

The class defines three `get` methods to access elements of the `Array`. The first method requires the type system to guarantee that the index is in bounds and that the access will be local. This is the array access as described in the core language. The second method requires the index to be in bounds but allows the element to be non-local. Internally, the implementation will do a remote access; consequently, the method is annotated with metadata warning the programmer that calling this method will result in "slow" code. A development environment might use such annotations to warn the programmer at the call site, for example using syntax highlighting. Finally, the third variant is unsafe – it does not even require the index to be in-bounds. Assuming that this variant internally checks for the index to be in-bounds and otherwise throws an exception, it is annotated to be "very slow" and to possibly throw an exception (since it lacks the `nonexcepting` modifier).

Using this style of overloading enables programmers to draft code without worrying about locality or memory safety. However, it will be obvious where memory safety may be violated and where performance is lost due to lack of locality. If performance becomes important, the code can then later be improved by changing types and constructions appropriately. This way, overloading can help focus programmer activity on critical sections of the code and thus improve programmer productivity.

Note that methods that are used to determine the dependent type of an object must not rely on dependent overloading – this could result in undecidable situations. For the compiler implementation, this means that an additional compilation stage is required. After first computing the class hierarchy, the compiler has to compute the signatures of members but disregard dependent type information at this stage. Then, in a separate stage specific to compilers supporting dependent typing, members with dependent type information are computed, – possibly using the members without dependent type information. Use of members that have the same signature without dependent types will not be possible at this stage because the use would be obviously ambiguous. Once the dependently typed member signatures have been computed, the last stage, code generation, can proceed as usual.

3.7.3 Defaults

The XTC-X10 implementation differs from IBM's X10 v0.4 in a few respects centering around what we believe to be an important language design principle, the *API change rule*. The rule can be phrased as follows:

> "Unless the programmer specifically says otherwise, any API should be as restrictive to clients as possible."

The reasoning here is that as code evolves, it is always easy to make an API more permissive; on the other hand, it is generally difficult to change an API to be more restrictive since this breaks backwards compatibility with existing clients. We do not want to ask programmers to think about possible useful restrictions or limitations of their API – in general, programmers start with design requirements.

Java takes a different approach which focuses on flexibility and extensibility – the default for classes, methods and fields is non-private and non-final. We believe that code should not be extensible by design but become extensible by being extended. Consequently, lifting API limitations can be deferred until the point where extensions require it.

Examples of the use of the API change rule in our implementation of X10 include method modifiers such as `nonblocking` and `nonexcepting`. By default, the methods make no guarantees – they may block or throw exceptions. The default visibility for fields and methods is `private`. Fields and local variables are `final` by default. The modifier `var` is required to make them updateable. As a result, the compiler can immediately construct dependent type information with dependencies on most local variables (since they are guaranteed not to change). In summary, the API change rule improves programmer productivity by making it easier to evolve code without breaking backwards compatibility.

3.7.4 Type declarations

Writing region type expressions for all local variables is not only tedious but also error-prone. Worse, in a design that uses overloading, it may often be important to give a precise type for the local variable in order to achieve the desired effect. Figure 3.18 presents a simple client that uses `region` (with a standard `java.util`-style `Iterator`) and the `Array` class from Figure 3.17. Based on the type declaration given for `it`, the `Iterator`'s `next` method will return elements of type `point`. Consequently, for this code, the compiler would invoke the "very slow" variant of `get`.

```
int m() {
    Array<int> a = new Array<int>([0:4], 1);
    Iterator<point> it = a.dist.reg.iterator();
    var int sum = 0;
    while (it.hasNext())
        sum += a.get(it.next());
    return sum;
}
```

Figure 3.18: Local variable type declarations causing trouble with overloading.

The programmer can address this problem by giving more precise types for the `Array` and `Iterator`. Figure 3.19 gives the most precise types that are legal for our implementation. Obviously, writing these huge type annotations for every variable is cumbersome and likely to cause errors. While the type checker will detect the use of an illegal type, it would accept any type that is a parent type of the right hand side – like those types in Figure 3.18. This loss of type information due to imprecise local variable declarations, in particular in combination with overloading, can be confusing to the programmer. Specifically, the programmer might be surprised to see method resolution fail or result in unexpected resolutions that go contrary to his reasoning in terms of the semantic operations.

```
int m() {
    Array<int;([0:4]*here):([0:4]*here)#1> a = new Array<int>([0:4], 1);
    Iterator<point<:[0:4]#1>> it = a.dist.reg.iterator();
    var int sum = 0;
    while (it.hasNext())
        sum += a.get(it.next());
    return sum;
}
```

Figure 3.19: Local variable type declarations can be tedious.

A simple solution to this dilemma is to allow the programmer to simply not declare types for final local variables (this is another important reason why local variables should be final by default). In this case, the compiler can simply infer the most specific type, which is usually what the programmer wants anyway. In our implementation, programmers can use a "." instead of a type for local variables and have the compiler automatically "fill in the dot(s)". The code given in Figure 3.20 is equivalent to that of Figure 3.19 but more pleasant to read and write.

```
int m() {
    . a = new Array<int>([0:4], 1);
    . it = a.dist.reg.iterator();
    var int sum = 0;
    while (it.hasNext())
        sum += a.get(it.next());
    return sum;
}
```

Figure 3.20: Local variable type declarations – not always necessary.

3.7.5 Decision Procedure for Subtyping

This section describes the implementation of the decision procedure for deciding subtyping relationships between region types in XTC-X10. From the point of view of the compiler, the decision procedure takes two region expressions and determines if they are in a subset relationship (or, for places, in an equality relationship). The details of the expression algebra are described in Section A.3.7. If the implementation can show that the subset (or equality) relationship holds for the two expressions, it returns `true`, otherwise `false`.

The heuristic that provides the decision procedure for region types in XTC-X10 consists of three main components. First, the decision procedure relies on a simplification pass which produces a simplified version of the region expressions. For example, an integer expression $1 + 2$ would be simplified to 3, or a region expression $r_1 \cup r_1$ would be simplified to r_1. The implementation applies this simplification pass incrementally and immediately during the construction of the region expressions.

The heuristic also contains a function to determine whether two expressions can be shown to be equal. This pass is important for operations that involve integer arithmetic, such as shifting of regions by a point, as well as place operations. Equality testing is most expensive for commutative expressions, such as union and intersection of regions and addition of integers and points.

The main function of the decision procedure is used to determine subset relationships for regions and distributions as well as less-than relationships for points and integers. The decision procedure contains a list of rules corresponding to the types of the terms that are being compared. A rule is specified by giving the types of the top-level expressions that are being compared, together with code that tries to establish that the desired relationship holds. This is generally achieved by decomposing larger terms and recursively applying the decision procedure to the smaller terms. If a particular rule fails to establish the subset relationship, other rules that apply to the particular types of expressions are considered. Only if none of the applicable rules can establish the relationship does the heuristic return `false`.

The rules are implemented as simple methods that take two arguments of appropriate type. The decision procedure selects applicable methods based on the types of the arguments and invokes these methods using reflection.

The decision procedure includes rules for integer arithmetic (Pressburger), point arithmetic and the region operations from Section 3.6. The implementation performs both symbolic reasoning (e.g. $r_1 \cap r_2 \subseteq r_1$), arithmetic reasoning (e.g. $[0 : 5] \subseteq [0 : 7]$) and combinations of the two (e.g. $r_1 \cap [0 * 5] * p\%q \subseteq [0 : 6] \cap r_1$).

3.8 Experiments

We have converted some of the benchmarks from the IBM reference implementation of X10 to work with XTC-X10. Since those benchmarks were originally written for languages without regions, most of the work has been put into making the code use regions. However, simply using regions is not enough in general – regions must be used in ways that enable the type system to show safety of array accesses. This sometimes requires a measure of creativity, as illustrated by the two versions of the crypt benchmark (in Appendix B).

The current XTC-X10 prototype uses an interpreter written in Java to execute the generated code. As a result, giving meaningful performance numbers based on XTC-X10 is not possible at this time. However, the performance impact of eliminating bounds checks can be estimated by crudely disabling all checks in an existing implementation. Disabling dynamic checks in IBM's X10 reference implementation improves performance by a factor of 3 for some benchmarks. Similarly, disabling bounds checks in IBM's Java Virtual Machine can improve the performance of benchmarks with intensive array access patterns by a factor of up to 1.75.[4]

In addition to raw performance, another important metric is programmer productivity. Using region expressions can help the programmer express complex constructions concisely. The DSOR benchmark is a good example of this. Static checking also benefits programmer productivity – both array bounds violations and locality violations would otherwise have to be resolved discovering the error and runtime and researching the code to locate bugs; with the type system, the type checker will point out such bugs at compile time.

3.8.1 The ArrayBench Benchmark Suite

The ArrayBench Benchmark Suite consists of seven benchmark programs. This section briefly explains the functionality of each benchmark, the style of parallelism (if any) and the overall amount of communication.

The X10 language model features two levels of parallelism: parallel execution on different places and parallel execution at the same place. Consequently, for each benchmark program we will give three figures: *PP*, *SP* and *SW*. The figure *PP* is the amount of place-parallelism (for a maximum of P places available) and describes how many places compute in parallel. A value of 1 indicates that a computation is not distributed, a value of P is used for a computation that uses all available places in parallel. The figure *SP* is the amount of single-place

[4]Personal communication with Christopher Donawa.

name	LOC	# IR	PP	SP	SW	O_M	O_S
Series	87	2018	P	1	n/P	0	0
KMP	74	2407	1	1	$m+n$	0	0
Reverse	96	3659	P	1	n/P	P^2	n
Crypt	250	5759	1	P	n/P	0	0
Crypt-2	220	5873	1	P	n/P	0	0
SOR	70	1702	1	n	n	0	0
DSOR	68	1742	P	n/P	n	n	n^2

Table 3.1: Size (in lines of code (LOC) and number of nodes in the intermediate representation (# IR) of the compiler) and classification of parallelism for the benchmarks.

parallelism, in other words, how many activities are running in parallel at the same place. In particular, these places will be able to access the same share of the global partitioned address space. A value of 1 indicates that there is only one activity per place involved in the computation. Finally, the figure *SW* is the amount of sequential work that each parallel activity performs. The product of *PP*, *SP* and *SW* gives the total amount of work required for the benchmark (for example, $O(n^2)$ for SOR and DSOR).

For communication, we give two figures. O_M is the number of messages exchanged. O_S is the sum of the size of these messages. The figures for communication do not include initial distribution of the computation and data (which for all parallel benchmarks can be done with $O_M(P)$ messages transmitting $O_S(n/P)$ data with P being the number of places).

Table 3.1 gives some fundamental benchmark statistics. The ArrayBench benchmarks implement the following algorithms:

Series: Calculates the first n Fourier coefficients of the function $(x + 1)^x$ defined on the interval $[0, 2]$. Uses one dependent cast in source code.

KMP: Sequential implementation of Knuth-Morris-Pratt string searching algorithm (with pattern of size m and string of size n). Uses six dependent casts in source code.

Reverse: Given an array distributed across places, reverses the order of the elements. Uses two dependent casts in source code.

Crypt: Implements the IDEA symmetric blockcipher (encrypt and decrypt) using integer increment operations to iterate over a stream. Uses 9 dependent casts in source code.

Crypt-2: Implements the IDEA symmetric blockcipher (encrypt and decrypt) using region iterators to iterate over a stream. Uses 3 dependent casts in source code.

SOR: Given a 2D array, performs successive over-relaxation [PFT92] of an $n \times n$ matrix. Uses two dependent casts in source code.

DSOR: Given a 2D array, performs distributed successive over-relaxation of an $n \times n$ matrix. Uses no casts.

The source code for Crypt, Crypt-2 and DSOR is given in Appendix B.

3.8.2 Region and Place Casts in the Benchmarks

The region casts and place casts in the benchmarks roughly fall into three categories:

1. Required casts due to the fact that the type checker is flow insensitive. The classical Java equivalent for this kind of type cast is of the form `if (a instanceof B)  B b = (B) a;` . Here, the cast itself is always guarded by an dominating branch that yields an assertion that the cast will succeed. These casts should be considered to be free at runtime since a reasonable compiler should be able to completely eliminate the check. They could be avoided entirely if the compiler was flow-sensitive to begin with; however, such a choice is likely to result in problems with respect to programmers' understanding of overloading resolution. In terms of language design, we believe it is better to require the programmer to put in explicit casts even if the control-flow already yields equivalent assertions.

2. Casts are used to cover certain corner cases that could be avoided (but at the expense of using significantly more complex type constructions). For example, a function may operate on arrays of arbitrary size *as long as they are not empty.* Such a corner case might be covered by requiring the programmer to supply an additional point and have the array satisfy the condition that it must contain this point and only points larger than it. A programmer might choose to instead obtain the minimum point of the array using the build-in `min` operator and use a cast (to not-null) to establish that the point exists. Our design allows the programmer to decide that the simplicity of a cast might be a better choice than a complex type construction. Typically, the cost of these casts for corner cases is minimal – programmers are likely to use them outside of loops, and often the particular checks themselves are also rather inexpensive. The reason for this is that if the cast is in a critical section of the code, the programmer has the option of using more elaborate types.

```
Array<point<#1>> overlap(int m, ValueArray<int:([0:m-1])#1> pat) {
    if (m <= 0)
        throw new Exception("Empty pattern!");
    . overlap = new Array<point<#1>>([0:m], 0p);
    overlap[(point<:([0:m])>)0p] = -1p; // CAST #1
    for (p : [1:m]) {
        . prev = p - 1p;
        overlap[p] = overlap[prev] + 1p;
        while ( (overlap[p] > 0p) &&
                (pat[prev] != pat[(point<:pat.reg>) // CAST #2
                                  (overlap[p]-1p)]) )
            overlap[p] = 1p + overlap
                                 [(point<:pat.reg>) // CAST #3
                                 (overlap[p]-1p)];
    }
}
```

Figure 3.21: Overlap computation for Knuth-Morris-Pratt.

3. Casts used to produce loop invariants which the type rules are unable to establish and that are necessary for type checking the loop or code depending on the result of the computation performed by the loop. In these cases, the programmers must add casts to produce the necessary invariants. Naturally, the compiler may still be able to use flow information to reduce the cost of these casts; however, eliminating the check completely would require a theorem-prover that is stronger than what our type-system can offer.

For example, function `overlap` (Figure 3.21) computes the partial match table (or failure function) of the Knuth-Morris-Pratt string searching algorithm [KJP77]. The syntax is similar to Java and C++. `ValueArray` is an immutable array, which means that accesses are not required to be local – only in-bounds. The language uses "." for the type of a local variable that the compiler is supposed to infer from the right hand side of the assignment. The type annotation `<#1>` adds the requirement that the respective point, region or array is one-dimensional. The type annotation `<:r>` specifies that the respective point must be contained in the region `r`.

Cast `#1` (identified by comments in Figure 3.21) in `overlap` falls into both category 1 and 2. The fact that `m` was tested to be positive in the first line of the function establishes that $0p$ is in the (now non-empty) interval $[0 : m]$. However, because the type checker is flow-insensitive, a cast is needed. The programmer

name	entailment checks		dynamic dependent casts			
	total	max.	without types		with types	
	number	size	S	L	S	L
Series	7324	24	12	23	2	2
KMP	11705	42	150	618	124	496
Reverse	48138	46	114	240	12	48
Crypt	24898	24	2684	9980	2591	9887
Crypt-2	65316	31	2684	9980	15	15
SOR	62488	95	192	1200	2	2
DSOR	105374	115	192	1200	0	0

Table 3.2: Numbers of dynamic checks required for the benchmarks.

might have chosen to declare `m` to be strictly positive – a minimal and sane restriction of the API – and avoided both the cast and the sanity check in the first line. Capturing such corner cases with types is often possible, but programmers are likely to use such "dirty" casts wherever they fail to find appropriate types.

Cast `#2` highlights the problem that the type-system may not always be able to establish proper loop invariants (category 3). For the points in the overlap array, the type-system does verify that all points are one-dimensional. However, it cannot establish a loop invariant that would show that the assignment of the form `overlap[p] = overlap[q] + 1p` never produces points with a value larger than $m+1$. Cast `#3` is simply repeating the same cast as cast `#2` and could thus be considered falling into both categories 1 and 3.

3.8.3 Measurements and Assessment

Our implementation can type check and execute the benchmark programs listed above along with the five type-safe example programs from Section 3.3. We collected our measurements by instrumenting the implementation of our X10 variant.

Table 3.2 table shows the number of dynamic checks required for the various benchmarks. We ran each benchmark on two input sizes (marked as "S" for small input, and as "L" for large input).

3.8.3.1 Dependent casts

Our implementation supports dependent casts in order to allow the programmer to execute programs for which the reasoning performed by the type system is insufficient to prove their safety. Such casts show the limitations of our approach – in essence, at this point the language falls back to dynamic checks. In order to improve performance, programmers might therefore be interested in eliminating these dependent casts. The classification scheme described earlier details which casts actually impact performance at runtime and which casts may be eliminated by restructuring the application.

Using the classification scheme, the majority of the static type casts required for the ArrayBench suite falls into the category (3), followed by casts in category (1). Casts in category (1) are usually obvious to the programmer and have no runtime overhead. Determining that a cast falls into category (2) or (3) is less obvious – the reason for this is that there might be non-obvious ways to change the structure or typing of the code which would allow the cast to be eliminated. For ArrayBench, there is on average one such cast in 50 lines of code. Because these casts are infrequent, the effort required from the programmer to investigate possible restructuring of the code to eliminate such casts – should they be in performance-critical sections of the code – is acceptable.

3.8.3.2 Compilation cost

Using the types, the type checking pass of the compiler will verify that all array accesses are in bounds and local using a decision procedure that tries to determine subset relationships between symbolic expressions. Note that the XTC-X10 compiler allows overloading of methods based on dependent typing, resulting in many more invocations of the decision procedure than there are static array accesses in the code. The heuristic used to determine subset relationships that is implemented in our prototype has exponential complexity. However, the problem sizes are relatively small (up to 115 nodes in the symbolic expression tree for ArrayBench). We expect this to continue to be true even for larger benchmarks than the ones studied since type checking can be done per method, and individual methods are unlikely to become extremely large. For the size of the expressions studied in our experiments, the execution time of our heuristic is so fast that it cannot be properly measured, especially given that the implementation is currently in Java where noise from the garbage collector and JIT compiler interfere with measurements on that scale. The total compile time of the ArrayBench benchmarks, including parsing and compilation of 3.000 lines of core X10 runtime libraries, is about 5s on a modern machine for a cold run of the JVM.

3.8.3.3 Performance impact

Our prototype does not allow us to gather meaningful runtime performance data for the generated code. XTC-X10 compiles the benchmarks into SSA-form which is currently interpreted using a multi-threaded interpreter which is written in Java and simulates a distributed runtime. While this does not allow us to give specific speed-up data, it is possible to count the number of bounds and place checks that a language without region types would have to perform and compare it to the number of dynamic region and place casts (which are equivalent to those bounds and place checks) in the typed language. We do not distinguish between bounds checks and place checks because for array locality, any place check is effectively a bounds check for place-adjusted bounds. Consequently, for some particular checks, the distinction would often not be possible.

As expected, the typed language always outperforms the untyped language in terms of the total number of dynamic checks required. For some benchmarks (KMP, Crypt), the reduction that can be achieved is rather small – here, most accesses had to be converted into casts of category (3). For other benchmarks, only a handful of casts remain, and these are often in code that is run only once. This is illustrated by running the benchmarks with two different input sizes. For Series, Crypt2, SOR and DSOR, the total number of dynamic checks does not change if the problem size is increased. The reason for this is that the casts here deal with corner cases, such as initialization. Note that the particular problem sizes chosen for the benchmarks are tiny – for example, the smaller version of Crypt uses a stream of 128 bytes, SOR uses a 6x6 array, and Series computes 3 Fourier coefficients. For larger benchmark sizes, the reduction in the number of dynamic checks will clearly be more dramatic, as shown by the respective second dynamic values.

3.8.3.4 Style matters

The Crypt-2 benchmark deserves some further discussion. The difference between Crypt and Crypt-2 is that most casts were eliminated by replacing integer-arithmetic that was used to walk over the stream (`i++`) with iterators over regions. These iterators are equivalent to the generators of the ordered point list in the operational semantics of the `for` statement in the core language. In particular, they are guaranteed to yield only points that are inside of the region (unlike the `i++` statement which, if used in a loop, does not have an obvious bound). Permitting the programmer to use the (region-typed) iterators directly instead of a `for` loop allows preservation of the original structure of the code. Iterators do have the disadvantage that there is an implicit check – as part of the iterator logic, the iterator verifies a next element actually exists. This check is a range-check

Language	X10	Fortress	Chapel	Titanium	CA-Fortran	ZPL
Place structure	flat	hierarchical	flat	flat	flat	single
Access model	non-value access local	uniform	uniform	global/local	global/local	uniform
Array shapes	arbitrary	rectangular	arbitrary	rectangular	rectangular	arbitrary
Array size	finite	finite	finite or inf.	finite	finite	finite
Region algebra	rich	none	small	rich	none	rich
Dist. algebra	rich	small	build-ins	none	SPMD only	n/a
Memory safety	yes	unknown	yes	optional	no	unknown

Table 3.3: Features of HPC languages.

that could be seen as a bounds check; however, the check of the iterator is also similar to the bounds check performed by any `for` loop. The numbers given for Crypt-2 do not include the test performed by the iterator, just as the numbers in all benchmarks do not include tests performed for the execution of `for` loops.

3.9 Related Work

As already discussed in Section 3.2, many recent HPC languages feature a construct similar to regions. However, the languages differ significantly in their support of regions and use of terminology. Table 3.4 gives an overview of the terminology used by various HPC languages for constructs that related to the type system presented in this chapter. Often these developing languages use different terms for the same or very similar abstractions. In this chapter, we have been using the terminology used by X10. Table 3.3 compares the features of various HPC languages with respect to arrays and memory. We say that the model of distributed memory is uniform if from the programmers point of view there is only one kind of memory. The model is flat if different partitions of the memory exist, but they have no structured relationship between them. The model is hierarchical if the partitions represent a memory hierarchy with multiple levels. Access is uniform if the the programmer does not need to distinguish between accesses to different partitions. With global/local access, the programmer is able to access remote memory using specific constructs. In X10, programmers cannot directly access mutable remote memory – the computation must move. In other words, in X10, non-value access must be local with respect to the computation. Note that some of the listed HPC languages are still experimental. We base our statements on the limited documentation available at this time, which often reflects the state of early prototypes. When the language description does not seem to use a term or if it does not seem to specify the behavior, the respective entry in the table lists *unknown*. If the concept does not apply to the language, the table lists *n/a*.

Language	X10	Fortress	Chapel	Titanium	CA-Fortran	ZPL
Locality	Place	Region	locale	demesnes	image	n/a
set of array indices	region	n/a	domain	domain	n/a	region
array index	point	index	index	point	index	index
immutable data	value	value	unknown	immutable	n/a	n/a
dimensionality	rank	unknown	unknown	arity	rank	rank

Table 3.4: HPC language terminology overview.

ZPL [CCL00, Cha01, CS01, DCS02] does not feature places or distributions; however, its region algebra is extremely rich. For example, it contains operations that allow the specification of a sparse region using a bitmask, as well as multiplication of a region with an integer to obtain a banded region. ZPL is able to generate efficient code for these constructs in part by restricting the way regions can be used. In particular, regions in ZPL are not first class values. Other languages, such as Titanium, feature more general regions but restrict their practical use to rectangular shapes which are easier to implement efficiently. For X10, the assumption is that sophisticated compiler support will enable efficient implementation of complex, ZPL-style region constructs while allowing the programmer the flexibility of regions as first class values. How this can be achieved is still an open research question. For this chapter, the described region algebras are not designed around the needs of the code generator but rather for the needs of programmers and the type system. Efficient code generation for these constructs is left for future work.

HPC languages also widely differ in their specific support for distributed memory. Traditional research has focused on automatic data distribution [GB93, LK02, KK98, Fea93]. In contrast, recent proposals for languages for HPC, such as X10 [CDE05], Chapel [Inc05], and Titanium [UCB05] show a tendency towards exposing the memory hierarchy of modern systems to the programmer. Languages like X10 and Fortress [ACL05] allow or even force the programmer to specify data distributions.

The type system presented in this chapter presents a foundation that full-featured type systems for these languages can build upon in order to statically check operations on distributed arrays. Our type system is inspired by that of Xi and Pfenning [XP98, XP99]. Like Xi and Pfenning, we use dependent types to avoid array-bounds checks. Xi and Pfenning use a decision procedure based on Pressburger arithmetic [Pug91] in order to show the safety of array accesses. In contrast to Xi and Pfenning's language and type system, we study a programming model and type system based on regions. Our type system uses types that are parameterized over regions. Regions or region-like constructs are already present in the designs of many modern HPC languages, making the use of a region algebra instead of (or more specifically, in addition to)[5] Pressburger arithmetic a natural choice.

An alternative to using types to eliminate dynamic checks is generally the use of static analysis. Early work on using static analysis to eliminate bounds checks investigated the use of theorem proving [SI77] to eliminate checks. This work is related in that we use types to guide a decision procedure, a technique which is also used in proof carrying code [Nec97]. For just-in-time compiled languages such as Java where compile time is crucial, the ABCD algorithm [BGS00] describes a light-weight analysis based on interval constraints that is capable of eliminating on average 45% of the array bounds checks. However, the results range from 0 to 100% for the various individual benchmarks, which may make it hard for programmers to write code that achieves consistently good performance. For distributed arrays, we are not aware of any published static analyses that would guarantee locality of access.

When speed is of utmost concern, a language designer may decide to not require any dynamic checks altogether. For example, the reference manual for Titanium [UCB05], a modern language for high-performance computing, defines that operations which cause bounds violations result in the behavior of the rest of the program being *undefined.* A normal continuation of a program after an array bounds violation is clearly not possible. However, this is not the case for non-local accesses.

An alternative to saying that program behavior is undefined for non-local accesses is to generate code that performs a remote memory access if data is not available locally. The type system of Liblit and Aiken [LA00] describes a simple static analysis that enables the compiler to distinguish between accesses that are guaranteed to be local and those that may be remote. A language system using this design would execute programs that the type system presented in this chapter would reject. The problem with this approach is that it makes it more difficult for the programmer to reason about the actual cost of an access. Compared to the work presented in this chapter, Liblit and Aiken's type system also ignores distributed arrays and place-shifting operations. Consequently, an implementation using their design would frequently have to generate conservative access code. Note that using the design presented in this chapter does not preclude a language system from generating conservative access code for operations that merely violate locality constraints. However, in our design, overloading already enables the programmer to do so explicitly, which is in our opinion a better

[5]For deciding region inclusions of the form $[i : j] \subseteq [k : l]$ the decision procedure essentially includes a Pressburger solver.

choice since it ensures awareness of the associated cost.

CHAPTER 4

Conclusion

This thesis described two extensions of type systems for object-oriented languages that help programmers express additional important and common invariants of their code. The extensions can be used to improve code modularity, obtain additional safety guarantees and to emit better code. Experimental results demonstrate the usefulness of the extensions. The type systems have been implemented in the XTC-X10 framework, which is available at

`http://grothoff.org/christian/xtc/.`

The first extension, confined types, protects encapsulation and helps prevent software defects. Confinement is an important property; it bounds aliasing of encapsulated objects to the defining package of their class, and helps in re-engineering object-oriented software by exposing potential software defects, or at least making (often subtle) dependencies visible. By giving programmers the means to easily reason about aliasing and encapsulation, confined types provide an avenue for systematic and consistent object protection in object-oriented languages.

We have demonstrated that inferring confined types is fast and scalable and have developed the `Kacheck/J` tool for inferring confinement in Java programs and used the tool to analyze over 46,000 classes. The number of confined types found by the analysis are surprisingly high, about 24% of all package-scoped classes and interfaces are confined on average for the analyzed benchmarks. Furthermore, we discovered that many of the confinement violations are caused by the use of container classes and thus might be solved by extending Java with genericity, this would increase confinement to 30%. The biggest surprise was the number of violations due to badly chosen access modifiers. After inferring tighter access modifiers, 45% of all package-scoped classes were confined. We expect that these numbers will rise even further once programmers start to write code with confinement in mind.

The second extension, region types, guarantees the safety of array accesses and obviates the need for doing dynamic checks of such accesses. Out-of-bounds array accesses remain a leading cause of security problems and, according to the

National Vulnerability Database [MKG06], buffer overflows are responsible for 233 out of 863 CERT technical alerts or vulnerability notes in the years 2004 and 2005. The type system presented in Chapter 3 can guarantee that no out-of-bounds array accesses will happen, thereby significantly reducing this set of security risks and at the same time, reducing resource usage by eliminating costly dynamic checks of array accesses.

In addition to addressing the array bounds problem, region types provide data locality guarantees for distributed systems. Computations with arrays using distributed memory are fundamental to high-performance computing (HPC). Even HPC languages that present the programmer with a flat, uniform memory model must deal with data distribution and communication issues in their implementation. The type system presented in Chapter 3 can guarantee that all accesses to data, including distributed arrays, are local. This can help programmers using languages that require data locality write correct code, and allows compilers for these languages to generate more efficient access sequences.

Even HPC languages that do not impose locality requirements on the programmer can benefit from this type system. Language systems for these languages internally still need to determine good data distributions to eliminate or reduce communication [CM05]. While the distribution aspect of our core language is clearly not useful for source-level type checking of these languages, the core language does represent a potential type-system for a typed intermediate representation to be used by compilers for those languages. Given the challenges inherent in automatic data distribution [GB93, LK02, KK98, Fea93], another possible application would be the use of automated algorithms to suggest data distributions which could then be manually tuned for improved performance. The type system might be useful to represent the results of the automatic data distribution to the programmer, as well as in describing the manual optimizations performed by the programmer. Regardless of the level at which the language system captures information about arrays and data distributions, a type-system modeled around the core concepts presented in Chapter 3 can be used to give desirable static correctness guarantees.

Future Work

Confined Types

The current type rules for confined types are specific to Java – specifically, Java bytecode. However, relying on Java bytecode has some disadvantages, particularly when it comes to the use of generics (specifically, containers). The homogeneous translation with type erasure used by the Java compiler results in a significant loss of type information for a confinement checker. The estimates for generic-confinable classes given in this thesis show that much could be gained with a better handle on generics.

There are two possible directions for improving the situation. One approach would be the implementation of a source-level confinement checker that analyzes the code before type erasure. Alternatively, an implementation could analyze code written in a language such as X10 with a compiler supporting heterogeneous translation of generic constructs. Integrating confined types into a new language design might also open possibilities for handling native and reflective code by imposing appropriate constraints as part of the specification of the runtime system.

Region Types

The main problem for adoption of region types by the HPC community is the need for an efficient implementation of X10 and, in particular, the region constructs. Considering the complexity of the language, it is expected that an implementation of X10 that is competitive with existing C, C++ and Fortran systems could take the better part of a decade. Using type information and knowledge about region operators should enable whole-program optimizing compilers for X10 to generate code that is as (or possibly more) efficient than code generated by compilers that have less knowledge about the application. Memory safety and type safety can theoretically enable optimizations that are not possible in unsafe languages. However, this will only matter after the costs for safety, such as garbage collection and dynamic checks, have been amortized. By eliminating the need for some of the dynamic checks, this thesis takes the language a step in that direction.

In future work, we plan to explicitly study richer constraint systems that can represent the peculiarities of specific programming idioms. Our existing prototype already supports an extended constraint algebra beyond that used in the core calculus. In particular, the algebra includes support for arithmetic constraints. The extended algebra is needed in order to type check common constructs in actual applications. The underlying principles of the type system presented in this chapter are independent of the particular choice of constraint algebra. Consequently, one major step might be a type checker that is completely parametric in the region algebra, allowing programmers to define their own extensions. We expect our future work on specific region algebras to evolve in step with the power of constraint solvers, the needs of application developers, and our ability to implement the region operators efficiently.

In addition to extending the region algebra, it might be useful to introduce an algebra for places. For example, in X10, we know that $p.prev().next() \equiv p$. It would probably be useful if the language did not predefine-define a specific place algebra since a place algebra should match the topology of the interconnect of the machine, the communication pattern in the application, or a mixture of the two. A trivial algebra that arranges the places in a cycle and features `next` and `prev` operations is defined in the current X10 v0.4 specification. Extending the existing decision procedure with type equivalence rules that can handle `prev` and `next` is trivial. However, a more practical approach would be to make the type system parametric over many place algebras. The system designer could supply a place algebra in the form of operators defining a group (with the empty sequence of operators being the neutral element) using an API that allows the compiler to statically determine the shortest sequence of operations that has the same result. Such an API could also be beneficial for compiler optimizations.

APPENDIX A

The XTC Framework

This chapter discusses various aspects of the XTC framework and the XTC-X10 compiler. The XTC framework is the foundation that is used to implement both the `Kacheck/J` tool and the XTC-X10 compiler with the region types type-checker.

Section A.1 introduces the Runabout [Gro03], a class frequently used in the processing of code (ASTs and intermediate representations) throughout the framework. The Runabout is generally used by XTC instead of the Visitor pattern [GHJ94]. The resulting code is shorter and more extensible, which in turn makes extending the various intermediate representations relatively painless.

The sections A.2 and A.3 describe the internal representations of Java bytecode and X10 IR in XTC. The specific approach of handling of Java bytecode is crucial for the high performance achieved by `Kacheck/J`. The X10 IR is the first attempt of formalizing an IR capable of expressing X10 and the presented type system. The description highlights some of the differences between the theory presented in Chapter 3 and the actual implementation. X10 implementors might also find the description enlightening since it abstracts over the still evolving X10 syntax, giving a more robust presentation of the language design.

Finally, Section A.4 documents the abstract interpretation framework used by XTC for program analysis of both bytecode and X10 IR.

A.1 The Runabout

A fundamental problem in programming language design is making software extensible while both avoiding changes to existing code and retaining static type safety [KFF98]. For example, a programmer may want to add functionality that operates on a number of existing objects, or he may want to introduce a new object to existing code. For such purposes, one strength of object-oriented programming is that it is easy to introduce a new class to existing code. Adding functionality to existing classes is a more difficult proposition, particularly because this typically requires access to the source code. It also may be undesirable to add the new functionality to all subclasses when expanding an existing class.

We call this the extensibility problem and define it as follows:

> **Extensibility Problem:** Devise a mechanism for adding functionality and classes to existing code while avoiding recompilation and retaining efficiency and static type safety.

One traditional solution to this problem is to use the Visitor pattern [GHJ94]. The Visitor pattern allows the addition of functionality in the form of `visit` methods that are invoked from an `accept` method defined in each visitee object. This `accept` method is only specific with respect to the type of an abstract visitor. Visitors do not completely solve the extensibility problem, however. If the set of visitee classes is changed, the type of the abstract visitor changes. Using visitors, it is difficult to change the set of visitees, since all visitors must be adjusted to provide a `visit` method matching the visitee types.

Another solution to the extensibility problem is to use multi-methods; these allow both new functionality and new classes to be added in a flexible and concise manner. Most existing languages do not feature multi-methods. Extending these languages with multi-methods would require significant changes to the syntax, compiler and possibly runtime system of the language. The Runabout is a step toward achieving many of the benefits of multi-methods for Java without requiring any changes to the language.

In this section we specifically address the extensibility problem for Java, giving a solution that supports changing sets of visitee types and provides both acceptable performance (only 2-10 times slower than visitors) and the minimum amount of programming effort. Our solution is based on an approach that was proposed by Palsberg and Jay [PJ98] called Walkabout. Their approach takes advantage of Java's reflection mechanism to implement double-dispatch.

The Runabout implementation presented in this section is an extension of the Java libraries which adds two-argument dispatch to Java. The Runabout is itself implemented in Java (without any native methods); the code for the Runabout is about 600 lines of code and is part of the XTC framework (but does not depend on any other code in the framework). Like the Walkabout [PJ98], the Runabout uses reflection to *find* visit methods. But instead of invoking the visit methods with reflection, the Runabout uses dynamic code generation to create verifying bytecode that will invoke the appropriate visit method. The dynamically generated bytecode is type-safe and can be analyzed and optimized by the compiler.

Generating bytecode for multi-dispatching is also the function of the MultiJava compiler [CLC00]. MultiJava compiles Java with multi-methods to ordinary Java bytecode. Unlike MultiJava, the Runabout generates the invocation code when the application is executed, not at compile time. Thus, the Runabout does not require changes to the compiler or the virtual machine. Contrary to previous beliefs [PJ98], the approach using reflection to determine `visit` targets does not automatically imply an extraordinary runtime overhead. In fact, for 100 million visit invocations on 2,000 visitee classes, the Runabout is slower by less than a factor of two compared to visitors (217s vs. 137s).

The remainder of this section is structured as follows. First, an example for programming with runabouts is given and the semantics of the Runabout are described in detail. In Section A.4.6 the implementation of the Runabout is presented. More details about the Runabout can be found in [Gro03].

A.1.1 Using the Runabout

Writing runabouts is similar to writing visitors or using multi methods. In order to demonstrate how to write code with runabouts, an example which implements the same functionality using dedicated methods, visitors, MultiJava and the Runabout is first presented. Next, the semantics of the `visitAppropriate` method of the Runabout are described. Finally, the specific benefits and drawbacks of each of the implementations in terms of expressiveness and restrictions imposed on the programmer are discussed.

For our example, we are going to use a set of visitee classes $\texttt{A}_i$ which implement the common interface `A`. Given an array `a` of instances of type `A`, the goal is to compute $\sum_{a \in A} I(a)$ where $I(a) = i$ if a is of type $\texttt{A}_i$.

Dedicated methods Dedicated methods can be used to solve the problem efficiently. The problem with dedicated methods is, that for every operation that is to be performed on the visitee classes, a method must be added to each of the visitee classes. This spreads the code used by a particular operation over many classes and makes it often hard to maintain. Figure A.1 shows the solution using a dedicated method.

Visitors Figure A.2 details the code for expressing a solution with visitors. The example uses overloading for the `visit` methods. Overloading is not needed for visitors and it is used here to emphasize the similarities with MultiJava and the Runabout. For simplification, we assume here that only one visitor is being used and that there is thus no need for a visitor interface for the `accept` methods to dispatch upon. In practice, the code would consist of multiple visitors for multiple computations that would be performed over the visitee objects.

```
interface A {
    int dedicated();
}
class A0 implements A {
    int dedicated()  return 0;
}
class A1 implements A {
    int dedicated()  return 1;
}
class A2 implements A {
    int dedicated()  return 2;
}
long run(A[] a) {
    long sum = 0;
    for (int j=0;j<a.length;j++)
        sum += a[j].dedicated();
    return sum;
}
```

Figure A.1: The visitee classes with a dedicated method (`dedicated`).

```
interface A {
    void accept(Visitor v);
}
class A0 implements A {
    void accept(Visitor v) { v.visit(this); }
}
class A1 implements A {
    void accept(Visitor v) { v.visit(this); }
}
class A2 implements A {
    void accept(Visitor v) { v.visit(this); }
}
class Visitor {
    long sum = 0;
    public void visit(A0 a) { sum += 0; }
    public void visit(A1 a) { sum += 1; }
    public void visit(A2 a) { sum += 2; }
}
long run(A[] a) {
    Visitor v = new Visitor();
    for (int j=0;j<a.length;j++)
        a[j].accept(v);
    return v.sum;
}
```

Figure A.2: Visitors require `accept` methods in the visitees.

```
interface A {}
class A0 implements A {}
class A1 implements A {}
class A2 implements A {}
class MultiJavaSum {
    long sum = 0;
    public void visit(A a) { throw new Error(); }
    public void visit(A@A0 a) { sum += 0; }
    public void visit(A@A1 a) { sum += 1; }
    public void visit(A@A2 a) { sum += 2; }
}
long run(A[] a) {
    MultiJavaSum v = new MultiJavaSum();
    for (int j=0;j<a.length;j++)
        v.visit(a[j]);
    return v.sum;
}
```

Figure A.3: MultiJava indicates multi-dispatch using minimal changes to the syntax.

Multi-methods The implementation shown using MultiJava (Figure A.3) does not require the accept methods. Instead, the compiler can see that multi-dispatch is declared (@) and generates code to invoke the appropriate visit method.

Runabouts The Runabout code (Figure A.4) lies somewhere between visitors and MultiJava. The visit methods do not require any additional syntax; all that is required is that the class extends Runabout and that `visitAppropriate` (a method provided by the parent class) is invoked instead of `visit`. As in MultiJava, no `accept` method is required in the visitees.

A.1.2 Semantics

In order to create a Runabout, the client code must create a public subclass of `Runabout`. The `Runabout` class provides the method `visitAppropriate` which can be used for two-argument dispatch. The arguments of the two-argument dispatch are (1) the receiver of `visitAppropriate` and (2) the first and only argument of `visitAppropriate`. The callee of the dispatch is determined by the `lookup` method.

```
public class A0 {}
public class A1 {}
public class A2 {}
public class RunaboutSum extends Runabout {
    long sum = 0;
    public void visit(A0 a) { sum += 0; }
    public void visit(A1 a) { sum += 1; }
    public void visit(A2 a) { sum += 2; }
}
long run(Object[] a) {
    RunaboutSum v = new RunaboutSum();
    for (int j=0;j<a.length;j++)
        v.visitAppropriate(a[j]);
    return v.sum;
}
```

Figure A.4: Runabouts extend the Runabout class to inherit `visitAppropriate`.

visitAppropriate

The callee in the dispatch performed by `visitAppropriate` is either one of the `visit` methods defined in or inherited by the class of the receiver or `visitDefault`. The concrete selection of the `visit` method is performed by the `lookup` function, which, given a `Class`, returns `Code` to invoke one of the `visit` methods. `lookup(T)` may only select non-static `visit` methods that have a return type of `void` and take only a single argument of public type `S` where `S` must be a supertype of `T`. `lookup` may return `null`, in which case `visitDefault` is invoked. If not overridden, `visitDefault` throws a runtime exception to indicate that no `visit` method was found. `lookup` may also throw runtime exceptions (for example, to indicate ambiguities in the method resolution).

Note that `visitAppropriate` does *not* require that all `visit` methods have a common base class other than Object. Thus, the Runabout does not require the interface `A` that most of the other discussed implementations use to declare the dedicated method, to declare the accept method, or as assistance to the compiler in the form of the `A@`.

The fact that the Runabout does not require `accept` methods or a common interface in the visitees is often beneficial when dealing with code where adding an `accept` method is not possible, as with `String`, for example. A simple example for this is given in Figure A.5.

```
public static void main(String[] arg) {
    MyRunabout mr = new MyRunabout();
    mr.visitAppropriate("Hello");
    mr.visitAppropriate(new Integer(1));
    assertTrue(mr.cnt == 3);
}
public class MyRunabout extends Runabout {
    int cnt = 0;
    public void visit(String s) { cnt += 2; }
    public void visit(Integer i) { cnt += i.intValue(); }
}
```

Figure A.5: Using the Runabout on any kind of visitee.

lookup

Which `visit` method is invoked by `visitAppropriate` is specified by the *lookup strategy* implemented by `lookup`. Defining a lookup strategy is similar to defining how a compiler (like `javac`) resolves method invocations for overloaded methods [GJS96, section 15.11.2]. The main difference is that instead using of the static type of the argument object, the dynamic type is used. As with overloading, multiple methods may be applicable. In the case of `javac`, the method with the closest matching signature is chosen, and a compile error is generated in the case of ambiguities.

A simple example of an implementation of `lookup` that does not consider interfaces is given in Figure A.6. The helper method `getCodeForClass(c)` tests whether a `visit` method for the type `c` exists, and, if so, returns the `Code` instance for that `visit` method.

The Runabout has the following lookup strategy: if `visit` methods for both classes and interfaces are applicable to the given dynamic type, the `visit` method for the *class* closest to the dynamic type is chosen; if no `visit` method for a superclass of the dynamic type exists and if there is only one `visit` method matching any of the interfaces implemented by the dynamic type, then that `visit` method is selected; if `visit` methods for multiple interfaces implemented by the dynamic type (but none for its parent classes) exist, a runtime exception indicating the ambiguity is thrown; finally, if no applicable `visit` method exists at all, `null` is returned, causing the invocation of `visitDefault`.

```
protected Code lookup(Class c) {
    while (c != null) {
        Code co = getCodeForClass(c);
        if (co != null)
            return co;
        c = c.getSuperclass();
    }
    return null;
}
```

Figure A.6: Example of a `lookup` method.

A.2 Representation of Java Code

Each IR instruction corresponds to a subclass of the `Instruction` class. Analyses written for the framework use the flyweight pattern [GHJ94] to avoid the need for multiple instances of the same instruction class. The semantic specification of every instruction is provided in the parameterless constructor of the respective class. Each instruction must at least implement the methods `size()`, which returns the size of the instruction in bytes, and `getOpcode()`, which returns a numeric instruction identifier. Further behavior is added only by subclasses for which the behavior is meaningful.

The `InstructionBuffer` class maintains a notion of the current program counter as well as the definitions of constants referenced from the bytecode stream. Because of this design, instruction objects can retrieve and interpret their immediate operands without any state of their own. For example, the concrete instruction `GETFIELD` subclasses the abstract class `FieldAccess`. `FieldAccess` provides a method `getSelector(InstructionBuffer)` to return information about the name and type of the field being accessed. The state required by this method is encapsulated in the instruction buffer argument. This design follows the flyweight pattern [GHJ94], allowing the `InstructionSet` class to hold a single instance of each concrete instruction.

The IR specification contains approximately 3,900 lines of code and 270 instruction definitions which results in about 14 lines of code per definition. The counts include abstract classes and supporting methods. Easily recognizable syntactic conventions (class and constructor declarations) account for approximately 4 lines per instruction; hence, it requires about 10 lines of nontrivial code to specify an instruction. The instruction set itself corresponds to the definition given in the Java Virtual Machine Specification [LY97].

A.2.1 Static Analysis

XTC includes a number of classes for performing static program analysis. In particular, the framework contains a general abstract interpretation framework (described in more detail in Section A.4) which in particular requires clients to provide an implementation of small step abstract semantics over the intermediate representation for an analysis specific abstract domain.

For bytecode analysis, XTC provides a default set of abstract values that corresponds to the set used by a Java bytecode verifier. This abstract domain distinguishes between four basic primitive types (int, float, double, long), the null reference, jump targets (JSR), initialized objects and uninitialized objects. XTC also provides an implementation of the abstract semantics for bytecode for this abstract domain. A step in this implementation corresponds to the execution of a single basic block in the Java bytecode. This is important in order to reduce the number of abstract states that the abstract interpretation framework has to deal with. Each basic block is executed using runabouts that access the bytecode using the flyweight pattern [GHJ94].

Programmers can provide alternative abstract domains or extend the provided domain with additional distinctions. Defining new abstract values requires code that provides tests for value equality and merging of abstract values at join points, as well as an implementation of the operational semantics. If programmer extends the default abstract domain, implementing the operational semantics generally only requires overriding the `visit` methods for those instructions that impact the values in the extended abstract domain.

A typical analysis uses this basic form of abstract execution and interposes calls to analysis specific visitors that inspect the state of the abstract interpreter for information relevant to the particular analysis. If the value abstractions are extended to better match the different abstract domain of a given analysis, visit methods of the abstract interpreter must be overridden to ensure proper handling of the new values. Examples of existing extensions of the abstract value set in XTC include the addition of a special value for the `this` reference in Kacheck/J [GPV01] and the use of type sets for the implementation of 0-CFA [Shi91].

The instruction specification isolates the analysis code from irrelevant changes in the IR. Often a single visit method covers the behavior of multiple instructions that are equivalent from the point of view of the analysis. For example, the default abstract interpreter has generic code for instructions that merely perform basic operations such as moves or arithmetic. Thus, instructions that fall into these categories can be added trivially without changing the abstract interpreter.

Example Figure A.7 gives a simple example of a `visit` method that overrides the default behavior of the abstract interpreter for the `NEW` instruction. In the context of 0-CFA, a `NEW` instruction pushes a flow set on the stack that contains the type of the object being constructed. The visit method is located within the body of a runabout called within the fixpoint iteration. The method starts by querying the `NEW` instruction for the name of the class under construction. Since the instruction is a flyweight object, the runabout passes the current instruction buffer to the instruction so that it can retrieve the type name.

```
void visit(NEW instruction) {
    TypeName type = instruction.getClassName(ibuf);
    frame.push(valueFactory.makeSet(type));
}
```

Figure A.7: The visit method for the `NEW` instruction.

The `valueFactory` object is an abstract value factory that creates the flow set which is then pushed onto the operand stack. The implementation of `getClassName` in the `NEW` instruction class is shown in Figure A.8. The method is written in terms of an operation on the state of the instruction buffer and an auxiliary `getCPIndex` method which returns the constant pool index immediately following the opcode of the current instruction.

```
TypeName getClassName(InstructionBuffer ibuf) {
    int index = getCPIndex(ibuf));
    return ibuf.getConstantPool().getTypeNameAt(index);
}
int getCPIndex(InstructionBuffer ibuf) {
    return ibuf.getCode().getChar(ibuf.getPC() + 1);
}
```

Figure A.8: The `NEW` flyweight instruction object extracts context dependent information from the instruction buffer.

A.2.2 Performance

The code needed to extract information from the abstract execution is typically small. For instance, the addition of the `this` pointer to the abstract value set for our confinement checker, Kacheck/J [GPV01] is specified in 144 lines of code. Flow sensitive types for the 0-CFA algorithm are implemented in 499 lines. The code for constraint generation in Kacheck/J is merely 584 lines; obtaining type set information for the 0-CFA takes 471 lines.

The use of the flyweight pattern for instructions is vital to achieving high throughput. Kacheck/J, which can be run as a standalone application and performs what amounts to a slightly extended variant of bytecode verification, has competitive running times: the tool processes roughly 10MB per second, which appears competitive with similar tools written in C. Figure A.9 shows the running time of the analysis on a set of large benchmark programs. The graph plots the number of MB of bytecode (size of class files inclusive of constant pools) in the benchmark against the time required for the analysis proper (we did not include the time it takes to load the bytecode and parse the constant pools).

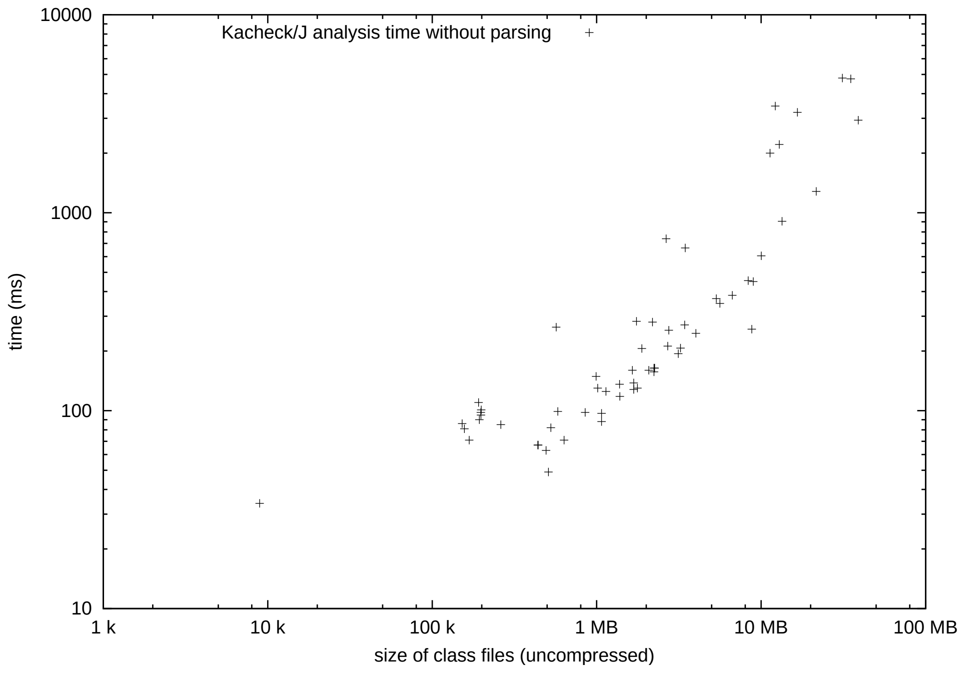

Figure A.9: Performance of the analysis framework for Kacheck/J on a PIII-800 running Sun JDK 1.4.1.

A.2.3 Code Manipulation

Code manipulation is performed by `Editor` objects. Editors operate on instruction buffers and provide two abstractions, `Cursors` and `Markers`. Cursors are used for inserting instructions. Markers act as symbolic jump targets. Like the analyses, editing is typically performed by a visitor that iterates over the code in some application-specific order. A transformation consists of a sequence of edit operations followed by a commit. The original code remains visible until the commit is performed.

Cursors have methods to create all of the concrete instructions. These methods often provide slightly higher-level abstractions than the actual instructions of the IR. For example, the cursor will emit appropriate code for a *branch* instruction, selecting either a short branch or a sequence of instructions that use a combination of short branch and long jump. Similarly, an insertion operation like *load constant* will automatically choose the best instruction for the given value (such as `ICONST3` for 3, or `BIPUSH(42)` for 42). The required constant pool entries are also automatically generated.

The editor uses stateful instruction objects. The reason for this is that code generation requires context information, such as the location of a jump target, that is not always available before a commit. Correctness of the resulting code is not enforced. In fact, the cursor API allows the insertion of arbitrary sequences of bytes into the instruction stream.

Complex modifications such as code motion are possible within the framework, but can quickly become complicated to express since the client code is not provided with any automated mechanisms to deal with data-flow. Moving a single instruction such as `IADD` will always be difficult because the surrounding instructions that first prepare the operand stack and later process the result must also be moved. For non-trivial code optimizations, the XTC framework also provides an SSA-based IR based on [CP95].

A.3 Representation of X10 Code

The current specification for X10 only describes a source-level representation of the language. The IBM reference implementation converts the corresponding abstract syntax tree (AST) to Java source code, linking against a runtime system implemented in Java. However, Java source and Java bytecode are not particularly suitable for analysis and optimization of X10 programs. X10 is a new language that deviates significantly from Java; some of the original design goals behind Java bytecode, such as fast interpretation and compactness, do not apply to X10. Java bytecode is rather complex with hundreds of instructions for handling eight different primitive types, JSR and implicit exceptions in almost any opcode. Furthermore, it has almost no features to express low-level optimizations such as array bounds or null-check elimination. X10 is not intended to have any primitive types, and the type system is designed to eliminate runtime exceptions; this is a significant departure from Java.

What is possibly more detrimental about using Java bytecode for representing compiled X10 code is that the stack-based IR of Java bytecode is extremely difficult to manipulate, especially for tools that are not integrated with a Java just-in-time compiler (which may then avoid limitations of bytecode by abandoning bytecode internally as soon as possible). Optimization stages for bytecode must be careful to preserve stack semantics and are thus either limited to relatively simple operations or required to first be compiled into an internal IR and then transpiled back into bytecode – with the hope that the result can actually be expressed in bytecode. This last concern is a rather substantial issue due to size limitations in the bytecode. Limits on the total method size and the number of constants in a class file may not be a problem for Java source compilers, but generated Java code, optimizers using inlining, parser generators and other source generators are known to easily violate these bounds. For example, one development version of the IBM parser generator for IBM's version of X10 pushed method sizes beyond Java's limit. Splitting up methods and classes artificially as a solution to this problem may not always be an option (due to class hierarchy and visibility limitations).

Consequently, a new intermediate representation based on a variant of static single assignment form (SSA) [CP95] was designed to be used in the XTC framework for analysis and optimization of X10 code. This section describes the fundamental elements of this representation and tries to motivate some of the design decisions.

The description of the IR given in this section is not complete. Some details, such as a binary format, have not yet been implemented. Other details, such as visibility rules and extended modifiers for methods, are omitted because they are either identical to Java, irrelevant in relation to the thesis, or trivial.

A.3.1 Types

The following is the list of types used in the IR:

- Base types (represented in the implementation using instances of `X10Type` for the concrete instance and `X10TypeName` for symbolic names) describe simple source-level classes, such as `Object`, `int` or `String`. The package structure and visibility modifiers are the same as in Java, except that the default package does not exist. Static nested classes are also allowed, the IR has no special provisions for non-static inner classes.

- Parameterized types (represented by instances of `X10InstaniatedType` and referred to symbolically using `X10ParameterizedTypeNames`) are used to represent classes that are parameterized over other types or symbolic type parameter names (such as `Array<T>`). Type parameters can be annotated with a bound and a variance (covariant, invariant and contravariant). When a symbolic type parameter is bound to a specific type, the code of the methods of the type is replicated, replacing occurrences of the symbolic name with the concrete type.

- Function types (`X10Function` and symbolically `X10FunctionName`) are used to represent higher-order functions. A function type always contains a constructor (which initializes the closure) and a method `f` which represents the code of the function. Function types are different from normal `X10Types` in terms of subtyping (covariant for return type, contravariant for arguments). Instances of type `X10Function` can be used as the receiver of an `FCALL`, which is a special IR instruction for evaluation of higher-order functions.

- Region types only exist symbolically in the type system (using instances of `X10RegionTypeName`). The reason for this is that a dependent type (and region types are a special form of dependent types) should have no runtime equivalent – the concrete values of the instance give the dependent type. For example, the region type of an integer 5 may express that the value has dependent type "integer of value 5". Obviously, having both the value 5 and a type "integer of value 5" at runtime makes no sense. The runtime type of a dependent type is the underlying base type (such as `int` or `region`) without the constraints described by the dependent type clause. Section A.3.7 gives details on the forms and representations of dependent constraints that can be expressed by region types in X10 IR. Note that after type checking, an X10 runtime may choose to completely ignore dependent type information. In particular, region types cannot be used in IS_INSTANCEOF tests in the IR. Note that such `instanceof` tests are legal in X10 source code; in the IR they are represented by simple operations on values, not types.

All `TypeNames` come in two variants: *nullable* and *non-nullable*. This distinction is not made for `Types`, since the type of a concrete instance is obviously never *nullable* with the exception of the `null` pointer, which is modeled by a special `X10TypeName` called `NULL` which lacks the non-nullable variant. Methods returning `void` use a special `X10TypeName` called `VOID`.

In addition to Java's type modifiers (`abstract`, `final`, `public`, etc.), X10 IR features two additional type modifiers. The modifier `value` is used to indicate that all fields of the type are immutable (for function types, this means that the variables captured in the closure of the function cannot be assigned to). The modifier `extern` is used to indicate that *all* methods (including constructors) of the type are not implemented in X10 but in another language (such as Java or C). X10 IR does not allow individual methods to be marked as `native`.

The various types in the IR can be combined freely. For example, it is possible to specify a function that takes a parametric type as its first argument, and where the parametric type is a `nullable` basetype with a dependent constraint on its values. For example, the types can represent the X10 code

function int(Array<**nullable** point<#3>> a);

as a function that returns an int (base type), takes a (non-nullable) Array (parameterized type) which must contain 3-dimensional (#3) `points` (region type) or `null` pointers.

A.3.2 IR Elements

Code in X10 IR is represented using a graph combining data and control-flow, inspired by Cliff Click's design for an intermediate representation for an optimizing Java compiler [CP95]. The basic structure of the graph is a directed graph of data dependencies. The IR can be executed directly using an interpreter; however, the primary design goal is to make transformations easy with the intent of eventually compiling the IR to a lower-level representation that would be more suitable for interpretation or direct execution. Consequently, direct interpreters for the IR are likely to be rather slow. The following description of how to interpret the IR should thus be seen as describing the general operational semantics of the representation, not as the desired model of execution.

A.3.2.1 Graph structure

All nodes in the graph of a method are reachable from the unique RETURN node of a method. There are four main categories of nodes in the graph:

- **Control nodes** represent the control-flow of the code.
- **Heap nodes** capture ordering constraints on updates to the heap.
- **Value nodes** represent objects and values that are used in the computation.

- **Projections** represent control, heap or value nodes that are the result of a particular operation that produced multiple results.

Example A branch (IF) is a control node. It has two inputs, another control node (the preceding control point) and a value node (representing the boolean condition that the branch depends on). It produces one successor for the case that the condition is `true`, and another successor for the case that the condition is `false`. These different results are represented using two control nodes which are also projections. Projections always have a single input, the node for which they represent the different "results"; in this case, that input would be the IF node. Since executing the branch itself does not change the heap, IF has no direct relation with heap nodes.

A.3.2.2 Execution

The basic execution model for the X10 IR is pretty much identical to the semantics described in [CP95]. The only major difference between the two IRs are in the specific instruction sets. Execution of each method uses a state consisting of the current control node, the current heap node, a mapping of evaluated nodes to instances and possibly the node that is currently being evaluated.

Execution begins with the ENTRY node as the current control node. The entry node projects the initial heap node and one projection per method argument. Execution of the entry node requires binding the argument projections to the concrete instances passed as parameters to the method call. The ENTRY node, like all non-branching control nodes, should have exactly one control node that uses it as its input (remember the direction of the references is "depends on", reaching control of one other node in the graph should depend on entering the method). Control continues with that node. Control nodes (like, for example, a branch) may depend on nodes representing values. If control reaches such a value, the respective value tree is evaluated. The mapping of evaluated nodes to instances is considered to avoid re-evaluations.

Nodes may also depend on a particular state of the heap. If a particular heap state is desired and that heap state is not the current heap state, the interpreter must consider two possibilities. Either the current heap state is a predecessor to the desired heap state, in which case there must exist a sequence of operations on the heap that will result in the desired heap state. If the current heap state is a successor of the desired heap state, the desired computation has already been performed and the desired value (if any) should be taken from the mapping of evaluated nodes. Note that one of the effects of evaluating a node that changes the heap is to advance the current heap state to the successor state (which is usually a projection of the node changing the heap).

REGION nodes are control nodes that are used to represent control-flow joins. The ordered inputs to a REGION are n control predecessors. They depend on a list of PHI nodes, which must be evaluated when the REGION node is reached. Evaluating a PHI node requires evaluating its k-th input, where k is the index of the control predecessor that was executed before reaching the REGION. PHI nodes can be either value nodes or heap nodes. Interpretation of a method ends upon reaching the RETURN control node. The inputs of a RETURN node include a return value, the desired final heap state, the previous control node and an exceptional return value (which will evaluate to `null` if the method does not throw an exception). The return value will be invalid (also represented by `null`) if the method does not throw an exception or returns void.

A.3.2.3 Overview of the IR elements

Each figure in the following sections illustrates the inputs and outputs of a particular node or a set of closely related nodes. The names of the nodes that are the focus of the figure are written in all caps. The other nodes contain a description of the function that these nodes have in the context. Values are represented using round borders; control nodes are drawn with rectangular boxes. Value dependencies use normal arrows ($\rightarrow$) and control edges use double arrows ($\Rightarrow$). PHI nodes are given with dotted arrows. Finally, END bindings (a new concept) are given with squiggly arrows.

A "$*$" on an arrow indicates that any number of edges are possible. The "$*$" is only shown for the final outbound edges (which have no target in the figure) and in cases where an arbitrary number of arguments is possible.

The direction of the arrows does not (always) indicate the direction of the references in the graph. The arrows indicate the direction of the control and data flow (source to sink), whereas the references in the implementation generally point in the inverse direction. The exception to this rule are the edges between a node and its projections and a REGION and its respective PHIs. The implementation uses references in both directions for those edges.

A.3.2.4 TYPE

The `typename` field gives the name of the type (i.e. "x10.lang.String") and possibly conveys other properties such as `nullable` and type parameters (`Vector<T>`). The `type info` field is an object that contains information about methods, fields and subtyping relationships of instances of the respective base type. The TYPE IR element does not have any operational semantics by itself, it is merely used to

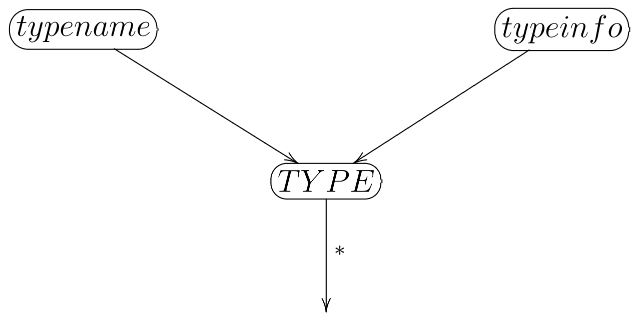

Figure A.10: TYPE defines an X10 type (value or object, nullable). TYPES may depend in their definition on other values in scope.

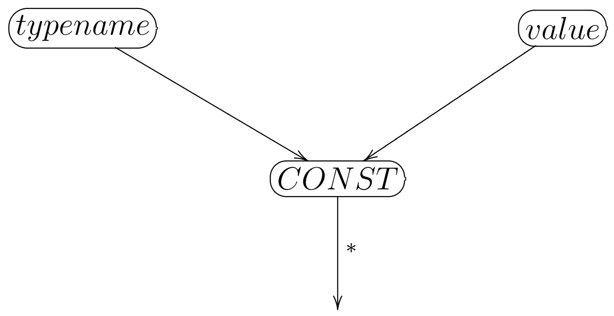

Figure A.11: CONST defines a value.

refer to a particular type in the IR. TYPE nodes are used for object allocation, casts and `instanceof` tests.

A.3.2.5 CONST

CONST is used to specify basic values such as integers and strings. Each CONST instruction contains the value represented as a string (to be converted by the compiler into the appropriate binary representation) together with the name of the respective primitive value type (for example, `x10.lang.int`). In our implementation the conversion is achieved by requiring that the native implementation of the primitive value defines a constructor that takes a string as an argument and produces the correct primitive value. The compiler simply uses this constructor in order to initialize primitive values that it is unaware of.

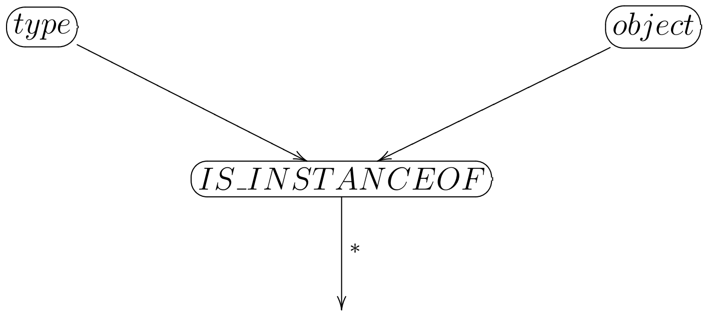

Figure A.12: instanceof

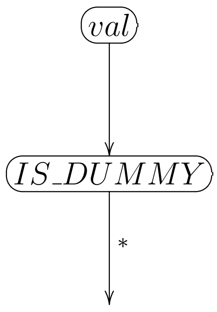

Figure A.13: Tests if the argument is 0 (or NULL).

A.3.2.6 IS_INSTANCEOF, IS_DUMMY and IS_NOT

The IS_INSTANCEOF instruction takes an object and a type and returns true (technically an `x10.lang.boolean`) if object is an instance of the given type.

IS_DUMMY returns true if the argument (which should essentially always be a pointer) is NULL. The name is used because NULL is often used as a special dummy value (e.g. to indicate *no exception*).

IS_NOT takes an argument which must be of type boolean and returns a boolean that is true if the argument was false and vice versa.

A.3.2.7 IS_AND and IS_OR

IS_AND takes two arguments which are guaranteed to be of type boolean and returns a boolean that is true if and only if both arguments evaluate to true.

IS_OR takes two arguments which are guaranteed to be of type boolean and returns a boolean that is true if and only if either argument is true.

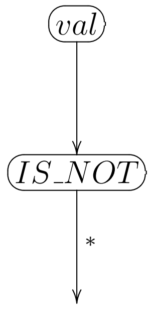

Figure A.14: Boolean NOT operation.

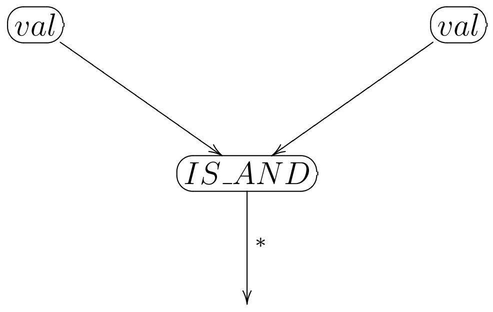

Figure A.15: Boolean AND operation.

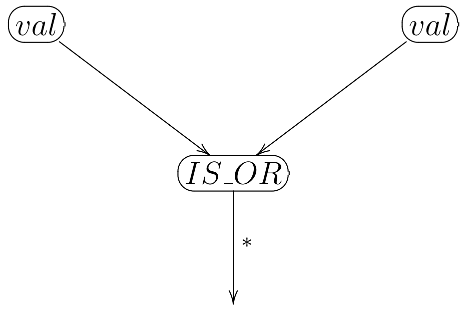

Figure A.16: Boolean OR operation.

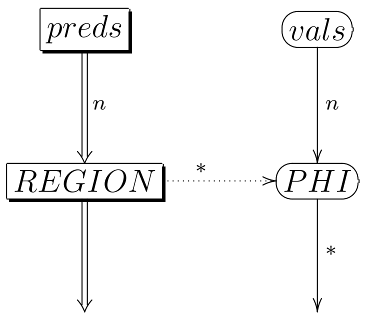

Figure A.17: REGION is essentially the place where control flow merges.

A.3.2.8 REGION

REGION has nothing to do with X10 regions. The term REGION was used by Cliff Click [CP95] for nodes that represent control-flow merges. Since this work builds on his design, his terminology is preserved. The use of REGION vs. region should be sufficient to avoid confusion.

The IR uses PHI nodes to represent choices between values based on control-flow [CFR91]. A REGION can have any number of PHIs. If the REGION has n (ordered!) control-flow predecessors, each of the PHIs must have n (ordered) input edges. A PHI must belong to one and only one REGION.

Evaluating a REGION requires the compiler to update all of the attached PHI instructions in parallel (!) to the values that correspond to the respective control predecessor. This is why the order of the control predecessors as well as the values in the PHIs matters.

After evaluating the PHIs, control proceeds with the next control node that follows the REGION node. While a REGION may have many control-in edges, there must only be one control-out edge.

A.3.2.9 INVOKEVM

The INVOKEVM instruction is a general escape mechanism to make a call into the virtual machine to do something special. It is parameterized with a string argument that specifies the desired VM service.

The current X10 IR uses only one such service, namely object allocation. For object allocation, the VM is expected to simply allocate an object of the specified type. The desired type is specified as an argument to the INVOKEVM instruction. INVOKEVM's allocation should not invoke a constructor and does not initialize any fields (zeroing maybe required for the garbage collector). However,

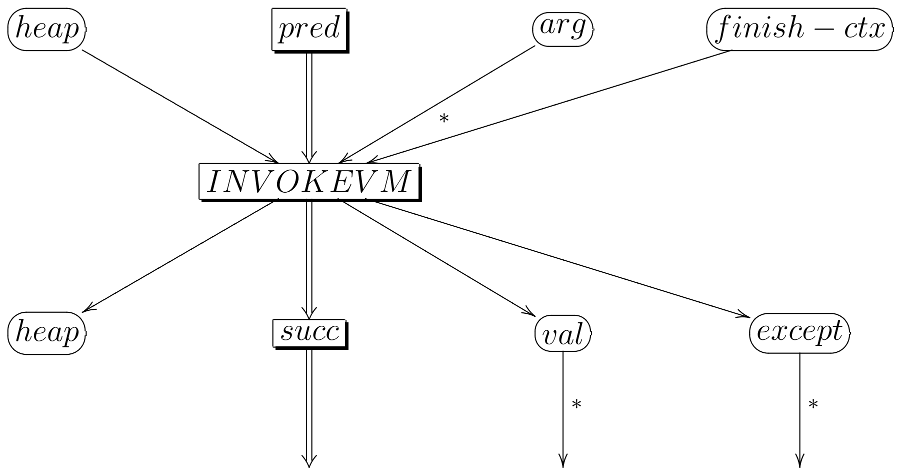

Figure A.18: The INVOKEVM instruction.

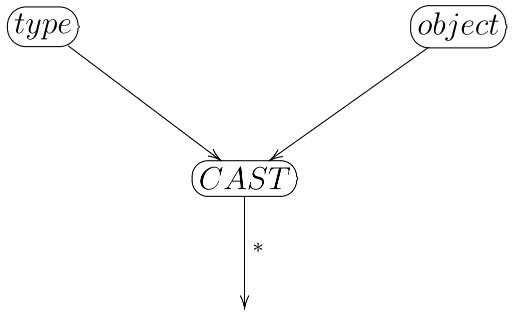

Figure A.19: Cast.

the type and VTBL information for the object must be initialized.

INVOKEVM's allocation should never fail with an exception. In contrast to Java, where any allocation may fail with a *catchable* out of memory exception, the default allocation mechanism in X10 IR says that a failing memory allocation results in an immediate abort of the entire application.

Future extensions should provide alternative allocation services that provide stack allocation and allocations with the possibility of catching out of memory exceptions.

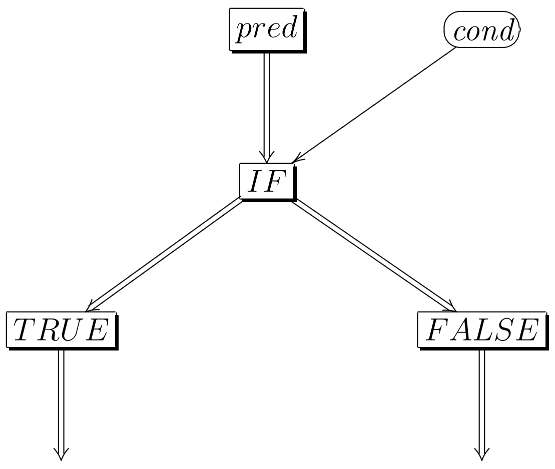

Figure A.20: Branch.

A.3.2.10 CAST

Note that semantically, a CAST is essentially a NO-OP: the output is exactly the input. CASTs are only used for the compiler to determine the static type of an object. For example, consider the following code:

```
void m(String a) {
    Object o = a;
    m(a); // (1)
    m(o); // (2)
}
void m(Object o) {
}
```

Here, the first call marked (1) must resolve to `m(String)` and the second call marked (2) to `m(Object)`. In order for the method overloading resolution to properly resolve `m(o)` the compiler will insert a CAST to `Object` into the IR. CASTs are always safe and a type checker should always be able to statically check this.

A.3.2.11 IF

The IF instruction expects a boolean conditional as an input. It proceeds with the control after the TRUE projection if the conditional evaluates to true, and

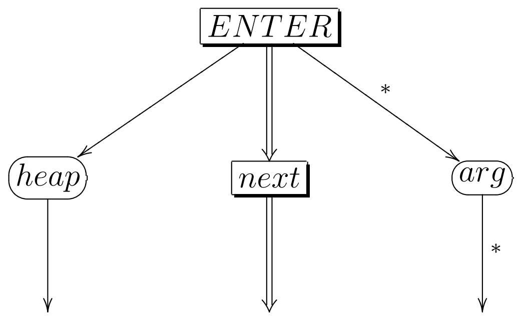

Figure A.21: Method entry.

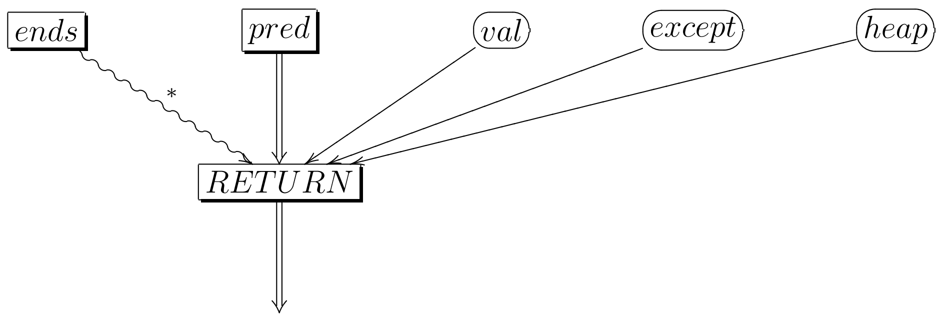

Figure A.22: Method exit.

otherwise proceeds with the FALSE projection.

Remark. The XTC-X10 IR also contains an instruction SWITCH with the usual semantics.

A.3.2.12 ENTER

The ENTER instruction projects special arguments for the starting heap, current place (arg0) and the finish context (arg1), as well as normal arguments (possibly including the receiver). Execution proceeds with the unique control successor (next). There is exactly one ENTER instruction per method.

A.3.2.13 RETURN

The RETURN instruction must be reached from a single control edge. It anchors all END nodes (from ASYNCs, discussed in more detail in Section A.3.2.19). It

$$\begin{tikzcd}
object \arrow[d] & heap \arrow[dl] \\
GET \arrow[r] \arrow[d, "*"] & heap \arrow[d] \\
{} & {}
\end{tikzcd}$$

Figure A.23: GET. Reads a value from the heap (using an object as the base address, plus an offset obtained from the particular field that is read).

$$\begin{tikzcd}
object \arrow[dr] & value \arrow[d] & heap \arrow[dl] \\
& PUT \arrow[d, "*"] & \\
& {} &
\end{tikzcd}$$

Figure A.24: PUT. Writes a value to the heap (using an object as the base address, plus an offset obtained from the particular field that is read).

also takes the final heap, return value and the return exception value as inputs. The method terminates normally if the return exception is a dummy (NULL) value. There is exactly one RETURN instruction per method.

A.3.2.14 GET and PUT

Note that the selector that describes the specifics of the field for GET and PUT nodes is not shown for brevity. The PUT node itself is treated as a heap node, avoiding the need to project a new heap state explicitly. GET projects a new heap state; the GET node itself is used to represent the value obtained from the heap.

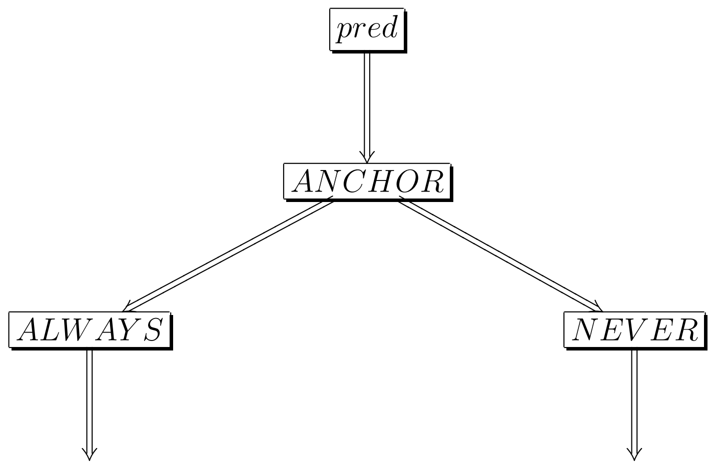

Figure A.25: Anchoring non-terminating loops.

A.3.2.15 ANCHOR

The ANCHOR instruction always proceeds with the control after the ALWAYS projection. Control never takes the path of the NEVER projection. ANCHORs are needed to represent infinite loops (`while(true);`) since the all of the code must be reachable by graph traversal from the RETURN instruction.

A.3.2.16 CALL

CALL essentially invokes the specified method with the specified arguments and upon completion produces the return value and exception state of that was consumed by the RETURN of the callee. The CALL is evaluated when the heap needs to be advanced to the heap projected by the CALL instruction. CALL is **not** a control instruction.

Note that the selector of the method and a `boolean` flag distinguishing virtual from non-virtual calls are not shown for brevity. Also, the current place and the receiver are shown only as some of the arguments.

A.3.2.17 FCALL

FCALL is very similar to CALL except that the callee is not a method in a class but a function. Functions are created by a NEW instruction with a special type (function type). Note that IBM's X10 implementation does not have function types; this is an XTC extension. Other than the different invocation

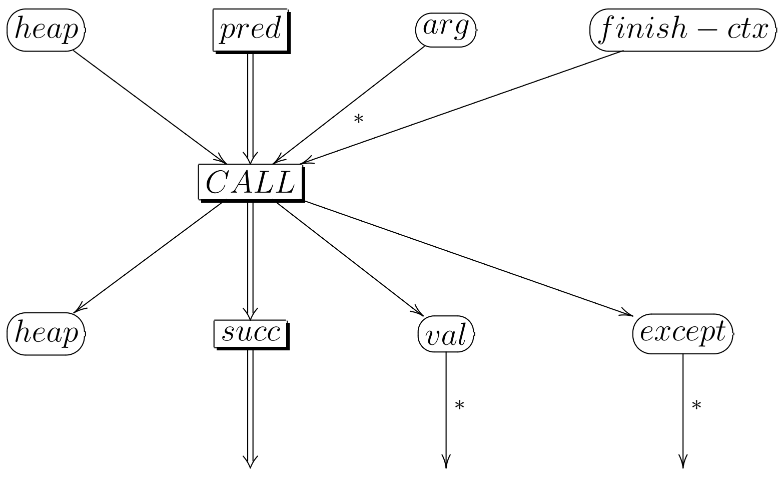

Figure A.26: Method call.

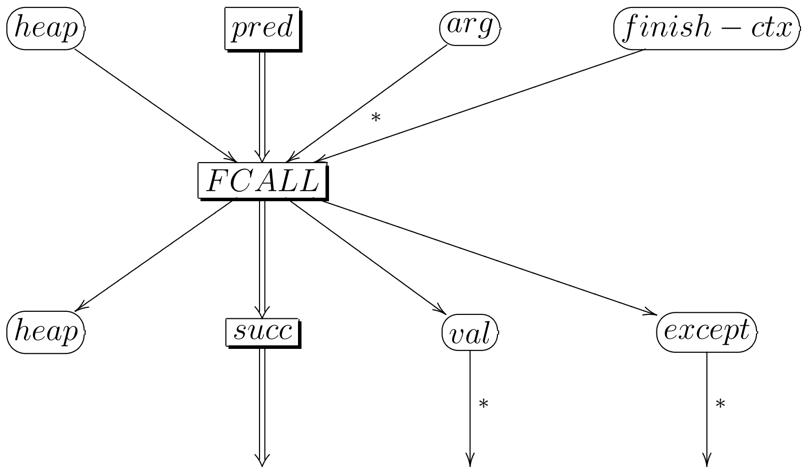

Figure A.27: Function call.

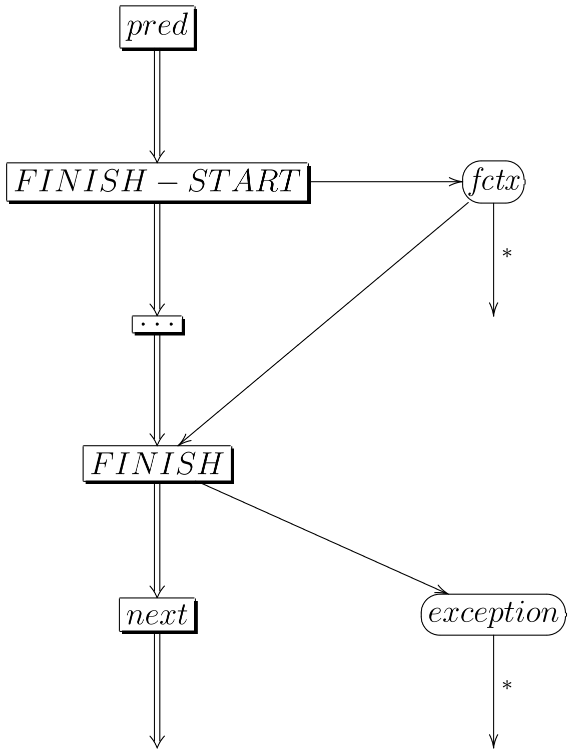

Figure A.28: FINISH waits for all activities registered with the finish-context (fctx) to complete.

mechanism (and the lack of a selector and the virtual/non-virtual distinctions), FCALL behaves just like CALL.

A.3.2.18 FINISH

Each `finish` block starts with a FINISH-START instruction that merely creates a fresh finish-context (*fctx*). This context is passed to the various ASYNC statements in the scope of the `finish` block. The FINISH instruction itself must then block until all of the asynchronous activities have completed. FINISH projects an exception which is NULL if all asynchronous activities completed successfully. Otherwise, an exception of type `x10.lang.MultipleException` is created which internally captures the various exceptions thrown by all ASYNCs inside of the FINISH.

Note that a `finish` block encloses any X10 execution, ensuring that a finish-context is always available.

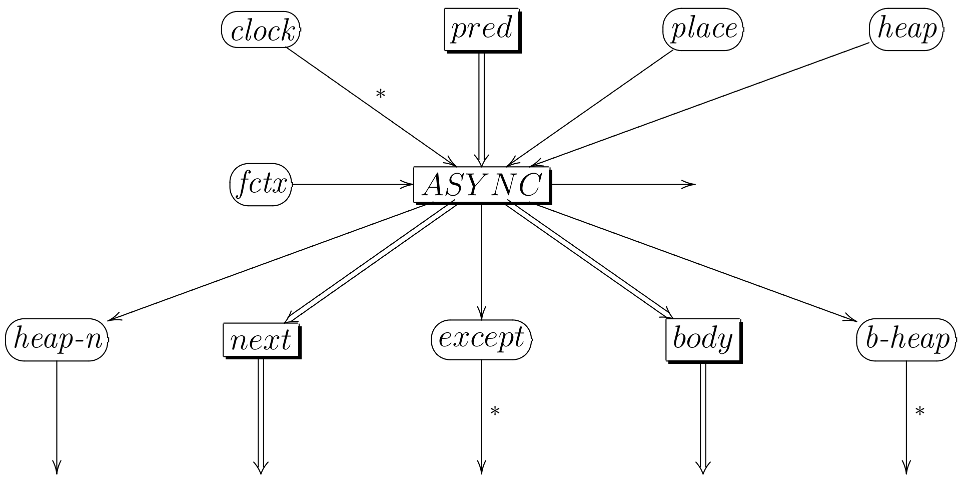

Figure A.29: Async. Note that the "body" must eventually end with an END node. The END node must be consumed by the RETURN node of the method.

A.3.2.19 ASYNC and END

ASYNC initiates running a parallel activity (body) at the specified place starting with the projected heap (*b-heap*). The activity is registered with the (optional) set of clocks. Specifically, each of the specified clocks may only advance into the next phase if the newly spawned activity is ready to advance (by having called `resume`, `next` or `done` on the clock). X10 guarantees that the clocks passed to ASYNC are currently registered with the current activity and that the activity has not yet declared quiescence in the current phase of the clock. This is currently assumed to be checked by a runtime check in `clock.register()`. In the event that `clock.register()` fails, ASYNC should not spawn the new activities but instead project the respective exception. Execution continues with the *next* control and the *heap-n* projected by the ASYNC. The *next* control then generally has to check for the possibility of a non-null *except* being projected by the ASYNC. In the absence of *clock* arguments, or based on flow information about the clocks, the compiler may be able to eliminate that test.

ASYNC also takes a special boolean flag (not shown) that indicates if the spawned activity must be added to the finish-context (*fctx*). This flag is false for X10's `future` constructs.[1]

[1]In other words, Vijay Saraswat is wrong saying that `future` is a defined construct, because `future` could not be constructed from ordinary `async` statements due to the different `finish` semantics. In my opinion `future` should be removed from the language, which would make this flag obsolete.

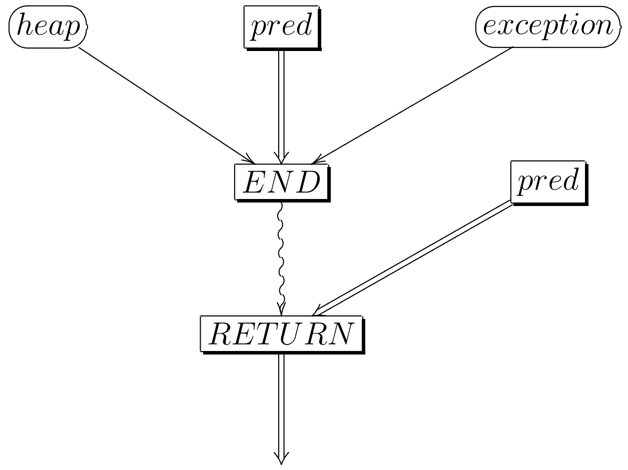

Figure A.30: END. The end of a parallel activity.

ENDs must be bound to the RETURN node (to anchor the graph). The exception object consumed by the END must be communicated to the respective FINISH that started the activity.[2] Depending on the implementation, END may also need to update the *fctx* at the termination of the activity such that FINISH can eventually continue.

Remark. END currently does not have a link to the respective *fctx*, but this could easily be changed.

A.3.2.20 FOR

Executing FOR begins with reaching the FOR control instruction from the *predecessor* with a *heap* and a *region*. The FOR also has an iteration *order* (i.e. forwards or backwards). Running FOR produces the FOR-ITERATOR, which should be seen as the state of the `for` loop. After running FOR, the heap advances to the *s-heap* to indicate the evaluation of the *region*. The REGION between FOR and FOR-BODY is used to update variables modified in the loop. There can be many PHIs, though only the PHI for the heap is shown here. The FOR-BODY uses the FOR-ITERATOR to produce the FOR-POINT, the current index for the iteration. FOR-BODY has two control projections, one for the normal loop body (BODY) and one for exiting the loop (EXIT). A heap projection exists for each

[2]`future` constructs are implemented in a way that catches any exception and thereby guaranteed to always terminate normally.

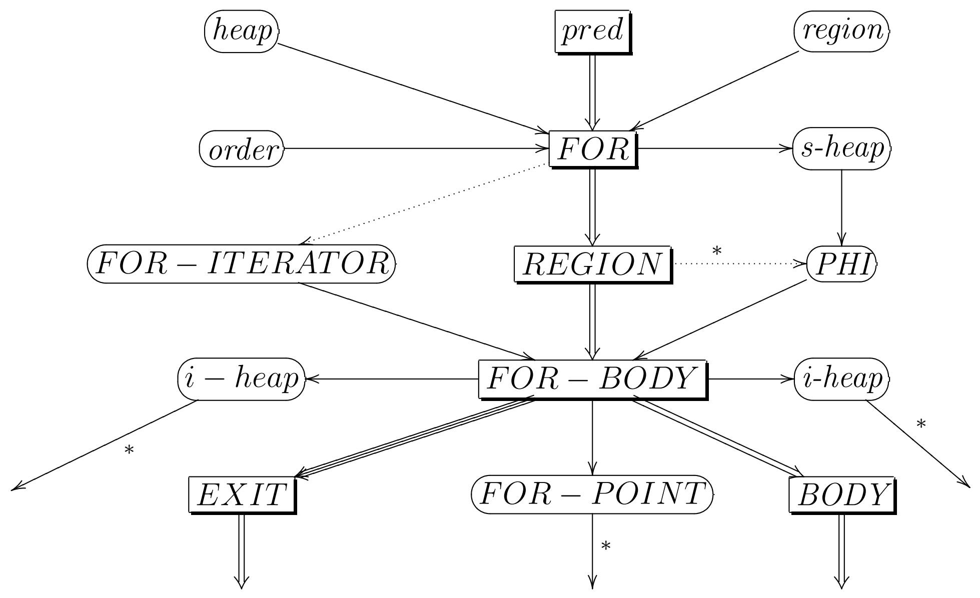

Figure A.31: For-loop.

of these control projections (e-heap and i-heap respectively). The control after BODY either eventually reaches the REGION between FOR and FOR-BODY (continue and fall-through case) or a REGION in the control after EXIT (for break statements).

Note that it is easily possible to represent FOR using a combination of CALL, IF and REGION instructions. FOR is preserved in this form in the IR mostly to allow the compiler to easily recognize and optimize such loops.

A.3.2.21 ATOMIC

XTC-X10 translates both conditional and non-conditional atomic sections into the same IR instruction, which as a result is probably the most complex construct in the IR. The presentation is thus broken into three parts, ATOMIC, CONDITIONAL-BLOCK and CEND-END.

An ATOMIC instruction has multiple CONDITIONAL-BLOCKs as successor control nodes, as well as one EXIT control node. This EXIT control node is chosen if the evaluation of the conditional expression in any of the conditional blocks results in an exception. In that case, this exception is projected and control continues with the *e-heap* after the EXIT control node.

Executing ATOMIC requires evaluating the CONDITIONAL-BLOCKs atomically in random but fair order until either one of the conditions evaluates to true

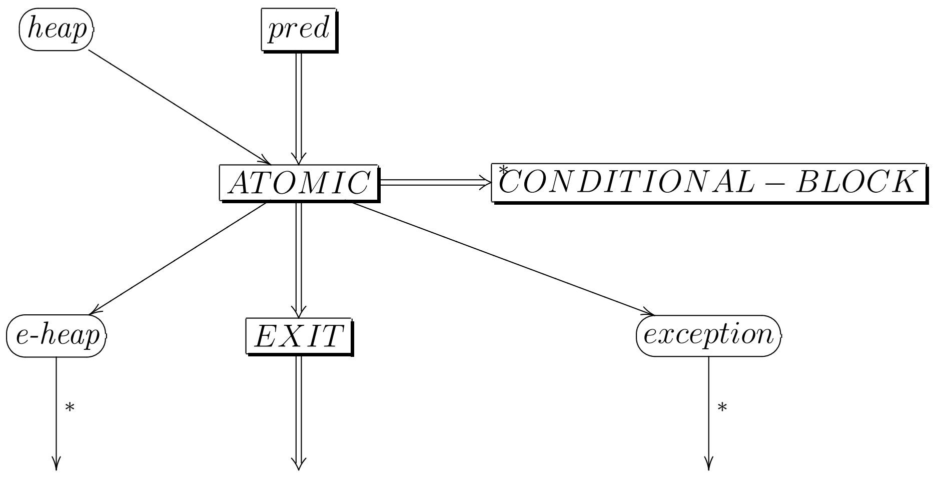

Figure A.32: Start of an ATOMIC section.

or results in an exception.

The CONDITIONAL-BLOCK projects a starting heap and control point. The control must eventually reach the CEND instruction. CEND consumes the final heap.[3] CEND also consumes an exception and a conditional value. If the exception is non-NULL, control continues with the exceptional EXIT from the respective ATOMIC instruction. If the condition evaluates to true, control continues after CEND. If the condition evaluates to false, the next CONDITIONAL-BLOCK is evaluated.

Note that while each conditional block must individually be executed atomically (possibly including the atomic statement following the guard), the system must allow other activities to execute atomic sections eventually. That is, it is illegal to always simply acquire a global lock on entry to ATOMIC and only relinquish the lock after some conditional section allows exiting the atomic section.

The compiler should also optimize the case of unconditional atomic sections where there exists only one CONDITIONAL-BLOCK with no code between BODY and CEND and where *cond* is always `true`.

Control after CEND must eventually reach a unique END instruction. This END instruction consumes the final heap and an exception. If the exception is NULL, control continues without atomicity guarantees after the END. If the exception is non-NULL, the exceptional EXIT of the respective ATOMIC in-

[3]Technically the conditional expression should not modify the actual heap, but this is currently not enforced. Either way, the heap is passed around to ensure that GET and CALL instructions can have their usual structure.

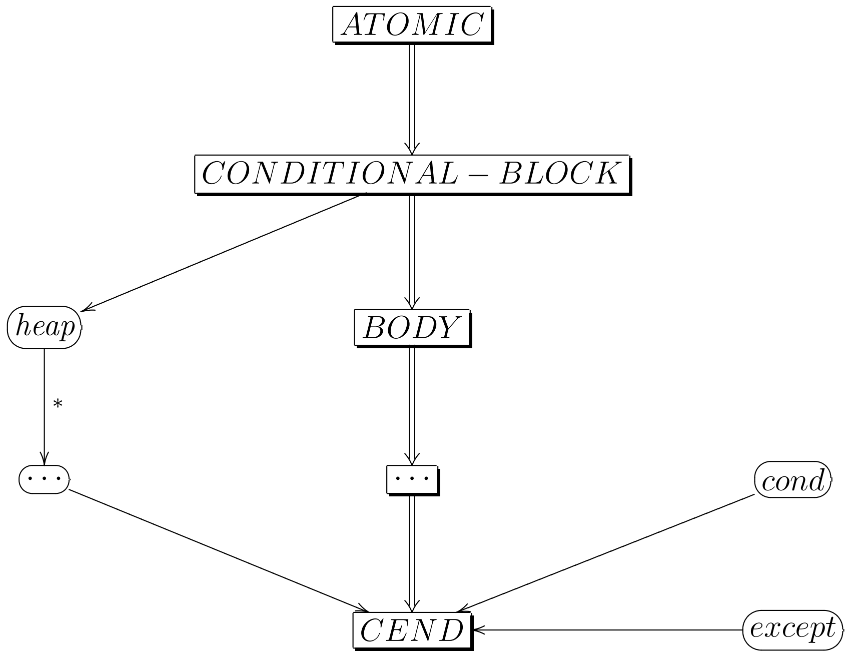

Figure A.33: Conditional block in an `atomic` section.

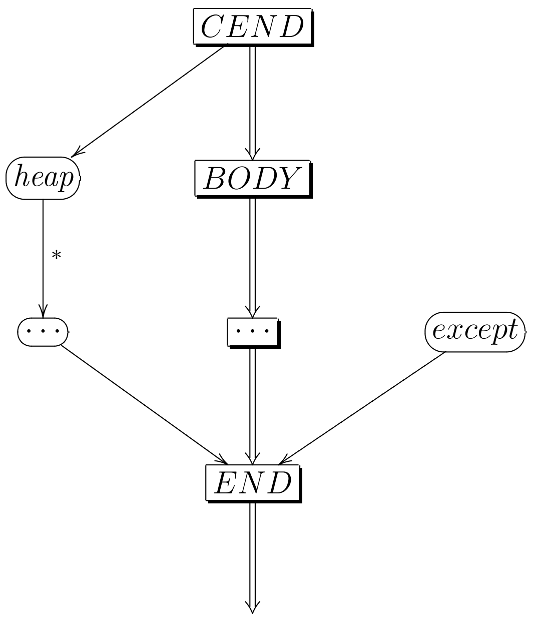

Figure A.34: Atomic statement between CEND and END.

struction is taken.

A.3.3 Graph invariants

Any valid graph of IR elements must obey the following invariants:

- There must be exactly one FINISH node per FINISH-START.
- There must be at least one END node per ASYNC.
- There must be exactly one ENTER and one RETURN (per method).
- Any control node other than END and RETURN must have at least one control node in its successor set.
- Any control node other than IF, ANCHOR, SWITCH, FOR, ASYNC and ATOMIC must have at most one control node in its successor set. IF, ANCHOR, FOR and ASYNC must have exactly two control node successors.
- Any control node other than REGION must have at most one control node in its predecessor set.
- Any control node other than ENTER must have at least one control node in its predecessor set.
- A flow-sensitive data-flow analysis must be able to establish that any value flowing into a CAST has a subtype of the type specified by the CAST.
- Execution as described in Section A.3.2.2 does not result in the use of undefined values; in particular PHIs can only be used after control has reached the respective REGION and execution will not require the same HEAP to be used as the input to two different instructions

A.3.4 Compiling X10 to X10 IR

This section lists how X10 constructs are compiled to X10 IR. The section focuses on constructs that lack an obvious IR counterpart or that require other special considerations.

A.3.4.1 Clock

X10's `next` construct is modeled using a function call on `Clock`. `Clock`s only occur in the IR as possible inputs to ASYNCs.

A.3.4.2 Future

X10's `future`s are also mostly a runtime library construct and do not occur in the IR as such; however, ASYNC nodes distinguish between futures and normal asynchronous calls in that a boolean flag is set to indicate that the enclosing FINISH does not need to wait for the `future` to terminate.

A.3.4.3 `foreach` and `ateach`

The compiler implements `foreach` and `ateach` using a combination of the code corresponding to `for` and `async`. Note that we may want special nodes in the IR for these more heavyweight instructions in the future in order to make detection of the respective pattern easier for the optimizing compiler.

A.3.5 `Array`, `region`, `distribution` and `point`

`Array`, `region`, `distribution` and `point` are pure library constructs in this design (except for dependent typing, as described in Section A.3.7). As far as the runtime is concerned (after type checking), these classes live in `x10.lang` and receive no special treatment in the IR itself.

While the IR lacks specific constructs for these types, this does not preclude the VM from knowing details about these classes and their semantics and optimizing accordingly.

A.3.6 Runtime and runtime library interactions

The X10 runtime requires certain properties from the implementation of certain runtime classes in order to be able to execute certain IR constructs. The details of these APIs may depend on the specific runtime and are thus not part of the IR specification.

Specifically, for `async`, the implementation of `Clock` must provide an appropriate method for registering activities with the clock.

For `for`, the implementation of `region` must provide appropriate methods for iterating over the points of the region.

Constant values (like `int` and `String`) are expected to provide appropriate default constructors to enable the runtime the instantiation of these constants.

A.3.7 Region types

Region types represent constraints on the possible set of values for a particular base type. The IR represents these constraints using a restricted expression language. In order to avoid confusion with the types of X10, we use the term *kind* to describe the types of expressions in this constraint language. Section A.3.7.1 lists the kinds of expressions used in the constraint language. Section A.3.7.2 lists the various operations and gives the kinds of their inputs as well as the kind of expression that they produce. Finally, Section A.3.7.3 lists what kinds of expressions are used by the region types to constrain the values for a particular base type in X10.

A.3.7.1 Kinds

The dependent type information captured by region types uses the following kinds of expressions:

- INT
- POINT
- REGION
- PLACE
- OBJECT
- DISTRIBUTION
- ARRAY
- VALUEARRAY

A.3.7.2 Operations

The region type system uses the primitive LOCAL to represent values in the X10 IR. The kind of a LOCAL corresponds to the base type of the respective value. For example, for a method `m(region r, region<:r> s)` a LOCAL of kind REGION referencing the IR node of `r` would be used to represent the first argument in the region constraint imposed on the second argument `s`. Another special primitive ICONST is used to represent constant integer expressions. The kind of ICONSTs is always INT.

In addition to these two fundamental constructors, the constraints can use the following basic operations:

- PointConstruction: INT* $\rightarrow$ POINT
- RegionConstruction: POINT $\times$ POINT $\rightarrow$ REGION
- DistributionConstruction: REGION $\times$ PLACE $\rightarrow$ DISTRIBUTION
- ValueArrayConstruction: REGION $\rightarrow$ VALUEARRAY
- ArrayConstruction: DISTRIBUTION $\rightarrow$ ARRAY
- ObjectAtLocation: PLACE $\rightarrow$ OBJECT
- LocationOfObject: OBJECT $\rightarrow$ PLACE
- DistributionOfArray: ARRAY $\rightarrow$ DISTRIBUTION
- RegionOfDistribution: DISTRIBUTION $\rightarrow$ REGION
- RegionOfValueArray: VALUEARRAY $\rightarrow$ REGION
- PointInRegion: REGION $\rightarrow$ POINT
- RankOfPoint: POINT $\rightarrow$ INT
- RankOfRegion: REGION $\rightarrow$ INT
- RegionOfRank: INT $\rightarrow$ REGION
- DistributionOverRegion: REGION $\rightarrow$ DISTRIBUTION
- PointOfRank: INT $\rightarrow$ POINT
- ZeroPoint: INT $\rightarrow$ POINT

The basic operations are extended with an operational algebra. The current algebra contains the following expressions:

- IntAdd: INT $\times$ INT $\rightarrow$ INT
- IntSub: INT $\times$ INT $\rightarrow$ INT
- IntMax: INT $\times$ INT $\rightarrow$ INT
- IntMin: INT $\times$ INT $\rightarrow$ INT
- IntMul: INT $\times$ INT $\rightarrow$ INT
- PointAdd: POINT $\times$ POINT $\rightarrow$ POINT

- PointSub: POINT × POINT → POINT
- PointMax: POINT × POINT → POINT
- PointMin: POINT × POINT → POINT
- PointMul: POINT × POINT → POINT
- PointProjection: POINT × INT^* → POINT
- RegionUnion: REGION × REGION → REGION
- RegionIntersection: REGION × REGION → REGION
- RegionMulRegion: REGION × REGION → REGION
- RegionMove: REGION × POINT → REGION
- RegionMulDistribution: REGION × DISTRIBUTION → DISTRIBUTION
- RegionProjection: REGION × INT^* → REGION
- DistributionUnion: DISTRIBUTION × DISTRIBUTION → DISTRIBUTION
- DistributionIntersection: DISTRIBUTION×DISTRIBUTION → DISTRIBUTION
- DistributionMove: DISTRIBUTION × POINT → DISTRIBUTION
- DistributionRestrict: DISTRIBUTION × PLACE → REGION
- DistributionMul: DISTRIBUTION × REGION → DISTRIBUTION

The semantics of the constraints represented by these operations correspond to the semantics of the region operations described in Section 3.6.

A.3.7.3 Constraints

This section describes the kinds of dependent constraints that the region type system captures for the various base types.

int The dependent type information for an `int` captures a lower and an upper bound (both of kind INT).

point The dependent type information for a `point` captures a lower and an upper bound (both of kind POINT), the rank of the point (of kind INT) and a region which represents an upper bound on the set of points that the respective value may belong to.

region The dependent type information for a `region` captures a lower and an upper bound (both of kind REGION) that describe sets of points that the region must contain (lower) and may contain (upper). The dependent type information also describe the rank of the region (of kind INT).

place The dependent type information for a `place` is a set of place expressions that represent the same place.

Object The dependent type information for an `Object` is the set of place expressions that are equivalent to the location of the object.

ValueArray The dependent type information for a `ValueArray` is the same as that for a region.

distribution The dependent type information for a `distribution` captures a lower and an upper bound (both of kind DISTRIBUTION) that describe sets of points that the associated region must contain (lower) and may contain (upper) together with the respective places that these the region's points must be mapped to. The dependent type information also describes the rank of the underlying region (of kind INT).

Array The dependent type information for an `Array` is the same as that for a distribution.

A.4 The Abstract Interpretation Framework

Since [CC77], abstract interpretation is known as a common approach to many program analyses. In fact, it may be *the* most common idiom in the domain of optimizing compilers and program verification. While the theoretical construct of abstract interpretation is well understood, actual implementations of program analyses still use hand-crafted implementations specific to the problem at hand.

The goal of the abstract interpretation framework in XTC is to provide a general framework for the implementation and execution of any kind of abstract interpretation based analysis.

The quest for universal abstractions in building compilers is not new. The idea of automatic parser (and lexer) generation [LMB92] from a grammar has been widely adopted. A general optimizer that is useful for many languages and target platforms (as outlined in [BD94]) is at the heart of the popular GNU compiler collection. Developing good abstractions for the intermediate representation has also resulted in a wide spectrum of solutions, from bytecode [LY97, PBF03] to graphs [CP95] and even XML [Yot04]. Other researchers have investigated generalizations for specific categories of program analyses, such as call graph construction [GC01]. The design of the XTC abstract interpretation framework attacks various problems that arise when the idea of abstract interpretation is turned into a general system for program analysis.

A large set of applications depends on abstract interpretation. The XTC framework [Gro06] uses abstract interpretation for bytecode verification [LY97], control-flow analysis [Shi91], and inference of confined types [GPV01]. Other applications include data-flow analysis [RRL99], worst-case execution time analysis [BBP02, EES01] and shape analysis [SRW99]. An extensive list of known applications can be found in [Cou01].

The key issues for providing a general abstract interpretation mechanism all relate to various efficiency concerns. It is difficult to devise a mechanism that scales to large abstract interpretation problems. The range of problems spans generalizing the properties of abstract values, limiting memory consumption and scheduling of transfer functions. Current abstract interpreters use hand-optimized, domain specific heuristics to scale abstract interpretation to relatively small problem sizes. The goal of the thesis is to produce better solutions that perform well for any abstract interpretation problem while also reducing the amount of code the compiler-writer needs to write.

The rest of the section is structured as follows: after an introduction to abstract interpretation, the core of the framework is described, followed by a discussion of useful extensions to the framework.

A.4.1 Abstract Interpretation

Abstract interpretation is a general theory for statically predicting program behavior that is used in optimizing compilers and software engineering tools [CC77, CRC92]. The goal of abstract interpretation is to obtain information about all possible states of a program by simulating its execution. The set of all possible states is called the state-space. Since the state-space is generally too large to allow an exact simulation to terminate, abstract interpretation uses *approximations*. Considering that the state-space of a program can be represented using sets (for example, sets of values for variables) one fundamental aspect of the theory of abstract interpretation is the approximation of sets and set operations (see Section A.4.2).

Another concern for abstract interpretation is showing that the result is sound (that is, formally proving that the predicted program behavior is a safe approximation of the actual behavior (see section A.4.3)). In the presence of loops in the analyzed program, abstract interpretation must also approximate control-flow. The computation of loop invariants is done using iteration. Thus, another concern is making sure that the any such iteration terminates. This is achieved using fixpoint theory and monotonicity properties of the abstract interpretation function in a partially ordered set of bounded height (see Section A.4.4).

If the height of the partially ordered set is constant, the worst-case complexity of the fixpoint iteration is cubic. A program of size n can have $O(n)$ program points. In the worst case, the analysis may perform $O(n)$ statement evaluations before increasing the abstract value representing the state at one program point. These $O(n^2)$ operations may be needed $O(n)$ times, until the abstract value for each program point has reached its maximum. Hence, abstract interpretation is often rather slow and is thus possibly impractical for larger programs, resulting in the use of less precise alternatives of linear complexity for certain applications [BW94, PS94].

Note that the notion that fixpoint iteration is of cubic complexity in the size of the program disregards the specifics of the abstract domain chosen for the particular analysis. As illustrated, the height of the poset representing the abstract domain is another factor in the worst-case complexity. An abstract value poset that “merely” models all possible types has a worst-case height of $O(n)$, resulting in a total worst-case performance of $O(n^4)$ for the abstract interpretation.

A.4.2 Abstract Domain

In order to make static predictions possible, the infinite number of states that can occur in actual programs must be reduced. This is achieved by mapping the domain of actual values in the program to a finite *abstract domain.* The mapping function from the concrete to the abstract domain is called the *abstraction function.* Its inverse maps abstract values back to concrete values and is called the *concretization function.* The abstract domain is usually a *poset*, which is a special kind of *poset.*

Definition 1 (Poset) *A* partially ordered set *(poset) is a pair* $\langle D, \subseteq \rangle$ *where* D

is a set and $\subseteq$ is a reflexive, transitive and antisymmetric relation on D.

A function $f : D \rightarrow D$ is called *monotone* if for all $d, d' \in D$, $d \subseteq d'$ implies that $f(d) \subseteq f(d')$. A sequence $d_1, d_2, \dots$ of elements in D is called a *chain* if $d_i \subseteq d_{i+1}$. It is called *strictly increasing* if $d_i \neq d_{i+1}$. The *height* of the poset is the length of the longest strictly increasing chain. The poset is of *finite height* if it contains no infinite, strictly increasing chain.

For $X \subseteq D$, let $\bigcup X$ be the least upper bound of all elements $x \in D$, that is, $x \subseteq \bigcup X$ for all $x \in X$ and $\bigcup X \subseteq d$ for all $d \in D$ for which $x \subseteq d$ for all $x \in X$. Furthermore let $\bigcap X$ be the greatest lower bound of all elements $x \in D$, that is, $\bigcap X \subseteq x$ for all $x \in X$ and $d \subseteq \bigcap X$ for all $d \in D$ for which $d \subseteq x$ for all $x \in X$. A poset $\langle D, \subseteq \rangle$ is *complete* if for every $X \subseteq D$ both $\bigcup X$ and $\bigcap X$ exist.

An *abstract value* is an element of a complete poset of finite height. From now on the term `poset` will always refer to complete posets of finite height. Finite sets form a poset with the canonical definition of the set operations $\cup$ and $\cap$ yielding the least upper and greatest lower bounds. A common construction for abstract domains is the use of a product-poset (Lemma 7). Product posets are typically used to capture the values of multiple variables in one abstract value. For example, if a pair of variables is modeled by two posets, the corresponding product-poset is a precise model of both variables expressed now as only one abstract value. The construction generalizes to any (finite) number of variables.

Lemma 7 (Product-Poset) *Given posets $\langle P_1, \subseteq_1 \rangle$ and $\langle P_2, \subseteq_2 \rangle$ the* product-poset *$\langle P, \subseteq \rangle$ defined as*

$$P := P_1 \times P_2$$
$$(a_1, a_2) \subseteq (b_1, b_2) \Leftrightarrow (a_1 \subseteq_1 b1) \wedge (a_2 \subseteq_2 b_2)$$

is again a poset.

The specific choice for the abstract domain depends on the application, i.e. the information to be obtained about program behavior. The precision of the analysis is largely determined by the properties modeled by the abstract domain, but naturally a richer abstraction is usually more costly. As we will see, the previously mentioned requirement for the abstract domain to be *finite* is a necessary (but not sufficient) condition to ensure that the abstract interpretation will terminate.

The correctness of the results obtained using abstract interpretation relies on the concretization function to yield a conservative estimate of the actual values for any concrete execution. Proving the correctness of an abstract interpretation is generally done by showing that the abstract semantics used in the abstract interpreter "correspond" to the concrete semantics of the target language. For this, the correspondence between concrete and abstract domain is described as a *Galois connection*:

Definition 2 (Galois connection) *Let A and B be partially ordered sets under the relations $\subseteq_A$ and $\subseteq_B$ respectively. Then the pair of functions (f, g) with $f : A \rightarrow B$ and $g : B \rightarrow A$ is a* Galois connection *of A and B if and only if for all $a \in A$ and $b \in B$*

$$f(a) \subseteq_B b \quad \equiv \quad a \subseteq_A g(b) \tag{A.1}$$

f is called the lower adjoint *and g is the* upper adjoint*.*

The abstraction function, mapping from the concrete to the abstract domain, corresponds to the lower adjoint. The upper adjoint is the concretization function, mapping results from the abstract domain back to concrete values. Important properties of Galois connections are that the lower adjoint uniquely defines the upper adjoint, that lower and upper adjoint are duals, and that for each supremum-preserving monotone function between two partially ordered sets, there exists a Galois connection for which that function is the lower adjoint. An extensive introduction to Galois connections can be found in [Bac02].

A.4.3 Abstract Semantics

Given an abstract domain, an *abstract interpreter* can be obtained by writing for each language construct an abstract evaluation rule that performs an equivalent operation on abstract values. The abstract interpreter is called *sound* if for any evaluation of the concrete program, each expression can only yield concrete values that are included in the set of possibilities obtained by applying the concretization function to the abstract values corresponding to the expression. More formally, let γ be the concretization function and $[\![\cdot]\!]$ the evaluation of a statement according to the concrete semantics of the language. Let σ describe the abstract semantics of the abstract interpreter in the form of a function that maps expressions in the concrete language to the resulting abstract values in the abstract domain. Then soundness of the abstract interpreter can be stated compactly as

$$[\![e]\!] \in \gamma(\sigma(e)) \qquad \text{for all expressions } e. \tag{A.2}$$

The soundness of an abstract interpreter can be proven easily by showing the soundness of the given abstract semantics for each language construct. Suppose the abstract semantics for some operation "$\circ$" of the concrete language are defined as $\sigma(e_1 \circ e_2) = \sigma(e_1) \bullet \sigma(e_2)$ where "$\bullet$" is the corresponding operation in the abstract domain. This definition is sound if for all abstract values a_1 and a_2

$$\{n_1 \circ n_2 | (n_1, n_2) \in \gamma(a_1) \times \gamma(a_2)\} \subseteq \gamma(a_1 \bullet a_2).$$

This realization can be used to derive an abstract interpreter from a concrete interpreter by showing this correspondence for each operation. The ($\bullet$-)functions in the abstract interpreter that correspond to the concrete ($\circ$-)operations are called transfer functions.

Definition 3 (Transfer function) *Let P be a poset. A* transfer function *is a monotone function of the form $f : P \to P$.*

Given the (abstract) state of the program before the operation, a transfer function computes the next (abstract) state of the program. This approach of constructing an abstract interpreter is quite natural in that it only requires the compiler writer to implement a set of transfer functions, all of which can be derived as a straightforward abstraction from functions handling concrete statements in an interpreter for the analyzed language [CHY95]. Note that the input to (and the result of) the transfer functions is a poset that captures the entire state of the application and not just the smaller set of values that this particular operation is concerned with. The next section describes why transfer functions must be monotone and how this is achieved.

A.4.4 Termination and Fixpoint Theory

In the presence of loops, the straightforward implementation of a sound abstract interpreter from a concrete interpreter as described so far does not result in an algorithm that terminates. In order to guarantee termination in the presence of loops, the abstract interpretation must abstract control flow such that the abstract execution of loops in the analyzed application is guaranteed to complete. This is done by expressing the process as a *fixpoint iteration.*

Definition 4 (Fixpoint) *An element $d \in D$ is called a* fixpoint *of a function $f : D \to D$ if $f(d) = d$. If D is a poset, a fixpoint $d \in D$ is called the* least fixpoint *if $d \subseteq d'$ for all fixpoints $d' \in D$.*

A fixpoint iteration of a monotone function $f : D \to D$ is the process of computing the chain of values $f_i = f(f_{i-1})$ starting at some initial value f_0. The result of a fixpoint iteration is the minimum value f_∞ for which $f_i \subseteq f_\infty$ for all *i.* f_∞ can be computed by fixpoint iteration for all monotone functions f if the poset D is of finite height.

The abstract interpreter can be described as a finite set of functions f that each take the abstract state of the application and compute the next state. Thus, it is possible to express the result of abstract interpretation as the computation of this common fixpoint. For a finite set $\mathcal{F}$ of functions of the form $f : D \to D$, an element $d \in D$ is called a *common fixpoint* of $\mathcal{F}$ if d is a fixpoint of every function in the set. The least common fixpoint can be computed according to the following theorem:

Theorem A.4.1 *Let μf be the least fixpoint of f. Let $\langle x :: f(x)\rangle$ describe a monotone function that maps an element x of a poset to another poset element $f(x)$. Then*

$$(\mu\langle x :: x \odot p.x\rangle, \mu\langle y :: q.y \otimes y\rangle) = \mu\langle x, y :: (x \odot y, x \otimes y)\rangle \tag{A.3}$$

where $p.x = \mu\langle v :: x \otimes v\rangle$ and $q.y = \mu\langle u :: u \odot y\rangle$.

In other words, the mutually recursive fixpoint of the function $\langle x, y :: (x \odot y, x \otimes y)\rangle$ can be computed using the individual least fixpoints $p.x$ and $q.x$. A formal proof of this theorem can be found in [Bac02, Theorem 106]. Stated as an algorithm, the theorem guarantees that by iteratively computing the fixpoints $p.x$ and $q.y$ (that is, $p.q.p.q.p.q.(\ldots).\bot$) until neither value changes, one can compute the fixpoint $\mu\langle x, y :: (x \odot y, x \otimes y)\rangle$. By repeated application the theorem generalizes to the composition of any finite number of mutually recursive transfer functions.

Definition 5 (Chaotic iteration) *Let $\mathcal{F}$ be a finite set of monotone functions over a poset. A* chaotic iteration *is any sequence of functions $f_i \in \mathcal{F}$.*

A given element $d_0 \in D$ induces a sequence of values $d_i = f_i(d_{i-1}) \in D$. If the transfer functions commute, [CC79] guarantees that any chaotic fixpoint iteration will always produce a least fixpoint within a finite number of iterations. While any chaotic fixpoint iteration will eventually terminate with the global fixpoint, there typically exist specific orderings that produce the global fixpoint faster. Determining a good order of execution for the transfer functions is one of the primary goals in the implementation (see Section A.4.6).

Given the finite height of the poset representing the program state, fixpoint theory guarantees that a fixpoint iteration terminates if all transfer functions $f \in \mathcal{F}$ are *monotone*. In the case of transfer functions from an abstract interpreter, monotonicity of f can be achieved by defining $f(x) := x \cup f'(x)$ where

```
class Engine {
    Engine(EntailmentGraph eg);
    void add(TransferFunction tf, AbstractValue arg);
    void run();
}
```

Figure A.35: The `Engine` API (simplified).

$a \cup b$ is the computation of the least upper bound in the product-poset and f' is a function that models the abstract semantics of the analyzed application without monotonicity considerations. By applying the $\cup$ operation, the result is guaranteed to be monotonically increasing toward more conservative approximations and therefore achieving monotonicity while preserving soundness. The global fixpoint can then be computed by iterating over the (now monotone) transfer functions until a fixpoint has been reached [Yi93].

Just as selecting the appropriate order of transfer functions is important for performance, the monotonicity constraint also allows for optimizations. Sometimes it is possible to guarantee monotonicity without performing the $\cup$ operation (which is potentially costly in terms of both precision and time) after every step. Determining at which points in the iteration it is necessary to "merge" is another important goal for the implementation.

A.4.5 API

For clients, the abstract interpretation framework presents a relatively simple API. The client starts the abstract interpreter by calling `add` on `Engine` (Figure A.35) providing an initial `TransferFunction` (Figure A.36) and `AbstractValue` (Figure A.37). The abstract interpretation is then started with the `run()` method of `Engine`. The transfer function implementation provides a function that given an abstract value computes a (possibly empty) set of successor transfer functions and abstract values, which it also `add`s to the `Engine`. The `run` method terminates once the fixpoint has been reached: all transfer functions have been evaluated with the most conservative abstract value in the abstract domain that could be `add`ed for the respective transfer function.

Note that due to implementation limitations creating the `Engine` requires supplying an *entailment graph*, which is be discussed in more detail in Section A.4.7.

```
interface TransferFunction {
    void run(AbstractValue arg);
}
```

Figure A.36: The `TransferFunction` interface (slightly simplified).

```
interface AbstractValue {
    boolean includes(AbstractValue v);
    AbstractValue merge(AbstractValue e);
}
```

Figure A.37: The `AbstractValue` interface (slightly simplified).

A.4.6 Implementation

The major concern for the implementation is to *minimize the amount of state* that needs to be kept for the fixpoint iteration in addition to minimizing the number of transfer function evaluations. Minimizing the amount of state improves performance in many ways, from reduced pressure on the memory management subsystem to fewer branches for versioned abstract values. Previous designs for abstract interpretation have not focused on this aspect, instead computing the global fixpoint. As discussed in section A.4.1, the system described in this section decomposes the common (global) fixpoint into the smaller inputs to the individual transfer functions. Consequently, it is not necessary to have the entire global fixpoint represented in memory at any given time. The analysis is assumed to process (or record) the necessary information while the fixpoint iteration is still ongoing.

Since abstract interpretation is at the heart a fixpoint iteration, any random (chaotic) execution order for transfer functions is guaranteed to yield correct results. Still, most such chaotic orders are not going to result in a minimal number of transfer function evaluations. The number of evaluations can be systematically improved by carefully selecting the evaluation order in accordance with data-dependencies between the transfer functions. The information about which transfer function may change the abstract value inputs of which other transfer functions can be captured in an *entailment graph* [CHY95]. That information can then be used to optimize the evaluation order.

Using Tarjan's algorithm [Tar72], the entailment graph can be split into strongly connected components (SCCs). The SCCs can then be totally ordered such that transfer functions never depend on inputs created (or modified) by transfer functions in an SCC of greater value in the total ordering. The total ordering of the SCCs gives a natural ordering of the transfer functions that minimizes the number of evaluations (disregarding cycles within the SCCs). An SCC that contains a transfer function that has not yet been evaluated with the latest available abstract value input is called active. Given a transfer function evaluation order that always evaluates transfer functions belonging to the SCC with the minimal ordering, all abstract values belonging to transfer functions in an SCC that becomes inactive can be freed since the ordering guarantees that those transfer functions will not be evaluated in the future. This limits the amount of state which must be kept by the fixpoint iteration to abstract values of transfer functions in the currently active SCC and to at most one abstract value for each `add`ed transfer function in other SCCs.

A.4.6.1 Breaking Cycles

In the absence of cycles in the entailment graph, the approach described so far minimizes the number of transfer function evaluations and the amount of state that needs to be kept. In the presence of cycles, at least one abstract value must be kept for a transfer function in every cycle in order to guarantee the eventual termination of the fixpoint iteration. The reason for this is that for a cycle of length n, the abstract value must be changed every n evaluations; otherwise the $\subseteq$ test performed by `add` holds. Since the change is required to be strictly monotonically increasing in a poset of finite height, the iteration will eventually terminate.

The goal of minimizing the amount of state that needs to be kept could be reached by selecting a minimal set of transfer functions in the entailment graph to break all cycles. For the selected transfer functions, the fixpoint iterations would have to keep the "last" abstract value around in order to ensure termination. The problem with this selection is that it both requires both a rather costly process of finding all cycles and conflicts with the goal of minimizing the number of transfer function evaluations. This is because the number of transfer function evaluations would be minimized if every transfer function in the SCC was selected. Otherwise, it is possible that $n-1$ redundant transfer function evaluations could occur in a cycle of length n (because it can take up to $n-1$ steps until the $\subseteq$ test succeeds).

Consequently, it often makes sense to use a heuristic to strike a balance between the amount of state that is kept and the number of transfer function evaluations. An exception to this rule are large SCCs with a significant number of cycles where the amount of state that might need to be kept around becomes more critical than the execution time of transfer functions and the selection algorithm. In either case, the selection algorithm must guarantee that at least one transfer function is selected in every cycle. In the case of the heuristic, the process should be inexpensive (i.e., linear in the size of the SCC) and reasonably reduce the number of transfer functions selected. Without prior knowledge about the relative cost of transfer function evaluations to keeping additional state, it is not possible to determine the optimal trade-off, leaving the cost of the heuristic itself as the only criteria that always matters.

The heuristic used by XTC proceeds by first selecting nodes (transfer functions) that directly refer to themselves (since this is the only way to eliminate these cycles). Then nodes X of the form $A \to X \to C$ where $A \neq X \neq C$ are removed from the entailment graph by reducing the cycle to the smaller graph $A \to C$. Furthermore, nodes X of the forms $A \to X \to \{B, C\}$ and $\{A, B\} \to X \to C$ are reduced to $A \to \{B, C\}$ and $\{A, B\} \to C$ respectively. After this, a remaining node of maximum degree is selected. Note that each time a node is selected the adjacent edges are removed, allowing for more applications of the graph reduction rules. The selection process is iterated until no nodes remain.

A.4.6.2 Intra-SCC Ordering of Transfer Functions

The heuristics described so far only order the SCCs of the entailment graph and describe where to keep state in order to break cycles. This is insufficient for a general abstract interpretation abstraction where SCCs can become rather large and a simple chaotic fixpoint iteration per SCC does not scale. For example, given an entailment graph like the one shown in Figure A.38, a chaotic fixpoint iteration may take an exponential number of evaluations. Since such cycles actually occur in practice for intraprocedural analyses, the abstract interpretation runtime must also determine an appropriate ordering for the transfer functions within an SCC.

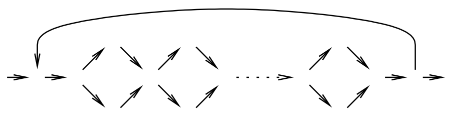

Figure A.38: Example of an entailment graph with a loop where the order of evaluation of the transfer functions within the same SCC matters.

The key idea for the algorithm is to split the SCC into subgraphs at the transfer functions selected to break the cycles. The remaining subgraphs can then be totally ordered. The merge points themselves are all assigned the “maximum” ordering value within the SCC. The resulting fixpoint iteration alternates between running the shortest possible sequence of transfer functions that are not selected such that only selected transfer functions are pending, and running one of the selected transfer functions. The resulting worst-case complexity of the fixpoint iteration is $O(n \cdot h)$ where n is the number of nodes in the SCC and h is the maximum height of the abstract domain. Figure A.39 shows a possible ordering for the example that could be obtained using this algorithm.

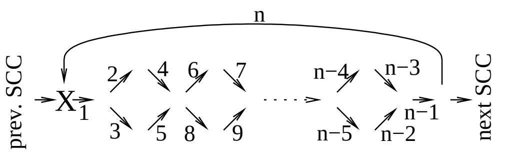

Figure A.39: Example for an ordering on the SCC that will guarantee linear execution time. The selected merge point is marked with an X. Note that this is just one possible selection and ordering and not the only possibility.

A.4.7 Obtaining the Entailment Graph

Until now, nothing has been said about how the abstract interpretation runtime obtains the entailment graph which is used to optimize the evaluation order of the transfer functions and to decide at which points state needs to be kept. An entailment graph is correct if it captures which transfer functions may `add` which other transfer functions. By definition, one extreme but correct choice for the entailment graph would be a clique. The result of using a clique would be an abstract interpreter that must keep state for every transfer function (and thus explicitly compute the global fixpoint) and which runs transfer functions in a chaotic order – the clique cannot help guide the ordering. The other extreme is the minimal entailment graph, which is the minimal graph that is correct with respect to the actual interpretation of the transfer functions for the particular order implied by the minimal entailment graph. Clearly, neither extreme is desirable since the clique does not allow optimization of state or transfer function evaluations and the minimal entailment graph cannot be obtained efficiently (in part due to its recursive definition, but primarily because it requires a precise evaluation of the actual transfer functions for its construction).

The current framework requires the client to explicitly supply a correct entailment graph. This can always be done as a straightforward simplification of the transfer function implementation that abstracts away the abstract values. The resulting entailment graph construction is equivalent to the data-flow insensitive construction of a control-flow graph.

In addition to the entailment graph, users of the framework must implement the abstract values (possibly re-using existing implementations provided by the framework) and the actual transfer functions. Given those three implementations and an initial pair of transfer function and starting state, the main `Engine` of the abstract interpretation framework can be `run`, resulting in the computation of the fixpoint. Various flags in the `Engine` can be used to supervise the abstract interpretation process. When activated, these instrumentations slow down the abstract interpreter, but they can help detect problems such as non-monotone definitions of the abstract domain and mismatches between the entailment graph and the transfer functions.

A.4.8 Future Work

The fact that the entailment graph must be provided by hand at the moment is a major drawback in the sense that the implementation that computes the entailment graph duplicates logic that is also present in the transfer functions themselves. It would thus be desirable to extend the framework with introspective capabilities that can deduce the entailment graph automatically from the transfer functions. This would further reduce the amount of code that a programmer needs to write and also eliminate a significant source of defects.

Note however, that the existing framework provides reasonably general implementations for entailment graphs for the Java and X10 IRs that can be easily re-used for analyses that essentially follow the ordinary intraprocedural control flow of methods.

A.5 License for XTC-X10

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long

as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder

APPENDIX B

ArrayBench Benchmarks

This chapter gives the source code for some of the more interesting ArrayBench benchmarks. The source code for all ArrayBench benchmarks is available at `http://grothoff.org/christian/xtc/x10/`. We also give the X10 version of the example programs from Section 3.3.

B.1 Example Programs

Note that in order to see the example programs without region types, the reader should simply ignore the dependent type annotations given between "<" and ">".[1]

B.1.1 init

```
void init(Array<int> a) {
  for (p : a.dist.reg) {
    finish async(a.dist[p]) {
      a[p] = 1;
    }
  }
}
```

B.1.2 partialinit

```
void partialinit(place h, Array<int> a) {
  finish async(h) {
    for (p : a.dist \% h) {
      a[p] = 1;
    }
  }
```

[1]The type parameter "int" for `Array` is not part of the region type annotations.

```
  }
```

B.1.3 copy

```
  void copy(distribution d, Array<int;d:d> a, Array<int;d:d> b) {
    finish ateach (h : distribution.unique()) {
      for (p : a.dist \% here) {
        a[p] = b[p];
      }
    }
  }
```

B.1.4 expand

```
  Array<int> expand(Array<int> a, region<;a.dist.reg> x) {
    . b = new Array<int>(distribution.blocked(x), 0);
    finish ateach (h : distribution.unique()) {
      for (p : a.dist.reg & b.dist \% here) {
        b[p] = future(a.dist[p]) { a[p] }.force();
      }
    }
    return b;
  }
```

B.1.5 shiftleft

```
  void shiftleft(Array<int#1> a) {
    . inner = (a.dist.reg + 1p) & a.dist.reg;
    for (p : inner) {
      finish async(a.dist[p]) {
        a[p] = future(a.dist[p-1p]) { a[p-1p] }.force();
      }
    }
  }
```

B.2 Crypt

```
void transcode(int id,
               region R,
               Array<byte;(R*here):(R*here)#1> text1,
               Array<byte;(R*here):(R*here)#1> text2,
               Array<int;([0:51]*here):([0:51]*here)#1> key) {
  int tslice = text1.dist.reg.size() / 8;
  int ttslice = (tslice + place.count()-1) / place.count();
  int slice = ttslice*8;
  int ilow = id*slice;
  var int iupper = (id+1)*slice;
  if (iupper > text1.dist.reg.size())
  iupper = text1.dist.reg.size();
  var int i1 = ilow;
  var int i2 = ilow;
  for (p[i8] : [ilow/8:iupper/8-1]) {
    var int ik = 0;
    var int r = 8;
    var int x1 = getAt(R, text1, i1++) & 0xff;
    x1 |= (getAt(R, text1, i1++) & 0xff) << 8;
    var int x2 = getAt(R, text1, i1++) & 0xff;
    x2 |= (getAt(R, text1, i1++) & 0xff) << 8;
    var int x3 = getAt(R, text1, i1++) & 0xff;
    x3 |= (getAt(R, text1, i1++) & 0xff) << 8;
    var int x4 = getAt(R, text1, i1++) & 0xff;
    x4 |= (getAt(R, text1, i1++) & 0xff) << 8;
    do {
        x1 = new int(new long(x1) * kgetAt(key, ik++)
                     % 0x10001L & 0xffffL);
        x2 = x2 + igetAt(key, ik++) & 0xffff;
        x3 = x3 + igetAt(key, ik++) & 0xffff;
        x4 = new int(new long(x4) * kgetAt(key, ik++)
                     % 0x10001L & 0xffffL);
        var int t2 = x1 ^ x3;
        t2 = new int(new long(t2) * kgetAt(key, ik++)
                     % 0x10001L & 0xffffL);
        var int t1 = t2 + (x2 ^ x4) & 0xffff;
        t1 = new int(new long(t1) * kgetAt(key, ik++)
                     % 0x10001L & 0xffffL);
        t2 = t1 + t2 & 0xffff;
        x1 ^= t1;
        x4 ^= t2;
        t2 ^= x2;
        x2 = x3 ^ t1;
```

```
        x3 = t2;
    } while(--r != 0);  // Repeats seven more rounds.
    x1 = new int(new long(x1) * kgetAt(key, ik++)
                    % 0x10001L & 0xffffL);
    x3 = x3 + igetAt(key, ik++) & 0xffff;
    x2 = x2 + igetAt(key, ik++) & 0xffff;
    x4 = new int(new long(x4) * kgetAt(key, ik++)
                    % 0x10001L & 0xffffL);
    setAt(R, text2, i2++, x1);
    setAt(R, text2, i2++, x1 >>> 8);
    setAt(R, text2, i2++, x3);
    setAt(R, text2, i2++, x3 >>> 8);
    setAt(R, text2, i2++, x2);
    setAt(R, text2, i2++, x2 >>> 8);
    setAt(R, text2, i2++, x4);
    setAt(R, text2, i2++, x4 >>> 8);
}

void setAt(region R,
           Array<byte;(R*here)> text2,
           int idx,
           int val) {
  text2[(point<:R>) new point(idx)] = new byte(val);
}

long kgetAt(Array<int;([0:51]*here):([0:51]*here)#1> key,
            int idx) {
  return new long(key[(point<:([0:51])>) new point(idx)]);
}

int igetAt(Array<int;([0:51]*here):([0:51]*here)#1> key,
           int idx) {
  return key[(point<:([0:51])>) new point(idx)];
}

static int getAt(region R,
                 Array<byte;(R*here)> text1,
                 int idx) {
  return new int(text1[(point<:R>) new point(idx)]);
}
```

```
public class IdeaContext {

  final ValueArray<short;([0:7]):([0:7])#1> userkey;

  final Array<int;([0:51]*here):([0:51]*here)#1> Z;

  final Array<int;([0:51]*here):([0:51]*here)#1> DK;

  IdeaContext() {
    Z  = new Array<int>([0:51], 0);
    DK = new Array<int>([0:51], 0);
    Random rndnum = new Random(136506717L);
    userkey = new ValueArray<short>([0:7],
                                function short(point p) {
        return new short(rndnum.nextInt());
      });
    calcEncryptKey();
    calcDecryptKey();
  }

  void calcEncryptKey() {
    for (p : [0:7])
      Z[p] = new int(userkey[p]) & 0xffff;
    for (p[i] : [8:51]) {
      int j = i % 8;
      if (j < 6) {
        Z[p] = ((Z[p - 7p]>>>9) | (Z[p - 6p]<<7)) & 0xFFFF;
        continue;
      }
      if (j == 6) {
        Z[p] = ((Z[p - 7p]>>>9)
             | (Z[(point<:([0:51])>) (p - 14p)]<<7)) & 0xFFFF;
        continue;
      }
      assert j == 7;
      Z[p] = ((Z[(point<:([0:51])>) (p - 15p)]>>>9)
             | (Z[(point<:([0:51])>) (p - 14p)]<<7)) & 0xFFFF;
    }
  }

  void calcDecryptKey() {
```

```
  var int t1 = IdeaContext.inv(Z[0p]);
  var int t2 = - Z[1p] & 0xffff;
  var int t3 = - Z[2p] & 0xffff;
  DK[51p] = IdeaContext.inv(Z[3p]);
  DK[50p] = t3;
  DK[49p] = t2;
  DK[48p] = t1;
  var int j = 47;
  var int k = 4;
  for (p : [0:6]) {
    t1 = gZ(k++);
    sDK(j--, gZ(k++));
    sDK(j--, t1);
    t1 = IdeaContext.inv(gZ(k++));
    t2 = -gZ(k++) & 0xffff;
    t3 = -gZ(k++) & 0xffff;
    sDK(j--, IdeaContext.inv(gZ(k++)));
    sDK(j--, t2);
    sDK(j--, t3);
    sDK(j--, t1);
  }
  t1 = gZ(k++);
  sDK(j--, gZ(k++));
  sDK(j--, t1);
  t1 = IdeaContext.inv(gZ(k++));
  t2 = -gZ(k++) & 0xffff;
  t3 = -gZ(k++) & 0xffff;
  sDK(j--, IdeaContext.inv(gZ(k++)));
  sDK(j--, t3);
  sDK(j--, t2);
  sDK(j--, t1);
}

private void sDK(int idx, int val) {
    DK[(point<:([0:51])>) new point(idx)] = val;
}

private int gZ(int idx) {
 return Z[(point<:([0:51])>) new point(idx)];
}
```

```
  private static int inv(int x) {
    if (x <= 1)
      return x;
    var int t1 = 0x10001 / x;
    var int y = 0x10001 % x;
    if (y == 1)
      return (1 - t1) & 0xFFFF;
    var int t0 = 1;
    var int xu = x;
    do {
      var int q = xu / y;
      xu = xu % y;
      t0 += q * t1;
      if (xu == 1)
        return t0;
      q = y / xu;
      y = y % xu;
      t1 += q * t0;
    } while (y != 1);
    return (1 - t1) & 0xFFFF;
  }
}
```

B.3 Crypt-2

```
void transcode(int id,
               region<#1> R,
               Array<byte;(R*here):(R*here)#1> text1,
               Array<byte;(R*here):(R*here)#1> text2,
               Array<int;([0:51]*here):([0:51]*here)#1> key) {
  int tslice = text1.dist.reg.size() / 8;
  int ttslice = (tslice + place.count()-1) / place.count();
  int slice = ttslice*8;
  int ilow = id*slice;
  var int iupper = (id+1)*slice;
  if (iupper > text1.dist.reg.size())
  iupper = text1.dist.reg.size();
  . rk = [0:51];
  . ir = [ilow:iupper-1] & R;
  . i1 = ir.iterator();
. i2 = ir.iterator();
for (p[i8] : [ilow/8:iupper/8-1]) {
  . ik = rk.iterator();
  var int x1 = new int(text1[i1.next()]) & 0xff;
  x1 |= (new int(text1[i1.next()]) & 0xff) << 8;
  var int x2 = new int(text1[i1.next()]) & 0xff;
  x2 |= (new int(text1[i1.next()]) & 0xff) << 8;
  var int x3 = new int(text1[i1.next()]) & 0xff;
  x3 |= (new int(text1[i1.next()]) & 0xff) << 8;
  var int x4 = new int(text1[i1.next()]) & 0xff;
  x4 |= (new int(text1[i1.next()]) & 0xff) << 8;
  for (r : [0:7]) {
    x1 = new int(new long(x1) * new long(key[ik.next()])
                 % 0x10001L & 0xffffL);
    x2 = x2 + key[ik.next()] & 0xffff;
    x3 = x3 + key[ik.next()] & 0xffff;
    x4 = new int(new long(x4) * new long(key[ik.next()])
                 % 0x10001L & 0xffffL);
    var int t2 = x1 ^ x3;
    t2 = new int(new long(t2) * new long(key[ik.next()])
                 % 0x10001L & 0xffffL);
    var int t1 = t2 + (x2 ^ x4) & 0xffff;
    t1 = new int(new long(t1) * new long(key[ik.next()])
                 % 0x10001L & 0xffffL);
    t2 = t1 + t2 & 0xffff;
    x1 ^= t1;
    x4 ^= t2;
    t2 ^= x2;
    x2 = x3 ^ t1;
    x3 = t2;
  }
  x1 = new int(new long(x1) * new long(key[ik.next()])
               % 0x10001L & 0xffffL);
  x3 = x3 + key[ik.next()] & 0xffff;
  x2 = x2 + key[ik.next()] & 0xffff;
  x4 = new int(new long(x4) * new long(key[ik.next()])
               % 0x10001L & 0xffffL);
  text2[i2.next()] = new byte(x1);
  text2[i2.next()] = new byte(x1 >>> 8);
  text2[i2.next()] = new byte(x3);
  text2[i2.next()] = new byte(x3 >>> 8);
```

```
    text2[i2.next()] = new byte(x2);
    text2[i2.next()] = new byte(x2 >>> 8);
    text2[i2.next()] = new byte(x4);
    text2[i2.next()] = new byte(x4 >>> 8);
  }
}

public class IdeaContext {

 final ValueArray<short;([0:7]):([0:7])#1> userkey;

 final Array<int;([0:51]*here):([0:51]*here)#1> Z;

 final Array<int;([0:51]*here):([0:51]*here)#1> DK;

 IdeaContext() {
   Z  = new Array<int>([0:51], 0);
   DK = new Array<int>([0:51], 0);
   Random rndnum = new Random(136506717L);
   userkey = new ValueArray<short>([0:7],
                                   function short(point p) {
       return new short(rndnum.nextInt());
     });
   calcEncryptKey();
   calcDecryptKey();
 }

 private void calcEncryptKey() {
   for (p : [0:7])
     Z[p] = new int(userkey[p]) & 0xffff;
   for (p[i] : [8:51]) {
     int j = i % 8;
     if (j < 6) {
       Z[p] = ((Z[p - 7p]>>>9) | (Z[p - 6p]<<7)) & 0xFFFF;
       continue;
     }
     if (j == 6) {
       Z[p] = ((Z[p - 7p]>>>9)
            | (Z[(point<:([0:51])>) (p - 14p)]<<7)) & 0xFFFF;
       continue;
     }
    assert j == 7;
    Z[p] = ((Z[(point<:([0:51])>) (p - 15p)]>>>9)
           | (Z[(point<:([0:51])>) (p - 14p)]<<7)) & 0xFFFF;
  }
}

private void calcDecryptKey() {
   . r = [0:51];
   . k = r.iterator();
   . j = r.reverse_iterator();
   var int t1 = IdeaContext.inv(Z[k.next()]);
   var int t2 = - Z[k.next()] & 0xffff;
   var int t3 = - Z[k.next()] & 0xffff;
   DK[j.next()] = IdeaContext.inv(Z[k.next()]);
   DK[j.next()] = t3;
   DK[j.next()] = t2;
   DK[j.next()] = t1;
   for (p : [0:6]) {
     t1 = Z[k.next()];
     DK[j.next()] = Z[k.next()];
     DK[j.next()] = t1;
     t1 = IdeaContext.inv(Z[k.next()]);
     t2 = -Z[k.next()] & 0xffff;
     t3 = -Z[k.next()] & 0xffff;
     DK[j.next()] = IdeaContext.inv(Z[k.next()]);
     DK[j.next()] = t2;
     DK[j.next()] = t3;
     DK[j.next()] = t1;
   }
   t1 = Z[k.next()];
   DK[j.next()] = Z[k.next()];
   DK[j.next()] = t1;
   t1 = IdeaContext.inv(Z[k.next()]);
   t2 = -Z[k.next()] & 0xffff;
   t3 = -Z[k.next()] & 0xffff;
   DK[j.next()] = IdeaContext.inv(Z[k.next()]);
   DK[j.next()] = t3;
   DK[j.next()] = t2;
   DK[j.next()] = t1;
 }
```

```
  private static int inv(int x) {
    if (x <= 1)
      return x;
    var int t1 = 0x10001 / x;
    var int y = 0x10001 % x;
    if (y == 1)
      return (1 - t1) & 0xFFFF;
    var int t0 = 1;
    var int xu = x;
    do {
      var int q = xu / y;
      xu = xu % y;
      t0 += q * t1;
      if (xu == 1)
        return t0;
      q = y / xu;
      y = y % xu;
      t1 += q * t0;
    } while (y != 1);
    return (1 - t1) & 0xFFFF;
  }
}
```

B.4 DSOR

```
void sor(distribution<#1> dist,
         region<#1> seq,
         double omega,
         Array<double;(dist * seq)#2> G,
         int num_iterations) {
  . iseq = (seq + 1p) & (seq - 1p) & seq;
  . idis = (dist.reg + 1p) & (dist.reg - 1p) & dist.reg;
  double omega_over_four = omega * 0.25d;
  double one_minus_omega = double.ONE - omega;
  for (iter[off] : [1:num_iterations*2]) {
    int om2 = off % 2;
    finish for (dp[i] : idis) async (dist[dp]) {
      if (i % 2 == om2) {
        for (sp : iseq) {
          . ij = dp * sp;
          G[ij] = omega_over_four * (G[dp * (sp + 1p)] +
                                     G[dp * (sp - 1p)] +
                                     rget(G, (dp + 1p) * sp) +
                                     rget(G, (dp - 1p) * sp))
                + one_minus_omega * G[ij];
        }
      }
    }
  }
}

double rget(Array<double> G,
            point<:G.dist.reg> pt) {
  return future (G.dist[pt]) { G[pt] }.force();
}
```

Index

n-package-scoped, 47
0-CFA, 5, 9, 12

ABCD algorithm, 129
abstract domain, 144, 176
abstract interpretation, 175
abstract semantics, 178
abstract value, 177
access modifier, 44
activities, 73
aggregate object, 74
aliasing, 4, 16
ANCHOR, 161
annotation, 18, 53
anonymity, 21, 22
anonymity rule, 22
anonymous inner class, 43
API change rule, 116
app, 10
arity, 128
array, 75
array subtyping, 4
AST, 147
ASYNC, 164
async, 74, 170
ateach, 170
ATOMIC, 166

bytecode, 143

C++, 2
C#, 2
CALL, 161
CAST, 158
CHA, 5, 8
chaotic iteration, 180
Chapel, 127, 128
clock, 73, 169
cluster, 70
Co-Array Fortran, 127, 128
code manipulation, 147
coinduction, 5
commutativity, 111
conditional atomic sections, 73
confinable, 45
confined types, 17
confinement rules, 24
constraint, 26
constraint solving, 15
containers, 50
contravariance, 3
covariance, 3
Cursor, 147

demesnes, 128
dependent type, 72
dependent types, 128
distributed arrays, 75
distribution, 71, 73, 75, 111, 128
  access, 112
  construction, 111
  intersection, 111
  move, 113
  multiplication, 113
  restriction, 111
  union, 111
DLCW, 10
DLOW, 10
domain, 128
dynamic code generation, 136
dynamic loading, 7

EJB, 26
encapsulation, 14, 16
END, 164
entailment, 90
entailment graph, 182, 185
ENTER, 159
evolution, 48
extensibility, 117, 135

extensibility problem, 136

FCALL, 161
featherweight Java, 18
final, 117
FINISH, 163
finish-context, 163
fixpoint, 179
flyweight pattern, 143, 146
FOR, 165
for, 78, 170
foreach, 78, 170
Fortress, 127, 128
Freenet, 56
future, 170

Galois connection, 178
generic-confinable, 45
generics, 3, 18, 43
GET, 160

hidden widening, 21
hierarchical array, 78
HPC, 69

IF, 158
image, 128
immutable, 128
index, 75, 128
induction, 5
inference, 14
inlining, 8
intersection, 75
invariance, 3
INVOKEVM, 156
iteration, 75

JAMIT, 44
Java, 2

kind, 171

lambda calculus, 72
lib, 10
liveness, 4
local variable, 117
locale, 128
locality, 116

Marker, 147
memory hierarchy, 69
memory safety, 15, 71, 116
memory-safe, 71
metadata, 55
MPI, 71
multi-methods, 136
MultiJava, 137

native code, 32
nonblocking, 117
nonexcepting, 116, 117

object, 74
operational semantics, 144
overloading, 115, 142
ownership, 16

place, 71, 72, 110, 128
    shifting, 71
place-local, 71
place-safe, 71
place-shifting, 71, 72
plugin, 7, 9
point, 75, 109, 128
    addition, 109
    multiplication, 112
    order, 109
    projection, 112
    subtraction, 109
poset, 176
Pressburger arithmetic, 128
product-poset, 177
productivity, 116
projection, 114
Purdue Benchmark Suite, 34

PUT, 160

rank, 109, 114, 128
real-time Java, 18
refactoring, 56, 60
reflection, 2, 32, 136
REGION, 156
region, 15, 73, 75, 78, 110, 128
  construction, 110
  intersection, 110
  move, 110
  multiplication, 112
  of distribution, 112
  projection, 113
  union, 110
requirements engineering, 116
RETURN, 159
revirtualize, 7
root activity, 74
RTA, 8, 10
Runabout, 135

satisfiability, 90
scalar object, 74
SCC, 183
sealed package, 17
set difference, 75
sharing, 16
side-effects, 4
SMP, 70
speculative optimization, 7
SSA, 147, 148
stack allocation, 4
static analysis, 71, 129, 144
static check, 71
static widening, 21
subtyping, 78

Tarjan's algorithm, 183
Titanium, 78, 127–129
transfer function, 179
  monotone, 181
TSMI, 8
TYPE, 152
type checking, 71
type soundness, 71

union, 75

value, 128
var, 117
visitAppropriate, 141
visitee, 136
visitor, 136
visitor pattern, 136

walkabout, 136
widening, 21, 41, 60

X10, 15, 71, 78, 127, 128
X10 IR, 147
XTC, 15

ZPL, 73, 78, 127, 128

References

[ACL05] Eric Allan, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. "The Fortress Language Specification Version 0.618." Technical report, Sun Microsystems, April 2005.

[AH05] David Aspinall and Martin Hofmann. "Dependent Types." In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter Dependent Types, pp. 45–86. The MIT Press, 2005.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. "Alias Annotations for Program Understanding." In *OOPSLA Proceedings*, pp. 311–330, November 2002.

[AKV93] Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. "The Complexity of Set Constraints." In *CSL*, pp. 1–17, 1993.

[Alm97] Paulo Sérgio Almeida. "Balloon Types: Controlling Sharing of State in Data Types." In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pp. 32–59, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[Alm99] Paulo Sérgio Almeida. "Type-Checking Balloon Types." *Electrical Notes in Theoretical Computer Science*, **20**, 1999.

[Bac97] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, University of California, Berkeley, December 1997. Report No. UCB/CSD-98-1017.

[Bac02] Roland Backhouse. "Galois connections and fixed point calculus." In *Algebraic and coalgebraic methods in the mathematics of program construction*, pp. 89–148. Springer-Verlag New York, Inc., 2002.

[BBP02] Iain Bate, Guillem Bernat, and Peter Puschner. "Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis." In *Proc. 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 83–90, Apr. 2002.

[BD94] Manuel E. Benitez and Jack W. Davidson. “Target-specific Global Code Improvement: Principles and Applications.” Technical Report CS-94-42, University of Virginia, Charlottesville, VA 22903, 1994.

[BDF04] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. “Verification of Object-Oriented Programs with Invariants.” *Journal of Object Technology*, **3**(6):27–56, 2004. Preliminary version in Proceedings of Fifth Workshop on Formal Techniques for Java-like Programs, 2003.

[BGS00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. “ABCD: eliminating array bounds checks on demand.” In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 321–333, 2000.

[BH99] Jeff Bogda and Urs Hölzle. “Removing Unnecessary Synchronization in Java.” In *OOPSLA’99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pp. 35–46, Denver, CO, October 1999. ACM Press.

[Bla99] Bruno Blanchet. “Escape Analysis for Object Oriented Languages. Application to Java.” In *OOPSLA’99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pp. 20–34, Denver, CO, October 1999. ACM Press.

[Bla03] Bruno Blanchet. “Escape Analysis for Java: Theory and Practice.” *ACM Transactions on Programming Languages and Systems*, **25**(6):713–775, 2003.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. “Ownership Types for Safe Programming: Preventing Data Races and Deadlocks.” In *OOPSLA Proceedings*, pp. 211–230, November 2002.

[BN02] Anindya Banerjee and David A. Naumann. “Representation Independence, Confinement and Access Control.” In *Proceedings of POPL’02, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 166–177, 2002.

[BNR01] John Boyland, James Noble, and William Retert. “Capabilities for Aliasing: A Generalisation of Uniqueness and Read-Only.” In *ECOOP’01 — Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pp. 2–27, Berlin, Heidelberg, New York, 2001. Springer.

[Bok99] Boris Bokowski. “CoffeeStrainer: Statically-Checked Constraints on the Definition and Use of Types in Java.” In *Proceedings of ESEC/FSE’99*, pp. 355–374, Toulouse, France, September 1999.

[Boy01] John Boyland. “Alias burying: Unique variables without destructive reads.” *Software—Practice and Experience*, **31**(6):533–553, 2001.

[BS96] David F. Bacon and Peter F. Sweeney. “Fast Static Analysis of C++ Virtual Function Calls.” In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’96)*, pp. 324–341, San Jose, CA, 1996. *SIGPLAN Notices* 31(10).

[BSB03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. “Ownership Types for Safe Region-Based Memory Management in Real-Time Java.” In *ACM Conference on Programming Language Design and Implementation*, pp. 324–337, June 2003.

[BV99] Boris Bokowski and Jan Vitek. “Confined Types.” In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’99)*, pp. 82–96, Denver, CO, 1999.

[BW94] Lars Birkedal and Morten Welinder. “Binding-Time Analysis for Standard ML.” In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pp. 61–71, 1994.

[CC77] Patrick Cousot and Radhia Cousot. “Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points.” In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. “Constructive versions of Tarski’s fixed point theorems.” In *Pacific Journal of Mathematics 82*, pp. 43–57, 1979.

[CCD04] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. “The high-level parallel language ZPL improves productivity and performance.” In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[CCL00] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. “ZPL: A Machine Independent Programming Language for Parallel Computers.” *IEEE Transactions on Software Engineering*, **26**(3):197–211, March 2000.

[CDE05] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing.” In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 519–538. ACM SIGPLAN, 2005.

[CFR91] Ron Cryton, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.” *ACM Transactions on Programming Languages and Systems*, **13**(4), October 1991.

[Cha92] Craig Chambers. “Object-oriented multi-methods in Cecil.” In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pp. 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[Cha01] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language.* PhD thesis, University of Washington, November 2001.

[CHY95] Liling Chen, Luddy Harrison, and Kwangkeun Yi. “Efficient computation of fixpoints that arise in complex program analysis.” *Journal of Programming Languages*, **3**(1):31–68, 1995.

[Cla01] David Clarke. *Object Ownership and Containment.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[CLC00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. “MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java.” In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pp. 130–145, 2000.

[CM05] Daniel Chavarría-Miranda and John Mellor-Crummey. “Effective communication coalescing for data-parallel applications.” In *PPoPP*

'05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 14–25, New York, NY, USA, 2005. ACM Press.

[CMH02] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. "Protecting Free Expression Online with Freenet." *IEEE Internet Computing*, **6**(1):40–49, 2002.

[Cou01] Patrick Cousot. "Abstract Interpretation Based Formal Methods and Future Challenges." *Lecture Notes in Computer Science*, **2000**:138–156, 2001.

[CP95] Cliff Click and Michael Paleczny. "A Simple Graph-Based Intermediate Representation." In *The First ACM SIGPLAN Workshop on Intermediate Representations*, pp. 35–49, San Francisco, CA, 1995.

[CPN98] David G. Clarke, John M. Potter, and James Noble. "Ownership Types for Flexible Alias Protection." In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pp. 48–64. ACM, October 1998.

[CRC92] Patrick Cousot, Radhia, and Cousot. "Abstract Interpretation Frameworks." *Journal of Logic and Computation*, **2**(4):511–547, 1992.

[CRN03] Dave Clarke, Michael Richmond, and James Noble. "Saving the World from Bad Beans: Deployment-time Confinement Checking." In *OOPSLA Proceedings*, pp. 374–387, Anaheim, CA, November 2003.

[CS98] Robert Cartwright and Guy L. Steele, Jr. "Compatible Genericity with Run-time Types for the Java Programming Language." In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pp. 201–215. ACM, 1998.

[CS01] Bradford L. Chamberlain and Lawrence Snyder. "Array language support for parallel sparse computation." In *Proceedings of the ACM International Conference on Supercomputing*, pp. 133–145, 2001.

[CSW00] Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." In *Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in Lecture Notes in Computer Science, pp. 46–66. Springer-Verlag, 2000.

[CW03a] Dave Clarke and Tobias Wrigstad. "External Uniqueness Is Unique Enough." In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pp. 176–200. Springer, 2003.

[CW03b] David Clarke and Tobias Wrigstad. "External Uniqueness." In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA, January 2003.

[DC94] Jeffrey Dean and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis." Technical Report 94-12-01, Department of Computer Science, University of Washington at Seattle, December 1994.

[DCS02] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. "High-Level Language Support for User-Defined Reductions." *Journal of Supercomputing*, **23**(1):23–37, 2002.

[Deu95] Alain Deutsch. "Semantic Models and Abstract Interpretation Techniques for Inductive Data Structures and Pointers." In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 226–229, La Jolla, California, June 21–23, 1995.

[DG84] William F. Dowling and Jean H. Gallier. "Linear-time algorithms for testing the satisfiability of propositional Horn formulae." *Journal of Logic Progamming*, **1**(3):267–84, October 1984.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of object-oriented programs using static class hierarchy analysis." In Walter G. Olthoff, editor, *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7-11, 1995, Proceedings*, volume 952 of *Lecture Notes in Computer Science*, pp. 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[DLN96] David Detlefs, K. Rustan M. Leino, and Greg Nelson. "Wrestling with rep exposure." Technical report, Digital Equipment Corporation Systems Research Center, 1996.

[EES01] Jacob Engblom, Andreas Ermedahl, Mikael Sjoedin, Jan Gubstafsson, and Hans Hansson. "Worst-case execution-time analysis for embedded real-time systems.", 2001.

[ESS04] Kemal Ebcioğlu, Vijay Saraswat, and Vivek Sarkar. "X10: Programming for Hierarchical Parallelism and NonUniform Data Access (Extended Abstract)." In *Language Runtimes '04 Workshop: Impact of Next Generation Processor Architectures On Virtual Machines (colocated with OOPSLA 2004)*, October 2004. www.aurorasoft.net/workshops/lar04/lar04home.htm.

[ESS05] Kemal Ebcioğlu, Vijay Saraswat, and Vivek Sarkar. "X10: an Experimental Language for High Productivity Programming of Scalable Systems (Extended Abstract)." In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[FBB99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Fea93] Paul Feautrier. "Toward automatic partitioning of arrays on distributed memory computers." In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pp. 175–184, New York, NY, USA, 1993. ACM Press.

[Fou04] Eclipse Foundation. "Eclipse." http://eclipse.org/, 2004.

[GB93] Manish Gupta and Prithviraj Banerjee. "PARADIGM: a compiler for automatic data distribution on multicomputers." In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pp. 87–96, New York, NY, USA, 1993. ACM Press.

[GC01] David Grove and Craig Chambers. "A Framework for Call Graph Construction Algorithms." *ACM Transactions on Programming Languages and Systems*, **23**(6), November 2001.

[GHJ94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[GP04] Neal Glew and Jens Palsberg. "Type-Safe Method Inlining." *Science of Computer Programming*, **52**:281–306, 2004. Preliminary version in Proceedings of ECOOP'02, European Conference on Object-Oriented Programming, pages 525–544, Springer-Verlag (*LNCS* 2374), Malaga, Spain, June 2002.

[GPG05] Neal Glew, Jens Palsberg, and Christian Grothoff. "Type-Safe Optimisation of Plugin Architectures." In *Proceedings of SAS'05, Static Analysis Symposium*, pp. 135–154. Springer-Verlag (*LNCS* 3672), London, UK, September 2005.

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. "Encapsulating Objects with Confined Types." In *OOPSLA 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tampa, Florida*, pp. 241–253, 2001.

[Gro03] Christian Grothoff. "Walkabout revisited: The Runabout." In *ECOOP 2003 - Object-Oriented Programming*, pp. 103–125. Springer-Verlag, 2003.

[Gro06] Christian Grothoff. "XTC." http://grothoff.org/christian/xtc/, 2006.

[GTZ98] Daniela Genius, Martin Trapp, and Wolf Zimmermann. "An Approach to improve Locality using Sandwich Types." In *Proceedings of the 2nd Types in Compilation workshop*, volume LNCS 1473, pp. 194–214, Kyoto, Japan, March 1998. Springer Verlag.

[Han72] Per Brinch Hansen. "Structured multiprogramming." *CACM*, **15**(7):574–578, July 1972.

[Her03] Stephan Herrmann. "Object Teams: Improving Modularity for Crosscutting Collaborations." In *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in Lecture Notes in Computer Science, pp. 248–264. Springer-Verlag, 2003.

[HLW92] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. "The Geneva Convention on the Treatment of Object Aliasing." *OOPS Messenger*, **3**(2):271–285, April 1992.

[Hoa74] C.A.R. Hoare. "Monitors: An Operating System Structuring Concept." *CACM*, **17**(10):549–557, October 1974.

[Hog91] John Hogg. "Islands: Aliasing Protection in Object-Oriented Languages." In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 271–285, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Inc05] Cray Inc. "The Chapel Language Specification Version 0.4." Technical report, Cray Inc., February 2005.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ." *ACM Transactions on Programming Languages and Systems*, **23**(3):396–450, May 2001.

[KFF98] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. "Synthesizing Object-Oriented and Functional Design to Promote Reuse." *Lecture Notes in Computer Science*, **1445**:91–113, 1998.

[KJP77] Donald Knuth, Jr James H. Morris, and Vaughan Pratt. "Fast Pattern Matching in Strings." *SIAM Journal on Computing*, **6**(2):323–350, 1977.

[KK98] Ken Kennedy and Ulrich Kremer. "Automatic data layout for distributed-memory machines." *ACM Trans. Program. Lang. Syst.*, **20**(4):869–916, 1998.

[KM95] Stuart J. Kent and Ian Maung. "Encapsulation and Aggregation." In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*, pp. 227–238. Prentice Hall, 1995.

[LA00] Ben Liblit and Alexander Aiken. "Type Systems for Distributed Data Structures." In *Proceedings of POPL'00, 27nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 199–213, 2000.

[LK02] Peizong Lee and Zvi Meir Kedem. "Automatic data and computation decomposition on distributed memory parallel computers." *ACM Trans. Program. Lang. Syst.*, **24**(1):1–50, 2002.

[LM04] K. Rustan M. Leino and Peter Müller. "Object Invariants in Dynamic Contexts." In *Proceedings of ECOOP'04, 16th European Conference on Object-Oriented Programming*, pp. 491–516, 2004.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly & Associates, 1992.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, USA, 1997.

[Mad95] Ole Lehrmann Madsen. "Open issues in object-oriented programming – a scandinavian perspective." *Software–Practice and Experience*, **25**(S4):3–43, December 1995.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. "Parameterized Types for Java." In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 132–145, New York, NY, 1997.

[Mic06] Sun Microsystems. "The Java Extension Meachanism for Support of Optional Packages." http://java.sun.com/j2se/1.5.0/docs/guide/extensions/, 2006.

[MKG06] Peter Mell, Karen Kent, Ashish Goel, Ellery Horton, Tanyette Miller, Robert Chang, Michael Reilly, and Kathy Ton-Nu. "National Vulnerability Database." http://nvd.nist.gov/, 2006.

[MMM90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. "Strong typing of object-oriented languages revisited." In *Proceedings of OOPSLA'90*, pp. 140–150. SIGPLAN, ACM Press, 1990.

[MP99] Peter Müller and Arnd Poetzsch-Heffter. "Universes: A Type System for Controlling Representation Exposure." In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[Mul01] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Also as LNCS 2262, Springer-Verlag, 2002.

[Nec97] George Necula. "Proof-Carrying Code." In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 106–119, 1997.

[Nie89] Flemming Nielson. "The Typed lambda-Calculus with First-Class Processes." In *Proceedings of PARLE'89*, pp. 357–373, 1989.

[NPV98] James Noble, John Potter, and Jan Vitek. "Flexible Alias Protection." In *Proceedings of ECOOP'98*, volume 1543 of *LNCS*, Brussels, Belgium, July 20 - 24 1998. Springer-Verlag.

[OAC05] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stephane Micheloud, Michel Schniz, Erik Stenman, and Matthias Zenger. "The Scala Language Specification 1.0." Technical report, Programming Methods Laboratory, EPFL, Switzerland, October 2005.

[OVM04] OVM Consortium. "OVM." http://ovmj.org/, 2004.

[Pal98] Jens Palsberg. "Equality-Based Flow Analysis versus Recursive Types." *ACM Transactions on Programming Languages and Systems*, **20**(6):1251–1264, 1998.

[PBF03] Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. "Engineering a Customizable Intermediate Representation." In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 2003)*. ACM SIGPLAN, 2003.

[Pes04] Slava Pestov. "jEdit." http://jedit.org/, 2004.

[PFT92] William H. Press, Brian. P. Flannery, Saul A. Teukolsky, and William T. Vetterling. "Successive Overrelaxation (SOR)." In *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pp. 866–869. Cambridge University Press, 1992.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[PJ98] Jens Palsberg and C. Barry Jay. "The Essence of the Visitor Pattern." In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pp. 9–15, 1998.

[PNC04] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. "Featherweight Generic Confinement." In *Workshop on Foundations of Object-Oriented Languages*, 2004.

[PO95] Jens Palsberg and Patrick M. O'Keefe. "A Type System Equivalent to Flow Analysis." *ACM Transactions on Programming Languages and Systems*, **17**(4):576–599, July 1995. Preliminary version in Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

[PP01] Jens Palsberg and Christina Pavlopoulou. "From Polyvariant Flow Information to Intersection and Union Types." *Journal of Functional Programming*, **11**(3):263–317, May 2001. Preliminary version in Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 197–208, San Diego, California, January 1998.

[PS91] Jens Palsberg and Michael I. Schwartzbach. "Object-Oriented Type Inference." In *Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 146–161, Phoenix, Arizona, October 1991.

[PS94] Jens Palsberg and Michael I. Schwartzbach. "Binding-time analysis: abstract interpretation versus type inference." In *ICCL'94*, pp. 289–298, 1994.

[Pug91] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis." In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 4–13, New York, NY, USA, 1991. ACM Press.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. "Data-Flow Analysis of Program Fragments." In *7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'99)*, pp. 235–252, 1999.

[RS06] Gabriel Dos Reis and Bjarne Stroustrup. "Specifying C++ concepts." In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 295–308, New York, NY, USA, 2006. ACM Press.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, CMU, May 1991. CMU–CS–91–145.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. "Implementation of an array bound checker." In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 132–143, New York, NY, USA, 1977. ACM Press.

[SLG99] Anthony Skjellum, Ewing Lusk, and William Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface.* MIT Press, 1999.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-Valued Logic." In *Symposium on Principles of Programming Languages*, pp. 105–118, 1999.

[SS05] Christian Skalka and Scott F. Smith. "Static Use-Based Object Confinement." *International Journal on Information Security*, **4**(1-2):87–104, 2005. Preliminary version in Proceedings of Foundations of Computer Security, volume 02-12 of *DIKU technical reports*, pages 117–126.

[Tar72] Robert E. Tarjan. “Depth First Search and Linear Graph Algorithms.” *SIAM Journal on Computing*, **1**(2):146–160, June 1972.

[Tho97] Kresten Krab Thorup. “Genericity in Java with Virtual Types.” *Lecture Notes in Computer Science*, **1241**:444–471, 1997.

[UCB05] P. N. Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Ben Liblit, Geoff Pike, Jimmy Su, and Katherine Yelick. “Titanium Language Reference Manual.” Technical report, U.C. Berkeley, 2005.

[VB01] Jan Vitek and Boris Bokowski. “Confined Types in Java.” *Software Practice and Experience*, **31**(6):507–532, 2001.

[Wad90] Philip Wadler. “Linear types can change the world!” In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pp. 347–359. North Holland, 1990.

[Wei94] Pei Y. Wei. “A Brief Overview of the VIOLA Engine, and its Applications.” http://xcf.berkeley.edu/~wei/viola/violaIntro.html, 1994.

[WF94] Andrew Wright and Matthias Felleisen. “A Syntactic Approach to Type Soundness.” *Information and Computation*, **115**(1):38–94, 1994.

[XP98] Hongwei Xi and Frank Pfenning. “Eliminating Array Bound Checking Through Dependent Types.” In *Proceedings of PLDI’98, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 249–257, 1998.

[XP99] Hongwei Xi and Frank Pfenning. “Dependent Types in Practical Programming.” In *Proceedings of POPL’99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp. 214–227, 1999.

[Yi93] Kwangkeun Yi. *Automatic Generation and Management of Program Analyses.* PhD thesis, University of Illinois at Urbana-Champaign, August 1993. Report UIUCDCS-R-93-1828.

[Yot04] Kamen Yotov. “Bernoulli Compiler Construction Kit.”, 2004. http://iss.cs.cornell.edu/bcck.htm.

[ZFA00] Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. “Sealed Calls in Java Packages.” In *OOPSLA ’2000 Conference Proceedings*, ACM SIGPLAN Notices, pp. 83–92. ACM, October 2000.

[ZNV04] Tian Zhao, James Noble, and Jan Vitek. "Scoped Types for Real-time Java." In *Proceedings of 25th IEEE Real-Time Systems Symposium*, pp. 241–251, 2004.

[ZPV06] Tian Zhao, Jens Palsberg, and Jan Vitek. "Type-Based Confinement." *Journal of Functional Programming*, **16**(1):83–128, 2006. Preliminary version, entitled "Lightweight confinement for Featherweight Java", in Proceedings of OOPSLA'03, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pages 135-148, Anaheim, California, October 2003.